The cT Programming Language

Version 3.0

Created by David Andersen, Bruce Sherwood, Judith Sherwood, and Kevin Whitley

Center for Innovation in Learning
Carnegie Mellon University
Pittsburgh

http://cil.andrew.cmu.edu/ct.html
Contents

1. THE CT PROGRAMMING LANGUAGE .................................................... 4
   cT Menu Options ............................................................................................ 11
   Additional Language Aspects ............................................................................. 17

2. GRAPHICS & TEXT .............................................................................. 32
   Basic Graphics & Text Commands ....................................................................... 34
      Describing the Screen ...................................................................................... 34
      Displaying Text and Variables ........................................................................ 39
      Lines, Circles, Boxes, Etc ............................................................................... 52
   Mode, Pattern, Thick, Cursor, & Clip .................................................................. 60
   Inhibit and Allow in Graphics ........................................................................... 65
   Color ............................................................................................................. 71
   Images ........................................................................................................... 88
   Animations ....................................................................................................... 98
   Making a Graph ............................................................................................... 102
   Relative Graphics Commands ......................................................................... 112

3. VIDEO & SOUND ............................................................................... 117
   Video Commands .......................................................................................... 117

4. MOUSE & KEYSET INTERACTIONS .................................................. 124
   Mouse, Single Key, & Timed Pause ................................................................... 125
   Pull-down Menus ........................................................................................... 136
   Buttons, Dialog Boxes, Sliders, & Edit Panels ................................................. 142
   Scrolling Text Panels ..................................................................................... 149
   Word & Number Input .................................................................................... 158
      Basic Judging Commands .............................................................................. 159
      Modifying Judging Defaults .......................................................................... 169
      Inhibit and Allow in Judging ......................................................................... 177
      Specialized Judging Commands .................................................................. 181
1. The cT Programming Language

Purpose of cT

This is a print version of the on-line help for cT 3.0 (August 1999). See the on-line help on the Window menu for updated information, and a history of previous versions. Also see http://cil.andrew.cmu.edu/ct.html.

The cT programming language is an algorithmic language like C, Pascal, Fortran, and Basic, but greatly enhanced by multimedia capabilities, including easy-to-use support for color graphics, mouse interactions, and even movies in QuickTime or Video for Windows format.

The cT programming language offers easy programmability of multimedia programs, with portability across Macintosh, Windows, and Unix.

The cT programming environment offers

on-line help with executable program examples,
a graphics editor for automatic generation of graphics commands,
incremental compiling to provide rapid turnaround, and detailed error diagnosis.

When is cT the right tool?

There are many excellent applications available for creating pictures and diagrams, and for making multimedia presentations, without having to write your own computer program.

However, it is sometimes the case that doing something really new and different is hard to do with these nonprogramming applications, because they often don't provide enough control of interactions and enough calculational capability to do what you really want to do.

cT offers the open-ended flexibility and power associated with programming languages but eliminates many of the difficulties and complexities usually associated with using a programming language.

Credits

cT has been developed in the Center for Innovation in Learning at Carnegie Mellon University in Pittsburgh by David Andersen, Bruce Sherwood, Judith Sherwood, and Kevin Whitley. cT is a trademark of Carnegie Mellon University.

Special thanks to: Andrew Appel, Bill Arms, Steven Bend, Ruth Chabay, Preston Covey, Andy van Dam, Ken Friend, Fred Hansen, Stacie Hibino, Chris Koenigsberg, Peter Kornelisse, Jay Laefler, Jill Larkin, Michael LoBue, David Madden, Gregg Malkary, Ben McClure, Jim Morris, Tom Neuendorffer, Tom Peters, Carol Schectic, and David Trowbridge.

License information


Permission to reproduce and use cT for internal use is granted provided the copyright and "No Warranty" statements are included with all reproductions. cT may also be redistributed without charge provided that the copyright and "No Warranty" statements are included in all redistributions.
An Example Program

Given below is a short program written in cT. This program illustrates graphics, rich text, animation, and mouse interactions.

**Copy** the example into the start of the program window (after the $syntaxlevel line).

**Execute** the program by choosing "Run from beginning" on the Option menu.

Click the button; draw in the box; use the pull-down menu to clear the box.

**Change the size of the window** and see how the program starts all over again.

(On a Macintosh, drag the lower-right corner of the window; no drag box is shown.)

To **stop**, choose "Quit running" from the Option menu, which does **not** quit cT.

After running the program, read through the program carefully, one line at a time, to see how the effects are achieved. You may want to look up detailed information on individual commands. Try changing the program and running it again. **Be sure to read the next topic, "Further Discussion".**

**Example program to be copied, down to "End of example program":**

```
unit  cTexample
* (An asterisk * at the start of a line introduces a comment.)
* cT programs are made up of "units", like subroutines in other languages.
* Declare a "button"-type variable named "animate" to be used by unit cTexample
*    in the creation of a "button", an object to click:
    button: animate $$ declare a "button" variable; note initial TAB
    $$ a "$$" introduces a comment at end of a line

*-- Specify the font to be used to output text:
font  zsans,15 $$ choose sans-serif font, 15 pixels high
* The TAB key is **required** to separate "commands" (such as "font") from
*    the "tag" (the arguments for that command, such as "zsans,15").

*-- Display text that is red, bold, and italic; upper-left corner is location 0,0:
color zred $$ choose color palette slot number "zred"
    at 10,20 $$ 10 pixels to right, 20 pixels down from upper-left corner
write Look! $$ display Look! at location 10,20

*-- Display a solid blue rectangle:
color zblue $$ choose color palette slot number "zblue"
    fill 70,12;115,40 $$ fill a rectangle (in blue)

*-- Erase a disk inside the blue rectangle, making a white hole (you might
*    rerun the program to see this white hole in the blue rectangle):
mode erase $$ start erasing
```

NO WARRANTY. cT IS FURNISHED ON AN "AS IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY OF RESULTS OR RESULTS OBTAINED FROM USE OF cT. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK OR COPYRIGHT INFRINGEMENT.
INTRODUCTION

at 92,26 $$ centered at location 92,26
disk 9 $$ draw solid circular disk with radius 9
mode write $$ stop erasing; reset to normal displaying

*-- Display a thick "vector" (a line with an arrowhead at the end):
color zblack $$ choose black color
thick 3 $$ lines will be 3 pixels thick
* Give starting and ending locations of "vector", with arrowhead at end:
vector 125,26;170,26 $$ thick "vector" (line with arrowhead)
thick $$ reset to default 1-pixel thickness

*-- Display a solid red oval, specified by a bounding rectangle:
color zred $$ choose color palette slot number "zred"
* Give corners of an invisible rectangle that bounds the oval:
disk180,15;235,37 $$ red solid (oval) disk bounded by this rectangle

*-- Connect three points to draw two straight black lines:
color zblack $$ black lines
* Note semicolons separating the three sets of x-y coordinate pairs:
draw225,7; 240,26; 225,45 $$ draw two lines

*-- Display window size near the lower-right corner of the window,
* by writing text with numbers, and displaying a vector:
* System variables zxmax, zymax give the window size, and
* we choose a location 90 to the left and 25 up from the
* lower-right corner of the window:
at zxmax-90,zymax-25 $$ near lower-right corner
* In the -write- statement below, note special "embed" symbols <| and |>
* an "embedded" -show- command displays a numerical value.
* The effect is to display a left parenthesis, the value of zxmax,
* a comma, the value of zymax, and a right parenthesis:
write (<|show,zxmax|>,<|show,zymax|>)

* System variables zwherex, zwherey give current writing location;
* in this case, at the location on the screen of the end of the text displayed
* by the previous -write- statement:
vector zwherex,zwherey+8; zxmax,zymax; 10 $$ 10-pixel arrowhead

*-- Display "rich" text (that is, text with styles such as bold, italic,
* subscript, or superscript, applied with the "Style" menu):
at 10,60 $$ position for writing at 10,60
write Text with styles -- H₂O, x³.

at 10,90 $$ position for writing at 10,90
write Click this button:

*-- Create a "button" to click:
* The tag (the arguments after the TAB) lists the button variable (animate),
* two opposite corners of the rectangular button,
* the unit (MoveBall, found below) to be done when you click the button,
* and the text ("Animation") to appear in the button:
button animate; 10,105;100,125; MoveBall; "Animation"
at 10,140 $$ position for writing at 10,140
write Click and drag in the box below.
Use "Special" menu to clear.

-- Create a pull-down menu item:
* The tag lists the name (Special) that will appear in the menu bar,
* the item (Clear Drawing Area) to appear on the pull-down menu,
* and the unit (ClearSketch) to be done when you choose that menu item:
menu Special; Clear Drawing Area: ClearSketch

-- Provide a sketch pad to draw on with the mouse:
* The following -do- command calls the subroutine (unit) named Sketch,
* passing to the subroutine the values of four arguments needed by Sketch,
* a unit found near the end of the program. The four arguments are the
* coordinates of two corners of the box to draw in:
do Sketch(10, 170, zxmax-10, zymax-30)

***************

unit MoveBall
* This subroutine unit is executed when you click the "Animation" button.
* Declare variables to be used by this unit:
  integer: xx, yy $$ integer variables for ball location
  integer: BALL=68 $$ constant -- will display icon no. 68
  integer: STEP=3 $$ constant -- step size of animation
  integer: X0=130, Y0=110 $$ constants -- initial location of ball
  float: time $$ floating-point variable for keeping time

* An icon file contains a set of small images; "zicons" is a set included with cT:
icons zicons $$ choose icon file "zicons" (in which icon no. 68 is a ball)
color zblue

* A -calc- statement assigns values to variables; ":=" means "assign value".
* In successive assignment statements, you need not repeat the command "calc":
calc time := zclock $$ get current time (zclock) in seconds
  xx := X0 $$ assign xx and yy to initial location
  yy := Y0

* Iterative loop, with xx assigned values from 130 to (zxmax+10) in steps of 3:
loop xx := X0, zxmax+10, STEP $$ go beyond zxmax (outside window)
  * Note the required TAB indenting inside this loop.

  * Move icon no. 68 (BALL) from old location to new location:
  * The tag specifies that we're moving "icons" (from "zicons"),
  * xx, yy is the old location of the ball,
  * xx+STEP,yy := Y0+15sin(xx/20) is the new location of the ball,
  * and BALL is the number (68) of the icon to be moved:
move icons; xx, yy; xx+STEP,yy := Y0+15sin(xx/20); BALL

  * Wait 0.02 seconds before making another move, to make the
  * animation run about the same speed on fast and slow computers.
  * Loop while elapsed time (zclock-time) is less than 0.02 sec:
INTRODUCTION

loop \( (z\text{clock-time}) < 0.02 \) $$ \text{loop while this is TRUE}$$
* Do nothing in this loop; just wait for the right time.
endloop

calc time := \text{zclock} $$ \text{$$reset to current time}$$
endloop

color \text{zblack} $$ \text{$$reset to black after using blue for the ball}$$

***************

unit Sketch(x1, y1, x2, y2) $$ \text{$$subroutine for drawing}$$
* The 4 arguments give the corners of a box within which to draw with the mouse
integer: x1, y1, x2, y2 $$ \text{$$declare arguments are integers}$$
integer: drawcolor $$ \text{$$declare another integer variable}$$

box x1, y1; x2, y2 $$ \text{$$draw bounding box}$$
calc drawcolor := \text{zred} $$ \text{$$first drawing color will be red}$$

loop $$ \text{$$unconditional loop; loop forever}$$
pause keys=touch $$ \text{$$wait for mouse click}$$
* System variables \text{ztouchx, ztouchy} give location of the mouse click:
at ztouchx,ztouchy $$ \text{$$start drawing at mouse location}$$

* The -clip- command restricts drawing to stay inside a box:
clip x1+1, y1+1; x2-1, y2-1 $$ \text{$$sketch with lines 2 pixels thick}$$
thick 2 $$ \text{$$choose color for sketching}$$
loop $$ \text{$$nested unconditional inner loop}$$
* Wait for a mouse "move" event, or a mouse "up" event
* (release of the mouse button), for left mouse button
* (a single-button mouse has a "left" button):
pause keys=touch(left: move,up) $$ \text{$$"move" or "up"}$$

* Starting with ";" means draw from current location:
draw ; ztouchx,ztouchy $$ \text{$$draw to mouse location}$$

* If mouse button comes up, get out of the inner loop.
* The system variable \text{zkey} contains a numerical code
* representing the most recent event, and \text{zk(left: up)}
* is the numerical code for a mouse "up" event:
outloop zkey = zk(left: up)
endloop

calc drawcolor := drawcolor+1 $$ \text{$$increment palette slot}$$
* In the following expression, "=" means "equality test", in contrast
* to ":=" which means "assign a value to a variable".
if drawcolor = 8 $$ \text{$$start over if reached color slot no. 8}$$
calc drawcolor := \text{zred} $$ \text{$$reset to slot no. "zred"}$$
endif

* Display a message in the current color, overwriting old message:
at x1,y2+2 $$ just below left corner of box
write You can draw again!
endloop

***************
unit ClearSketch $$ subroutine driven by menu selection
erase 11,171; zxmax-11,zymax-31 $$ erase inside of box
* End of example program.

Further Discussion

Execution options: To execute an arbitrary unit, click the mouse somewhere in the unit and choose "Execute current unit" from the Option menu. Or use "Select unit" to specify which unit to start from. Try executing unit "MoveBall" in the example program to see what happens.

The difference between "Run" and "Execute" is that if you "Execute", cT automatically quits running the program when there is nothing left to do, which is often convenient. If you "Run", you have to explicitly "Quit running", or click in the editing window to stop execution.

You cannot run or execute a subroutine unit that has arguments such as "Sketch(x1, y1, x2, y2)", because in that case cT doesn't know what values to use for these arguments.

Graphics editing: Be sure to study the topic "Graphics Editing", which explains how you can get cT to generate or modify program statements graphically, rather than having to type the statements yourself.

Graphics coordinates: The standard "absolute" coordinate origin is at the upper-left corner of the window, with x running to the right and y running down, so that 200,50 is 200 pixels to the right and 50 pixels down from the origin. You can also set up your own coordinate systems using "graphing" or "relative" graphics commands. You can read about these features by clicking a topic in the "See Also" list below.

What next? You now know enough to get started writing your own programs. At some point you should read through the next few topics:

The topic "cT Menu Options" explains the function of the various cT menu options.

The topic "Additional Language Aspects" provides overviews of various general aspects of the cT language, including a section on "Differences from Other Languages" of particular interest if you have written programs before.

The topic "Calculation Introduction" listed below gives an overview that is very important if you have not written computer programs before, and it can usefully be skimmed by experienced programmers to see the basic syntax of cT numerical calculations.

Sample programs: Included with cT are many medium- and large-size programs that you can study, use, or modify. See "Sample Programs" for descriptions of these programs. The program exercise.t contains a set of exercises to help you learn the basic concepts of programming in cT. To work with this program, choose "Open" on the File menu.

See Also:

Graphics Editing (p. 10)
Making a Graph (p. 102)
Relative Graphics Commands (p. 321)
cT Menu Options (p. 11)
Additional Language Aspects (p. 17)
INTRODUCTION

Differences from Other Languages (p. 17)
Calculation Introduction (p. 187)
Sample Programs (p. 28)

Graphics Editing

cT lets you generate graphics statements in an automated way:

Create a new unit just after the initial $syntaxlevel line:
Type the command name "unit" at the start of the line,
then press TAB, then type "test" for the name of the unit.

Press Return, then type the command name "draw", then press TAB.
Make sure that you leave the editing cursor located just after the TAB.

Choose "Select unit" from the Option menu and choose "test" for the unit.

Click the mouse somewhere in the execution window (the graphics window).
Note that an x-y coordinate pair for that point is added to your program.

Move the mouse to another location in the execution window and click again.
A second x-y coordinate pair is added to your program, and the program
is automatically compiled and executed, showing a straight line.
If you don't see a straight line, choose "Select unit" on the Option menu
and give the name of the unit ("test") that your -draw- statement is in.
It is the selected unit that is executed when you do graphics editing.

Be sure you're not running ("Quit running" if necessary),
click again, and you will see two straight lines connecting the points.

Next, select one of the x-y coordinate pairs in the program window (that is, highlight the x-y pair by
dragging the mouse across it). Then click in the execution window. The selected coordinate pair is changed, the
modified unit is recompiled and reexecuted, and you see the altered lines.

There is a useful graphics editing trick for finding out the coordinates of some part of a display. Start a new line
with an "*" so that this line is a mere comment. Then click in the execution window at one or more locations
of interest, and the coordinates are shown in the comment line, without reexecution of the program.

Command list: You can also get cT to type command names for you. On the Window menu, choose
"Commands" to see a complete list of all cT commands, organized into groups (graphics, calculational, etc.).
Position the commands window in such a way that you can see the program and execution windows as well.

In the commands window, find the word "box" and click it.
The command "box" followed by a TAB appears in the program window.
Click at two locations in the execution window to complete and see the box.

In addition to providing a typing shortcut, the commands list is useful for seeing a complete list of all the cT
commands, grouped by command type.

When the commands window is active, its menu lets you choose which group to display; you can also scroll to
that group, or expand the window to see all the commands.
cT Menu Options

File Menu

On the File menu are standard options such as Open, Save, and Quit. There are also some cT-specific options that are explained here.

**Auxiliary file:** The "Auxiliary file" option lets you examine auxiliary files, such as library (-use-) files or data files. The "Open" option lets you change the primary program file, which is the file that is compiled and executed when you choose "Run from beginning". Use "Open" when you want to change from working on one program to working on a different program.

**Insert file:** This option lets you insert program statements contained in another file, or an image. The text or image is inserted at the current location of the editing cursor. An image would normally be inserted into a -write-, -text-, or -string- statement. If you choose a file containing an image, you are asked whether to insert just the text that may accompany the image, or to insert the image either in monochrome or color format.

Images can be in PICT or BMP or PPM or PCX format, and once the image has been inserted into your program it is portable across different types of computers.

**Saving and checkpointing:** While you are developing your program, you should remember to save it regularly. That way, if there is some sort of a malfunction (e.g., a loss of power or a system error), you will lose only the work you have done since you last saved your file. You can do this at any time by choosing "Save" on the File menu.

In addition, cT has a special protection feature that regularly and automatically saves a special kind of backup file, called a "checkpoint" file, if you do not save the file yourself. A checkpoint file differs from a regular backup file in that

- It is automatically created and saved by cT (in the same directory as your program).
- It has the same name as your program plus ".CKP" at the end.
- It is automatically removed by cT if you "Save" your program before quitting cT.

If you do not "Save" your program before leaving it, then the checkpoint file will be left on your disk as a safeguard. Whenever you return to programming, you will find a file ending with ".CKP" on your disk. If you decide to open the ".CKP" file and work with it, you need to rename it to indicate that it is no longer a cT-generated backup file, which may be removed when you quit.

In the "Preferences" on the Option menu you can specify the time between checkpoints.

*See Also:*
   Images & Files (p. 91)

Edit Menu: Comments, Indents, and Hidden Units

On the Edit menu are standard options for Cut, Paste, and Copy. There are also some cT-specific options for controlling comments, indents, and hidden units.

**Comments:** A well-commented program is easier for you to work with and makes it easier for another person to maintain your program. Blank lines may be left in the program to improve readability.

* any line that starts with an asterisk is ignored
INTRODUCTION

do someunit $$ an end-of-line comment

* An asterisk and/or blank lines
* at the end of each unit makes
* your code easier to read
*

There are no end-of-comment markers in cT. Comment markers refer to one line at a time. To write a long comment or to temporarily disable several lines of code, you must put an asterisk on each line. The $$ marker is used to put a comment at the end of any line of code.

The editor provides a way to insert or remove asterisks before several lines of code with one action. Use the mouse to select the lines to be modified, then choose "Comment Lines" or "Un-Comment Lines" from the Edit menu. This is extremely useful when debugging a program. A standard debugging technique is to insert a -show-command to display the value of some variable that is in doubt, followed by a -pause- to suspend execution. Using the menu options you can quickly comment in or comment out this debugging code.

**Indents:** Indenting is required for commands inside a -loop-, -if-, or -case- structure. If such a structure is nested inside another one, multiple indents are required:

```ct
loop ii := 1, 5
  loop jj := 1, 10
    ***
  endloop
endloop
```

You can use either TAB or period-TAB, whichever you find more readable:

```ct
loop ii := 1, 5
  . loop jj := 1, 10
  . ***
  . endloop
endloop
```

You can easily change the indent level of a group of lines. Use the mouse to select the lines to be changed, then choose "Indent Lines" or "Un-Indent Lines" on the Edit menu one or more times.

**Hidden units:** When working on a long program, it is convenient to "hide" the many units that have already been completed, so that only the units you are working on are fully displayed. To hide a unit or to redisplay a hidden unit, place the mouse cursor in that unit, then choose "Hide Units" or "Show Units" on the Edit menu. A hidden unit shrinks to just the unit statement with a line below it indicating that there are hidden contents. To hide or show several adjacent units, select them and then choose "Hide Units" or "Show Units". There are also options to "Hide All Units" or "Show All Units".

**Compose character:** This option on the Edit menu (or its keyset equivalent) is used to input non-English characters. See the topic, "Typing Non-English Text".

**See Also:**
Arithmetic Operators (p. 200)
Typing Non-English Text (p. 184)
Other Menus

Search menu: The Search menu lets you search for specific strings of characters in your program, and optionally to replace these strings with other strings.

Style menu: The Style menu offers the usual options for making mouse-selected text italic, or bold, or superscripted, etc. The Bigger and Smaller options make the selected text bigger or smaller than the default font size set by a -font- command (or by the default font size). The difference between Left Justify and Full Justify is that left-justified text does not automatically wrap to the next line at the right margin of the text. By default, in a new file lines are left justified, so that the display of long lines is truncated at the right edge of the window. If you would like some lines to wrap around, make them full justified.

Color menu: This is an extension of the Style menu. It lets you impose a color on mouse-selected text, and this color overrides the color specified by a preceding -color- command.

Font menu: The Font menu lets you impose a particular font on mouse-selected text. For an explanation of which fonts correspond to the standard cT fonts zsans, zserif, zfixed, and zsymb, see "Generic Font Names". The "Icon" entry on the Font menu lets you insert icons (graphic characters) into your text (see the topic "icons Selecting an Icon").

Option menu: See "Preferences" and "Making and Saving Binaries". The various Run and Execute options are described in "An Example Program".

Window menu: Choose to see the Help or the Commands window, or which window should come forward. On the Macintosh, choosing "Mac SE" or "Mac II" makes the execution window be the size that is standard on those Macintosh models.

See Also:
- Generic Font Names (p. 327)
- Preferences (p. 13)
- Making and Saving Binaries (p. 16)
- An Example Program (p. 5)
- icons Selecting an Icon (p. 93)

Preferences

It is possible to specify editing preferences for such things as font, font size, and default color of the program displayed in the program window, and for more technical options as well. Choose "Preferences" from the Option menu and make your choices.

The choices for what font to use while editing your program include zsans, zserif, and zfixed. Font size is in points. Font face is typically "normal" but can be italic or bold.

Tab width is the width of a tab in digit widths (that is, a value of 8 means the width of eight 0's).

Foreground and background colors specify the foreground and background colors of the program text. Colors can be black, white, red, green, blue, cyan, magenta, and yellow.

Checkpoint specifies the number of seconds between saving a .CKP checkpoint version of the file.

Temp size is the maximum number of bytes of disk and extended-memory space that cT will use for its own purposes while running. This space is used whenever the program you are running needs more storage space than is available inside the computer's working memory. The default is one megabyte.
INTRODUCTION

Turning snapshot on means that cT makes a copy of the execution screen whenever you stop running. The menu option "Show exec window" shows you that snapshot. This is especially useful when doing graphic editing on a small screen, if the program window covers up the execution window.

Check "window positions" if you would like cT to remember your final arrangement of source and execution windows and use these positions the next time you use cT.

These preferences are stored in a preferences file. On Unix, if there is no machine-specific preference file (".ct.prf.dec3100 for example), it will look for a machine-independent copy (.ct.prf).

Printing

Printing the program listing: To print your program statements, choose the option "Print..." on the File menu. If you want to print only a portion of the program, copy and paste the statements into a new file. Also note that hidden units take up only two lines on the printout. On some machines, styles (such as italics) or embedded pictures may be ignored in the printout.

Printing sections of the on-line help: Create a new file. Copy the section of the on-line help and paste it into the edit window, then print as with printing any program listing.

Printing directly to the printer: Any marker (string) variable can be sent directly to a connected printer by using the -print- command. The marker can contain styled text (bold, italic, multiple fonts, etc.) and even embedded (pixel) graphics, as is discussed in the section below ("Printing a copy of the screen directly to the printer").

dialog page setup $$ optional; asks for portrait or landscape, etc.
dialog print $$ asks how many copies, etc.
print (marker variable) $$ contents of marker sent to printer

If you have never executed -dialog page setup- during this session, -dialog print- will automatically precede the print dialog box with the page setup dialog box. If you have never executed -dialog print- during this session, the -print- command will fail and set zreturn to 3 ("file not open").

For all three commands, zreturn is set to -1 if it works; otherwise it is set like file errors (e.g. 4 for "printer not found," etc.). Also, zretinf is 0 if the dialog box did not work, 1 if the "ok" button was pressed, 2 if the "cancel" button was pressed. It is important to check the zreturn and zretinf values, because lots of things can go wrong, including the user canceling the operation (in which case zreturn will be -1, but zretinf will be 2).

Since the printer dialog boxes and status messages come up in locations that are not under cT's control, the entire executor window is saved and restored before and after the dialog boxes and the print status message. This can look wierd, but it does avoid having to take in a reshape event and starting execution over again from the main unit. This is also a possible source of zreturn failure on a machine that is too short on memory to save a copy of the screen.

Printing a copy of the screen directly to the printer: The preceding discussion explained how to print the contents of a marker (string) variable. To print a pixel copy of the screen directly to the printer, first use -get- to place the image into a screen variable, then use -style- to impose this image on a marker variable, which can then be printed:

unit PrintScreen
screen: pimage
marker: pstring
get pimage; zxmin,zymin; zxmax,zymax
Creating a file of screen graphics to be printed later: There are several different methods for printing a display generated by cT in the execution window. While running your program from within cT Create, you can make

- a PICT image file on a Macintosh, or
- a BMP image file on MS-Windows, or
- a PPM (Portable Pix Map) image file on Unix

by choosing "Make PICT" or "Make BMP" or "Make PPM" or on the Option menu. This menu choice triggers a reshape event, with execution starting over from the start of the current main unit. An image file is created that is labeled "yourfile" or "yourfile.bmp" or "yourfile.ppm" or "yourfile.pcx". The image file is completed when you reach the end of the main unit, or execute a -pict- command, or execute a -jump- command, or quit running. If you make additional pictures, the file names contain a number indicating the sequence in which the image files were made. These menu options are not present when running a compiled binary with the cT Executor.

The -pict- command lets you start and stop the creation of an image file under program control, including when running a compiled binary with cT Executor:

```
pict "filename" $$ can be a marker expression
if ~zreturn $$ if cannot create or use file (already exists)

... $$ blank tag completes the image file
```

The initial -pict- command attempts to create the specified file (and returns a FALSE zreturn value if it cannot create the file). If the file already exists, the contents of the file are deleted in preparation for creating the image. You can, of course, use a preceding -setfile- command to check whether the file already exists. In addition to a blank-tag -pict- command, an image file is completed if you reach the end of a main unit, or execute a -jump- command, or quit running. By default the format of the image file is PICT on Macintosh, BMP on Windows, and PPM on Unix. To save an image in PPM format on a Macintosh or Windows, use a file name ending in ".ppm", such as "image.ppm".

The -put- command in the form -put screenvar; "file name"- creates an image file from a screen variable that had previously obtained a portion of the screen using a -get- command (see "Images & Files"). By default the format is PICT on Macintosh, BMP on Windows, and PPM on Unix. If the name of the file ends in ".ppm" the image is in PPM format on all platforms. Note that while the -pict- command and the cT Create menu options create an image of the entire window, the -put- command lets you save a portion of the window.

In all of these cases the image is stored in a pixel or "paint" format, with one exception. On a Macintosh, the -pict- command and the cT Create menu options store the image in an object-oriented "draw" format, and only the graphics commands executed after the -pict filename- statement go into the image file. On other platforms, when the blank-tag -pict- command is encountered the entire screen image visible at that time is saved. It is possible that in future versions of cT the -pict- command will be able to generate object-oriented image files on other platforms.
Once you have created an image file, you can read the image into your favorite drawing or paint program, or insert the image into a word processor document, or even insert it into your own cT program with "Insert file" on the File menu.

See Also:
Images & Files (p. 91)

Making and Saving Binaries

The source file is the file into which you type your cT program. This source file can be moved to any computer that supports cT and used without change (but see the discussion "Moving to Other Computers"). When you execute a unit, a process called compiling produces a binary if one does not already exist. The binary contains instructions that the computer understands and is unique for each machine.

When testing individual units in the programming environment, you will rarely see long delays due to compilation because units are compiled one at a time, as needed. However, if all units have to be compiled the compile time can be rather long. This happens if new units are added or old ones deleted, or if no up-to-date binary already exists. You may find it convenient to keep a dummy unit around to avoid delays when you want to create a little unit to test some cT statements.

The menu choice "Make Binary" forces cT to produce and save a binary file. This binary file is read the next time you edit your program, eliminating the need to recompile and therefore speeding up the editing process. When you are editing, it makes sense to make a binary just before leaving cT, so that the source and binary are in agreement.

When a program is compiled, the conventional ".t" extension is dropped and the extension ".ctb" is added. Other extensions are preserved.

On Unix machines, the extension for a binary becomes ".machinename.ctb", as ".sun3.ctb", and a binary can be run without the program source by including a "-x" command-line option, with a program name that need not include the binary extension:

```
cT -x programname
```

The last-modification date and time of the source and all -use-d files are compared with the date and time stored in the binary. If the source file or a -use-d file has been altered since the binary file was created, the binary file will be recognized as obsolete and will not be used. There are additional internal checks to make sure that the version of cT being used is compatible with the binary.

When you want to distribute your program, prepare a binary and distribute the binary along with the freely distributable cT Executor (the run-time package). Include any auxiliary data or icon files, which should be placed in the same directory as your binary. For Windows, you also need to supply the fonts that came with cT: the ".fon" fonts must be installed using the fonts Control Panel, or as part of an installation procedure.

When you distribute your own programs to others, you should provide the following information in case they already have some old cT binaries accompanied by an old Executor: "Different versions of the cT Executor expect different data formats in cT binaries. If you have old binaries, you should retain a copy of the old Executor to run those binaries."

See Also:
Moving to Other Computers (p. 21)
Additional Language Aspects

Differences from Other Languages

This summary is intended for experienced programmers, to compare calculations in cT with calculations in other programming languages that you might already know.

Compared to most programming languages, cT is unusual in providing built-in and easy-to-use support for graphics, mouse interactions, and multimedia. However, the core calculational facilities in cT are similar to those in other algorithmic programming languages such as C, Pascal, Fortran, or Basic. Nevertheless, you should be aware of the following calculational aspects of cT that may be different from a language you have used.

**Floating rounds to integer:** cT automatically makes all necessary conversions between real numbers (floating-point) and integers, including assignment statements and pass-by-value subroutine arguments. There is essentially no difference between "23" and "23.00"; cT will compile either a floating-point or integer value depending on which is more efficient in the particular context. Floating-point values are rounded (not truncated) to the nearest integer (4.8 rounds to 5).

**Floating and integer division:** A related issue is that in ordinary division with the "/" operator, the quantities are made floating-point before carrying out a floating-point division, and the result is floating-point (for example, 3/4 is treated as 0.75, not 0 or 1). The $divt$ and $divr$ operators perform integer division, either truncating or rounding the result.

**Fuzzy zero:** In order to compensate for round-off errors caused by the finite precision of computers, cT considers a = b to be TRUE if abs[(a-b)/a] < 10^{-11} or abs(a-b) < 10^{-9} (similarly for a < b etc.). Essentially, for purposes of numerical comparison, cT considers very small quantities to be equal to zero (and two such values are taken to be equal to each other), because the finite precision of computers can lead to two quantities that should be identical actually differing very slightly. If you need to compare two very small quantities to each other, scale them up before making the comparison (for example, check for 1E10*a = 1E10*b). Alternatively, the statement -inhibit fuzzyeq- turns off the use of the fuzzy zero.

**Iterative loops can have a floating index:** In most languages an iterative loop must have an integer index (i goes from 1 to 10 by 2's), but in cT you can use a floating index (i goes from 0.1 to 0.8 in steps of 0.05). The decision whether to end the loop uses fuzzy-zero comparison. The statement -inhibit fuzzyeq- turns off the use of the fuzzy zero.

**Case-sensitive:** Uppercase B and lowercase b are considered to be different letters (as in C but not in Fortran). Some cT programmers take advantage of this distinction by using "ALLCAPS" for naming constants, "Caps" for naming global variables, and "lowercase" for naming local variables. This convention helps in reading programs.

**Declaring constants:** Defining "Variable=10" actually defines a constant, not a variable with the initial value of 10. Initialization of variables must be done in assignment statements (-calc-).

**Pass by value and address:** As in Pascal, subroutine arguments can be "pass by value" or "pass by address". For details, see the -do- command (the cT command used to call subroutines).

**Array bounds checking:** The bounds on arrays are checked, and an execution error results if array indices are out of bounds. This prevents programming mistakes from leading to a hard crash.

**Run-time evaluation:** The powerful -compute- command compiles and numerically evaluates alphanumeric expressions at run time. For example, the sample program grapher.t distributed with cT lets the user enter...
INTRODUCTION

systems of algebraic or differential equations, and the -compute- command is used to evaluate and graph these equations.

(a < b < c) is legal: In most languages a two-sided comparison must be performed in two steps (a < b and b < c), but cT correctly evaluates a < b < c to be TRUE if b lies between a and c.

Character strings: In order to deal with rich text, including bold, italic, superscript, and subscript styles, Japanese characters, and embedded images, cT provides unusual "marker" variables that are quite different from and more powerful than simple ASCII string variables which you may have used. A marker variable is a pointer to a starting point in a block of text, with a length bracketing some text following that starting point. The -append-, -replace-, and -style- commands let you modify the bracketed text, and all other markers on the block of text are automatically adjusted appropriately, so that they continue to mark their respective sections of text. For details, see "Introduction to Strings".

Formatted text output: There is no format statement such as that found in Fortran. In cT one uses "embedded" -show- and -showt- commands to generate text involving variables. Here are examples of screen display (direct text) and file output (marker expression with concatenation) involving a marker variable "Country" and a numeric variable "Pop":

write <|show,Country|>: <|showt,Pop/1E6,3,0|> million people.
dataout file; <|show,Country|>:+":"+<|showt,Pop/1E6,3,0|>" million people."

Restoring the screen: A limitation of cT that is shared with other programming languages is that when the execution window is brought forward from behind other windows, or reshaped, it is the responsibility of your program to repaint the screen, because the window-oriented operating systems that cT runs on do not automatically save and restore the screen. To enable you to restore the screen, cT reexecutes the current "main" unit, which is why it is important to understand the concept of main units (see "Main Units").

Debugging: See the topic "Debugging Strategies" for useful suggestions.

What cT can't yet do: This version of cT does not support rich data structures and pointers such as are found in Pascal and C; cT only provides multidimensional arrays of its various types of variable. Currently only one window is supported during execution. Only rectangular regions are supported for mouse clicks and certain kinds of graphics operations such as -clip-. Although the -get- and -put- commands can save and restore a portion of the screen, graphics commands such as -draw- or -fill- cannot write directly into off-screen memory, which puts restrictions on making some kinds of animation run smoothly.

Remember that to see a topic listed under "See Also", just click the topic:

See Also:

Calculation Introduction (p. 187)
compute Storing and Evaluating Inputs (p. 165)
compute Computing with Marker Variables (p. 254)
Introduction to Strings (p. 246)
Main Units (p. 228)
Debugging Strategies (p. 20)
inhibit/allow fuzzyeq (p. 71)

Conditional Commands

In cT, as in other languages, you can use an "if" structure to do different things depending on the value of some expression, as in the following example:
if expression < 0
  . write minus
elseif expression = 0
  . write zero
elseif expression = 1
  . write one
elseif expression = 2
  . write two
else
  . write greater than or equal to three
endif

In addition, cT offers "conditional commands" which are convenient in some situations. The following conditional -write- command is equivalent to the "if" structure shown above:

write \expression \minus\zero\one\two\greater than or equal to three

The first argument after an initial backslash (\) is a numerical expression, which is rounded to an integer before being used to select among the possibilities. The second argument tells what action to take when the expression rounds to any negative integer. The third argument tells what action to take when the expression rounds to zero. There may be as many arguments as you need, separated by backslashes. The final argument (illustrated here for the value 3) tells what to do when the expression rounds to 3 or any integer greater than 3.

The initial backslash, which signals that the command is a conditional command, must immediately follow the TAB. The tag of a conditional command may be continued over several lines, as in the conditional -write- example shown above. Each new line must have a TAB followed immediately by a backslash.

All conditional commands used in cT follow the same pattern. Most cT commands may use a conditional format except those which deal with structures and definitions. Commands that may NOT be conditional:

define if loop arrow
use elseif endloop ifmatch
unit else case endarrow
endif endcase edit

Conditional commands are often used in true/false situations. Since TRUE is defined to have the value -1 and FALSE has the value 0, a conditional command with a true/false expression in the tag is

command expression\ trueaction\ falseaction

For example:

do \ x<y \ littleX\ bigX

See Also:
Logical Operators
  if IF Statements
case CASE Statements
INTRODUCTION

**Debugging Strategies**

There is a built-in debugger for cT. After describing how to use it, we list some other features of cT that facilitate finding bugs in programs.

**Using the built-in debugger:** Before running the program, choose "Debug" from the "Window" menu. In the Debug window, click the "Step" button to step through the program, one command at a time.

Alternatively, you can insert -step- or -step TRUE- statements anywhere in your program to halt normal execution and put you into step mode. Place -step FALSE- wherever you want the step mode to end (you can also end step mode by clicking "Run" in the Debug window). The -step- commands in your program are ignored if the Debug window is closed.

Important: The Debug window must not overlap the execution window. In that case the program will continually restart at the beginning of the main unit, due to the "reshape" events associated with having another window in front of the execution window.

When a -do- command is encountered in step mode, clicking "Step" will execute all of the statements in the -do- ne unit and halt on the statement that follows the -do-. Alternatively, if you want to go into the details of the subroutine, click "Into" to go into the -do-ne unit and execute it step by step. (You can click "Into" at other times as well to take one step.)

At any time that you are in step mode you can click "Stack" to display the current sequence of nested -do-s.

Type the name of one of your variables in the "Expr" slot, press Enter or Return, and you are shown the current value of the variable. You can even evaluate an expression such as "x+3y". The display of the value is updated at every step. The values of local variables can be shown only when you are stepping through the unit where they are defined. The Stack window is used to display styled text (that is, text that may be italic, bold, etc.).

While stepping the program, if either the Debug or Stack window is covered by the other one, you can bring the window to the front by choosing Debug or Stack from the menu.

**-show- and -pause-:** A very useful debugging strategy is to insert temporary code such as the following, which erases a small region of the screen, displays the values of some variables in that region, and pauses to let you observe and think about these values before proceeding:

```
erase 10,100; 200,150
at 10,100
write count = <|show,count|>, ratio = <|showt,time/count,1,3|>
pause
```

When you have the information you need, delete this debugging code or, if you think you might need it again soon, comment out this code by selecting all four lines with the mouse and choosing "Comment Lines" from the Edit menu. Later you can uncomment these lines to reinstall the debugging code.

**-menu-:** You might put the debugging code in a menu-driven unit, so that you can examine the current value of the variables at various times just by choosing your debugging unit from a pull-down menu (created with a -menu- command). This is feasible only for global or "group" variables, and it may be necessary to move some local variables temporarily into global or group variables in order that the menu unit can access them.

**-beep-:** If you are trying to understand the flow of control of your program, it can be useful to insert -beep- commands at various points so that you get an audible indication of whether the program passes through some section of interest.
**Graphics debugging:** If you are trying to debug a complicated numerical algorithm, it can be very useful to use -draw- or other graphics to visualize some aspect of the algorithm. Using different colors in the graphics can be helpful for marking stages through the algorithm.

**Choose the unit to execute:** You can execute a particular unit simply by placing the editing cursor somewhere in that unit and choosing "Execute current unit" on the Option menu. If you are debugging a subroutine that accepts arguments, you can't execute the subroutine directly, but if the subroutine is done by a unit named "Driver", use "Select unit" on the Option menu to specify "Driver" as the selected unit. Then whenever you choose "Run from selected unit" or "Execute selected unit" you start from an appropriate place in the program.

If there are important initializations in the first unit of the program, but you really want to start at unit "zonk", temporarily put -jump zonk- at the end of your first unit, and run from the beginning of the program.

When trying out a command you have not used before, or if you want to test a fragment of code, create a new unit and try things there, without disturbing the rest of the program. Creating or deleting a unit, or changing global definitions (before the first -unit- command), currently leads to a complete recompilation of the entire program, which can cause a noticeable delay if the program is very long. For that reason you may find it useful to keep one test unit around at all times, the contents of which you can change without triggering a complete recompilation.

**Scrolling text:** If you need an extensive sequence of debugging information, you might use the -edit-command to create an edit panel with a vertical scroll bar, and use -append- to add debugging information a little at a time to the marker associated with this edit panel. At any time you can scroll through the debugging information.

**Write to a file:** A similar technique is to create a file with -addfile- and use -dataout- to append data to the file. After running the program, use "Auxiliary file" on the File menu to examine the debugging information.

See Also:
- show Displaying Variables (p. 47)
- showt Specifying the Format (p. 48)
- pause Single Key & Timed Pause (p. 125)
- Summary of Menu Formats (p. 136)
- beep Making an Audible Tone (p. 122)
- append Adding Characters to a String (p. 259)
- Scrolling Text Panels (p. 149)
- addfile Create a File (p. 289)
- setfile Select a File (p. 291)
- dataout Write Data to a File (p. 298)

**Moving to Other Computers**

One of the advantages of cT is that source files can be moved from one type of computer to another, recompiled, and run without change, but a fully satisfactory porting may require a bit of planning (and the availability of the cT compiler on the other computer). Here are the main considerations that affect portability.

1) You can use the sequence -font-, -fine-, and -rescale- to make the program adjust its size to fit the available display space. Here is an example:

```
font zsans,20 $$ font type & newline height
fine 500,350 $$ graphics coordinates in an area 500 by 350
rescale TRUE, TRUE, TRUE, TRUE $$ rescale graphics and text
```
INTRODUCTION

The sample program sample.t included with cT uses this technique, and you might like to see what happens in that program as you change the window size. Another technique is to customize your displays to fit in the available space given by the system variables zwidth and zheight, or zxmax and zymax (for an example, see "An Example Program"). As you write your program, occasionally try different window sizes and shapes to make sure that the display rescales gracefully, as an indication that it will also run on another computer with a different screen size.

2) Don't use local quirks such as an unusual font that is available on only one kind of computer. Unfortunately, even the same font on different kinds of computers may not take up exactly the same amount of horizontal and/or vertical space, so that text which fills a box on one kind of computer might spill outside the box on other kinds of computers. You might use a larger box and use a "center" style to center the text inside the larger box. Font differences are the single most serious problem with porting cT programs from one platform to another.

Ultimately it is important that you try running the program on another computer to make sure that font or other problems are not serious. In some cases it may be necessary at the beginning of the program to choose slightly different font sizes (with a -font- or -fontp- command) based on the value of system variable zmachine, like this:

```c
case zmachine
  "macintosh"
    font zsans,13
  "windows"
    $$ MS-Windows
    font zsans,16
  else
    font zsans,14
endcase
if ~zreturn
  write Cannot obtain font.
  pause
  jumpout
endif
```

A related issue is that on some computers, especially machines running Windows or Unix, the number of pixels per inch is quite variable. For that reason a font that has a carriage-return height of 15 pixels might be almost too small to read. You can use -sysinfo default newline, myvar- to find out what is a normal readable font size.

It is possible to move a font from the Macintosh to Windows: see the -font- command for instructions.

3) Keep in mind the limitations of your target machine(s), especially with respect to screen size and processing speed. You simply cannot display 50 lines of text on the small screen of some machines. A complicated plot that takes 5 seconds on a fast computer might take 20 seconds on another. While 5 seconds may be an acceptable delay, 20 seconds may not be acceptable.

On the other hand, an animation that looks fine on a slow machine may run much too quickly on a fast machine. A way around this is to use the system variable zclock before and after a few iterations of a loop, and use the elapsed time to determine how much the animation should be slowed down on a very fast machine (for an example, see the sample program hill.t included with cT; for a simpler scheme involving zclock see "An Example Program").

4) If you use color, use "fallback" options in -palette- commands to enable the program to run on non-color computers, or on computers with fewer colors. The -sysinfo- command gives more detailed information about color availability than zncolors does.
5) Color images that have been inserted into your source program are portable to other platforms without change. It is possible that colors may be somewhat different if the standard "system" palette is somewhat different on the different computers. If your color images are in files accessed with -get-, you can use PPM format as a universal format recognized by cT on all platforms (see "Images & Files").

6) Icons (bit-map images) do not rescale. If you are moving among noticeably different screen sizes you may need to prepare several sets of icons in graduated sizes.

7) To move an icons file from one computer to another, first use the icon editor program (Icon Maker on Macintosh, icon.t on other platforms) to prepare a universal ".fdb" form of the icons file and transfer it to the other computer. Then use the icon editor on the other computer to convert to the format appropriate to that computer. (But on Unix machines, conversion from the .fdb format is automatic when your program first executes an -icons- command.)

See Also:
An Example Program (p. 5)
font Selecting a Typeface (p. 43)
fontp Selecting a Specific Typeface (p. 46)
fine Declaring a Screen Size (p. 35)
rescale Adjusting the Display (p. 37)
Sample Programs (p. 28)
Images & Files (p. 91)

cT Datastream

The following information is provided for the benefit of experienced programmers who may wish to make conversions between cT rich-text source files and other rich-text formats such as those used by particular word processors.

When you save on disk a styled cT file containing italics, bold, images, etc., cT represents those styles and images in a form that is 7-bit ASCII and has a maximum of 70 characters per line. This allows the files to be sent by electronic mail and by file transfer mechanisms. Here is an example of what is called the cT "datastream":

```
write An italic word. $$ how it looks when editing
write An @1ff 2italic@1ff8000 word. $$ how it is stored on disk
```

The "at" sign (@) introduces special codes that describe the style. In the sequence "@1ff 2" above, "1" indicates a style change, "ff" indicates a font-face style such as italic or bold, and the "2" following three spaces indicates italic, while the sequence "@1ff8000" indicates a change to plain (unitalicized) text. The format for a style change is @1SSxsyy, where SS is a two-letter code such as ff, and xx and yy are two-letter hex numbers (base 16); leading zeros can optionally be represented by spaces. The total description following the @ is always exactly 8 characters long.

Here are all the two-letter style codes, with the meanings of the two hex numbers that follow them:

```
fs  font size
    percent bigger or smaller than the default size
    0 = default size, 1 = 1% larger, -1 = 1% smaller
    if number <= a certain big negative number ABSSIZE,
        then -(number-ABSSIZE) is the absolute point size

ff  font face
```
1 = bold, 2 = italic, 4 = underline, 8 = outline, 16 = shadow
32 = subscript, 64 = superscript
bits are or-ed together; 0 = plain text

pl paragraph
right-most bit = invisible
next two bits [(number $\text{srh}\$ 1) $\text{smask}\$ 3] are justification
  0 = left justify, 1 = centered, 2 = right justify, 3 = full justify
remainder [number $\text{srh}\$ 3] is a paragraph layout index in the PARA table discussed later

cl color
palette number (0-7) for foreground color of text, or
background color in mode inverse

fg font group or family (zserif, zsans, etc.)
number refers to entries in a FONT table discussed later

ht hot text
index into a HOTT table discussed later; index = 8000 means not hot

For each of these styles, the number 8000 indicates the default style at the time of display, as determined by the current environment (for example, in relation to a font specified with a -font- command).

In addition to style sequences, text may contain other special sequences as follows:

1) Ordinary characters with ASCII codes less than 127 represent themselves (except for "@").

2) The character "@" introduces a special sequence, such as @1ff8000.

3) The sequence "@ @" represents the @ character itself.

4) There are two special sequences that delimit the beginning and end of the datastream:
   
   "@cT datastream n" starts the datastream (version number n).
   "@cT end" terminates the datastream.

Usually there is just one datastream in a file, but if N -dataout-s of markers are made to a styled file, there are N datastream segments in the file, each beginning with "@cT datastream n" and ending with "@cT end". For an example of a file editor that deals with such files, see the program editfile.t in the cT sample programs.

5) The sequence of "@" followed by a newline character (<|cr|>) represents a newline that has been added to the datastream to limit the length of the line (because some electronic mail programs cannot deal with long lines). This sequence is discarded when the file is read into the computer.

6) The sequence "@0xx" represents a single ASCII character, whose hexadecimal encoding is xx. For instance, the yen sign is "@0a5". The sequence @0xx always has exactly four characters.

7) The sequence @4SSSS introduces a special data segment in the datastream, and such a data segment is terminated by @5SSSS. Currently the four-letter SSSS codes include FONT (a table of fonts used in the file), PARA (a table describing types of paragraph layouts), HOTT (information associated with hot text), and STXT (a graphic element). The details of these data segments follow.
a) FONT: a font-family table. The font-family table maps font names to the font-family numbers of the original file. This allows a font-family style sequence (@1fgxxxy) to refer to a font by a number that can be looked up in the font-family table to get the actual name of the font. Here is an example:

@4FONT
6
2 zserif
3 zsans
17 zsymbol
15 Palomino
14 Technical
@5FONT

The first entry in the table ("6" in this case) is the decimal number of entries in the table. For every entry in the table, the font-family number (a 3-character hex number) is followed by a space and then the font family name.

b) PARA: a paragraph layout table. An example of a special paragraph is one with a centered or right-adjusted style. The paragraph table maps paragraph layout numbers to the actual paragraph layout data. This is needed because the actual layout numbers are assigned sequentially to layouts as they are created. Two versions of cT with different histories will assign different layout numbers to the same layout data. Here is an example:

@4PARA
3
8 180 9 9 0 24
10 54 9 9 0 24
18 180 0 108 0 24
@5PARA

The first entry in the table ("3" in this case) is the decimal number of entries in the table. For every entry in the table, there are six parameters:

a) paragraph layout number (hex)
b) tab size (decimal)
c) left margin (decimal)
d) right margin (decimal)
e) paragraph indent (decimal)
f) extra line height (decimal)

c) HOTTEXT: a hot-text table. The hot-text table contains the "hot info" text associated with each piece of hot text (for a discussion of hot text, see "edit Scrolling Text Panels"). Suppose there are two words of hot text in the file, "first" and "second", and the hot info associated with "first" is "Happy" and the hot info associated with "second" is "Two". Here is the text with hot styles (@1ht n):

The @1ht first@1ht8000 item and the @1ht second@1ht8000 item.

Here is the corresponding HOTTEXT hot-text table, which is placed at the end of the file, just before the final @cT end:

@4HOTT
2
5
3
@cT datastream 1
HappyTwo@cT end
@
The first entry in this hot-text table is the number of hot-text items (2 in this case). For each item, the length is given (5 for "Happy", 3 for "Two"). All the hot-info text is concatenated ("HappyTwo") and contained within another cT datastream. Currently the execution of hot text (what happens when the text is double-clicked) only handles plain-text hot info strings, but eventually cT may support fully styled text in the hot info, which is why this text is enclosed in a regular datastream context.

d) STXT: a graphic element. The first entry in the STXT table is a numerical code for the type of graphic element: 134 is a monochrome image, 135 is a color image, and 136 is an icon reference (associated with the Icon menu item on the cT Font menu, or created by applying an "icon" style to a marker). The second entry is a (negative) version number. If the second entry is positive, we assume the version number is absent and give the value of -1 (for original version). The remaining entries depend on the type of graphic element. Here are the current possibilities:

Type 134: a monochrome image. The third entry (252) is the raster size, padded out to full computer words. The fourth and fifth entries give details about offsets and origins for this image. The sixth entry is the width and height (91 pixels wide by 21 pixels high). This is followed by the character image written 4 hex characters per word, 16 words per line, left-to-right, top-to-bottom.

Type 135: a color image (version 2). The third and fourth entries (200 and 85) are the width and height in pixels:

Type 136: an icon reference. The third entry (1) is the number of icons stored successively. The fourth entry (11 12 13) is a list of the icon numbers. The fifth entry is the name of the file in which these icons are found.
Note that the @1 styles for font group, paragraph, and hot text reference the data from the @4 data segments. Without the @4 segments those styles have no real meaning. The @4FONT and @4PARA segments are placed at the beginning of the datastream, the @4HOTT is placed at the end. There can only be one of each type per datastream. The @4STXT segments can be anywhere and there can be as many of them as desired in a datastream.

A good way to study the cT datastream format is to create a cT file containing the styles, hot text, or graphics of interest, save the file, then examine the file with a standard word processor, which will display all the special sequences without interpreting them the way cT would to mean italic, etc. Or examine the file using the file editor found in "A File Editor Application" after deleting the "styled" parameter on the -setfile- command, in order to see the datastream characters without the styles being interpreted.

See Also:
A File Editor Application (p. 155)
Scrolling Text Panels (p. 149)

Syntax Level

As new features are added to cT, it is occasionally necessary to make a change such that the language would not be upwardly compatible. When this is done, provisions are made for automatic conversion of source code. The syntax level shows which level of cT the source code was prepared for. The syntax level is given on the first line of the program and has the form:

$syntaxlevel n

Known Bugs

Here is a list of bugs in cT known at this time.

GLITCHES in the PROGRAM WINDOW

Graphic Editing

The graphics editor does not work on continued lines. That is, a click in the execution window to add or modify a point is not recognized if the cursor or the selected region is on a continued line. Workaround: Put all points on one line (which can have wraparound).

The graphics editor only reports positions to the nearest tenth (“224.7”). If you have extreme scaling, where one conceptual unit is several pixels, you will not be able to use graphics editing on the display. For example, -fine 10,10- (or -scalex .001- or -size 20-) produces a display that cannot be edited graphically, because several different points on the display collapse down to the same value in the program.

LANGUAGE BUGS: MORE SERIOUS and/or HARDER to WORK AROUND

-rdisk-

The -rdisk- command rotates correctly only if the x-scale equals the y-scale (that is, if the disk is a circle, not an ellipse).
"zspell" system variable

When -specs okspell- is used, zspell does not report correctly; it is always TRUE. Workaround: Check for the desired response, then issue a -specs okspell- and check again. Set your own flag for correct spelling.

-getkey-

A loop does not check for inputs on every cycle. Thus, you may see some "coasting" with -getkey-. That is, if a counter says "45" when you press a key, the counter may get up to "48" before the input is noticed.

-jump- with arguments

Reshaping the window during execution causes the current main unit to be reexecuted. If the main unit was initiated by a -jump- command with arguments, such as -jump start(3)-, the passing of arguments is not repeated on the reshape, so that the local variables in the main unit retain whatever value they had at the time of the reshape. There are a couple of bugs associated with this process. Local marker variables are not preserved. At the end of the main unit, choosing a menu option followed by a reshape causes local variables to have unpredictable values. Note that these problems occur only when the window is reshaped during execution.

Workaround: Avoid initiating a main unit with an argumented -jump-. Put the information in a global variable before executing the -jump-. Or, if you don't like using global variables that way, just hope that your users won't reshape the window until after the next release.

BUG or FEATURE?

graph labels in -mode xor-

Because the lines that display axes and tick marks cross and recross, the commands that make tick marks along graphing axes (labelx, labely, markx, marky) ignore -mode xor-. This may be a "feature" not a bug, but it means that you cannot use -mode xor- to erase tick marks. Use -mode erase- instead. If you want a white-on-black graph, draw the black area first and use -mode erase- for the graph.

Sample Programs

In addition to the small programming examples in the cT help, a number of sizable programs written in cT have been included with cT in order to provide examples of what can be done with the language and to offer models of how to achieve certain effects. You are welcome to do whatever you want with these programs, including modifying them or incorporating them into your own programs.

Note that in many cases most or all of the units in these programs are "hidden," showing only the unit name. To see the contents of hidden units, select the units of interest with the mouse and choose "Show Units" from the Edit menu.

General

sample.t -- This program is a sampler of many of the basic capabilities of cT: color graphics, animations, pull-down menus, mouse clicks and drags, multifont text, "hot" text, calculations, graphing of functions, and response analysis. After running the program you might want to study the program code to see how the effects are achieved.

exercise.t -- A set of exercises to help you learn the basic concepts of programming in cT. The program contains a number of incomplete units with suggestions on how to complete them. The cT help contains the information needed to do these exercises.
**Graphics**

**draw.t** -- A basic drawing editor, with many of the features of commercial drawing editors, such as grouping objects together, applying patterns, colors, and arrowheads, and designing your own palette of colors. Drawings are saved in the form of cT source code, so you may find draw.t useful in creating portions of your own programs, including color -palette- commands. This program also illustrates one approach to "object-oriented" programming in cT.

**map.t** -- A little program that displays a map of the 48 contiguous states of the United States, with an ability to zoom in and out. It uses the -use- file states.t.

**Color**

**palette.t** -- A -use- file that provides a set of useful additional colors beyond the basic eight cT colors, including dark red, dark green, and light, regular, and dark versions of slate, teal, coral, gray, gold, lavender, and cerise.

**setcolor.t** -- A -use- file that lets you experiment with the color of an object, in the full context of your running program. This is useful for deciding exactly what color the object should be in relationship to other elements of your display.

**chaos.t** -- A plot of the chaos in a simple population growth scheme. Usually what is plotted is just the final-state population. This program uses hues from blue to red to show the approach to the final state. The red dots correspond to the usual plot.

**spiro.t** -- Make geometrical color designs by choosing the total number of vertices and the number of vertices to skip. It uses the -use- file colorpic.t.

**Video**

**video.t** -- A program that gives an example of how to build your own special-purpose video controller, using the basic video commands. This file can be used as a -use- file by other programs. A short video clip for testing purposes is provided with video.t for Macintosh and Windows.

**Games**

**BigForty.t** -- A solitaire card game that uses -touch- regions, so that event-handling routines are driven automatically; there are no -pause- commands in the program. The program uses the -use- file animate.t to provide the option of sliding cards smoothly over the background on sufficiently fast computers.

**rilato.t** -- A Mah Jong-like game in which you match corresponding pairs of tiles. Pairs can be chemical elements and their symbols, American presidents and their years in office, English kings and their years of reign, or American states and their capital cities. You can also create your own lists of pairs.

**Physics and math**
grapher.t -- Solves and graphs systems of algebraic or ordinary differential equations. This program was a First Prize winner in the 1990 Educational Software Contest of the journal *Computers in Physics* (Sept./Oct. 1990, p. 540). The program includes an interactive explanation of how to use the program.

hill.t -- Draw a hill with the mouse, place a block on the hill, give it an initial speed, and watch it move. If there is a valley, the block may move back and forth forever (no friction), or slowly come to rest (if you add some friction with the slider control). While the block moves, bar graphs display the kinetic, potential, and total energy. At the beginning of the program, the program makes a measurement to determine the animation step size, so the animation runs at about the same speed on fast and slow computers.

orbits.t -- Study 2-body and 3-body gravitational orbits. Some setup files are provided that specify particularly interesting orbits. There is an option to display continuously the potential energy, kinetic energy, and total energy.

optics.t -- Place lens and mirrors along a bench, then flash a light. Rays spread out and are bent by the optical elements, producing a spot on a piece of film. There is an interactive explanation of how to use the program.

quantumw.t -- Study the quantum behavior of an electron in various kinds of potential wells. If the well is symmetrical, the bound states have symmetrical wave functions.

sonar.t and voltage.t -- Simple examples of microcomputer-based-laboratory software. Connect a Universal Lab Interface (ULI) and Sonic Ranger (distributed by Vernier Software of Portland, Oregon, phone 503-297-5317) to the serial port of either a Macintosh or a PC machine, and sonar.t will track your motion in front of the motion detector. If you don't have this equipment, use the mouse to make motions that are graphed on the screen. The program uses the -use- file ULI.t and the icons file imotion. The program voltage.t uses the ULI to plot voltage as a function of time.

xyplot.t -- Plot a function of two variables, f(x,y), using a set of icons of differing dot densities (icon file random).

**Intercomputer programs using sockets**

InterDraw.t -- Run this program on two different Macintoshes in the same AppleTalk zone, or two different Unix workstations on the same network (provided that a cT server has been established), and two people can draw on each other's screens, or run two copies of the program on one Unix workstation. This program is a simple example of the use of the -socket- command to link separate programs together.

Battleship.t -- The classic "battleship" game played on two different Macintoshes in the same AppleTalk zone, or two networked Unix workstations (provided that a cT server has been established). This is another example of the use of the -socket- command. It also provides another example of how to do object-oriented programming in cT. The program uses the icons file SHIPicn.

These intercommunication programs involve two cT programs communicating with each other. An important use of sockets is to connect between a cT program that handles the graphical user interface and a program written in some other language. At present this is supported on Macintosh System 7 and on Unix. Sample C-language programs are provided in the programs distributed with cT. Interprocess communication on Windows has a very different structure, and cT supports Dynamic Data Exchange rather than sockets on Windows.

See Also:

A File Editor Application (p. 155)

In addition to the suite of sample programs, there are a few other files that are distributed with cT but which are not described explicitly in the basic installation procedure.
To create small images to design icons, cursors, and patterns, use the program Icon Maker on the Macintosh or icon.t on other machines. The use of these programs is documented in the on-line help under "Making Icons on Macintosh" and "Making Icons on PC & Unix".

The program MacFDB is used on the Macintosh to convert a Macintosh font to the cT universal "FDB" format, and these files can be converted to Windows fonts using the programs fdbpc, fpwin, and fntfon (which are installed with cT on Windows). The procedure is described in "font Selecting a Typeface".

See Also:
- Making Icons on Macintosh (p. 95)
- Making Icons on PC & Unix (p. 96)
- font Selecting a Typeface (p. 43)
2. Graphics & Text

Graphics Introduction & Defaults

Screen positions are expressed in the x,y coordinate system, with the x- and y- positions separated by a comma. When several points are given on one command line, the points are separated by a semicolon:

\[ X1,Y1; \ X2,Y2; \ X3,Y3 \]

There are three coordinate systems available for making graphics:

- Absolute -- most ordinary graphics
- Graphing -- graphs & "real world" coordinates
- Relative -- rotated and scaled graphics

In the "absolute" coordinate system, the points refer to pixel positions (dots on the screen). The point 0,0 is at the upper-left corner of the available display space. The "x" coordinate grows larger as a point moves to the right. The "y" coordinate grows larger as a point moves downward. For the graphing and relative coordinate systems, the 0,0 point and the direction of increase are controlled by the program. If coordinates are given as floating-point values (numbers with fractions), pixel positions are rounded to the nearest pixel.

All three coordinate systems may be used on the same display. Which display system you use depends both on the task and on your own programming style. At the beginning of the program, default positions and scaling factors are set so that the three coordinate systems are equivalent until the program explicitly defines the graphing and relative systems.

In a multiwindow environment, the available display space refers to the space controlled by the program and not the entire display screen. In the cT programming environment, there is a "source" window and an "execution" window. The available display space refers only to the execution window.

Any number given in an example for a graphics command can be replaced by a variable or an expression.

\[
\begin{align*}
\text{at} & \quad 150,50 \\
\text{at} & \quad X\text{position}, Y\text{position} \\
\text{at} & \quad 3\text{count} + 27, Y\text{initial} + \text{steps} / 7
\end{align*}
\]

cT automatically makes all necessary conversions between real numbers (floating-point) and integers. If a floating-point number or variable is used where the program "expects" an integer, the floating-point number is rounded to the nearest integer. The value is not truncated. For example, 4.8 rounds to 5.

Graphics Defaults: At the beginning of a program, the absolute coordinate system puts the 0,0 point at the upper left of the display and makes one unit be one pixel. That is, the point 1,1 is one pixel to the right and one pixel below the point 0,0. The position of 0,0 and the meaning of 1,1 can be modified with the commands -fine- and -rescale-. The discussions in this manual assume that moving from 0,0 to 0,1 is a movement of one pixel.

Upon entry into a program, default values are set for the graphing parameters and the relative parameters. If the program does not explicitly initialize a graphing system, a "g-type" command behaves as if it were in absolute coordinates. Similarly, unless a relative system is defined, the "r-type" commands behave like absolute commands. That is, -gdraw- and -rdraw- behave just like -draw-, -gbox- and -rbox- behave just like -box-, etc.

The default parameters for graphing are
gorigin  0,0
bounds  1,1
scalex  1
scaley  -1
polar    FALSE

The default parameters for the relative coordinate system are

rorigin  0,0
size     1
rotate   0

**Main Unit Defaults:** At the beginning of every main unit, the screen is erased (filled with the window color), the current screen position (zwherex, zwherey) is set to 0,0, and the left and right margins are set to the edges of the available display space. Graphing and relative parameters are not reset to default values.

*See Also:*
  - Inhibit and Allow in Judging  (p. 177)
  - fine Declaring a Screen Size   (p. 35)
  - rescale Adjusting the Display  (p. 37)
Basic Graphics & Text Commands

Describing the Screen

Window Size and Title

A "$\text{window}$" statement at the start of a program lets you specify the initial size of the execution window:

```plaintext
$syntaxlevel 2
$\text{window} 100,100  \text{\$ execution window will be 100 by 100, if possible}
```

In the absence of a $\text{window}$ specification, the \text{cT} executor.

The \text{-wtitle-} command sets (or resets) the title that appears at the top of the execution window:

```plaintext
\text{wtitle} "Esperanto Verbs"  \text{\$ title in quotes,}
\text{ \$ or marker expression (character string)}
```

\text{See Also:}

- \text{fine Declaring a Screen Size (p. 35)}

at: Positioning Graphics

The \text{-at-} command specifies a position on the display and sets a left margin for the display of text produced by \text{-text-}, \text{-write-}, and \text{-show-}. It optionally also sets a right margin and lower bounds for text.

```plaintext
\text{at} beginx,beginy
\text{at} x,y; rightx,bottomy
\text{at} ; rightx, bottomy
```

The \text{-at-} command with one screen position sets a left margin for the display of text and sets the bottom-right margins to the bottom-right corner of the available screen area. The \text{-at-} with two screen positions sets both right and left margins and sets a bottom boundary below which text will not appear. If the first screen position is omitted, the position and left margin are set to the current screen position. These two commands are equivalent:

```plaintext
\text{at} z\text{where}x,z\text{where}y; rightx,bottomy
\text{at} ; rightx, bottomy
```

If \text{-fine-} and \text{-rescale-} are not in effect, the position is expressed in actual screen pixels relative to the upper-left corner of the window. If \text{-fine-} and \text{-rescale-} are used, the arguments of \text{-at-} express positions relative to a conceptual screen.

Entering a new main unit resets the margins to the left and right edges of the display area.

\text{Example:}

In this example, the margins chosen are too small. Not all of the text is displayed. Changing the "200" to "300" will make all of the text visible.
unit xat
at 0,0
write This is at the upper left.
at 100,50; 200,160
text This text starts at 100,50. The right margin is at 200 and the lower bound, below which the text may not extend, is at 160.
\ *

See Also:
Graphics Introduction & Defaults (p. 32)
clip Limiting the Display Area (p. 63)

atnm: Positioning with No Margin

The -atnm- command specifies a position on the screen. It differs from the -at- command only in that it does not set margins for the display of text and variables. The previous margins are left unchanged, and text outside those previously set margins will not be displayed (to remove these constraints, execute -at zxmin,zymin; zxmax,zymax-).

\atnm x-position, yposition $$ fine grid

Example:

unit xatnm
box 50,50; 250,200
at 50,50; 250,200
text This little paragraph uses the margins set by the -at- command.
\ atnm 120,100
text The -atnm- specifies the beginning of this paragraph, but it does not change the margins set by the earlier -at- command.
\ atnm 21,178
write Some of this text will not display because it is outside the margins.
*

fine: Declaring a Screen Size

The -fine- command specifies the range of coordinate positions used by a program. If there is no -rescale-command, each coordinate position (such as "115,237") corresponds to the actual pixel position (within the program's window) on the screen. A position that falls outside the available display area is not shown.

fine x-dimension, y-dimension
fine x1,y1; x2,y2
The -fine- with one coordinate pair specifies that the coordinates used by the program range from (0,0) to (x-dimension, y-dimension). If the space available is greater than required (and -rescale- is not active), the display is centered within the available space. If the available space is too small (and -rescale- is not active), the lower and right portions of the display are lost.

The form with two coordinate pairs specifies that the coordinates of the program range from (x1,y1) to (x2,y2). This form is used when the program must be fit onto a small display space and the author knows that the area above and to the left of (x1,y1) can be ignored. This format can also be used with -rescale- to expand a section of the display.

-fine- and -rescale-

The -fine- and -rescale- commands, working together, allow the author to regard all screen positions as conceptual positions. When the program is executed, the display is automatically adjusted so that it fits into the space available.

When using -fine- and -rescale-, imagine that all graphics are drawn on a piece of stretchy graph paper. As the size of the display expands and contracts, the paper is stretched or squeezed in one direction or the other, but all of the paper is visible.

You should pick a -fine- dimension that is convenient to visualize and use it consistently. In general, the -font-, -fine-, and -rescale- commands belong at the very beginning of a program before the first -unit- command, or in the first unit.

Rescaling is not the only way to deal with variable window sizes. Another way is to use \texttt{zwidth} and \texttt{zheight} to detect when a window is too small, and display a message saying that you need a bigger window. Yet another scheme involves using the values of \texttt{zwidth} and \texttt{zheight} to tailor your display to use the available space. See "An Example Program" for an example of this.

Examples:

Execute the unit below and experiment with different window sizes. Then try it again with the FALSE (3rd argument, aspect ratio) changed to TRUE.

```verbatim
unit xfine
font zsans,20 $$ font type & size
fine 500,350 $$ conceptual area 500 by 350
rescale TRUE, TRUE, FALSE, TRUE
at 250,175 $$ center of the screen
circle 50
write Hello!
*
```

The next examples illustrate using -fine- and -rescale- to display a selected area of the screen. Notice that the coordinates used in unit "FDraw" range from (0,0) to (399,399). The blank-tag -box- command outlines the active area of the display. The -rescale- and -fine- at the end of FDraw return to default conditions.

```verbatim
unit xfine1 $$ as much of the drawing
fine 400,400 $$ as possible is displayed;
do FDraw $$ the drawing is centered
*
unit xfine2 $$ the drawing fills the window
fine 400,400
```
rescale TRUE,TRUE,FALSE,FALSE
do FDraw
*
unit xfine3 $$ only a portion of the
draw 50,100; 300,250 $$ drawing is visible
do FDraw
*
unit xfine4 $$ a portion of the drawing
draw 50,100; 300,250 $$ fills the entire window
rescale TRUE,TRUE,FALSE,FALSE
do FDraw
*
unit FDraw
draw 200,0; 399,200; 200,399; 0,200; 200,0
at 200,200
circle 50
box $$ put a box around the display area
*

See Also:
font Selecting a Typeface (p. 43)
rescale Adjusting the Display (p. 37)
Graphics Introduction & Defaults (p. 32)
Current Screen Size (p. 321)
An Example Program (p. 5)

rescale: Adjusting the Display

The -rescale- command allows the display to expand or contract to fit the available display space.

rescale TRUE,TRUE,FALSE,TRUE
rescale -1, -1, 0, -1

The -rescale- command permits rescaling of the display to fill the window. It has four arguments that are either TRUE (-1) or FALSE (0). The arguments specify

- use the full width of the window;
- use the full height of the window;
- maintain the aspect ratio;
- adjust the font size.

"Maintain the aspect ratio" means that the actual display distance represented by one unit in the x-direction should be the same as the distance represented by one unit in the y-direction. Rescaling the width and height is not independent of preserving the aspect ratio. When the third argument is TRUE, the display fills the window as much as possible while preserving the aspect ratio. (This can be thought of as making circular circles.) If the third argument is FALSE, rescaled circles may become ellipses.

The -rescale- command causes x- and y-scaling factors to be calculated internally. When "adjust font size" is TRUE, the smaller of the two factors is used to select a font size. If the display area is so small that an appropriate font is not available, a message appears at the bottom of the display saying "PLEASE MAKE WINDOW TALLER (or WIDER)." If the display already occupies the full screen, the message is "NO FONT SMALL ENOUGH."
GRAPHICS & TEXT

If -rescale- is omitted, the size of the display does not change size as the window size changes. Displays that are too large are clipped at the right and at the bottom.

If a font other than the default will be used as the "standard" font, the -font- command should precede the -rescale- command. For font rescaling to work, use the default font size for your basic text. If you put "Bigger" or "Smaller" styles on all your text, font scaling probably won't work satisfactorily.

See the -fine- command for examples of -fine- and -rescale-. Refer to the -font- command for a more detailed discussion of font rescaling.

The -rescale- command has an optional fifth argument. TRUE (the default) means constrain the graphics scaling to match available fonts. FALSE means do not constrain graphics: rescaling is not constrained to match available fonts.

Rescaling is not the only way to deal with variable window sizes. Another way is to use zwidth and zheight to detect when a window is too small, and display a message saying that you need a bigger window. Yet another scheme involves using the values of zwidth and zheight to tailor your display to use the available space. See "An Example Program" for an example of this.

See Also:

- fine: Declaring a Screen Size (p. 35)
- Graphics Introduction & Defaults (p. 32)
- An Example Program (p. 5)

coarse: "Typing-Paper" Coordinates

For historical reasons, cT provides the option to describe the display as a sheet of typing paper with letter positions: rows and columns. This is "coarse grid". One can specify screen positions in terms of rows and columns of letters instead of in terms of x,y coordinate positions. The -coarse- command specifies the width and height (in pixels) of a single "character" used by "coarse grid" coordinates.

```
coarse width, height
coarse 8, 16 $$ default values
at 812 $$ row 8, column 12
```

In coarse grid, the row and column are written as one number. Every column number must have two digits. The position "row 15 column 7" is written 1507. This is just like room numbers in a big building: floor 15 room 7 is written 1507.

If there is a -coarse- command at the beginning of the program, any screen position in any graphics command may be specified in either fine grid or coarse grid. The program can distinguish between the two grid types because every fine grid address has two numbers separated by a comma and followed by a semicolon, while coarse grid is just one number. Coarse grid is much less useful on a system with variable-width fonts than it is where fonts are fixed-width. This manual uses only fine-grid examples.

Coarse-grid coordinates cannot be used unless there is a preceding -coarse- command. If you intend to use coarse-grid coordinates, put an appropriate -coarse- command at the beginning of the program (before the first -unit- command).
Displaying Text and Variables

text: Putting Text on the Screen

The -text- command is used to display text. It allows the full range of display options (bold, italic, centered, etc.) as well as embedded variables and positioning options.

```
text left,top; right,bottom
```

This title is centered.

```
\text{This is additional text that will be displayed.}
```

```
text left,top; right,bottom; marker expr. $$ special form, no \" -- see below
```

The first line of the tag contains optional margins. The display begins at the position "left,top". The text is normally left and right adjusted ("full justified") between "left" and "right". Any text that would extend below "bottom" is omitted.

The body of the -text- command begins on the line immediately below the command itself. It is not indented. The text is terminated by a backslash at the beginning of a line.

```
The -text- command and the closely related -string- command are the only commands in cT that do not follow the pattern of command-at-the-left and tag-field-after-a-tab.
```

If desired, the margin specifications may be specified "permanently" by an -at- command:

```
at 100,100; 500,350
```

```
text

This is some text.
```

```
\text{The margins set by the -text- command are temporary; they do not affect the "permanent" margins set by an -at- command. It is usually better to set the -text- margins as part of the -text- command itself. It can be very confusing to lose some display because of a right margin or lower bound set by an -at- many lines earlier in the program.}
```

The "Insert file" menu option lets you insert text or graphics from another file into a -text- statement. On some machines you can paste graphics directly into a -text- statement. Programs containing inserted graphics are portable to other computers.

Embedded -show- commands are used to display the contents of variables. The embedded forms <|quote|>, <|cr|>, and <|tab|> insert double-quotes, carriage-return, and TAB.

With full justification there is automatic word-wrap at the right margin. This slows down execution, so if faster display speed is important don't use the -text- command.

In the special case that you need to display a marker (character-string variable) with all of its own styles, use the form -text left,top; right,bottom; marker expr.- (no \"). This form is needed because the embedded form <|s,marker|> imposes whatever style you use in editing the statement (such as left-adjusted). Or you can get nearly the same effect with the sequence -at left,top; right,bottom- followed by -show marker-, but in this case you have set "permanent" margins.
GRAPHICS & TEXT

Other ways to display text include the -write- command which is suitable for short texts, the -show- command for displaying marker variables (character strings), and the -edit- command for setting up an edit panel containing interactive text.

Examples:

```
unit xtext
box 50,50; 150,150
text 50,50; 150,150
This text is shown in the area bounded by (50,50) and (150,150).
\                  $$ initial backslash indicates end-of-text
*                  *
unit xtext2
box 25,50; 225,150
text 25,50; 225,150
You can display bold text and italic text.
  
\                  You can even center it!
*                  *
```

In conditional -text- commands, the backslash indicates the end of one argument within the tag. The end of a conditional -text- command has two backslashes. Notice that a null argument (a case where no text should be shown) is indicated by backslashes on consecutive lines.

```
unit xtext3
  f: temp,value
next xtext3
randu temp,5
calc value := temp - 2
  at 80,50
write temp = <|s,temp|>
    value = <|s, value|>
  at 50,120
text \ value \                  This text is for value < 0.
\  This text is for value = 0.
\  \ $\$ no text is displayed for value=1
Now value is 2.
\  \ $\$ end of conditional; no text for >2
*                  *
```

The optional positioning information appears on the backslash lines, which are delimiters between pieces of text. To try this example, select this unit and use "Run from Selected Unit," then press ENTER several times.

```
unit xtext4
  f: zip
next xtext4
randu zip,4
calc zip := zip - 2
  at 50,20
```
write current value of zip is $<$s.zip$>$
at 50,100; 200,500  $$ set default margins
text \ zip \50,50       $$ set left margin for this case only
For zip negative, the text starts at location 50,50.
\ $$ no position given; use default margin
For zip = zero, text is at the default position of 50,100.
\300,50; 400,500             $$ right and left margins set
If zip is $\geq$ 1, this text appears
in the area 300,50 to 400,500.
\ $$ end of conditional -text-
*

See Also:
write Another Way to Display Text (p. 41)
Scrolling Text Panels (p. 149)
show Contents of a Marker Variable (p. 250)
mode Changing Modes (p. 60)
newline Newline Height (p. 42)
supsub Super/subscript Height (p. 43)
inhibit/allow supsubadjust (p. 70)
Conditional Commands (p. 18)
Defining Variables (p. 190)
Logical Operators (p. 201)
Embedding Variables in Text (p. 50)
Using Embedded Marker Variables (p. 251)

write: Another Way to Display Text

The -write- command is a convenient way to write short pieces of text. It looks "tidier" in your program than
the -text- command, since the body does not extend over into the command field. Also, to achieve maximum
display speed it truncates rather than word-wraps at the right margin. Use the -text- command if you need
automatic word wrap.

The -write- command uses explicit carriage returns at the end of lines and a TAB is required before each new line
begins. The -write- command does not fill text out to the margin, as the -text- command does. If a line of text
extends beyond the margin, that line is truncated.

Embedded -show- commands are used to display the contents of variables. The embedded forms $<$quote$>$, $<$cr$>$,
and $<$tab$>$ insert double-quotes, carriage-return, and TAB.

The "Insert file" menu option lets you insert text or graphics from another file into a -write- statement. On
some machines you can paste graphics directly into a -write- statement. Programs containing inserted graphics
are portable to other computers.

Other ways to display text include the -text- command which is suitable for long texts, the -show- command for
displaying marker variables (character strings), and the -edit- command for setting up an edit panel containing
interactive text.

Examples:

unit xwrite
at 10,50
write Notice that the body of a -write- command
appears entirely in the "tag field." Successive lines must start with a TAB.

at 100,125; 200,300

write Because this text does not fit in the margins, the lines that are too long are truncated.

*

The next example illustrates a conditional write statement. Two backslashes together mean that nothing should be displayed for the corresponding value of "zip". No text is displayed for zip<0, zip=0, and zip=3. The value of "zip" is displayed with an embedded variable.

unit xwrite2 $$ use "Run from Selected Unit"
  f: zip
next xwrite2
randu zip,5 $$ random value 1-5
at 50,10
write This time zip = <|s,zip|>.
at 50,50
write \zip\zip is one
    \zip is two
    \zip is greater than three

*

See Also:
write Another Way to Display Text (p. 41)
text Putting Text on the Screen (p. 39)
Scrolling Text Panels (p. 149)
show Contents of a Marker Variable (p. 250)
mode Changing Modes (p. 60)
newline Newline Height (p. 42)
supsub Super/subscript Height (p. 43)
inhibit/allow supsubadjust (p. 70)
Conditional Commands (p. 18)
Defining Variables (p. 190)
Logical Operators (p. 201)
Embedding Variables in Text (p. 50)
Using Embedded Marker Variables (p. 251)

**newline: Newline Height**

The -newline- command lets you control the height of a line, measured from the top of a "T" on one line to the top of a "T" on the next line:

newline 30 $$ line height now 30 coordinate units high

**Example:**

unit xnewline
  i: newline
do xwriteit(10,10)
sysinfo default newline, newline $$ find system newline height
newline 2newline $$ double the newline height
do xwriteit(10,100)
DISPLAYING TEXT AND VARIABLES

*
unit xwriteit(xx,yy)
i: xx, yy
at xx, yy
write The second time this is displayed, all the lines are "double spaced," due to the -newline- command.
*

See Also:
sysinfo Get System Information (p. 319)

supsub: Super/subscript Height

The -supsub- command controls how high above or below the normal level superscripts and subscripts are displayed.

supsub 10 $$ shift super- and subscripts up or down 10 units

Example:

unit xsupsub
supsub 15 $$ big shift of super- and sub-scripts
at 10,30
write H\textsubscript{2}O and x\textsuperscript{3} displayed with shifts of 15.
*

See Also:
write Another Way to Display Text (p. 41)
text Putting Text on the Screen (p. 39)
inhibit/allow supsubadjust (p. 70)

font: Selecting a Typeface

The -font- command specifies a font family and a size. It affects the characters displayed by -text-, -write-, the -show- family, -labelx- and -labely-, and the user's input at an -arrow-. The -font- command does not affect displays made with area fill (see -pattern-) or with -plot- and -move- (see -icons-). The font used for menus is not under program control.

font family,height $$ comma required
font "myfont", 20$$ user font
font zserif, 15 $$ default system font
font $$ reset to base font

The -font- command selects a font from the named font family whose "newline" height is no larger than the specified height in the current (rescaled) coordinate system. The "newline" or "carriage return" height is the distance from the top of one line to the top of the next line. The height is not the same as the "point size" often used to characterize fonts. A blank-tag -font- resets to the font that was active at the time a previous -fine- or -rescale- command was executed.

43
Note that the -font- command lets you specify fonts in a rather generic way, and the font size is specified in terms of screen coordinates, not point size. In contrast, the -fontp- command lets you choose a very specific font and point size.

Several generic font family names are recognized to improve the portability of programs. These names are system marker variables that can be used anywhere that a font name can be used and are recognized on all systems:

- zserif - a standard font with serifs (Times, except for New York on Macintosh)
- zsans - a standard sans-serif font (Helvetica, except for Geneva on Macintosh)
- zfixed - a standard fixed-width font (Courier, except for Monaco on Macintosh)
- zsymbol - a math symbol font (Symbol on all platforms)

A "sans serif" font is one without the little decorative marks such as the crook at the top of an "l" or the foot on the bottom of an "f". A fixed-width font is one in which all characters use up the same width, thus leaving lots of white space around "i" and very little white space around "w".

Unfortunately, even a "standard" font like Times is not exactly the same on all platforms, with slight differences in spacing between letters, etc. This is the single most serious problem with porting cT programs from one platform to another. See "Moving to Other Computers" for more discussion.

Note that the -font- command affects the display of text when the program is run. There is a preferences file ct.prf that permits specifying what font to use in the program window for editing the program.

The selection of the font size is affected by the -rescale- command. If rescaling of fonts is active (4th argument of -rescale- is TRUE), the font changes size as the display space available expands or contracts. For example, -font "myfont", 20- produces text with a carriage return no greater than 20 dots high when no rescaling is in effect. When rescaling is active, -font "myfont", 20- produces text with carriage returns 20 units high in the conceptual system given by -fine-. With -fine 1000,1000-, font rescaling active, and an actual window size of 500 pixels by 500 pixels, a carriage return would be 10 pixels high.

**Font rescaling**

The -font- command scales using the newline size (i.e. how much a carriage return moves down). The rescaling is done in "jumps": graphics scaling factors are constrained by the availability of font sizes. Any text that would go below the bottom text margin is clipped off.

In essence, the largest font that fits the window is chosen, and then the graphics scaling factors are chosen to match the amount the text has actually rescaled. This leaves a white border around the display, but ensures a close match between text and graphics. (Use a blank-tag -box- command to surround the graphics area, and you will see a white border for most window sizes.) This scheme is not perfect. It depends on the constancy across various font sizes of the ratio of the height of a line of characters to the average width of the (proportionally spaced) characters. Occasionally, a few very wide characters such as "w" may throw this off, if in that particular font the "w" is proportionally a bit wider than it is in other font sizes.

The program cannot rescale itself every time a new font is introduced. **The -font- command that determines scaling must come earlier than a -rescale- and/or -fine- command.** A -font- command that is not followed by -fine- or -rescale- is simply a short-term special-purpose font change and does not affect scaling (a blank-tag -font- restores the base font). For example, either of the following will work:

```
font    zserif, 15
fine    500,300
rescale TRUE,TRUE,TRUE,TRUE
```
If the display space is very small, there may be no font available which would allow the rescaling to adjust as described above. In that case, the smallest available font is used and the message "NEED WIDER WINDOW" is displayed at the lower left of the window. The text and graphics will not be in correct proportions.

For font rescaling to work, use the default font size for your basic text. If you put "Bigger" or "Smaller" styles on all your text, font scaling probably won't work satisfactorily.

Rescaling is not the only way to deal with variable window sizes. Another way is to use \texttt{zwidth} and \texttt{zheight} to detect when a window is too small, and display a message saying that you need a bigger window. Yet another scheme involves using the values of \texttt{zwidth} and \texttt{zheight} to tailor your display to use the available space. See "An Example Program" for an example of this.

To move a font from Macintosh to Windows, use the Macintosh program MacFDB and the Windows programs fdbpc, fpcwin, and fntfon as follows (these programs are supplied with cT):

\begin{verbatim}
Use the application MacFDB on the Macintosh to create an .fdb font.
Move the .fdb font to a Windows machine.
On Windows, execute: fdbpc my.fdb my.fpc
Execute: fpcwin my.fpc my.fnt
Execute: fntfon my.fnt my.fon or fntfon my10.fnt my12.fnt my.fon
Then use the Windows control panel to install the font.
\end{verbatim}

Some characters for less frequently used European languages are not available on a Macintosh unless the special "ISO" fonts distributed with cT are installed.

\textit{Examples:}

The statement "font zserif,18" selects a font from the font family named zserif, such that, measured in current coordinates, the newline height is 18.

\begin{verbatim}
\textbf{Examples:}

\end{verbatim}
In the next example, notice that when the -fine- is changed, the position (10,50) of the text changes as well as
its size. In a real program, such a change of -fine- is not recommended.

```
unit xfont3
font zsans,15
fine 200,200
rescale TRUE,TRUE,FALSE,TRUE
at 10,50
write fine 200,200
fine 500,500
at 10,50
write fine 500,500

See Also:
- fontp Selecting a Specific Typeface (p. 46)
- icons Selecting an Icon (p. 93)
- pattern Making Textured Areas (p. 61)
- rescale Adjusting the Display (p. 37)
- Generic Font Names (p. 327)
- An Example Program (p. 5)
- Moving to Other Computers (p. 21)

fontp: Selecting a Specific Typeface

The -fontp- command lets you choose a specific font and point size:

```
fontp "Helvetica", 48
```

choose 48-point Helvetica, if available

After executing the -fontp- command, zreturn = TRUE if it actually sets to the font you specified, and zreturn
= FALSE if the font selected is not exactly the one you wanted (due to unavailability of that particular font
and/or point size).

Note that the -font- command lets you specify fonts in a more generic way, and the font size is specified in
terms of screen coordinates rather than point size.

Unfortunately, even a "standard" font like Times is not exactly the same on all platforms, with slight differences
in spacing between letters, etc. This is the single most serious problem with porting cT programs from one
platform to another. See "Moving to Other Computers" for more discussion.

See Also:
- font Selecting a Typeface (p. 43)
- Generic Font Names (p. 327)
- Moving to Other Computers (p. 21)
**show: Displaying Variables**

The `-show-` command displays the value of a variable or expression or the contents of a marker variable. The `-show-` is sensitive to the type of information in its tag (numerical or character string), and selects the display format accordingly. This section deals only with numerical displays (the "See Also" section includes a reference to using `-show-` to display marker variables).

```
  show    expression
  show    expression, significant figures
  show    expr, sig figs, minimum
  showz   expression, sig figs  $$ show trailing zeros (see below)
```

When the one-argument form is used, default values are used for "significant figures" and "minimum." The default number of significant figures is 4. The default minimum is $10^{-9}$. Any expression whose absolute value is less than $10^{-9}$ is displayed as 0 unless a different minimum is specified.

Very large and very small values are automatically displayed with "scientific notation." (In scientific notation, $1.19E+03$ means 1190 or $1.19\times10^3$). You can force this "exponential" notation with a `-showe-` command.

There are two special floating-point values that may be displayed by a `-show-` command: INF (infinity) is displayed when a nonzero value has been divided by zero; NAN ("Not A Number") is displayed when a zero value is divided by zero, which might also be considered an "indefinite" result.

If the value would require additional digits to express the number, then it is displayed in scientific notation. This depends on the number of significant digits that are displayed. For example:

```
<\$, 456, 2\$> shows  4.6E+02
<\$, 456, 3\$> shows  456
```

The `-show-` command does not show trailing zeros, so `-show 12.5-` displays "12", not "12.000". The `-showz-` command does show trailing zeros. For detailed format control, use the `-showt-` command.

If you don't want to use the system defaults, you might set "significant figures" and "minimum" in your `-define-` set at the beginning of the program:

```
define  i: MySigFigs = 2
        f: MyMinimum = 1E-12
....
show   value, MySigFigs, MyMinimum
```

**Examples:**

```
unit    xshow
  i: count
loop   count := 1, 15  $$ count = 1,2,3,4, ....,15
  . at  50, 18count  $$ space down 18 each line
  . show  count
  . at  100,18count
  . show  sqrt(count)  $$ square root
endloop
*
unit    xshow2
  i: x, y, yloc
```
calc yloc := 0
loop x := -20, 20, 4 $$ x = -20, -16, -12, etc.
. calc yloc := yloc + 20
. at 50,yloc
. show x
. at 100,yloc
. show 2**x $$ 2 to the power x
endloop
*

See Also:
Embedding Variables in Text (p. 50)
show Contents of a Marker Variable (p. 250)
Using Embedded Marker Variables (p. 251)

**showt: Specifying the Format**

The -showt- command displays the value of a variable or expression in a format suitable for producing aligned tables of numbers. (The "t" is for "tabular").

```
showt expression, left, right
showt expression, left
showt expression
```

The first argument of the -showt- tag is the variable or expression to be displayed. The second argument ("left") gives the number of digits to allow to the left of the decimal. The third argument ("right") gives the number of digits to display to the right of the decimal. The right-hand digits are always displayed, even if they are zero. The beginning of the displayed number is filled with spaces so that the decimals are aligned.

If "right" is omitted or zero, there are no digits to the right of the decimal and no decimal point is displayed. If both "left" and "right" are omitted, a default format of 4, 3 is assumed: 4 digits to the left and 3 digits to the right.

If the number to be displayed requires more digits than are allowed by "left", the decimal point is moved out of line. The -showt- never "refuses" to display a number.

When -showt- is embedded, it can be shortened to "t".

**Examples:**

```
unit xshowt1
at 100,50
showt 1357.2345, 5, 2
at 100,70
showt 9.999, 5, 2
*
```

The next example displays the values of $x^{2.3}$ for x running from 1 to 10. Notice that when left=5, it allows for 5 digits, leaving some space between the line and the actual beginning of the number. When left=2, there is not enough space, so that the column is not nicely aligned.

```
unit xshowt2
f: x
```
i: left, right

draw 50,10; 50,250
calc left := 5 $$ 5 digits before the decimal
   right := 2 $$ 2 digits after the decimal
loop x := 1, 10
   at 50, 20x
   showt x^2.3, left, right
endloop
draw 200,10; 200,250
calc left := 2 $$ 2 digits before the decimal
loop x := 1, 10
   at 200, 20x
   showt x^2.3, left, right
endloop

See Also:
Embedding Variables in Text (p. 50)
show Displaying Variables (p. 47)

**showb: Displaying in Non-Decimal**

The -showb-, -showo-, and -showh- commands display the value of a variable or expression in binary, octal, or hexadecimal. These commands are usually used with byte or integer variables. If a floating-point variable is displayed, it is first rounded to an integer and then the integer value is displayed.

```plaintext
showb x $$ embed as <|b,x|>
showo x $$ embed as <|o,x|>
showh x $$ embed as <|h,x|>
showb x,8 $$ show 8 digits
showo x,3 $$ show 3 digits
showh x,2 $$ show 2 digits
```

The first argument names the variable (or constant) to be displayed. The second argument gives the number of digits to display.

If the second argument is omitted, the number of digits displayed is "enough digits to display the maximum value that can be stored in that type of variable." For integers, -showb-, -showo-, and -showh- display 32, 11, and 8 digits, respectively. For bytes, they display 8, 3, and 2 digits.

If the second argument is 0 or less, nothing is displayed. If the second argument is larger than the default value, it causes an error.

**Example:**

```plaintext
unit xshowo1
i: N
at 5,5
write decimal hex octal binary
at 5,25
loop N := 1, 1000, N+1
write <|t,N,3|> <|h,N|> <|o,N|> <|b,N|> <|cr|>
```

49
See Also:
Embedding Variables in Text (p. 50)
show Displaying Variables (p. 47)

Embedding Variables in Text

The embedded form of the -show- command allows any variable to be displayed as part of a -text-, -write- or -string- statement. The variable is evaluated and its value is displayed as part of the text. The embedded command is written <|command,variable|>. All of the show-type commands may be embedded. Each may be abbreviated to a single letter:

- write x is <|show, x|> $$ "s"
- marker is <|show,m1|> $$ "s"
- binary is <|showb,x|> $$ "b"
- octal is <|showo,x|> $$ "o"
- hex is <|showh,x|> $$ "h"
- formatted: <|showt,x,5,3|> $$ "t"
- <|cr|>, <|tab|>, <|quote|> $$ special characters

Consider this -write- statement:

write I have <|show,N|> apples
     and <|show,p+3|> peaches.

If N=12 and p=19, it will display

I have 12 apples
and 22 peaches.

The example above is (essentially) equivalent to

write I have
show N
write apples
     and
show p+3
write peaches.

When an embedded show is used, all of the information for the sentence is included in one -write- statement, that is, in one command statement. When explicit -show- commands are used, it requires several command statements to display the same sentence. This difference is important in response judging (-arrow-s), because only one text-display statement is automatically erased after a response is finished.

Examples:

unit xembed
i: peaches, apples
randu peaches, 5 $$ select value from 1-5
randu apples, 10 $$ select value from 1-10
at 50,100
write I have <|s.peaches|> peaches and <|s.apples|> apples.
*

The next example uses embedded -show- commands to display variable contents. The embedded forms <|quote|>, <|cr|>, and <|tab|> are used to insert double-quotes, carriage-return, and TAB. Note how the form of the -show- changes the display of the numbers ("s" vs. "t").

unit xembed2
f: A(6)
m: phrase
set A := 1.011, 3, 5.7, 4567, 902, -27
calc phrase := "hello"+ <|cr|> +"hello"+ <|cr|> +"hello"
at 30,30
write <|s,phrase|>
at 30, 100
write <|s,A(1)|><|tab|><|s,A(2)|><|tab|><|s,A(3)|>
<|s,A(4)|><|tab|><|s,A(5)|><|tab|><|s,A(6)|>
at 30, 150
write <|t,A(1)|><|tab|><|t,A(2)|><|tab|><|t,A(3)|>
<|t,A(4)|><|tab|><|t,A(5)|><|tab|><|t,A(6)|>
*

See Also:
Using Embedded Marker Variables (p. 251)
show Contents of a Marker Variable (p. 250)
string Text in a Marker Variable (p. 248)
Lines, Circles, Boxes, Etc.

**draw: Drawing Lines**

The -draw- command displays continued lines. The argument is a series of screen positions separated by semicolons. Coarse grid and fine grid addresses may be mixed in the command. The word "skip" indicates a break in the figure.

```plaintext
draw 10,20; 100,20; skip; 50,60; 80,90

draw ; 200, 150 $$ draw from current position
```

If the -draw- starts with a semicolon, the drawing starts from the current screen position. The command -inhibit startdraw- is useful when starting a series of continued lines.

The thickness of the lines can be specified with a -thick- command.

**Examples:**

This example draws a right triangle whose base is at the top and whose tip is at the lower right:

```plaintext
unit xdraw1
draw 100,100; 200,100; 200,300; 100,100
```

This example uses the system variables `zxmin,zymin, zxmax,zymax` to draw using the full window:

```plaintext
unit xdraw2
draw zxmin,zymin; zxmax,zymax
draw zxmax,zymin; zxmin,zymax
```

Unit "xdraw3" uses `skip` in the first -draw- command to draw disconnected lines. The later -draw-s are "continued" to display several connected lines.

```plaintext
unit xdraw3
draw 40,100; 40,200; skip; 80,100; 80,200
draw 200,0; 150,50
draw ; 200,100; 150,150
draw ; 200,200; 150,250
```

In "xdraw4", after the -dot- is displayed, the system variables `zwherex` and `zwherey` are set to `zwherex=50` and `zwherey=100`. The `zwherex` and `zwherey` are not updated until after a statement is completed, so the following -draw- statements are equivalent:

```plaintext
unit xdraw4
dot 50,100
draw 10,20; zwherex-5,zwherey-5; 230,40
pause $$ wait for keypress
mode xor $$ change modes
draw 10,20; 45,95; 230,40
```
dot: Making Dots

The -dot- command displays a single dot. Multiple dots can be displayed with one command by giving several points, separated by semicolons. If the tag starts with a semicolon, the first dot is displayed at the current screen position.

```
dot 150,150

dot ; 75,50; 75,75
```

The -dot- command with one point is equivalent to a -draw- command with a single position:

```
draw 150,150
```

The thickness of the dot can be specified with a -thick- command.

Example:

```
unit xdot
i: n
thick 3
loop n := 50,150, 5
    dot n,100
endloop
```

See Also:

- thick Line Thickness (p. 62)
- System Variables for Graphics and Mouse (p. 321)
- inhibit/allow startdraw (p. 66)

circle: Drawing Circles and Arcs

The -circle- and -circleb- ("broken circle") commands are used for displaying circles, arcs, and ovals:

```
circle radius $$ circle centered on current screen position
circleb radius $$ broken or dashed circle centered on current screen position
circle radius, angle1, angle2 $$ circular arc centered on current screen position
circle x1,y1; x2,y2 $$ oval bounded by the specified rectangle
```

When drawing a circle centered on the current screen position, the first argument of the tag gives the radius of the circle. The optional second and third arguments specify the beginning and ending angles for drawing partial circles (arcs). The partial circle (arc) is drawn from the smaller angle to the larger angle, inclusive, regardless of the order in which the angles are specified.

The angles are normally measured in degrees, increasing in the same direction that y increases. That is, the angle increases in the direction determined by rotating the positive x-axis toward the positive y-axis. If 0,0 is in the upper-left corner of the display with y increasing as you move down, the angle is measured clockwise. In the
graphing coordinate system, y is usually defined to increase upwards, and the arc angles are measured counterclockwise.

The command -inhibit degree- changes to radian measure for the arc angles. Using radians is often convenient for consistency with the radians that are always used in the trigonometric functions (sine, cosine, arctan, etc.).

To draw an oval, specify the corners of a rectangle that the oval would fit inside.

After a circle is complete, the screen position (zwherex,zwherey) is set to the center of the circle (for an oval, to the upper-left corner of the bounding rectangle). If a partial circle (arc) is drawn, the screen position is set to the last point on the arc.

The thickness of the circle is affected by the -thick- command. See the -thick- command for issues that arise with circles that have large thicknesses.

Example:

```
unit xcircle
at 100,100 $$ set center of circle
circle 50 $$ radius 50
thick 2
circleb 75 $$ radius 75; thick dashes
thick
circle 60,80;140,120 $$ oval
*
unit xcircle2
at 70,100 $$ set center of arc
circle 50, 45, 270 $$ arc from 45 to 270 degrees
at 70,100 $$ must reset center
circle 60, -90, 45 $$ arc from 270 to 45 degrees
draw 170,100; 70,100; 150,180
at 175,95
write 0 degrees
at 155,175
write 45 degrees
draw 70,100; 70,20
at 75,15
write 270 degrees
*
```

See Also:
- disk Filling a Circle (p. 57)
- thick Line Thickness (p. 62)
- inhibit/allow degree (p. 70)

**vector: Line with Arrowhead**

The -vector- command displays a line with an arrowhead that can have an open head or a closed head. The head size can vary.

```
vector tailX,tailY; headX,headY
vector ; headX,headY; headtype
```
If the tag starts with a semicolon, the tail of the vector is at the current screen position \((z\text{where}x, z\text{where}y)\). After a -vector-, the current screen position is set to the point of the vector.

The optional head type specifies the type of vector drawn and the size of the arrowhead. The sign of "headtype" determines whether a closed or open arrowhead is displayed. When the head type is positive or omitted, the arrowhead is a filled triangle. When the head type is negative, the arrowhead is formed by two angled lines—the head is "open." If the vector is shorter than the standard head, cT shrinks the head to fit.

When the head type is an integer, it specifies the length of the arrowhead in screen units. If the head type is omitted, the length of the arrowhead is about 20 screen units. If the head type is a fraction, the length is not an absolute value, but is that fraction of the length of the vector's length.

When a -pattern- command is in effect, a closed vector head is textured rather than solid.

The thickness of the vector shaft can be specified with a -thick- command, but note that you may also want to increase the head size to match the thicker shaft.

**Examples:**

```
unit xvector
vector 25,50; 225,50 $$ default head size
vector 25,100; 225,100; 60 $$ 60-dot head size
vector 25,150; 225,150; -30 $$ open head
vector 25,200; 225,200; .5 $$ head size depends on length
*
```

In this example, the -rvector- command starts with a semicolon, so that the start (tail) of each vector is the point of the previous vector.

```
unit xvector2
f: angle
rorigin 130,150
rat 100,0 $$ start of first vector
loop angle := 0,360,30
  rotate angle
  rvector ;100,0
endloop
*
```

**See Also:**

- pattern Making Textured Areas (p. 61)
- thick Line Thickness (p. 62)

**box: Making a Rectangle**

The -box- command is used for drawing rectangles.

```
box x1,y1; x2,y2
box x1,y1; x2,y2; thickness
box ;x2,y2; thickness
box $$ available display area
```
The screen positions in the tag of -box- give diagonally opposite corners of the rectangle to be displayed. If the first position is omitted, one corner of the box is the current screen position (zwherex,zwherey).

The "thickness" argument causes the outline of the rectangle to be "thickness" screen units wide. If the "thickness" argument is positive, the thickness is added to the outside of the basic rectangle; if the argument is negative, the thickness is developed inwards. The thickness of the lines of the box can also be specified with a -thick- command, in which case the thickness is the number of pixels, rather than (possibly rescaled) screen units.

When a -pattern- is in effect, a thick box is displayed with that pattern.

The blank-tag form of -box- draws a frame around the region specified by a -fine- command. The blank-tag form of the graphing box command, -gbox-, draws a frame around the region of the graphing axes.

After the rectangle is drawn, the current screen position (zwherex,zwherey) is set to the first point mentioned ("corner1").

Examples:

Note that the three boxes are of the same basic size (100 x 50).

```
unit      xbox
box       50,50; 100,150
box       150,50; 200,150; 15   $$ thickness grows outward
box       250,50; 300,150; -7   $$ thickness grows inward
```

This example shows a box in the relative coordinate system. The walls of the box are of uneven thickness because one screen unit in the "x" direction is three times as large as one screen unit in the "y" direction.

```
unit      xbox2
rorigin   100,50   $$ set relative origin
size      3,1       $$ expand x units
rotate    30        $$ rotate axes around 100,50
pattern   zpatterns, 7   $$ set fill pattern for edges
rbox      0,0; 100,100; -10   $$ thickness 10 units inward
size      1         $$ back to default size
rotate    0         $$ back to default rotation
```

See Also:

Describing the Screen (p. 34)
Current Screen Position (p. 323)
pattern Making Textured Areas (p. 61)
thick Line Thickness (p. 62)

fill: Filling an Area

The -fill- command fills in a polygon whose corners are specified by the arguments of the tag.

```
fill      x1,y1; x2,y2 $$ rectangle
fill      x1,y1; x2,y2; ... ; xN,yN
fill      $$ full-screen fill with existing -pattern-
```
If the tag of the -fill- command only has two points specified, as in the first line above, it draws a filled-in rectangle. If there are more than two points specified, the result is as if those points were -draw-ing a figure and then filling it in. The final point, which closes the figure, does not need to be specified.

The -pattern- command may be used to specify a pattern for the -fill-. The default pattern is solid black. A blank-tag -fill- fills the entire window (like a blank-tag erase), using the current pattern, which is useful for giving texture to the window.

After the figure is drawn, the current screen position (zwherex,zwherey) is set to the first point mentioned ("corner1") .

The following sequence of triangular -draw- and -erase- statements leave the lower-right slanted line intact:

\[
\begin{align*}
\text{draw} & \quad 10,10;176,10;10,180; \quad 10,10 \\
\text{erase} & \quad 10,10;176,10;10,180 \quad \text{same area as a -fill-}
\end{align*}
\]

This is because a nonrectangular -fill- or -erase- goes to just inside the bottom and right sides of the polygon, and this -draw- is just outside the filled region.

**Example:**

\[
\begin{align*}
\text{unit} & \quad \text{xfill} \\
\text{fill} & \quad 50,50;150,100 \quad \text{$\$$ rectangle} \\
\text{fill} & \quad 50,200;75,150;130,130;175,250 \quad \text{$\$$ 4-sided polygon} \\
\end{align*}
\]

*See Also:*

erase Erasing an Area (p. 58)

pattern Making Textured Areas (p. 61)

**disk: Filling a Circle**

The -disk- command displays a filled-in circle or oval. The -pattern- command is used to make a textured fill instead of a solid color fill.

\[
\begin{align*}
\text{disk} & \quad \text{radius} \quad \text{$\$$ circular disk centered at current screen position} \\
\text{disk} & \quad x1,y1; x2,y2 \quad \text{$\$$ oval disk bounded by specified rectangle}
\end{align*}
\]

With one argument specifying the radius, the center of the disk is at the "current screen position". An -at-command is usually used to specify current screen position. With four arguments, the disk is a filled oval that lies inside the specified rectangle.

After the disk is complete, the screen position (zwherex,zwherey) is set to the center of a circular disk, or at the upper-left corner of the bounding rectangle for ovals.

**Examples:**

\[
\begin{align*}
\text{unit} & \quad \text{xdisk1} \\
\text{at} & \quad 100,100 \\
\text{disk} & \quad 50 \quad \text{$\$$ circle} \\
\text{disk} & \quad 30,150 ;170,180 \quad \text{$\$$ oval}
\end{align*}
\]
The second example shows patterned disks. Sometimes a disk looks better if it is outlined.

```
unit xdisk2
pattern zpatterns, 13
disk 50,80; 150,120
pattern zpatterns, 6
at 200,100
disk 75
circle 75 $$ outline disk
*
```

Although partial disks are not available directly, there are tricks one can use. Half-disks or quarter-disks can be achieved by using `-clip-`.

```
unit xdisk3 $$ partial disk with clip
clip 50,50; 300,150
box 50,50; 300,150 $$ show clipped region
at 50,100
disk 50 $$ right half of a disk
at 200,50
disk 50 $$ bottom half of disk
at 300,150
disk 50 $$ quarter disk
clip $$ don't forget to cancel the clip!
*
```

An `-erase-` can be used to remove part of a disk.

```
unit xdisk4 $$ partial disk with erase
at 150,100
disk 50
erase 176,44; 150,100; 1 97,69
*
```

See Also:
```
pattern Making Textured Areas  (p. 61)
```

**erase: Erasing an Area**

The `-erase-` command clears a region of the display. It erases a polygon (like `-fill-`) whose corners are specified by the addresses of the tag.

```
erase x1,y1; x2,y2 $$ rectangle
erase x1,y1; x2,y2; ... ; xN,yN
erase $$ erase entire display
```

If the tag of the `-erase-` command has two screen positions specified, it clears a rectangular area. If there are more than two points specified, the area erased is a polygon whose corners are specified by the addresses of the tag. The final point, which closes the figure, does not need to be specified. The erasure region is restricted to the display area defined by the `-fine-` command. After the `-erase-` command is executed, the current screen position `(zwherex,zwherey)` is set to the first point mentioned in the tag.
The `-erase` command is affected by the `-pattern` command, as are all commands that display a filled area. If a `-pattern` is in effect, the `-erase` removes only those dots that would normally be "turned on" when the pattern is displayed.

The blank-tag `-erase` clears the entire window, including the area outside the `-fine` area. It is unaffected by a `-clip`. The color is specified by the `-wcolor` command. If no `-wcolor` command has been executed, the color is the default background color, `zdefaultb`.

The following sequence of triangular `-draw` and `-erase` statements leave the lower-right slanted line intact:

```
draw 10,10;176,10;10,180; 10,10
erase 10,10;176,10;10,180
```

This is because a nonrectangular `-fill` or `-erase` goes to *just inside* the bottom and right sides of the polygon, and this `-draw` is just outside the filled region.

NOTE: If the tag does not give two screen positions, it is a form of `-erase` included partly for compatibility with systems that use "coarse grid". Character width and height are specified by the width of an "n" and the height of a line (or they are specified by a `-coarse` command).

```
erase characters $$ # chars to erase
erase characters,lines $$ for several lines
```

**Examples:**

These two units illustrate how the same effects can be achieved with either `-erase` or `-mode erase`.

```
unit xerase
fill 50,50; 300,200 $$ make a solid rectangle
erase 80,80; 150,100; 100,170 $$ erase a triangle
erase 200,80; 250,160 $$ erase a rectangle
*
unit xmode
fill 50,50; 300,200 $$ make a solid rectangle
mode erase
fill 80,80; 150,100; 100,170 $$ triangle
fill 200,80; 250,160 $$ rectangle
mode write
*
```

**See Also:**

- `clip` Limiting the Display Area (p. 63)
- `mode` Changing Modes (p. 60)
- `pattern` Making Textured Areas (p. 61)
- `wcolor` The Window Color (p. 79)
- Inhibit and Allow in Judging (p. 177)
Mode, Pattern, Thick, Cursor, & Clip

mode: Changing Modes

The -mode- command changes the way a graphics or text display command is treated.

- mode write -
- mode erase -
- mode xor -
- mode rewrite -
- mode inverse -

The mode is always set to -mode write- when entering a new main unit. In -mode write-, any new display appears on top of the existing display. In -mode erase-, what would have been written in the foreground color is written in the background color.

In -mode rewrite-, the -text- and -write- commands erase the area (filling with the background color) before displaying the new text. A pattern -fill- displays the pattern in the foreground color and the other areas in the background color. The -mode inverse- is like -mode rewrite-, but with the colors reversed.

In -mode xor- ("exclusive or") the color of each displayed pixel is reversed from its current color in such a way that precisely repeating this graphics operation restores the screen to its original form. This is useful for such things as "rubber-banding" a line or moving an icon across other objects.

The actual color of an object displayed in -mode xor- is not welldefined.

Typically, the color of an object in -mode xor- is unpredictable and completely independent of your foreground and background colors, even when drawing on a region filled with the background color. But there is reversibility: doing two -mode xor- displays in the same color at the same place always restores the screen to its original appearance.

The system variable zmode tells what the current graphics mode is (as set by the -mode- command).

Examples:

- unit xmode1
- mode write
- at 35,20
- write This is in -mode write-.  
- mode inverse
- text 50,50;400,130
- The black area surrounding this inverse -text- extends to the right margin and to the end of the sentence.  
- 
- mode write $$ be sure to return to "write" !
- *
- unit xmode2
- at 35,50
- write Notice that the lines from the -draw- are black outside the filled area and white inside it.  
- fill 104,173;68,143;164,126;142,163;202,227;75,234
- mode xor
- draw 111,117;68,173;139,244;182,165;110,117
mode write

* unit xmode3
draw 88,15;288,176
do xmode3a $$ show some text
pause
mode erase $$ prepare to erase
do xmode3a $$ remove the text
mode write $$ return to -mode write-
*
unit xmode3a $$ make a little display
at 55,40
write When -text- or -write- is displayed in -mode- erase, the letters are removed.

Notice how the line through the text is affected when you press a key.

* See Also:
  zmode Current Mode (p. 324)
erase Erasing an Area (p. 58)
clip Limiting the Display Area (p. 63)
Inhibit and Allow in Judging (p. 177)

pattern: Making Textured Areas

The -pattern- command specifies an icon or character whose pattern is used to fill areas in the -fill-, -box-, -disk-, -erase-, -vector-, -vbar-, and -hbar- commands.

pattern zpatterns,9
pattern "mypatterns",5
pattern $$ blank tag returns to default

The first argument of the tag names a file that contains an icon set. The second argument is the character number within that set of the selected pattern. The default fill pattern is all dots on; that is, solid.

The icon set zpatterns is a generic set that is available on all machine types. In this set, patterns 0 through 16 are 4 by 4 patterns containing 0, 1, 2, ...15, 16 dots. This makes it easy to choose algorithmically among different densities of patterns.

You can use Icon Maker on the Macintosh or icon.t on other platforms to create your own patterns. Note that in Icon Maker you must specifically mark an icon as being usable as a pattern.

The -pattern- command allows character numbers from 0 to 126.

The cT sample program showicon.t displays all the patterns in an icons set and shows the numerical code (0 to 126) for each pattern. To use pattern number 97, use either -pattern filename,97- or -pattern filename.zk(b)- (since the character "b" has numerical code 97).
Example:

```
unit xpattern
pattern zpatterns,8 $gray
fill 50,50; 300,150
pattern $return to default (black)
box 50,50; 300,150; 20
* 
```

See Also:
- Making Icons on Macintosh (p. 95)
- Making Icons on PC & Unix (p. 96)
- Generic Font Names (p. 327)
- File Name Specification (p. 286)

**thick: Line Thickness**

The `-thick-` command makes later line-drawn commands use thick patterned lines:

```
thick 4 $$ 4 times normal thickness, affected by -pattern-
thick $$ turn off thick option
thick 0 $$ turn off thick option
thick 1 $$ turn off thick option
```

When a thickness N greater than 1 is in effect, lines are drawn as though the center of a filled-in square of size N by N were dragged along the path of the line. This ensures that a sequence of connected lines join appropriately, but very thick circles may not appear as you would like (a very thick circle can be created by a `-disk-` command followed by a smaller concentric `-disk-` in a different color).

Line-drawn commands affected by a preceding `-thick-` command include `-box-`, `-circle-`, `-circleb-`, `-dot-` (useful for plotting dots on a graph), `-draw-`, `-vector-` (but note that you may also want to increase the head size to match the thicker shaft), `-axes-`, `-labelx/y-`, `-markx/y-`, and the corresponding graphing and relative graphics commands.

Example:

```
unit xthick
thick 6
pattern zpatterns,12
color zblue
draw 35,30;70,65;35,100;35,30
color zred
vector 50,65;200,65
dot 210,65
* 
```

**cursor: The Mouse Pointer**

The `-cursor-` command allows you to change the pointer that moves around on the display when the mouse is moved. The usual (default) pointer is a small, slightly curved arrow that points upward to the left.

```
cursor zcursors,zk(f)
```
cursor zcursors,3 $$ invisible cursor
cursor zcursors,72 $$ watch cursor (indicate waiting)
cursor "myobjects",zk(O)
cursor $$ blank tag returns to default

The first argument of the tag names an icon set. The second argument is the character number within that set of the selected pattern. The icon set zcursors is a generic set that is available on all machine types. You can use Icon Maker on the Macintosh or icon.t on other platforms to create your own cursors. Note that in Icon Maker you must specifically mark an icon as being usable as the mouse cursor.

Example:

```
unit xcursor
  i: ncursor
at 10,10
write Note: If the cursor is the standard one when you press a key, there is no cursor associated with that index.
loop ncursor := 0, 127
  cursor zcursors,ncursor
  erase 60,70; 100,100
  at 60,70
  show ncursor
  pause
endloop
```

* See Also:  
  Making Icons on Macintosh (p. 95)  
  Making Icons on PC & Unix (p. 96)  
  Generic Font Names (p. 327)  
  File Name Specification (p. 286)

**clip**: Limiting the Display Area

The -clip- command establishes a rectangular area of the display. Any text or drawing that would fall outside of this area is not shown.

```
clip 100,100; 300,300
clip $$ blank-tag cancels clipping
```

The -clip- command does not change the current screen position given by zwherex, zwherey. Clipping is not reset on entry to a new main unit.

The -rclip- and -gclip- commands take relative and graphing coordinates, but -rclip- clips a horizontal rectangle, unaffected by -rotate-.

**NOTE**: Clipping applies to all graphics commands other than -button-, -slider-, and -edit-, and only one clipping region may be in effect at a time. Thus a -gclip- affects commands from the absolute and relative coordinate systems as well as those from the graphing system. You cannot simultaneously have a -clip- and a -gclip- in effect; the most recent one overrides earlier clips.
Graphics objects (-button-, -slider-, and -edit-) cannot be created if a -clip- is in effect that would obscure part of the object, and these objects are unaffected by later -clip- commands. The clip region is temporarily reset when these graphics objects are manipulated.

Example:

The -gclip- command is particularly useful with graphs, to ensure that the graph does not go all over the display:

```
unit xclip
f: x
gorigin 40,170
axes 0,-50; 300,150
scalex 360
scaley 3
labelx 90,45,1
labely 1,0.5,1
gclip 0,-1; 360,3
gat 0,0
loop x := 0, 360, 5
  gdraw ;x, .01x +sin(x/15)
endloop
gclip
*
```

See Also:
- at Positioning Graphics (p. 34)
- inhibit/allow display (p. 67)
- mode Changing Modes (p. 60)
Inhibit and Allow in Graphics

-inhibit- and -allow- in Graphics

The -inhibit- and -allow- commands modify various default behaviors. The tag is a keyword naming the behavior to be modified.

```
inhibit startdraw
inhibit startdraw, degree $$ combined keywords ok

inhibit $$ blank tag; reset to defaults
allow $$ blank tag; reset to defaults
```

There may be several -inhibit- and/or -allow- commands in one unit; the results are cumulative. Several keywords may be combined in one tag. The following keywords are available as tags for -inhibit- with respect to graphics (in addition to the inhibit options `anserase`, `arrow`, and `blanks` for judging):

```
erase $$ do not erase on new unit
startdraw $$ do not show first -draw- output
display $$ do not display output
update $$ do not update screen immediately
objdelete $$ do not delete graphics object on -jump-
optionmenu $$ do not display Option menu
buttonfont $$ do not use system font for button text
supsubadjust $$ do not adjust text positioning with sup or sub
degree $$ do not use degrees with -circle-, -rotate-, -polar-
fuzzyeq $$ do not use "fuzzy zero" in comparisons
```

When these tags are used with -allow- all of the "do nots" are changed to "do". The effect of an -inhibit- or -allow- lasts until it is canceled by one of these actions:

1) a blank-tag -inhibit- or -allow-
2) a specific -inhibit- or -allow- command
3) the beginning of a new main unit

The blank-tag -inhibit- and the blank-tag -allow- have the same effect. All of their options are changed back to the default status, as if these commands had been executed:

```
inhibit blanks
allow anserase, arrow, erase, startdraw, display
allow update, objdelete, supsubadjust, degree
```

The keyword `reset` can be used to reset to the default options:

```
inhibit reset, erase $$ set to defaults, then -inhibit erase-
```

See Also:
Inhibit and Allow in Judging (p. 177)


**inhibit/allow erase**

This command modifies the full-screen erase that usually occurs when entering a new main unit. The default status is -allow erase-.

Normally, the entire display is erased when the user enters a new main unit by pressing ENTER, by selecting (Next Page) or (Back), or when the program goes to a new unit with -jump-. The -inhibit erase- prevents this automatic erasure.

*Example:*

```
unit xInhibitErase0 $$ use "Run from Selected Unit"
do circles $$ circles
inhibit erase $$ try removing this line
jump xInhibitErase
*
unit xInhibitErase
at 160,35
write Now in unit xInhibitErase.
*
unit circles $$ a group of circles
 i: r $$ circle radius
loop r := 75,175,25
   at 200,r
   circle r
endloop
*
```

**inhibit/allow startdraw**

The "startdraw" tag on -inhibit- causes the first element of the next -draw- or -move- command NOT to be displayed. When "startdraw" has an argument, the first element of the next N -draw- or -move- commands is not displayed. The default status is -allow startdraw-.

```
inhibit startdraw
inhibit startdraw(N)
```

When a display such as a graph or an animation is being generated by a loop, the first item must often have special treatment. The -inhibit startdraw- was designed to simplify loops where -draw- and -move- are used with initial semicolons to indicate "start from previous location."

*Examples:*

At the beginning of every main unit, the "current screen position" is set to 0,0. If -inhibit startdraw- is omitted from this example, an extra line is drawn from 0,0 to the beginning of the arc.

```
unit xstartdraw $$ show a parabola
 f: i, x, y
inhibit startdraw $$ try it without this line
loop i := -100,100,10 $$ i = -100, -90, ... 90,100
   . calc x := i+150
   . calc y := i*i/100
```
When `-inhibit startdraw-` is used with the `-move-` command, the first `-move-` command after `-inhibit startdraw-` does not erase the "from" location. Thus, the first iteration of the `-move-` command in this loop merely writes into the "to" location. Without the `-inhibit startdraw-`, an extra O is shown, right after the word "box."

```
unit xstartdraw2
f: x
fill 142,87;224,120 $$ a solid box
at 50,20
write Watch the letter move
through the box.
inhibit startdraw $$ try it without this line
mode xor $$ "flip dots" graphics mode
loop x := 100, 250, 2
move icons: ; x,100; zk(O)
pause .1 $$ slow it down
endloop
mode write $$ normal graphics mode
*
```

Do not mistake the first element for the entire command. The next example illustrates the treatment of an ordinary `-draw-`. The first line of each `-draw-` is suppressed.

```
unit xstartdraw3
inhibit startdraw(3) $$ try it without this line
draw 50,50; 50,100; 100,100 $$ draws an "L"
draw 150,50; 150,100; 200,100
draw 250,50; 250,100; 300,100
*
```

**inhibit/allow display**

This command modifies the normal display of output. The default status is `-allow display-`.

Occasionally it is useful to know what the current screen position would have been if a display had been plotted without actually plotting the display. The `-inhibit display-` prevents output of display commands. It does not stop these commands from executing; all processing is normal except the final step of output to the screen.

The `-inhibit display-` command is also used to make the input at an `-arrow-"invisible," as might be used for entering a codeword. The input is still processed; it just isn't visible.

**Example:**

This example illustrates a lazy way to draw a line at a specific angle. It "draws" an arc but does not display it. It then draws a line from the end of the arc back to the center of the circle.

```
unit xInhibitDisplay
inhibit display
at 100,100
circle 50, 0, 135 $$ arc to 135 degrees
allow display
```
inhibit/allow  update

This command modifies the normal display of output. The default status is -allow update-.

inhibit  update  $\$$ delay output
allow   update  $\$$ update the screen

Normally cT takes pains to ensure that the display appears on the screen smoothly, rather than in spurts. The statement "inhibit update" causes the (computer's) instructions to the screen to be accumulated in a buffer. An "allow update" sends the entire buffer to the display program. NOTE: A very extensive display may overflow the display buffer, which forces the display to update.

Notice the difference between the "update" and "display" tags for -inhibit- and -allow-. An -inhibit display- calculates the display information, but never sends it to the display buffer. With -inhibit update-, the display information reaches the display buffer, but is not sent to the screen display program until an -allow update- is issued.

Example:

Try this example with and without the -inhibit update- line.

```plaintext
unit  xupdate
f: angle
origin 200, 100 $\$$ prepare for graphing
bounds -150,0; 150,150 $\$$ set axes, no display
polar TRUE $\$$ use polar coordinates
scalex 15 $\$$ set scaling
scaley 15
inhibit update  $\$$ turn off screen updating
loop angle := 0, 1800, 15
gdraw ; angle/200, angle
endloop
allow update  $\$$ show accumulated display
polar FALSE $\$$ restore x,y coordinates
*```

inhibit/allow  editdraw

You can temporarily prevent updating the screen display in an edit panel that would normally occur as you change the associated marker text:

inhibit   editdraw  $\$$ don't update edit panel display yet
allow    editdraw  $\$$ now update the edit panel display

This can be useful when you make lots of changes to the text and want them all to appear on the screen all at once rather than one at a time.
inhibit/allow objects

You can temporarily prevent button, slider, edit panel, and video controller objects from processing events. Events which occur while -inhibit objects- is in effect are discarded.

```
inhibit    objects $$ stop processing "object" events
allow      objects $$ resume processing "object" events
```

This can be useful when you are executing some critical statements and don’t want to be interrupted by user inputs.

inhibit/allow objdelete

Normally a -jump- automatically destroys existing edit panels, buttons, sliders, and touch regions. With -inhibit objdelete- you can prevent this automatic destruction, and this also inhibits the normally automatic full-screen erase:

```
inhibit    objdelete
jump       somewhere
...         
allow      objdelete
```

For the undestroyed objects to function, their associated variables have to be global variables. If these variables are defined as local variables in the original unit, the object information cannot be preserved even with -inhibit objdelete-.

inhibit/allow optionmenu

cT normally creates an Option menu, or re-creates an Option menu when -next- or -back- or -arrow- are active (or -pause keys=next-). You can permanently remove the Option menu with -inhibit optionmenu-:

```
inhibit    optionmenu $$ remove Option menu (and do not re-create it)
allow      optionmenu $$ restore Option menu
```

Caution! The option to quit running the program is normally on the Option menu. If you delete the Option menu, you should offer another way out of the program, like this:

```
menu       Special; Quit: GetOut $$ make your own "Quit" option
inhibit    optionmenu $$ delete the Option menu
...         
unit       GetOut
jumpout    $$ quit the program
```

inhibit/allow buttonfont

Normally cT uses a special system font for the text displayed in a button, but you can inhibit this:

```
allow      buttonfont $$ use special system font for button text (default)
inhibit    buttonfont $$ use current font (set by -font- or -fontp-)
```

Normally styles such as subscript or italic are ignored in displaying the text of a button, but with -inhibit buttonfont- in effect, the displayed button text can contain styles and can even contain pasted-in images.
inhibit/allow supsubadjust

Normally, in order to fit within vertical margins, text is moved down if there is a superscript, and a line is also moved down if the preceding line contained a subscript. There is an -inhibit- option that causes cT to ignore superscripts/subscripts when aligning text in -write- or -text- statements:

```
unit xsupsubadjust
draw 10,100; 240,100
at 10,100
write H₂O and x³ moved down from upper margin,
and this line is moved down even more.
inhibit supsubadjust
draw 10,170; 240,170
at 10,170 $$ upper margin at y = 170
write H₂O and x³ not moved down; clipped,
and this line is not adjusted.
draw 10,240; 240,240
atnm 10,240 $$ no upper margin
write H₂O and x³ not moved down; not clipped,
and this line is not adjusted.
allow supsubadjust*
```

This currently has no effect on edit panels.

inhibit/allow degree

Angles in -circle- arcs, -rotate-, and -polar- coordinates are normally measured in degrees, but you can inhibit degrees and use radians instead:

```
inhibit degree $$ use radians in -circle-, -rotate-, and -polar-
allow degree $$ use degrees in -circle-, -rotate-, and -polar- (default)
```

Using radians in these commands is often convenient for consistency with the radians that are always used in the trigonometric functions (sine, cosine, arctan, etc.). The specification of inhibit/allow degree is not reset by a new main unit. For example, a single -inhibit degree- in the initial entry unit (IEU) is sufficient to turn off the use of degrees throughout the entire program.

Example:

```
unit xdegrees
* Draw a pie-shaped figure.
   f: ANGLE=0.7 $$ angle in radians
inhibit degree
draw 50,50; 150,50
```
atnm 50,50
circle 100, 0, ANGLE $$ angle in radians
draw 50+100\cos(ANGLE),50+100\sin(ANGLE); 50,50
*

See Also:
Trigonometric Functions (p. 202)

**inhibit/allow fuzzyeq**

In making logical comparisons such as "if x > y" involving floating-point numbers, cT normally uses a "fuzzy zero" so that tiny differences due to the accumulation of roundoff errors are considered to be equal to zero. You can inhibit this "fuzzy zero" or "fuzzy equal":

<table>
<thead>
<tr>
<th>inhibit</th>
<th>fuzzyeq</th>
<th>$$ do not use fuzzy zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow</td>
<td>degree</td>
<td>$$ use fuzzy zero (default)</td>
</tr>
</tbody>
</table>

This applies to all types of comparisons (=, !=, >, >=, <, and <=), and in terminating an iterative -loop-. When making logical comparisons with the "fuzzy zero" active, $X = Y$ if $\text{abs}((X-Y)/X) < 10^{-11}$, or if $\text{abs}(X-Y)<10^{-9}$.

Example:

```plaintext
unit xfuzzy
float: x, y
calc x := 3E-10
      y := 5E-10
at 10,10
show y > x $$ FALSE (x-y difference tiny)
inhibit fuzzyeq
at 10,30
show y > x $$ TRUE (fuzzy zero not used)
```

See Also:
Differences from Other Languages (p. 17)

**Color**

**Color Introduction**

Using color in cT to draw a red box or a blue line is very simple. However, understanding and using the full range of color possibilities requires a specialized vocabulary and careful attention to issues that don’t arise on a noncolor computer. This section introduces some vocabulary and concepts that will be useful when reading the command descriptions, including the following issues:

- Foreground, background, and window color
- Defining colors: RGB and HSV
- Palette color (256 colors) and true color (thousands or millions of colors)
- Experimenting with color -- a program that lets you play with colors
Basic Color Usage
To display something in one of the basic colors provided by cT (referred to as zblack, zwhite, zred, zyellow, zgreen, zcyan, zblue, or zmagenta), set the color with a -color- command, then make the display:

<table>
<thead>
<tr>
<th>color attribute</th>
<th>value</th>
<th>color used</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>zred</td>
<td>$set to red</td>
</tr>
<tr>
<td>fill</td>
<td>10,20; 100,50</td>
<td>$this is displayed in red</td>
</tr>
<tr>
<td>vector</td>
<td>110,10; 120,60</td>
<td>$this is also displayed in red</td>
</tr>
<tr>
<td>color</td>
<td>zblue</td>
<td>$change to blue</td>
</tr>
<tr>
<td>box</td>
<td>5,5; 130,70</td>
<td>$this is displayed in blue</td>
</tr>
</tbody>
</table>

Foreground, Background, and Window Color
To use colors effectively in cT it is important to understand the distinctions among "foreground," "background," and "window" colors.

Consider the display of text in -mode rewrite-. The color of the letters is called the foreground color, and the color of the area immediately surrounding the letters is called the background color:

<table>
<thead>
<tr>
<th>color</th>
<th>value</th>
<th>color used</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>zred, zcyan</td>
<td>$foreground red, background cyan (blue-green)</td>
</tr>
<tr>
<td>mode</td>
<td>rewrite</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>Hello</td>
<td>$letters red, background cyan</td>
</tr>
</tbody>
</table>

Similarly, in -mode rewrite- a patterned -fill- displays the pattern in the foreground color and the remaining areas (such as the inside of an "O") in the background color. In -mode inverse- the color roles are reversed. In -mode write- the background color plays no role, since the background areas are not affected. In -mode erase- the letters or patterns are displayed using the background color. In -mode xor- special considerations apply: see the description of the -color- command.

The window color is the color that fills the screen whenever there is a full-screen erase due to a blank-tag -erase-command, starting a new main unit (with -next-, -back-, or -jump-), or reshaping the window. The window color can be thought of as describing the "canvas" on which all further displays are made. Usually you will make the window color and the background color be the same.

A non-full-screen -erase- such as -erase 10,20; 100,150- displays the background color. Such -erase- statements cannot go outside the area defined by a -fine- command (which specifies the limits of coordinates to be used by the program). The blank-tag, full-screen -erase- command erases the entire window, even outside the -fine- area, and it uses the window color, not the background color.

To take a concrete example, suppose the foreground color is red, the background color is blue, the window color is yellow, and there is a -fine- command in effect. A -draw- statement displays in red (the foreground color) on a yellow canvas (the window color). In -mode rewrite- text is displayed red on blue (the background color). The statement -erase 0,0; zxmax,zymax- paints the entire -fine- area blue (leaving the surrounding area of the window yellow), and a blank-tag -erase- makes the entire window yellow.

Defining Colors: RGB and HSV
cT provides two schemes for defining new colors of your own. The RGB color definition scheme specifies a color by giving the percentages of red, green, and blue that make up that color. The HSV scheme specifies a color by giving the hue, saturation, and value (brightness) that make up that color.

At the end of this section is a program that lets you experiment with RGB and HSV descriptions of colors. Playing with the program will make the following discussion much more meaningful.

Our experience of color often starts with painting: red paint plus green paint makes brown paint. Paints and most things around us absorb light and have "subtractive colors." Computer monitors and other light emitters
have "additive colors." With additive colors, red plus green makes yellow, which may seem rather nonintuitive at first.

In the RGB scheme, white is the color with a maximum (100%) of red, blue, and green (100, 100, 100). Black has minimum amounts (0, 0, 0). The brightest red is represented by (100, 0, 0) -- that is, it has 100% red, no green, and no blue.

In the HSV system, the hue is a numbering of colors ordered from red to yellow to green to cyan to blue to magenta back to red again around a 360-degree "color wheel." Note that yellow lies between red and green because it is equal parts of red and green. Similarly, cyan lies between green and blue, and magenta lies between blue and red. Here are the hue values for the basic colors:

<table>
<thead>
<tr>
<th>Color</th>
<th>HSV</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>360 (or 0)</td>
</tr>
<tr>
<td>yellow</td>
<td>60</td>
</tr>
<tr>
<td>green</td>
<td>120</td>
</tr>
<tr>
<td>cyan</td>
<td>180</td>
</tr>
<tr>
<td>blue</td>
<td>240</td>
</tr>
<tr>
<td>magenta</td>
<td>300</td>
</tr>
</tbody>
</table>

Saturation refers to how "brilliant" the color is. A partially unsaturated color is one that has been diluted by adding some white. For example, pink can be made with RGB values of 100, 70, 70, which can be thought of as a pure red of 30, 0, 0 plus a lot of white made with RGB values of 70, 70, 70. We say that this pink is only 30% saturated. An RGB value of 25, 100, 100 represents a partially unsaturated cyan which can be thought of as the sum of a pure cyan (0, 75, 75) with some white (25, 25, 25). We say that the saturation is 75%.

The "value" or brightness is an overall intensity measure. A half-intensity yellow with RGB values 50, 50, 0 has a value of 50 in the HSV scheme.

Here are some colors expressed in the two different systems:

<table>
<thead>
<tr>
<th></th>
<th>RGB</th>
<th>HSV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>100,00,00</td>
<td>360,100,100</td>
</tr>
<tr>
<td>Pink</td>
<td>100,70,70</td>
<td>360,30,100</td>
</tr>
<tr>
<td>Rose</td>
<td>100,40,70</td>
<td>330,35,100</td>
</tr>
<tr>
<td>White</td>
<td>100,100,100</td>
<td>xxx,00,100</td>
</tr>
<tr>
<td>Black</td>
<td>00,00,00</td>
<td>xxx,xxx,00</td>
</tr>
</tbody>
</table>

In HSV, if saturation is 0 and value is 100, the color is white no matter what hue is chosen. If the value is 0 (no brightness), the color is black. Intermediate values give a range of grays.

**Gray Scale Values**

Some computer systems lack color but do provide a range of grays. With such systems cT automatically calculates a gray intensity in a manner comparable to the way a black-and-white television displays a color television program. If an RGB value is "red,green,blue" then the calculated intensity is

\[
\text{intensity} = 0.30 \text{red} + 0.59 \text{green} + 0.11 \text{blue}
\]

Note that the intensity for white is \(0.30 \times 100 + 0.59 \times 100 + 0.11 \times 100 = 30 + 59 + 11 = 100\). The intensity for black is 0.

**Color Mechanisms: Palette Color and True Color**

There are two quite different kinds of color mechanisms used in computers, "palette" color and "true" color. The older "palette" color mechanism permits displays that use a limited number of different colors, most often 256
different colors. This is rather like a “paint by numbers” scheme. You can define what each of these 256 colors looks like, but you can't display more than 256 different colors on the screen at the same time. If the display you want to make requires more than 256 colors, you have to compromise by using some colors that look as much as possible like the desired colors.

"True" color, on the other hand, hardly restricts what you can display, because it permits displaying many thousands or even millions of different colors at the same time. This can be particularly important for high-quality color images or digital movies. With the growing importance of multimedia presentations, true color is becoming increasingly popular.

**Palette Color**

On a computer with "palette" color you typically can use 256 different colors at the same time. When you say -color zred- you are really choosing the "palette slot number" that corresponds to a color that has been predefined to be red. cT predefines a palette with slot numbers 0 through 7: zblack (0), zwhite (1), zred (2), zgreen (3), zblue (4), zyellow (5), zcyan (6), and zmagenta (7). So -color zred- is actually the same as -color 2-.

The -palette-command lets you modify this predefined palette or create a completely new palette, using the RGB scheme to specify the desired colors. If you want to define the colors using the HSV scheme, use the -getrgb-command to translate from HSV to RGB. You don't need a -palette- command if the standard "system" palette is adequate. The "system" palette includes the 8 basic cT colors but may provide as many as 256 colors total.

Included among the set of cT sample programs is palette.t, a -use- file that gives names to a set of useful additional colors beyond the basic eight cT colors, including light, regular, and dark versions of slate, teal, coral, gray, gold, lavender, cerise, dark red, and dark green. Also useful on palette machines is the sample program setcolor.t, a -use- file that lets you experiment with the color of an object, in the full context of your running program.

A complication is that the color palette is shared among all the applications that you have running at any given moment. If the active application (the one whose window is most forward) sets up a palette with a shade of red in every palette slot, the sky in a color image in some other window will be red! For that reason, cT offers tools for finding out what colors have already been defined at some instant and which might be acceptable for your purposes.

A special form of the -palette- command lets you extract a palette from an existing color image, so that you can then display that image using a set of colors that is appropriate for that particular image.

**True Color**

It is more and more common for computers to provide "true color," in which there is no palette. This is sometimes referred to as offering "thousands of colors" or "millions of colors." Colors are specified directly in terms of their RGB or HSV values, unrestricted by the limitation to only 256 colors. cT supports true color, and there are versions of the color commands that let you set a true color in terms of RGB or HSV specifications. For convenience, palette commands work even on a true-color machine, which provides an easy way to refer repeatedly to colors you have designed, by referring to a slot number.

**Experimenting with RGB and HSV color**

Copy the following rather long program into the beginning of your program window and choose "Run from beginning" from the Option menu. Observe the effects of increasing or decreasing the amount of red, green, or blue in the color, or the settings for hue, saturation, and value.

```
define      group,color:
  i: Slot     $$ palette slot of varying color
  i: TrueColor $$ TRUE if true color, FALSE if palette machine
  slider: S(6) $$ six sliders for adjusting RGB and HSV color
```
This section requires a color display.

This monitor has color, but cT cannot modify the palette.

Use the mouse to adjust the red, green, and blue content of the colored area. Or adjust the hue, saturation, and value (brightness).

Note that reducing the saturation "dilutes" the color by mixing in some white (which is an equal mixture of red, green, and blue).
do  InitSlider(1,60,35,100,"R")
text  0.43;55,300
\text  0,124;55,300
\text  0,124;55,300
\do  InitSlider(2,80,35,0,"G")
do  InitSlider(3,100,35,0,"B")
gethsv  100, 0, 0; hue, saturation, value
\text  0,184+13(nn-1); 55,300
endloop
\do  InitSlider(4,60,175,hue,"H")
loop  nn := 1, 7
calc  \nn-2 \color:="red" \color:="magenta" \color:="blue"
\color:="cyan" \color:="green" \color:="yellow"
\color:="red"
text  0,184+13(nn-1); 55,300
endloop
\do  InitSlider(5,167,175,saturation,"S")
text  0,184; 162,300
\text  0,223; 162,300
\text  0,223; 162,300
\text  0,265; 162,300
\text  0,265; 162,300
\do  InitSlider(6,275,175,value,"V")
text  0.184; 270,300
\text  0.265; 270,300
\text  0.265; 270,300
\if  TrueColor
\text  color rgb, zvalue(S(1)),zvalue(S(2)),zvalue(S(3))
\else
\text  * Exploit modifiable slot on palette machine:
\text  rgb Slot, zvalue(S(1)),zvalue(S(2)),zvalue(S(3))
\text  color Slot
\endif
\do  ShowNewColor  $$ display initial color (red)
color  zwhite
\*  unit  UpdatePalette(id)  $$ update all color sliders
merge,color:
  i: id
  f: red,green,blue,hue,saturation,value
if  id <= 3  $$ user changed RGB, must adjust HSV sliders
if  TrueColor
  \text  color rgb, zvalue(S(1)),zvalue(S(2)),zvalue(S(3))
  \do  ShowNewColor
else
$$ -rgb- command instantly changes color on screen:
rgb Slot, zvalue(S(1)),zvalue(S(2)),zvalue(S(3))
endif
gethsv zvalue(S(1)),zvalue(S(2)),zvalue(S(3)); hue, saturation, value
slider reset, S(4); value, hue/3.6
slider reset, S(5); value, saturation
slider reset, S(6); value, value
else
$$ user changed HSV, must adjust RGB sliders
if TrueColor
color hsv, 3.6zvalue(S(4)), zvalue(S(5)), zvalue(S(6))
do ShowNewColor
else
hsv Slot, 3.6zvalue(S(4)), zvalue(S(5)), zvalue(S(6))
endif
getrgb 3.6 zvalue(S(4)), zvalue(S(5)), zvalue(S(6)); red, green, blue
slider reset, S(1); value, red
slider reset, S(2); value, green
slider reset, S(3); value, blue
endif
*
unit InitSlider(id, xx, yy, value, label) $$ create a color slider
merge,color:
i: id, xx, yy
f: value
m: label
rorigin xx, yy
rtext 0, -16; 10, 0
<|s,label|>
\
rslider S(id);0,0; 10,110; UpdatePalette(id); value, value
*
unit ShowNewColor
fill 122,20;285,145
*

color: Color Graphics & Text

The -color- command specifies the color used by subsequent text and graphics commands. Once a -color-
command is issued, the same color is used until a new -color- command is issued. Up to 256 different colors can
be displayed on a palette-based computer, or thousands or millions of different colors on a true-color computer.
The first part of the tag gives the color in which displays are drawn, the "foreground color." The second part
gives the "background" color, which is discussed below.

Simple form:

  color foreground color slot $$ choose from palette; background unchanged
  color foreground color slot, background color slot $$ choose from palette
  color , background color slot $$ choose from palette, foreground unchanged
  color zred $$ or zblack, zwhite, zyellow, zgreen, zcyan, zblue, or zmagenta

On a true-color system (which does not have a palette) you can specify the color directly, and the same
commands on a palette-based system search for a closest match across the entire current palette:
color rgb, rr, gg, bb $$ specify true-color rgb values, or search for best match
color hsv, hh, ss, vv $$ specify true-color hsv values, or search for best match
color rgb, rr, gg, bb; hsv, hh, ss, vv $$ foreground and background
color ; hsv, hh, ss, vv $$ background only

"Palette slot numbers" are in the range 0 to 255, even on true-color machines. Each palette slot describes a color. For convenience, a palette of 8 slots (numbers 0 through 7) is predefined and each slot is given a name. In order, they are: zblack, zwwhite, zred, zgreen, zblue, zyellow, zcyan (blue-green), zmagenta. Using additional colors, or changing the predefined colors, requires a -palette- command.

For most displays, only the foreground color is used. In -mode rewrite- and -mode inverse-, displaying text or filling an area with a pattern not only creates a "foreground" display but also changes the "background." The -erase- command and -mode erase- replace the erased display with the background color. (But note that the blank-tag, full-screen version of the -erase- command fills the entire window with the color set by the -wcolor- command, and this color can be different from the background color.)

The system variables zcolorf and zcolorb are the current foreground and background colors. The system variables zdefaultf and zdefaultb give the system default foreground and background colors.

If you want a normal display to show up against a normal background, it is better to use -color zdefaultf- than to use -color zblack- or -color zwwhite-. While all currently supported computers have normally white windows and zdefaultf represents black, it is possible that some future computers might have normally black windows and zdefaultf would represent white. If you were to use -color zblack- with a default background, nothing would appear on such computers!

Colored areas are opaque, even with light colors such as yellow and cyan. To make text or a drawing on a colored field, fill the area first, then make the drawing. Note that -fill- with a -pattern- only modifies some of the pixels, so a patterned fill may appear translucent (unless one uses -mode rewrite- or -mode inverse-). Similarly, an icon drawn with -plot- only modifies some pixels and may appear translucent.

The actual color of an object displayed in -mode xor- is undefined. Usually with -mode xor- the color of an object is unpredictable and completely independent of your foreground and background colors, even when drawing on a region filled with the background color. But there is reversibility: Doing two -mode xor- displays in the same color at the same place always restores the screen to its original appearance, as long as there isn't an intervening -palette- command.

Examples:

unit xRainbow1 $$ make a simple rainbow
i: index
clip 0,0; 350,150 $$ show only part of -disk-s
at 160,180
loop index := 1,6
  color \index\zblue\zcyan\zgreen\zyellow\zred\zwhite
  disk 150-10index
COLOR

endloop
clip
  *
  unit xBackground$$ illustrate background color
  color zblue $$ blue foreground, default background
  fill 50,50; 150,150
  erase 75,75; 125,125
  color , zyellow $$ foreground unchanged, yellow background
  fill 200,50; 300,150
  erase 225,75; 275,125
  *
  unit xBackground2 $$ illustrate -mode rewrite-
  color zred, zcyan $$ blue foreground, yellow background
  at 50,50
  write This text is in red.
    It does not affect the background.
  mode rewrite
  at 50,90
  write Text in -mode rewrite-
    changes the background.
  *
  unit xOpaque $$ illustrate opaqueness of colors
  at 120,40
  write Hello there
  at 120,150
  write Hello again!
  pause
  color zcyan
  fill 160,30; 250,70 $$ blocks out the text
  pattern zpatterns,2 $$ this is a "sparse" pattern
  fill 160,140; 250,180 $$ text peeks through pattern

See Also:
  Color Introduction (p. 71)
  Color Status (p. 326)
  zrgbn & zhsvn Finding a "Closest" Color (p. 88)

wcolor: The Window Color

The -wcolor- command specifies the basic color of the entire window but does not update the display immediately. The next time the window is cleared, it is restored to the color specified by -wcolor-. The tag of -wcolor- is a palette slot number (refer to the -color- description) or a true-color description.

    wcolor zblue $$ full-screen erase will turn entire window blue

On a true-color system (which does not have a palette) you can specify the color directly, and these same commands on a palette-based system search for a closest match across the entire current palette:

    wcolor hsv, 50, 50, 90 $$ RGB for true-color systems; or best match
    wcolor rgb, 80, 30, 40 $$ HSV for true-color systems; or best match

The display is updated by moving to a new main unit with -next-, -back-, or -jump-; by executing a full-screen -erase-; or by reshaping the window. To update the window color immediately, use a blank-tag -erase-.
It is important to understand the relationship of -wcolor-, -erase-, and -fine-. The -fine- command defines the region of the window that is the active display. The only command that is allowed to affect the area outside the active display is the blank-tag -erase- command. It sets the entire window to the color specified by -wcolor-. Other -erase- commands use the current background color and stay within the active display area.

To set your program to display in green on a black background, include these commands in the IEU (initial entry unit -- the statements preceding the first -unit- command):

```
  wcolor zgreen
  color zblack, zgreen
  erase
```

If there are no explicit commands to set the foreground, background, and window colors, the program behaves as if these commands were included in the IEU:

```
  wcolor zdefaultb
  color zdefaultf, zdefaultb
  erase
```

The palette slot number of the current window color is available in zwcolor.

Example:

These examples illustrate the use of a window color that is different from the background color. (Usually you would put the -fine- and color-setting commands in the IEU.)

```
  unit Xwcolor1
    fine 200,150 $$ limit display area to a small region
    color zblue,zyellow $$ foreground and background
    wcolor zred $$ declare full-window color
    erase $$ activate full-window color
    erase 0,0; zxmax,zymax $$ erase active display area at 50,50
    write HELLO
  *
  unit Xwcolor2
    fine 200,150 $$ limit display area to a small region
    color zblue,zyellow $$ foreground and background
    wcolor zred $$ declare full-window color
    erase $$ activate full-window color
    box $$ frame around active display area
    mode rewrite
    text 14,33; 177,124
  See how "rewrite" mode makes the background turn yellow?
  *
```

See Also:
- Color Introduction (p. 71)
- color Color Graphics & Text (p. 77)
- Color Status (p. 326)
palette: Creating a Color Palette

The -palette- command is used to reinstall the standard "system" palette provided on the particular computer system, or to extract a palette from a color image contained in a screen variable, or to modify or add to the current palette of colors:

```plaintext
palette        $$ blank tag: reinstall system palette (like obsolete -newpal-)
palette "ImageFile"     $$ extract palette from image file
palette 8, 80, 74, 20          $$ slot 8 has RGB values 80, 74, 20
         9, 30, 25, 8, 4 $$ slot 9 will fallback to slot 8 or 4
         11, 25, 75, 25, 0, 0, TRUE $ modifiable palette slot
palette slotarray, Rarray, Garray, Barray  $$ array form
```

**Reinstalling the system palette**
The blank-tag -palette- command resets the palette to contain the colors normally provided by the particular computer system. Whenever cT starts executing a program, the system palette is installed. The system palette offers a range of colors that is adequate for many purposes. The system palette includes the eight basic cT colors (black, white, red, yellow, green, cyan, blue, and magenta).

The obsolete -newpal- command had the effect of reinstalling the system palette before modifying the palette with new definitions. It is recommended that any -newpal- commands be replaced.

**Extracting a palette from a color image**
Although the system palette is adequate for many purposes, a particular color image may require a set of colors different from those in the system palette in order to display well. The second form of the -palette- command extracts color definitions from an image file (a PICT file on a Macintosh or a *.*BMP file on Windows).

If the new palette is successfully installed, zreturn is TRUE after executing the -palette- command, and zretinf gives the number of palette slots that were found. If not, zreturn gives a file error indication. For example, a value of 4 means "no such file," and a value of 5 means "improper file type."

*Priorities in installing the palette:* In building the palette, cT refuses to modify the colors black and white, which must always be available. cT tries to avoid modifying any colors used by the computer system to display standard controls such as sliders and window title bars, but if necessary will change these colors, which can lead to peculiar-looking sliders or title bars. How many such privileged colors there are can be obtained with the -sysinfo- command (-sysinfo default colors,var-). The last existing colors to be modified are any colors that you have explicitly set with the slot-oriented -palette- commands that are described below ("Designing your own palette of colors").

**Designing your own palette of colors**
The third form of the -palette- command lets you specify a slot number, with the next three arguments specifying the red, green, and blue parameters for that slot number. Once the palette has been established, the statement -color 9- would select the color defined for palette slot 9 to be used in displaying text and graphics.

Following the RGB parameters, one can specify optional fallbacks for foreground and background colors. In the -palette- example above, if there aren't enough different colors available to provide for a slot number 9, this color will "fall back" to slot 8 when used as a foreground color and will fall back to slot 4 when used as a background or window color. If the new palette is successfully installed, zreturn is TRUE after executing the -palette- command. If not, zreturn is a positive number indicating how many slots were created, and the specified fallbacks will be in effect for the other slots.
If your program absolutely needs to have 25 colors, you can check the value of \texttt{zncolors} (discussed separately) at the start of the program and refuse to proceed if there are fewer than 25 available slots. However, you might choose to specify a second-choice color for slots that are unavailable. The optional fallback arguments specify what palette slots should be used if the requested slots are not available.

\begin{verbatim}
palette 12,80,73,20, zdefaultf, zblue  $$\text{fall back to zdefaultf, zblue}
25,320,80,25,12,8  $$\text{fall back to slots 12, 8}
\end{verbatim}

In the example above, if palette slot 25 is unavailable, the color from slot 12 will be used for foreground and slot 8 for background (and window color). If palette slot 12 is unavailable, \texttt{zdefaultf} will be used.

Following the fallback parameters, there is an optional "modify" parameter that is FALSE by default. If this parameter is successfully set to TRUE, one can use -rgb- or -hsv- commands to make instant changes in displays \emph{already on the screen}. Using the example above, the color of any text or graphics that had been drawn on the screen with \texttt{-color 11-} can be instantly changed simply by executing an -rgb- or -hsv- command to alter the color definition for slot number 11.

Unless you really need to perform this special kind of instantaneous color change, do not set the modify parameter to TRUE. For one thing, these special colors are not available at all on true-color computers, which increasingly are becoming standard. Also, on computers that permit multiple programs to run simultaneously, specifying many palette slots to be modifiable locks up these slots in such a way that other programs may be prevented from operating normally. After a -palette- command, \texttt{zreturn} may not be TRUE if there aren't enough modifiable slots available, even if the total number of slots is within bounds. This can happen either because the computer system does not permit such color manipulations (this includes true-color systems) or because other simultaneously running programs have reserved many modifiable palette slots for their own purposes.

This doesn't mean that you can't modify your palette. The following command will modify two slots of the current palette and install the new, modified palette:

\begin{verbatim}
palette 5, 20, 30, 40
6, 40, 60, 80
\end{verbatim}

Installing a modified palette can be a rather slow process on some palette-based machines, so ideally one should use the command sparingly. Also note that the following two -palette- commands take longer than the previous example:

\begin{verbatim}
palette 5, 20, 30, 40
...
palette 6, 40, 60, 80
\end{verbatim}

These two -palette- commands involve modifying and installing the modified palette \emph{twice}. The previous form, in which succeeding command names are omitted, makes up the whole modified palette and then installs it, so there is only \emph{one} installation of a modified palette, not two.

The array form of the -palette- command accepts whole arrays as the arguments for slot numbers, red, green, blue, foreground fallback, background fallback, and modify parameter. An example is given below, in the program examples. In the array of slot numbers, a slot number greater than 255 is taken to mean "ignore this entry in the arrays."

At the start of a program, cT installs the system palette and orders the colors in such a way that the first eight colors are the basic cT colors, with appropriate fall-back options:
If only four colors are available (black, white, red, and green) blue will be shown as green, yellow as red, cyan as blue (which falls back to green), and magenta as red.

Included among the set of cT sample programs is `palette.t`, a `use`-file that defines names for a set of useful additional colors beyond the basic eight cT colors, including light, regular, and dark versions of slate, teal, coral, gray, gold, lavender, cerise, dark red, and dark green. Also useful on palette machines is the sample program `setcolor.t`, a `use`-file that lets you experiment with the color of an object, in the full context of your running program.

Some computers let you have several different programs visible simultaneously. Usually these computers have the color palette of just one of these programs in effect at a time. As a result, changing the palette may instantaneously change the colors everywhere! When you quit running a cT program, cT restores the system palette.

Some computers offer "true color," in which there is no palette. Colors are specified directly in terms of their RGB or HSV values. cT supports true color: there are versions of the color commands that let you set a true color in terms of RGB or HSV specifications. For convenience, palette commands work even on a true-color machine, which provides an easy way to refer repeatedly to colors you have designed.

**Examples:**

```
unit xpalette
palette zyellow,100, 50, 50  $$ change yellow to pink
if ~zreturn
   at 10,10
   write Cannot make pink.
else
   color zred
   fill 10,10; 160,160
   color zyellow
   fill 30,30; 140,140  $$ pink, not yellow
endif
pause
```

In the following example using the array form of the `-palette-` command, we need `-getrgb-` commands to convert from HSV (hue-saturation-value) to RGB (red-green-blue) because the `-palette-` command only recognizes RGB descriptions of colors:

```
unit xRainbow2  $$ a more realistic rainbow
* Note: this example uses palette slots 9 through 39.
i: radius, hue, paletteN
i: SKY=9, MAXC=39, NC=MAXC-SKY+1
i: slot(NC), r(NC), g(NC), b(NC)
i: S=70  $$ for dimmer rainbow, S=30
getrgb 200,20,100; r(1),g(1),b(1)  $$ rainy-day sky
```
calc  slot(1) := SKY
     paletteN := 2
loop  hue := 0.290, 10                          $$ go from red through indigo
     getrgb hue,S,100; r(paletteN),g(paletteN),b(paletteN)
     calc  slot(paletteN) := paletteN+SKY-1
           paletteN := paletteN + 1
endloop
palette  slot,r,g,b                          $$ add to basic cT palette
color    SKY
fill     10,40; 310,200                      $$ sky background
clip     0,0; 500,150
calc     radius := 100
loop     paletteN := SKY+1, MAXC
     color paletteN
     atnm 160,210
     circle radius
     calc radius := radius +1
endloop
pause
*

See Also:
Color Introduction (p. 71)
Color Status (p. 326)
sysinfo Get System Information (p. 319)
zreturn The Status Variable (p. 332)

rgb & hsv: Modifying a Palette Slot

The -rgb- and -hsv- commands are used to change modifiable "palette slots" for use by the -color- command. These commands are used only for very special effects, where something already on the screen must change color rapidly (for example, a large area must change color instantaneously as a slider is adjusted). For ordinary color definitions, use -palette- commands. Also note that the -rgb- and -hsv- commands cannot produce these special effects on true-color computers. Since true color is increasingly becoming the standard, the -rgb- and -hsv- commands have limited usefulness.

Each palette slot holds the "rule" for making one color. The -rgb- command specifies the color associated with a palette slot by giving the percentages of red, green, and blue that make up that color. The -hsv- command specifies the color associated with a palette slot in terms of the hue, saturation, and value. To use these special commands, you must first specify that the palette slot is modifiable:

```
palette 11, 25, 75, 25, 0, 0, TRUE  $$ make palette slot modifiable
...rgb 11, %red, %green, %blue, f fallback, b fallback
hsv 11, hue, saturation, value, f fallback, b fallback
rgb N $$ restore slot N to original palette setting
hsv N $$ restore all slots to original palette settings
```
The first argument of the tag is a palette slot number, an integer between 0 and 255. Slots 0 through 7 are predefined (zblack, zwhite, zred, zgreen, zblue, zyellow, zcyan, and zmagenta) to not be modifiable, although you change these basic palette entries with a -palette- command.

The next three arguments give the percentages of red, green, and blue (RGB), or the hue, saturation, and value (HSV). The percentages, saturation, and value range from 0 to 100. The hue indicates the color on a "color wheel" and ranges from 0 to 360.

You can give optional foreground and background fallback options, which modify the original palette specifications.

When only the slot number is given in the tag, that slot is restored to the color value (and fallback options, if any) set by the original -palette- command. When the tag is blank, all of the slots revert to the original palette settings, which may be a slow process.

It is not possible to use -rgb- and -hsv- commands to change the color definitions for a palette slot that was not successfully specified to be modifiable. If the color cannot be changed by the -rgb- or -hsv- command, zreturn will be set to FALSE.

The -rgb- and -hsv- commands reflect different ways of thinking about how color is produced. The red-green-blue (RGB) system specifies how much of each color is used. The greater the percentage, the more intense the color. In the hue-saturation-value system (HSV), the hue specifies the color, saturation gives the "brilliance," and value gives the brightness.

It is important to understand a major difference between the -rgb- or -hsv- commands and the -color- command: Executing a -color- command chooses a palette slot to be used by the following display commands, but does not affect what is already on the screen. Executing an -rgb- or -hsv- command on a palette-based computer instantly changes the color of anything already on the screen that had been displayed using that palette slot, but does not change which palette slot is used by the following display commands. (These commands do not affect the screen display on a true-color computer.)

If a region contains dynamically modifiable colors (those controlled by -rgb- or -hsv- commands), on some computers a -get- followed by a -put- will not restore the original colors.

NOTE: While testing sample units that use -rgb- or -hsv- commands, you must usually use either "Run from selected unit" or put a -pause- at the end of the unit. "Execute selected unit" or "Execute current unit" switches back to the computer's default palette as soon as execution is finished.

Examples:

```
unit xrgb
i: PS = 10
palette PS, 0, 0, 100, 0, 0, TRUE $ no red, no green, 100% blue
color PS $ choose color
fill 100,50; 200,150
vector 200,150; 50,200
pause $ wait for keypress
rgb PS, 80, 0, 80 $ 80% red, no green, 80% blue
pause
```

See Also:
- Color Introduction (p. 71)
- palette Creating a Color Palette (p. 81)
- Color Status (p. 326)
**zncolors: The Number of Available Colors**

The system variable `zncolors` gives the number of palette slots provided by the system. (On a true-color system `zncolors` gives a very large number.)

Color systems with palettes typically offer 16 or 256 palette slots. The number of colors available is reported in `zncolors`. On a monochrome system (e.g., black and white, or green and black) where there is no possibility of color changes, `zncolors` is zero.

In some cases there may be many palette slots available, but with unchangeable colors. The system variable `zreturn` reports FALSE (0) if an `-rgb-` or `-hsv-` command attempts to specify a color for an unchangeable slot. If the slot is successfully changed, `zreturn` is TRUE (-1). The `-sysinfo-` command provides additional details about the availability of colors.

**Example:**

```
unit Xzncolors
  i: index
at 15,25
write This system has <s,zncolors> palette slots available.
palette 4,50,50,0,0,TRUE  $$ make slots 4 and 5 modifiable
  5,75,75,75,0,0,TRUE
at 15,75
write List in bold slots which cannot be changed:
loop index := 0, 7
  rgb index,0,100,100 $$ make some change
  if ~zreturn
    write <s,index>
  else
    write <s,index>
  endif
  rgb index $$ change back to default colors
endloop
write done
```

**See Also:**
- Color Introduction (p. 71)
- palette Creating a Color Palette (p. 81)
- Color Status (p. 326)
- sysinfo Get System Information (p. 319)

**getrgb & gethsv: Find RGB and HSV Values**

The `-getrgb-` and `-gethsv-` commands are used to convert between RGB and HSV specifications for a color:

- `getrgb` 210, 43, 95; myred, mygreen, myblue  $$ HSV to RGB
- `gethsv` 100, 100, 0; myhue, mysat, myvalue  $$ RGB to HSV
The syntax of these commands is similar to that used for passing arguments to a subroutine. The first three arguments are the HSV or RGB values to convert, and the next three user-defined variables receive the converted values.

Another form of the -getrgb- and -gethsv- commands can be used to request information about an existing palette slot and receive back information about the foreground fallback slot and the color values. The commands return the current RGB or HSV values of a palette slot. If the palette slot number is greater than or equal to \texttt{zncolors}, the foreground fallback slot number is returned.

\begin{verbatim}
getrgb  N; ActiveSlot, myred, mygreen, myblue
gethsv  N; ActiveSlot, myhue, mysat, myvalue
\end{verbatim}

The slot number \texttt{N} is specified "by value," and the active slot number and RGB or HSV parameters are returned as though they had been passed "by address." The slot number, \texttt{N}, may be a user-defined variable, an integer, or a system-defined color such as \texttt{zgreen}. The other arguments are user-defined variables.

Usually \texttt{ActiveSlot} is equal to \texttt{N}. However, if \texttt{N} is greater than or equal to \texttt{zncolors}, \texttt{ActiveSlot} is the foreground fallback slot corresponding to slot \texttt{N}.

The -getrgb- command returns the RGB values of \texttt{ActiveSlot} in the variables \texttt{myred}, \texttt{mygreen}, and \texttt{myblue}. Similarly, the -gethsv- command returns in your defined variables \texttt{myhue}, \texttt{mysat}, and \texttt{myvalue} the actual HSV values used when plotting. If there has been a fallback from slot \texttt{N}, so that \texttt{ActiveSlot} is not the same as \texttt{N}, the original color values assigned to slot \texttt{N} cannot be retrieved.

\textbf{Examples:}

\begin{verbatim}
unit Xgetrgb1
  i: hue, sat, value
gethsv 90, 60, 40; hue, sat, value
at 10,20
write RGB 90, 40, 40 is equivalent to
    HSV <\texttt{s},<\texttt{s},hue|>, <\texttt{s},sat|>, <\texttt{s},value|>

unit Xgetrgb2
  i: Active, r, g, b
getrgb zmagenta; Active, r, g, b
at 50,50
write zmagenta is in slot #<\texttt{s},Active|>.
    Its RGB values are <\texttt{s},r|>, <\texttt{s},g|>, <\texttt{s},b|>.
palette 50, 70, 70, 45, 4
getrgb 50; Active, r, g, b
at 50,100
write Slot 50 is plotted as slot #<\texttt{s},Active|>.
    Its RGB values are <\texttt{s},r|>, <\texttt{s},g|>, <\texttt{s},b|>.
color zmagenta
fill 52,148;112,209
color 50
fill 160,148;220,209
pause
*\end{verbatim}

Some computer systems provide a way to specify the number of colors (for example, the Monitor section of the Macintosh Control Panel). You might want to change the number of colors when trying this example.
zrgbn & zhsvn: Finding a "Closest" Color

The system functions **zrgbn(red, green, blue)** and **zhsvn(hue, saturation, value)** give the palette slot number in the current palette whose RGB (or HSV) values most closely approximate the specified values. These functions are useful when you don't want to disturb colors that have already been assigned to the available palette slots. **zrgbn** or **zhsvn** take an arbitrary color description and find the slot that most nearly matches it. (Note that the -color- command also has an rgb and an hsv version that searches through the current palette.)

The example below chooses random values for red, green, and blue. It uses **zrgbn** to find the palette slot that most nearly matches those values and displays the color of that slot. Then the example assigns those values to slot 15 and displays the color that the randomly chosen values would actually produce.

**Example:**

```
unit Xzrgbn
i: R, G, B, slot, ourslot = 15
i: ActiveSlot, red, green, blue
next Xzrgbn
palette ourslot,100,100,100,0,0,TRUE  $$ modifiable slot
loop
  randu R,100  $$ choose random color values
  randu G,100
  randu B,100
  calc slot := zrgbn(R,G,B)  $$ find closest slot
  outloop slot ~= ourslot  $$ mustn't use same slot
rgb ourslot, R, G,B  $$ set palette slot to random colors
color ourslot  $$ select slot
fill 25,20; 100,70  $$ display the random color
getrgb slot; ActiveSlot, red, green, blue
color zdefaultf
at 125,20
write Random values chosen
  rgb: <s,R>, <s,G>, <s,B>
  color: slot
  fill 25,120; 100,170  $$ display closest color
  color zdefaultf
  at 125,120
write zrgbn found slot #<s,slot>:
  rgb: <s.red>, <s.green>, <s.blue>
```

```
* See Also:
  Color Introduction (p. 71)
  Color Status (p. 326)
```
get and put Portions of Screen

The -get- and -put- commands can be used
to save or restore an image of a rectangular portion of the screen, or
to convert an array of numeric data into a color image and display it.

"Screen" variables must be defined for use with -get- and -put-. The screen variable identifies the saved region:

```
define screen: scr1, myimage, datapic
byte: SomeData(100*50)
...
get scr1; x1,y1; x2,y2 $$ get this portion of the screen
put scr1; x3, y3 $$ put the image at a new location
...
get scr1; {in-line color image}  $$ transfer in-line image into screen var
...
* 24-bit rgb pixel info, 100 by 50 image:
get datapic; rgb; 24, 100, 50, SomeData
* 8-bit pixels; use current palette:
get datapic; palette; 8, 100, 50, SomeData
...
get scr1 $$ release memory used for stored image
```

After a portion of the screen has been saved with -get-, the -put- command places the saved image at a specified location (affected by a -clip- region). There are also -rgget-, -rgput-, and -gput- commands that use relative or graphing coordinates. Note when using -gput- or -rput- that you must specify the upper-left corner of the image, independent of which direction the x- and y-coordinates run.

You can use -get- to save a rectangle where you want to display a temporary message, then use -put- to restore that portion of the display. You can also use repeated -put- commands to move an image across the screen to make an animation.

The system functions `zswidth(screen variable)` and `zsheight(screen variable)` give the width and height of the image currently associated with that screen variable.

These commands can also be used to save or restore images in files: see "Images & Files". You may need to use a -palette- command to extract an appropriate set of colors from the image file, so that the current palette will be set appropriately for use by -put- in displaying the image:

```
palette "ImagA" $$ extract palette from color image
get scr1; "ImagA" $$ get color image from file (or in-line image)
put scr1; x1,y1 $$ display image using current palette
```

Multimedia issues: Note that in a multimedia program that utilizes many color images, some planning is required to ensure that images and other graphics display with appropriate colors. For example, if on a 256-color palette-based computer you try to display simultaneously one image that requires 250 blues and green, and another image that requires 250 reds and browns, you’ve got a problem! About the only thing you can do in that case is to modify the images to use fewer colors, so that one palette can span both images as well as possible. A related issue is that digital movies may install their own customized palettes and affect the display of other images.

Numeric data from an array: Here are more details on converting numeric data into a color image for visualization purposes, using the "palette" version of the -get- command:
define byte: array(picturesize) $$ or can be integer array
get screen variable; palette; pixel size, width, height, array
put screen variable; 10, 20 $$ display the color image

The -get- command transfers the array of data to an off-screen image, treating each element in the array as an index in the current color palette (hence the "palette" keyword in the command). At present, "pixel size" must be 8 for the palette option, signifying 8 bits per pixel.

An "rgb" form of this -get- command is also available:

define byte: array(picturesize, 3) $$ or can be integer array
get screen variable; rgb; pixel size, width, height, array
put screen variable; 10, 20 $$ display the color image

This -get- command transfers the array of data to an off-screen image, treating each three consecutive elements in the array as a red-green-blue triplet. At present, "pixel size" must be 24 signifying 24 bits per pixel, 8 bits for each of red-green-blue.

Mode does not apply: Note that "mode" commands do not apply to color pictures displayed with -put-. The picture completely replaces the specified region of the screen, as though you were using mode rewrite.

Use with modifiable colors: If any of the saved region of the screen contains dynamically modifiable colors (those controlled by -rgb- or -hsv- commands), on some computers a -get- followed by a -put- will not restore the original colors.

After a -get- command, zreturn is set as follows:

-1 TRUE, all ok
1 not enough memory to get the image (subsequent -put- does nothing)

Like file variables, screen variables can be passed by address to subroutines. Two screen variables can be compared with "=" or "~=".

Examples:

unit xget
screen: boxtext, ring, save
i: x, y
color zred
at 10,20
write Here is some text.
calc x := zwherex
box 10,20; x,35
get boxtext; 10,20; 120,35 $$ save the text
put boxtext; 70, 60 $$ place additional copies of text
put boxtext; 130,100
color zblue
box 0,70; 60,130; -3
color zgreen
at 30,100
disk 25
mode erase
disk 15 $$ make a hole in the disk to make a ring
mode write
get ring; 0,70; 60,130 $$ save the ring
erase 0,70; 60,130 $$ erase the ring
put ring; 210,20 $$ show ring
put ring; 160,170 $$ show another ring
*
unit xgetarray
screen: sc
byte: data8(16*16), data24(16*16,3)
i: index
* 16 by 16 palettized image, all red:
loop index := 1,16*16
calc data8(index) := zred
endloop
get sc; palette; 8, 16, 16, data8
put sc; 10, 10 $$ displays 16 by 16 red block at 10,10
*
* 16 by 16 rgb image, all blue (red=0, green=0, blue=maximum=255):
loop index := 1,16*16
set data24(index,1) := 0, 0, 255
endloop
get sc; rgb; 24, 16, 16, data24
put sc; 10, 30 $$ displays 16 by 16 blue block at 10,30
*

See Also:
Images & Files (p. 91)
move Making Animations (p. 98)
Animations with get and put (p. 100)

Images & Files

The -get- and -put- commands can be used to save or restore color images in files:

define screen: scr1, myimage
...
get scr1; x1,y1; x2,y2 $$ get this portion of the screen
put scr1; "Display1" $$ store image in file named "Display1"
put scr1; "image.ppm" $$ store image in file in universal PPM format
...
get myimage; "CarPic" $$ get color image from file "CarPic"
put myimage; x4,y4 $$ put color image on the screen
...
get scr1 $$ release memory used for stored image

There are also -rget-, -gget-, -rput-, and -gput- commands that use relative or graphing coordinates. Note when using -gput- or -rput- that you must specify the upper-left corner of the image, independent of which direction the x- and y-coordinates run.

After a successful -put- to a file, the system variable zreturn is -1 (TRUE); otherwise the value indicates a file error (see "File I/O Errors").

When using -put- to store an image in a file, by default the image is in PPM format (Portable Pix Map) on Unix, or PICT format on a Macintosh, or BMP format on Windows. To save an image in PPM format on a
Macintosh or Windows, use a file name ending in ".ppm", such as "image.ppm". In all of these cases the image is stored in a pixel or "paint" format. On a Macintosh it is possible to use a -pict- command to store an image in an object-oriented "draw" format (see "Printing").

When using -get- to retrieve an image from a file, the image can be in PPM format on any platform, or PICT format on a Macintosh, or BMP format on Windows.

The system functions \texttt{zwidth(screen variable)} and \texttt{zsheight(screen variable)} give the width and height of the image currently associated with that screen variable.

You may need to use a -palette- command to extract an appropriate set of colors from the image file, so that the current palette will be set appropriately for use by -put- in displaying the image:

\begin{verbatim}
  palette "ImagA" $$ extract palette from color image
  get scr1; "ImagA" $$ get color image from file (or in-line image)
  put scr1; x1,y1 $$ display image using current palette
\end{verbatim}

**Multimedia issues:** Note that in a multimedia program that utilizes many color images, some planning is required to ensure that images and other graphics display with appropriate colors. For example, if on a 256-color palette-based computer you try to display simultaneously one image that requires 250 blues and greens, and another image that requires 250 reds and browns, you've got a problem! About the only thing you can do in that case is to modify the images to use fewer colors, so that one palette can span both images as well as possible. A related issue is that digital movies may install their own customized palettes and affect the display of other images.

**Insert File:** You can also use "Insert file" to place a color image into your source program. Or, for images that are not too large, you can use "Paste" to insert a copied color image into your source program.

**Mode does not apply:** Note that "mode" commands do not apply to color pictures displayed with -put-. The picture completely replaces the specified region of the screen, as though you were using mode rewrite.

**Memory required:** Note also that you need quite a bit of memory to work with large color pictures: \texttt{cT} must use at least twice the number of bytes that are in the picture when doing an "Insert file", for example.

**Examples:**

\begin{verbatim}
  unit xgetimage
  screen: image
  file: choose
  marker: filename

  loop
    setfile choose; zempty; ro $$ choose an image file
    zreturn
    outloop zreturn = 18
    if
      at 5,5
      write Need bigger window.
      pause
    else
      jumpout
    endif
  endloop
  get image; zfilepath(choose)+zfilename(choose)
  put image; 10,10
  color zred
\end{verbatim}
fill 12,12; 20,20 $ add red square to image
get image; 10,10; 300,200 $ save a new image
loop
addfile choose; zempty $ choose a file
outloop zreturn
if zreturn = 18
at 5,5
write Need bigger window.
pause
else
jumpout
endif
endloop
calc filename := zfilepath(choose)+zfilename(choose)
delfile choose $ delete the new file
put image; filename $ store image in file
*

See Also:
get and put Portions of Screen (p. 89)
move Making Animations (p. 98)
File I/O Errors (p. 287)
Printing (p. 14)

icons: Selecting an Icon

The -icons- command specifies a set of icons. Icons from this set are displayed using -plot- and -move- commands. Unlike characters associated with the -font- command, there is no automatic rescaling of icons.

icons zicons $ a system icon set
icons "mycars" $ same directory as source
icons "/cmu/cdec/yourid/auto/mycars"

The tag of -icons- gives the name of the icons set. Once an icon set has been specified, it remains active until replaced with a different set of icons. The cT sample program showicon.t displays all the icons and patterns in an icons set.

If the icon set is a system-provided set, or if it resides in the same directory as your source file (your ".t" file), only the file name is required. If you create your own icons set in some other directory, the full path name must be specified.

The name zicons is a system marker variable that is recognized by all machine types.

A "set of icons" is a file of dot patterns. Usually such a set is created and edited using programs supplied with cT, including the Icon Maker program for the Macintosh or icon.t for other platforms.

When cT fetches an icon set, it keeps its own copy for fast plotting. There is a very special command -forget icons,pathname- that makes cT forget about this copy. This command is used by icon editor programs to permit revising the icon set during a session.

There are two other ways to plot icons. You can use the "Icon" item on the "Font" menu to insert icons into a -write- statement: give the name of the icons file, and a list of icon numbers separated by spaces and/or commas. You can also use a -style- command to put icons into a marker variable, and then -show- the marker variable.
An example of showing icons in these ways is provided in the sample programs distributed with cT: japan.t displays Japanese "Kanji" characters, using a set of icons "KANJI18.FCT".

Example:

```
unit xicons
at 50,50
plot zk(F) $$ default icons
icons zicons
at 100,50
plot zk(F)
```

See Also:
Making Icons on Macintosh (p. 95)
Making Icons on PC & Unix (p. 96)
cursor The Mouse Pointer (p. 62)
font Selecting a Typeface (p. 43)
plot Plotting Characters and Icons (p. 94)
move Making Animations (p. 98)
style Assigning Styles to Markers (p. 262)
pattern Making Textured Areas (p. 61)
Generic Font Names (p. 327)

plot: Plotting Characters and Icons

The -plot- command displays icons from the icon set specified by an earlier -icons- command, or a pasted-in image:

```
icons "myicons"
...
plot zk(A), zk(B), var+20
plot (pasted-in image) $$ no -icons- command needed
```

Individual icons within a set are numbered by the icon editor. To display an icon, it is referred to by its numerical character code or by using the \(\text{zk}\) function and the character: \(\text{zk(a)}\). The command above plots the icons named A and B and icon number "var+20" from the current icon set.

Alternatively, you can paste an image into the tag of the -plot- command, which has the advantage that you can see the image in the source program.

There are two other ways to plot icons. You can use the "Icon" item on the "Font" menu to insert icons into a -write- statement: give the name of the icons file, and a list of icon numbers separated by spaces and/or commas. You can also use a -style- command to put icons into a marker variable, and then -show- the marker variable.

The cT sample program showicon.t displays all the icons and patterns in an icons set and shows the numerical code (0 to 127) for each icon. To plot icon number 97, use either -plot 97- or -plot zk(b)- (since the character "b" has numerical code 97).

Example:

This example plots a figure over and over again, diagonally across the screen.
unit xplot
  i: temp
icons zicons
loop temp := 15,300, 15
  at temp, temp/2
  plot zk(F)
endloop
*

See Also:
icons Selecting an Icon (p. 93)
move Making Animations (p. 98)
style Assigning Styles to Markers (p. 262)

Making Icons on Macintosh

The Macintosh application Icon Maker provided with cT is used to create small icons or to translate graphic images made with other applications into icons that can be displayed using the -icons-, -plot-, and -move- commands. It is also used to create cursors and fill patterns for -cursor- and -pattern- commands.

Creating an icons file: Start up Icon Maker. Choose "New Mac" from the File menu. An Untitled window opens up containing a list of icons. The icon list starts with numbers 0 through 31, then proceeds through the ASCII character set of letters, digits, and punctuation marks. To display an icon stored in slot 3 near the beginning of the list, use the statement -plot 3- in your cT program. To display an icon stored in slot "R", use either -plot zk(R)- or -plot 82- (82 is the standard ASCII code for the letter R).

Creating an icon: Double-click a slot in the icon list. The "cT icon" window opens up, in which you can draw and erase with the mouse. Click or drag to draw the icon. The black squares represent individual pixels. If you start from a black square, clicking or dragging erases the pixels.

Specifying an origin: To specify an origin (which will be the positioning reference in your cT program), click the Origin box, then click at the desired location within the icon. The gray square shows the location specified for the origin.

Storing an icon: To store your icon into the icon list, choose "Save" on the Icon menu (not the File menu), or click the "close" box at the left of the title bar on the cT icon window. Note that the File menu is associated with the icon-list window, while the Icon menu is associated with the cT icon window.

 Cursors and patterns: In order to use an icon as a cursor or a pattern, select an icon by clicking that slot, then choose "Change type" on the Icon menu and specify all the ways you intend to use this icon.

Importing images: You can copy an image from the Scrapbook or other graphics application and paste it into an icon slot in Icon Maker.

Saving an icons file: When you have finished making icons and storing them in the icon list, choose "Save" from the File menu. You will be asked to choose a name for your icons file. If you call the file myicons, you will need a statement of the form -icons "myicons"- in your cT program in order to select these icons for use by -plot- and -move- commands. Similarly, this is the file name you use in -cursor- and -pattern- commands.

At a later time you can revise your icons file by starting up Icon Maker and choosing "Open" from the File menu.

The icons file should normally be stored in the same folder as any programs you write that use it.
You may also find the sample program `showicon.t` useful for inspecting an icons file.

**Manipulating imported images in Icon Maker**

*Opening a Scratch window:* Icon Maker provides "Scratch" windows for manipulating large images: you can cut out pieces of the image to store in icon slots. Choose "New" from the Scratch menu and paste an image from the Scrapbook or other graphic source into the Scratch window. Or choose "Open" from the Scratch menu to choose an existing MacPaint-format or PICT-format file.

*Selecting a portion of the image:* Now you can select a portion of the image in the Scratch window by dragging the mouse over the desired area. The pixels underneath the lines of the selection box are included in the icon. You can repeat this dragging to change the bounding box. When you are satisfied with your selection, choose "Copy" from the Edit menu, then click the window containing the list of icons. Scroll to the desired icon slot, click that slot, and choose "Paste" from the Edit menu to assign the image to that slot. You can repeat the process (selecting a different bounding box) for the current image to create another icon.

*Specifying an origin:* In order to specify an origin different from the upper-left corner of the image, double-click the slot in the icon list to edit the image and specify an origin. For a large icon you may have to use the cT icon scroll bars to move around in the image.

*Transferring icons to other computers:* The "Fdb" options in Icon Maker are used to create or convert between Macintosh icon files and a machine-independent "Fdb" format in order to be able to transfer icons, cursors, and fill patterns among different brands of computers. For example, if you are using an icons file named `myicons` on the Macintosh and you want to move these icons to a different computer, use Icon Maker to convert the file to "Fdb" format, giving it the name `myicons.fdb`. Transfer `myicons.fdb` to the other computer and use its icon conversion program to convert `myicons.fdb` into the form used on that computer.

See Also:
- `icons` Selecting an Icon (p. 93)

**Making Icons on PC & Unix**

The cT program `icon.t` provided with cT is used to create small icons that can be displayed using the `-icons-`, `-plot-`, and `-move-` commands. It is also used to create cursors and fill patterns for `-cursor-` and `-pattern-` commands. Because this program handles binary files, there are different versions for different Unix platforms.

*Creating an icons file:* Run the cT program `icon.t` and choose the option to create a new cT icons file, or to open an existing cT icons file. To display an icon stored in slot 3 near the beginning of the list, use the statement `-plot 3-` in your cT program.

*Creating an icon:* After specifying a file, you will see a numbered array of icons. Click an icon number, which is the number you would use in a `-plot-`, `-move-`, `-cursor-`, or `-pattern-` command. If the icon is empty, you will be asked for the width (x pixels) and height (y pixels). A rectangular area opens up for drawing the icon. Click or drag with the mouse to draw the icon. The little squares represent individual pixels. If you start from an already existing square, clicking or dragging erases the pixels.

*Specifying an origin and spacing:* Click "Options" to change the size of the icon, to specify an origin (which will be the positioning location in your cT program), or to specify the icon spacing (how `zwherex` and `zwherey` change after a `-plot-` command).
Storing an icon: When you have finished making an icon, click "Done." You will be asked whether to save the icons file at this time, which makes it possible to see the new icons in the numbered array of icons. If you don't save the file at this point, you will still have the opportunity to save the file when you are finished editing.

Importing images: Note that icon.t also offers the option to convert an image file into a one-icon cT icons file.

Referring to an icons file: If the file is called myicons.fpc, you will need a statement of the form -icons "myicons"- in your cT program in order to select these icons for use by -plot- and -move- commands. Similarly, this is the file name you use in -cursor- and -pattern- commands.

At a later time you can revise your icons file by starting up icon.t and selecting your existing file.

The icons file should normally be stored in the same folder as any programs you write that use it.

You may also find the sample program showicon.t useful for inspecting an icons file.

Transferring icons to other computers: If you have Macintosh images to transfer to the PC, use the Icon Maker application on the Macintosh to convert the icons to the "FDB" format, which is a machine-independent form. Transfer this converted file to the PC. Run the program icon.t, and choose "Convert." Then click "Convert FDB -> FPC" and choose the file you want to convert. If the file name is myicons.fdb, the converted file will be named myicons.fpc. In your program, use the name myicons in -icons-, -pattern-, and -cursor- commands. Similarly, you can use icon.t to convert a PC icons file to the machine-independent FDB form, which you can transfer to other computers and convert into cT icons.

See Also:
icons Selecting an Icon (p. 93)
Animations

move: Making Animations

The -move- command creates an animated display by moving icons from one place to another. On multi-process computers it can be difficult to get a smooth animation, due to interruptions from other processes. The internal workings of the -move- command (which you don't see) attempt to compensate for the idiosyncrasies of the system, whereas an animation created by separately writing (in mode write) and erasing (in mode erase) may not work very well.

```
icons "myicons" $$ need -icons- command before using -move-
...
move icons: ; to; object
move icons: from; to; object
move icons: from; to; old; new
```

Currently, the only initial keyword available is "icons". Someday, we hope to be able to move other types of objects, such as an area of the screen or the display generated by a unit. Also see "Animation with get and put" for another way to make animations.

The "objects" are character numbers from an icon set specified by an earlier -icons- command. An "object" may include several icons. The character numbers may be given as numbers, variables, or \texttt{zk(lettername)}. In both lines below, the "old object" is AB and the "new object" is ab.

```
move icons: x,y; p,q; 97,98; 65,66
move icons: x,y:p,q;zk(A),zk(B);zk(a),zk(b)
```

The "from" and "to" arguments give the starting and ending positions of the object. If the "from" position is omitted, it is set to the current screen position. The format that omits the "from" position is very common, because you are usually moving the object from "where it is now" to a new position. These two statements are equivalent:

```
move icons: ;x,y; zk(A)
move zwherex,zwherey; x,y; zk(A)
```

The -move- command removes the icons specified by "old" object from the "from" location. It then writes the icons specified in "new" object at the "to" location. If only one "object" argument is given, the new and old objects are the same.

The graphics mode used for removing old icons depends on the current mode (set by the -mode- command):

```
write <-> erase
xor <-> xor
rewrite <-> inverse
```

When using a -loop- to move objects, it is sometimes inconvenient to establish a "from location" for use by the first iteration of the -move- command. The command -inhibit startdraw- prevents the first item from being removed.

In the sample program \texttt{animate.t} distributed with cT is an example of how to use -get- and -put- to move rectangular objects around on the screen while restoring the background, so that the object seems to glide over the screen. The sample program \texttt{BigForty.t} is a solitaire card game that -use-s \texttt{animate.t} for dragging cards.
Example:

```
unit xmove1 $ move a ball across screen
i: x
icons zicons
loop x := 50, 250, 3 $ move 3 pixels at a time
move icons: ; x,100; zk(D)
pause .02
endloop
*

unit xmove2 $ using different modes
i: x, temp
icons zicons
loop temp := -1, 3 $ different mode each time
fill 78,28,206,184 $ solid area
at 10,10; 60,25 $ show mode
write \temp\write\rewrite\xor\erase\inverse
mode \temp\write\rewrite\xor\erase\inverse
inhibit startdraw $ else leaves icon
loop x := 50, 240, 2
move icons: ; x,100; zk(F)
pause .02
endloop
mode write $ back to default
at 75,200
write Press a key.
 erase keys=all $ wait for key
endloop
*

Alternating icons require two -move- commands. Notice that the "to" position for one -move- becomes the
"from" position for the next -move-. The path of the moved object need not be in a straight line. Here the path
is a parabola.

unit xmove3 $ alternating icons
i: x,y
icons zicons
loop x := 50, 250, 4
calc y := 20+( (x+2) -150)^2/100
   $ remove "up arrow" (85) and display "down arrow" (86)
move icons: ; x+2,y; 85; 86
pause 0.05 $ need to slow it down a bit
calc y := 20+( (x+4) -150)^2/100
   $ remove "down arrow" and display "up arrow"
move icons: ; x+4, y; 86; 85
pause 0.05
endloop
*
When moving objects along two different pathways, both the "from" and "to" positions must be given. The first -move- moves along a straight line. The second -move- makes a semicircle (using $a^2 + b^2 = c^2$). A graphing origin is set; the default graph scaling is used.

```
unit xmove4 $$ two pathways
icons zicons
gorigin 125,100 $$ graphing origin
loop x := -100,100, 1
  calc y := sqrt(10000-x^2)
  gmove icons: oldx,0; x,0; zk(F)
  gmove icons: oldx, oldy; x,y; zk(F)
  calcoldx := x $$ save x
  oldy := y $$ save radius
endloop
```

See Also:
icons Selecting an Icon (p. 93)
plot Plotting Characters and Icons (p. 94)
inhibit/allow startdraw (p. 66)
get and put Portions of Screen (p. 89)

Animations with get and put

The -get- and -put- commands can be used to move objects around on the screen, without changing the background. The basic idea is to use -get- to save the background, then display the object using -put- or -plot- or -move- (possibly through a -clip-), pause for the object to be seen, then use -put- to restore the background to its original state.

In the sample program `animate.t` distributed with cT is an example of how to use -get- and -put- to move rectangular objects around on the screen while restoring the background in such a way that the object seems to glide smoothly over the screen on computers that are fast enough. The program is structured to be usable as a -use- file in connection with your own programs. The sample program `BigForty.t` distributed with cT is a solitaire card game that -use-s `animate.t` for dragging cards and offers a choice among several animation schemes.

The following example shows the basic idea, but there is flickering due to seeing the background without the moving object for a brief instant. The appropriate cure for this flickering is to be able to do some of the graphics manipulations off-screen where the eye doesn't see the changes being made, and it is intended that future versions of cT will let you do this. The scheme used in `animate.t` is to use -clip- in a subtle way so that changes are made only at the edges of the moving object, so that only the edges flicker, and the animation looks quite smooth (on fast-enough computers).

Examples:

```
unit xgetanimate
  screen: boxtext, ring, save
  i: x, y

color zred
  at 10,20
  write Here is some text.
  calc x := zwherex
  box 10,20; x,35
```
**See Also:**

- move Making Animations (p. 98)
- get and put Portions of Screen (p. 89)
Making a Graph

Plotting a Graph

The "graphing" family of commands is designed for plotting graphs but is useful whenever you want to establish your own origin and coordinate grid. The graphing commands let you refer to the actual values represented by a point, such as height and weight, rather than absolute screen positions. Thus, after initializing a graphing grid, a point in a height versus weight display could be given as 65,145 for 65 inches and 145 pounds.

The following example puts together the key graphing commands needed to draw graphs, including bar graphs:

Example:

```
unit xgraph $$ plot a function on labeled axes
float: x
color zdefaultf
gorigin 40,170 $$ graph origin
axes185,150 $$ lengths of x- and y-axes
scalex 100 $$ x axis represents 100 units
scaley 10 $$ y axis represents 10 units
labelx 25,5 $$ major marks every 25, minor every 5
labely 2,1 $$ major marks every 2, minor every 1
inhibit startdraw $$ first iteration just positions
color zred
thick 2
loop x := 0, 100, 2 $$ 0 to 100 by 2's
* draw connected lines on graphing grid:
gdraw ; x,5+4cos(.2x)exp(-.01x)
endloop

thick
color zblue
delta 2 $$ width of vertical bars
loop x := 0, 100, 2PI/.2 $$ at maxima
* display vertical bar at maxima of curve:
vbar x,5+4cos(.2x)exp(-.01x)
endloop
*
```

See Also:
- Graphing Commands (p. 102)
- Plot Two User Functions Simultaneously (p. 281)
- Plotting Parametric Equations (p. 282)
- Relative Graphics Commands (p. 112)

Graphing Commands

All of the ordinary graphics commands can be used with graphing coordinates. The graphing form is prefixed with "g". For example, `-gdraw-` is the graphing version of the `-draw-` command. If coordinates are given as floating-point values (numbers with fractions), pixel positions are rounded to the nearest pixel.

The commands below use the same syntax as the absolute coordinate graphing commands. The description of `-vector-` also applies to `-gvector-`, etc., but note that the blank-tag `-gbox-` command draws a frame around the
graphing region, whereas the blank-tag -box- command draws a frame around the area defined by the -fine-
command.

\begin{verbatim}
garrow gbutton gdisk gerase gput
gat gcircle gdot gfill gslider
gatnm geicleb gdraw gget gtext
gbox gclip gedit gmove gvector
\end{verbatim}

In addition, these commands are specifically for use with the graphing coordinates:

\begin{verbatim}
axes scalex markx gorigin
bounds scaley marky lscalex
delta labelx hbar lscaley
polar labely vbar
\end{verbatim}

The system variables relating to mouse location also have graphing forms identified with "g". All system
variables have the prefix "z", so the "g" in this case is the second letter:

\begin{verbatim}
zgtouchx, zgtouchy
\end{verbatim}

These system variables give the mouse location in terms of the graphing grid in effect at the time the event was
accepted (for example, by a -pause- command), not the graphing grid in effect at the time you use these variables
(which might be different due to changing the graphing grid after the -pause-).

\textit{See Also:}
Plotting a Graph \hspace{1cm} (p. 102)
Plot Two User Functions Simultaneously \hspace{1cm} (p. 281)
Plotting Parametric Equations \hspace{1cm} (p. 282)
Relative Graphics Commands \hspace{1cm} (p. 112)

**Graphing Defaults**

The default values for graphing are set at the beginning as if your program had executed these commands:

\begin{verbatim}
gorigin 0,0
bounds 100,100
scalex 100
scaley -100
polar FALSE
\end{verbatim}

When the default values are in effect, the graphing commands (such as -gbox- or -gdraw-) behave just like the
absolute graphics commands (such as -box- or -draw-). These default values are \textbf{not} reset at the start of a new
main unit.

The default value of -delta- is one minor mark. If the minor mark is 0, the default value is 5\% of the positive
axis length.

**gorigin: Setting the Origin**

The -gorigin- command specifies the (x,y) position of the origin of a graphing coordinate system. The values
given in the tag are actual screen positions. The values given by -axes- are screen distances. All other graphing
coordinates are given in scaled values relative to the -gorigin-.
The blank-tag form of \texttt{-gorigin-} sets the origin of the graph to the current screen position (to "zwherex,zwherey"). The \texttt{-gorigin-} position is set to "0,0" at the beginning of the program.

After a \texttt{-gorigin-} command the current screen position is set to the graph origin.

\textit{See Also:}

Graphics Introduction & Defaults (p. 32)

\textbf{axes: Describing the Axes}

The \texttt{-axes-} command specifies the lengths of the axes and causes them to be displayed on the screen. The crossing point of the axes is at the position specified by \texttt{-gorigin-}.

\begin{verbatim}
axes x-positive, y-positive
axes x-negative,y-negative; x-pos,y-pos
axes $$ display previous axes
\end{verbatim}

When the tag of the \texttt{-axes-} command has two arguments, only positive x- and y-axes are displayed. When the \texttt{-axes-} command has four arguments, the axes extend in both the positive and negative directions from the graphing origin. The blank tag form of \texttt{-axes-} displays the axes set by a previous \texttt{-axes-} or \texttt{-bounds-} command.

The axis lengths are given in screen units, not explicit screen positions.

The command \texttt{-axes 300,400-} displays a positive x-axis 300 screen units long extending to the right from the origin and a positive y-axis 400 screen units long extending \textit{upward} from the origin. No negative axes are shown.

In the command \texttt{-axes -200,0; 200,400-}, the negative x-axis is -200, while the negative y-axis is 0. The axes drawn have both positive and negative x-axes, but only a positive y-axis.

\textbf{NOTE:} The \texttt{-axes-} command \textit{does not} define a restricted area for graphing. Points that are "off the graph" may be displayed. Use \texttt{-clip-} or \texttt{-gclip-} to restrict the graphing area.

Once axes values are set with an \texttt{-axes-} or \texttt{-bounds-} command, they retain these values for the entire program unless explicitly changed with a new \texttt{-axes-} or \texttt{-bounds-} command. When the program is entered, a default value is set with \texttt{-bounds 1,1-}.

After an \texttt{-axes-} command the current screen position is set to the graph origin.

The thickness of the axes can be specified with a \texttt{-thick-} command.

\textit{Examples:}

The first unit sets the origin and displays positive axes only. The x-axis is 150 screen units (dots) long and the y-axis is 100 screen units (dots) long. The second unit has both positive and negative axes.

\begin{verbatim}
unit xaxes
gorigin 50,125 $$ set graph origin
\end{verbatim}
bounds: Specifying Axes Lengths

The -bounds- command specifies the lengths of the axes but does not display them on the screen. The command has the same format as the -axes- command.

```
bounds x-positive, y-positive
bounds x-negative, y-negative; x-pos, y-pos
```

Minor Point: Since there is no requirement that a graph position be "inside" the specified axes, the only real effect of -bounds- is to set endpoints that -scalex- and -scaley- can use to calculate the relative scales for the x- and y-axes. Thus it is not necessary to have negative arguments for -bounds- unless you intend to use a blank-tag -axes- command later in the program. However, for program design and internal documentation it is useful to have -bounds- describe the actual area that you intend to use.

scalex: Setting the Scales

The -scalex- and -scaley- commands provide a way to automatically convert quantities expressed in your own convenient units (such as meters, pounds, minutes, etc.) to screen pixel coordinates. After you have set up the graphing environment using -gorigin- and -axes- (or -bounds-), the -scalex- and -scaley- commands specify the maximum x- and y-values on the positive axes and specify the value at the origin.

```
scalex x-maximum
scaley y-maximum
scalex x-maximum, axis-crossing
scaley y-maximum, axis-crossing
```

The first argument of the tag gives the x- or y- value that will appear at the "positive" (the right or the top) end of the axis. The second argument of the tag, "axis-crossing", gives the value that should occur at the origin. If the second argument is omitted, the value at the origin is 0.

Usually only the x-maximum and y-maximum are given:

```
scalex 500
scaley 10
```

If the length of the positive axis is 0, then the "x-maximum" or "y-maximum" argument gives the value at the end of the negative axis. In the example below, the positive x-axis has length 0. Therefore, the left end of the x-axis is -50.

```
axes -200,0; 0,200
scalex -50
```
NOTE: Except in the special case shown by the example just above, the values at the negative ends of the axes are determined by the positive end of the axis, the value at the origin, and the relative lengths of the axes. To adjust the value at the negative end of the axis, you must change the -axes- command.

Examples:

```
unit xscalex
gorigin 75,150
axes -50, -100; 150, 100
scalex 50
scaley 4
labelx 10
labely 1
```

If you were plotting literacy rate versus time, the graph might start as shown below. The positive x-axis represents the interval 1900-2000, and the negative x-axis represents 1850-1900.

```
unit xscalex2
gorigin 100,150
axes -60,0; 120,125
scalex 2000,1900
scaley 100
labelx 50,10
labely 50
```

**labelx: Putting Labels on the Axes**

The -labelx- and -labely- command are used to display labels and tick marks along the x- and y-axes. The -markx- and -marky- commands do not write numbers; they only make tick marks.

```
labelx major, minor, tick style, precise
labely major, minor, tick style, precise
markx major, minor, tick style, precise
marky major, minor, tick style, precise
```

Only the first argument of the tag is required; the other arguments are set to 0 when omitted.

```
labelx 10 $value and tick mark every 10
labelx 10,5 $minor tick mark every 5
labelx 10,5,2 $all tick marks extend to edge
labelx 10,5,2,TRUE $start numbering at origin
markx 25 $major tick marks every 25
markx 25,5 $minor tick mark every 5
markx 25,5,1 $major marks extend to the edge
```

The first argument, **major mark interval**, specifies the interval at which number labels and "major" tick marks are displayed. If the first argument is 0, no labels or tick marks are displayed.
The second argument, **minor mark interval**, specifies the interval at which "minor" tick marks are shown, but no numbers are displayed. If this argument is 0, or omitted if it is the last argument, no minor tick marks are displayed.

The third argument, **tick mark style**, specifies how the tick marks are displayed. It may be 0, 1, 2, or omitted if it is the last argument. For 0 or omitted, the tick marks are short lines. The major tick mark crosses the axis, while the minor tick mark is on only one side of the axis. When the argument is 1, the major tick marks extend from one side of the graph to the other. If the argument is 2, all tick marks extend to the edges of the graph.

The fourth argument, **label precision**, controls the selection of display intervals when the origin is not at 0. *The default case is FALSE.* In the default case, labels are shown on integer multiples of the major interval. That is, for an interval of 10, the values shown would be . . . -30, -20, -10, 0, 10, 20, 30 . . . regardless of the value chosen for the origin. If the fourth argument is set to TRUE (that is, to -1), the numbering is forced to start at the origin.

After -labelx-, -labely-, -markx-, and -marky- the current screen position is set to the graph origin.

The thickness of the tick marks can be specified with a -thick- command.

**Examples:**

This example puts labels every 10 along the x-axis and every 25 along the y-axis. Along the y-axis, the major marks are extended across the graph.

```
unit xxlabel0
gorigin 100,175
axes -70,-100; 140,150
scalex 40
scaley 100
labelx 10,5
labely 25,10,1
*  
```

The next two examples illustrate the effect of the "precise" argument. In unit xxlabel1, the "precise" argument is omitted (assumed to be FALSE). The x-axis labels are 1970, 1980, & 1990. Minor marks appear at 1965, 1975, 1985, and 1995. Note that there is no label on the x-axis at the origin.

```
unit xxlabel1
gorigin 50,175
axes 170,150
scalex 1995, 1965
scaley 100
labelx 10,5 $$ precise FALSE by default
labely 50,10
*  
```


```
unit xxlabel2
gorigin 50,175
axes 170,150
scalex 1995, 1965
scaley 100
```
See Also:

thick Line Thickness (p. 62)

The \texttt{-lscalex-} and \texttt{-lscaley-} commands change the x- and y-scales of a graph to log scales. Either the x-scale, the y-scale, or both may use log scaling.

Log scaling does not permit unusual intervals. The origin and the endpoints of the axes are integer powers of 10: $10^{-2}$, $10^{-1}$, 1, 10, 100, 1000, etc.

\textbf{Examples:}

\begin{verbatim}
unit xsemilog $$ semi-log graph
gorigin 30,250
axes 200,200
lscalex 100 $$ log scale
scaley 100 $$ regular scale
labelx 10,10,2
labely 10,5,1
*
unit xloglog $$ log-log graph
gorigin 35,225
axes 200,200
lscalex $10^{-3}$, $10^{-5}$ $$ 2 decades on x-axis
lscaley 1000 $$ 3 decades on y-axis
labelx 10,10,2
labely 10,10,2
*
\end{verbatim}

\textbf{Labeling Log Scales}

The general format of \texttt{-labelx} and \texttt{-labely} does not change when log scaling is in effect, but less flexibility is available. The command \texttt{-labelx 10,10-} usually makes satisfactory labels.

Log scaling does not permit unusual intervals. The origin and maximum of a log scale are always at integer powers of 10. Log scales always have labels at the integer powers of 10, so the major-mark argument is ignored. It is conventionally given as 10.

The minor marks may be shown in one of these patterns:

\begin{verbatim}
tag is 0  minor marks not plotted
tag is 3  minor marks at 1, 2, 5
tag is 5  minor marks at 1, 2, 3, 5, 7
tag is 10 minor marks at 1, 2, 3, 4, 5, 6, 7, 8, 9
\end{verbatim}

The interpretation of the third argument, length of tick marks, does not change for log scales: 0 means short tick marks, 1 extends major marks, and 2 extends both major and minor marks.
Example:

```
unit xloglabel
gorigin 30,225
axes 200,200
lscalex 1000
lscaley 100
labelx 10,10  $$ all minor marks shown
labely 10,3,2  $$ minor marks only at 1,2,5
gbox $$ outline graph area
```

**polar: Polar Coordinates**

The `-polar-` command changes the interpretation of the ",x,y" coordinates from rectilinear coordinates to the "radius,angle" interpretation of polar coordinates.

```
polar TRUE $$ start polar plotting
polar FALSE $$ return to (x,y)
```

The angle is normally measured in degrees, but the command `-inhibit degree-` changes to radian measure. Using radians is often convenient for consistency with the radians that are always used in the trigonometric functions (sine, cosine, arctan, etc.).

Example:

This example draws a spiral. As the angle gets larger, the distance from the origin (the length of the line in the `-gdraw-`) also gets larger. The spiral is narrowed because the x-axis is shorter than the y-axis.

```
unit xpolar
 i: angle
gorigin 125,200
axes -100,0; 100,150
polar TRUE $$ start polar
scalex 10
scaley 10
labelx 5.1
labely 5.1
gat 0,0
loop angle := 0, 1080, 15
  gdraw ; angle/200, angle
endloop
polar FALSE $$ return to rectilinear coordinates
```

See Also:
```
inhibit/allow degree (p. 70)
```
hbar: Bar Graphs

The -hbar- and -vbar- commands draw horizontal and vertical bars on a graph. The -delta- command specifies the thickness of the bar. Several bars may be displayed with one command by separating the points with semicolons. If the tag starts with a semicolon, the first bar is positioned at the current screen position.

\[\text{hbar} \quad \text{xvalue,yvalue} \quad\text{horizontal}\]
\[\text{vbar} \quad 15,3 \quad\text{vertical}\]
\[\text{vbar} \quad ;7,5; 10,7; 8,3 \quad\text{multiple bars}\]

One end of the bar is always at the axis. The other end of the bar is at the position given in the tag of the command. After an -hbar- or -vbar-, the current screen position is set to the last point mentioned.

Ordinarily the bars are solid black. The -pattern- command may be used to specify a pattern for filling the bars.

NOTES: These commands always relate to graphing coordinates. There are no corresponding commands in the absolute or relative coordinate systems. Do not use -hbar- and -vbar- with polar coordinates.

Examples:

\[
\begin{align*}
\text{unit} & \quad \text{xhbar} \\
\text{gorigin} & \quad 50,200 \\
\text{axes} & \quad 150,150 \\
\text{scalex} & \quad 20 \\
\text{scaley} & \quad 10 \\
\text{labelx} & \quad 10,5 \\
\text{labely} & \quad 5,1 \\
\text{hbar} & \quad 5,10; 10,8; 13,7; 15,5; 13,3; 7,1 \\
\text{vbar} & \quad 0,-3; 10,4; 20,8; 30,7; 40,10; 50,8 \\
\text{gdraw} & \quad 0,-3; 10,4; 20,8; 30,7; 40,10; 50,8 \\
\end{align*}
\]

See Also: Making Textured Areas (p. 61)

delta: Bar Width

The -delta- command specifies the thickness of the bars displayed by -hbar- and -vbar-. The thickness is specified with respect to the scale set by -scalex- (vbar) and by -scaley- (hbar). A delta of "2" specifies 2 units of thickness according to the scale on relevant axis. If -delta- is omitted or set to 0, the default values are used.

Default scaling: When there is no -delta- command, the thickness is set equal to one "minor mark" interval as specified by the -labelx- (vbar) or -labely- (hbar) command. If the "minor mark" is 0, the thickness is set to 5% of the axis length.
**Log scaling:** Values along a log scale are not evenly spaced, so -delta- values along a log axis must have a different interpretation. The value given by -delta- is interpreted as the percent of 1 decade. The default value is 10% of 1 decade.

For example, if the x-axis uses a log scale that runs from 1 to 1000, it covers 3 decades (1-10, 10-100, and 100-1000). If the x-axis is 300 screen units long, 1 decade is 100 screen units, and the default -delta- value for -vbar- is 10 screen units.

*Example:*

```plaintext
unit       xdelta
gorigin    125,175
axes       -100,0; 100,150
scalex     20
scaley     10
labelx     10,5
labely     5,1
hbar       20,1  $$ default thickness
delta      2
hbar       13,5  $$ 2 units thick
delta      0.5
hbar       -17,8 $$ .5 units thick
*```


Relative Graphics Commands

Introduction to Relative Graphics Commands

Relative commands are graphics commands that are drawn with respect to a 0,0 point established by an -rorigin- command and whose size and orientation are affected by -size- and -rotate- commands. These commands are very useful when one design must be drawn at a number of different positions, sizes, and angles. If coordinates are given as floating-point values (numbers with fractions), pixel positions are rounded to the nearest pixel.

All of the graphics commands that are prefixed with "r" (the "r-type" commands) follow the same formats as their absolute counterparts.

<table>
<thead>
<tr>
<th>rarrow</th>
<th>rbutton</th>
<th>rdisk</th>
<th>rerase</th>
<th>rput</th>
</tr>
</thead>
<tbody>
<tr>
<td>rat</td>
<td>rcircle</td>
<td>rdot</td>
<td>rfill</td>
<td>rslider</td>
</tr>
<tr>
<td>ratnm</td>
<td>rcircleb</td>
<td>rdraw</td>
<td>rget</td>
<td>rtext</td>
</tr>
<tr>
<td>rbox</td>
<td>rclip</td>
<td>redit</td>
<td>rmove</td>
<td>rvector</td>
</tr>
</tbody>
</table>

NOTE: -rdisk- does not work correctly when it is rotated.

The system variables relating to mouse position also have relative forms identified with "r". All system variables have the prefix "z", so the "r" in this case is the second letter:

zrtouchx, zrtouchy

These system variables give the mouse location in terms of the relative coordinates in effect at the time the event was accepted (for example, by a -pause- command), not the relative coordinates in effect at the time you use these variables (which might be different due to changing the origin or size or rotation after the -pause-).

Example:

```
unit xrtype
color zblue
rorigin 60,120
do xclown $$ display first clown
color zred
rorigin 185,110
rotate 30 $$ change angle & size
size 1, .7
do xclown $$ display second clown
rotate 0
size 1 $$ always return to size=1, rotate=0
*
unit xclown
rat 0,0
rcircle 40
rcircleb 27.40,140
rdraw -20,-35; 36,-88; 37,-18
rvector -17.0; -17,-16; 15
rvector 15.0; 15,-16; 15
rfill 0.0; -7.7; 0.14; 7.7
rbox 0.0; -100; 50,50; 4
*
```
**rorigin: Setting the Origin**

The -rorigin- command specifies a screen position that serves as the "0,0" position for all of the relative commands.

```
rorigin 150,150
rorigin $$ set to "zwherex, zwherey"
```

The location of the -rorigin- is given in absolute coordinates from the upper-left corner. If the tag is blank, the -rorigin- is set to the current screen position. After the -rorigin- command, the current screen position is at the relative origin.

Once an -rorigin- has been set, it remains set for the rest of the program unless changed by a new -rorigin- command. The default -rorigin- is at 0,0.

**size: Rescaling a Display**

The -size- commands affects the relative commands (all of the graphics commands prefixed with an "r").

```
size value
size x-value, y-value
```

The tag of the size command may have one or two arguments. When only one argument is given, the size affects both the x- and y- coordinates. When two arguments are given, the first argument is the multiplier for x and the second is the multiplier for y. The default size is 1; always return to -size 1- after a special display.

*Example:*

The two -rdraw- statements are exactly the same, but the environment has been modified by -rorigin- and -size-.

```
unit xsize
rorigin 50,150
rdraw 0,0; 25,-25; 50,0; 25,25; 0,0
rorigin 150,150
size 2,4
rdraw 0,0; 25,-25; 50,0; 25,25; 0,0
size 1
*```

**rotate: Rotating a Display**

The -rotate- command affects the relative commands (all of the graphics commands prefixed with an "r").

```
rotate angle $$ angle is normally expressed in degrees
```

The angles are measured clockwise from a horizontal line. Angles increase in the direction of increasing y-values. Math conventions usually have y-values increasing upwards, leading to angles that increase counterclockwise. However, since y increases downwards, cT angles increase in the clockwise direction.

The default angle for -rotate- is 0. You should always return to -rotate 0- after a special display or later displays may surprise you!
The angle is normally measured in degrees, but the command -inhibit degree- changes to radian measure. Using radians is often convenient for consistency with the radians that are always used in the trigonometric functions (sine, cosine, arctan, etc.).

Example:

First a sailboat is drawn relative to the -rorigin- at 100,100. Then the -rorigin- is reset to 200,100 and -rotate- is set. The second sailboat is rotated by 45 degrees.

```plaintext
unit xrotate
rorigin 100,100
do sailboat $ draw boat
rorigin 200,100
rotate 45
do sailboat $ draw boat
rotate 0 $ reset -rotate-
*
unit sailboat
rbox 0,0; 80,-20
rdraw 60,-20; 60,-90; 10,-30; 60,-30
rat 45,-50
rcircle 8
*
```

See Also: inhibit/allow degree (p. 70)

rcircle: Relative Circles

The formats of the -rcircle- and -rcircleb- commands are the same as the formats of the -circle- command:

- `rcircle radius`
- `rcircle radius, begin arc, finish arc`
- `rcircle x1,y1; x2,y2 $ ellipse`

The actual size and position of the -rcircle- depends on -size- and -rotate-. The start and finish of arcs are measured in degrees. If the -size- is different for the x and y directions, -rcircle radius- displays an ellipse.

In the form -rcircle x1,y1; x2,y2-, the ellipse cannot be rotated but causes an execution error. However, the other form of -rcircle- can be rotated.

Examples:

```plaintext
unit xrcircle $ $ default size & rotation
next xrcircle2
at 50,15
write y increases downward
rorigin 100,150
do xCirclePic
*
unit xCirclePic $ draw circle picture
rat 0,0
rcircle 100, 0,75
```
rdraw 0,0; 150,0 $$ line along +x axis
rvector 0,0; 0,100 $$ vector along +y axis
*
unit xrcircle2 $$ example with size change
next xrcircle3
at 50,15
write size 1,-1 makes positive y go upward
rorigin 100,150
size 1,-1 $$ reverse y orientation
do xCirclePic
size 1 $$ reset to default size
*
unit xrcircle3 $$ example with rotation
next xrcircle4
rorigin 100,150
at 50,15
write +y downward, rotate 30 degrees
rotate 30 $$ rotate 30 degrees in +y direction
do xCirclePic
rotate 0 $$ reset to default
rdraw 20,0;50,0;skip;60,0;90,0;skip;100,0;130,0;skip;140,0;170,0
*
unit xrcircle4 $$ example with both size & rotation
next xrcircle
at 50,15
write +y upward, rotate 30 degrees
rorigin 100,150
size 1,-1 $$ positive y upward
rotate 30 $$ rotate 30 degrees in +y direction
do xCirclePic
size 1 $$ reset size to default
rotate 0 $$ reset rotation to default
rdraw 20,0;50,0;skip;60,0;90,0;skip;100,0;130,0;skip;140,0;170,0
*

rtext: Sizing of Text

The -rtext- command is just like the -text- command except that the font size selected for the text is affected by the -size- command.

The -rtext- command cannot be rotated. Using -rtext- when -rotate- has been set to a nonzero value causes an execution error.

Example:

unit xrttext
rorigin 50,20
rbox 0,0; 150,40
rtext 0,0; 150,40
Here is some text in an -rtext- command.
\ 
robox 0,50; 150,130
rat 0,50; 150,130
size 1.5 $$ increase size by 1 1/2
rtext
The -size- command affects this text, but not the margins.
\rbox 0,100; 150,150
rtext 0,100; 150,150 $$ these positions are in "size 1.5"
Both the margins and the size are affected by the -size- command.
\size 1 $$ return to default size
*

See Also:
text Putting Text on the Screen (p. 39)
font Selecting a Typeface (p. 43)
3. Video & Sound

Video Commands

**video: Initialize Video**

cT can display video in QuickTime format or Video for Windows (version 1.1 or later). The -video- command selects a movie file to be played:

```plaintext
video movie; "Cartoon1"   $$ select file "Cartoon1" to play as a movie
video    $$ cancel video (stops playing; makes controller inactive)
```

You can also specify a rectangle where the movie will be displayed, and in that case add keywords to specify additional options (or use a later -vset- command to set or reset these options):

```plaintext
video movie; "Cartoon1"; x1,y1; x2,y2; keyword; keyword; etc.
```

The following command will play an entire movie file, scaled to fit the rectangle:

```plaintext
video movie; "Paris"; 10,20; 200,150; play
```

The keyword options for the -video- command are:

- **scale**
  - movie is scaled to fit specified rectangle, changing aspect if necessary
  - (this is the default)

- **no scale**
  - no scaling; if rectangle small, only upper-left video portion visible

- **center**
  - no scaling; movie is centered in specified rectangle

- **sound**
  - control sound volume (sound,volume -- where volume is 0 to 100)
  - (volume = -1 sets to default volume for this video)

- **play**
  - play the entire movie
  - (default is NOT to play, in which case use a subsequent -vplay-)

- **loop**
  - whenever end of movie is reached, start over from beginning
  - "loop" or "loop,variable" where variable is TRUE or FALSE
  - (default is NOT to loop)

- **controller**
  - (or **ctrl**) displays video controller (scroll bar, etc.)
  - (controller placed under image; default is that there is no controller)
  - If there isn't enough room for a controller, zreturn = 18.

- **ctrl rect**
  - place controller at specified location rather than under image
  - (ctrl rect, x1,y1; x2,y2 -- height however is zvcheight)

- **palette**
  - use palette that is included in the movie, if there is one
  - "palette" or "palette,variable"; variable is TRUE or FALSE
  - (default is TRUE, movie does use an included palette)
The -video- command chooses the video to be displayed, after which -vplay-, -vshow-, and -vstep- commands can be used to play video sequences, show a still frame, or step forward or backward. The -vset- command can modify various video options.

There are several system variables associated with the -video- command:

- **zvtime**: $$ gives the current time from the start of the selected movie
- **zvlength**: $$ gives the length (in seconds) of the selected movie
- **zvwidth**: $$ gives the full horizontal size (pixels) of the selected movie
- **zvheight**: $$ gives the full vertical size (pixels) of the selected movie
- **zvcheight**: $$ gives the vertical height (pixels) of the movie controller
- **zvplaying**: $$ TRUE while video (or sound only) is playing

You can play a movie in its full unscaled size with the following statements:

```
video movie; "MyFilm" $$ sets up zvwidth & zvheight
vset rectangle, 10,30; 10+zvwidth-1,30+zvheight-1; no scale
vplay
```

The video rectangle refers only to the video image; if you specify a video controller, it is placed below the image (unless placed elsewhere using the "ctrl rect" option) and has a height given by **zvcheight**.

**Effect of -clip-**: If there is a -clip- command in effect at the time of specifying the display rectangle with a -video- or -vset- command, the display rectangle is affected by the -clip- region, and the video controller is shrunk to fit within the clipped region. You can use this to play just the right side of a movie, for example.

When video is shown on color-palette machines (typically 256-color machines), by default the -video- command installs a palette included in the movie if there is one, which can affect other displays on the screen. If you don't want this to happen, choose the "palette, FALSE" option.

When running on Windows, a Video for Windows file must have the file extension ".avi" in order to be recognized as a video file. Suppose the Video for Windows file name is "racer.avi", and you have an equivalent QuickTime movie named "racer" on a Macintosh. Just use the name "racer" in the -video- command, and cT will automatically append the ".avi" when running on Windows, so you don't have to change the -video- command.

The -video- command sets **zreturn** in the same way as file operations, with the addition that zreturn = 20 for "operation not supported" if you try to play a movie on a machine that doesn't have QuickTime or Video for Windows installed.

Executing a new -video- command stops any video already in progress.

In the sample programs distributed with cT, **video.t** gives an example of how to create your own specialized video controller using the basic video commands. A short video clip for testing purposes is provided with **video.t** for Macintosh and Windows.

**Example:**

```
unit xvideo
i: curTime
file: choose

loop
  setfile choose; zempty; ro $$ choose a movie file
  zreturn
  outloop zreturn = 18
```

```
VIDEO COMMANDS

at 5,5
write Need bigger window.
else
jump out
endif
endloop

See Also:
- vset: Modify Video Options (p. 119)
- vplay: Play a Video Sequence (p. 120)
- vshow: Show a Single Frame (p. 121)
- vstep: Video Stepping (p. 121)
- Video Status (p. 327)

vset: Modify Video Options

The -vset- command modifies the conditions for playing a movie, including the rectangle in which the movie is displayed:

vset rectangle, left, top; right, bottom; keyword; keyword
vset sound, volume $$ "sound, 50" is half-volume
$$ volume = -1 sets to default sound volume for this video

Here is the complete set of optional keywords, similar to the -video- command:
**rectangle**  
(or **rect**) specify a new display rectangle  
(rectangle, x1,y1; x2,y2)

**scale**  
movie is scaled to fit specified rectangle, changing aspect if necessary  
(this is the default)

**no scale**  
no scaling; if rectangle small, only upper-left video portion visible

**center**  
no scaling; movie is centered in specified rectangle

**sound**  
control sound volume (sound, volume -- where volume is 0 to 100)  
(volume = -1 sets to default volume for this video)

**play**  
play the entire movie  
(default is NOT to play, in which case use a subsequent -vplay-)

**loop**  
whenever end of movie is reached, start over from beginning  
"loop" or "loop,variable" where variable is TRUE or FALSE  
(default is NOT to loop)

**controller**  
(or **ctrl**) displays video controller (scroll bar, etc.)  
(controller placed under image; default is that there is no controller)  
If there isn’t enough room for a controller, zreturn = 18.

**ctrl rect**  
place controller at specified location rather than under image  
(ctrl rect, x1,y1; x2,y2 -- height however is zvcheight)

**palette**  
use palette that is included in the movie, if there is one  
"palette" or "palette,variable"; variable is TRUE or FALSE  
(default is TRUE, movie does use an included palette)

For example, the following command will shift the playing of the movie to the new rectangle and center the display in the rectangle:

```plaintext
vset rectangle, 10,20; 200,150; center
```

**See Also:**
- video: Initialize Video  
(p. 117)
- Video Status (p. 327)

## vplay: Play a Video Sequence

The **vplay**- command plays a portion of a movie (a **video**- command must have been executed previously):

```plaintext
vplay start,end
```

where start and end are times measured from the start of the movie. Either of the arguments may be omitted. A missing start is interpreted as the current time (same as **zvtime**, the system variable that gives the current time within the movie). A missing end is interpreted as "don't stop." A time that is negative is the same as a missing time. Some examples:

```plaintext
vplay $$ blank tag, play without stopping from where we are  
$$ (like hitting the play button on the player)```
The -vplay- command sets `zreturn` = 20 for "operation not supported" if you try to play a movie on a machine that doesn't have QuickTime or Video for Windows installed.

Example:

```
unit xvplay
video movie; "somemovie"; 10,10; 150,100  $$ initialize movie
vplay zvlength/2,zvlength   $$ play 2nd half of movie
loop zvtime >= 0.75zvlength
    pause 0.1
endloop
write Have just passed 3/4 point in movie...
*
```

See Also:

```
video Initialize Video (p. 117)
Video Status (p. 327)
```

### vshow: Show a Single Frame

The -vshow- command displays a single still "frame" of a movie and holds it:

```
vshow tt  $$ show still frame tt seconds into movie
```

This is equivalent to -vplay tt,tt-. A -video- command must have been executed previously.

Example:

```
unit xvshow
video "somemovie"; 10,10; 150,100  $$ initialize movie
vshow 2  $$ show still frame 2 seconds into movie
*
```

See Also:

```
video Initialize Video (p. 117)
Video Status (p. 327)
```

### vstep: Video Stepping

If a video sequence is stopped, the -vstep- command plays the next or previous specified number of "frames" (more technically, that number of video "samples"):

```
vstep nn  $$ play nn frames; forward if positive, backward if negative
vstep 1   $$ step forward one frame (for a "slide show")
vstep -1  $$ step backward one frame
vstep 0   $$ stop playing
```

```
```
VIDEO & SOUND

A -video- command must have been executed previously. If a video sequence is currently playing, -vstep- will stop the sequence and then play the specified number of frames (no play if nn = 0).

After a -vstep- command the system variable zreturn is set to 14 (out of range) if the display reaches the end of the movie (or the start of the movie if stepping backwards).

**Example:**

```plaintext
unit xvstep
video movie; "somemovie"; 10,10; 150,100 $$ initialize movie
vplay $$ start playing
pause
vstep 0 $$ stop playing
```

**See Also:**

- [video Initialize Video](p. 117)
- [Video Status](p. 327)

(sound: Playing Recorded Sounds)

The -sound- command plays recorded sounds such as speech or music:

```plaintext
sound filename $$ play first sound in the file
sound filename,N $$ play sound number N in the file
```

The -sound- command handles regular sound files on a Macintosh; on Windows it handles *.wav files. Also, the -sound- command recognizes and plays the sound track of a QuickTime or Video for Windows movie, in which case any video clip currently running is shut down, and the system variables zvtime, zvlength, and zvplaying give information about the playing of the sound.

The difference between -beep- and -sound- is that the -beep- command plays tones specified by frequency, duration, and volume, whereas the -sound- command plays arbitrary sounds that have been prerecorded.

To play different sounds stored in the same file on a Macintosh, use ResEdit to examine the file. Double-click the sound icon ("snd" resource), and note the ID numbers listed for each sound of interest. Use these ID numbers in the -sound- command.

**See Also:**

- [beep Making an Audible Tone](p. 122)

(beep: Making an Audible Tone)

The -beep- command plays a tone with the computer's built-in tone generator.

```plaintext
beep $$ blank tag beep, gives simple beep
beep frequency, duration, volume
beep f1,d1,v1; v2,d2,v2; v3,d3,v3
```

The simplest format, a blank-tag -beep-, plays the "system default beep." On a Macintosh, this default beep can be set by the user from the control panel.
In the second format, a single tone is defined by a triplet of values: frequency, duration, and volume. One -beep- command can specify multiple tones, with the tones separated by semicolons.

Frequency is stated in hertz (cycles per second) and is rounded to the nearest musical note (including sharps and flats). The higher the frequency, the higher the tone. Middle C on the piano is about 260 hertz. An "ordinary" woman's singing voice might range from about 200 hertz (G below middle C) to 660 hertz (2nd E above middle C.) To move upwards one half-step, multiply by $2^{1/12}$. That is, if A is 440, then A-sharp is $440*2^{1/12}$.

Duration is in seconds. Volume is a number from 0-100, which is mapped onto the range of the machine. (You can think of this as percent of highest volume.) A volume of 0 is always silent.

Execution proceeds to the next command while the beep is playing, but if another -beep- command is encountered, cT waits for the first beep to finish before starting the next one.

Examples:

```
unit xbeep
    f: t = 0.45 $$ one unit of time
    f: volume = 30 $$ 30% of maximum volume
    i: tune(14) $$ list of notes for the tune
    f: length(14) $$ time duration of each note
    i: index
    i: c = 262, cs = 277, d = 294, ds = 311 $$ note names
    i: e = 330, f = 349, fs = 370, g = 392
    i: gs = 415, a = 440, as = 466, b = 494
    set tune := c, c, g, g, a, a, g, f, f, e, e, d, d, c
    set length := t, t, t, t, t, t, 2t, t, t, t, t, t, t, 2t
    loop index := 1,14
        beep tune(index), length(index), volume
    endloop
*
```

See Also:
```
sound Playing Recorded Sounds (p. 122)
```
4. Mouse & Keyset Interactions

Overview of Mouse & Keyset Interactions

In cT there are several commands that allow user input: -pause-, -getkey-, -arrow-, -button-, -edit-, -slider-, and -touch-.

The -pause- command is used for mouse and single keypress inputs, and it also permits timed pauses.

The -getkey- command is used for specialized processing of mouse and single keypress inputs in the middle of a calculational or graphics loop.

Keyboard inputs (words, phrases, numbers, formulas) can be accepted by an -arrow- command and evaluated by a variety of "response-judging" commands. This is discussed under "Word & Number Input."

The -button- command creates a labeled region that accepts and processes mouse clicks in that region.

The -edit- command establishes a scrolling text panel that responds to keyboard and mouse inputs.

The -slider- command creates a slider or scroll bar and lets the user manipulate the slider with the mouse.

The -touch- command establishes a region of the screen and specifies units to be executed automatically when there are mouse clicks and drags in that region (unlike the -button- command, no button is displayed in the region).

For input from external devices, see the -serial- command; for input from other programs, see the -socket- command; and for input from files, see the -setfile- command.

See Also:

- Mouse, Single Key, & Timed Pause (p. 125)
- Word & Number Input (p. 158)
- Buttons, Dialog Boxes, Sliders, & Edit Panels (p. 142)
- serial Serial Port (p. 309)
- socket Connecting to Another Process (p. 311)
- setfile Select a File (p. 291)
Mouse, Single Key, & Timed Pause

pause: Single Key & Timed Pause

The -pause- command can be used for timing and/or to collect user input from the mouse or the keyboard. This section only discusses keyboard input. Mouse input is in the next section.

```
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pause 0.5</td>
<td>wait for 1/2 second</td>
</tr>
<tr>
<td>pause keys=all</td>
<td>wait for any keypress</td>
</tr>
<tr>
<td>pause keys=touch</td>
<td>wait for mouse input</td>
</tr>
<tr>
<td>pause 3,keys=n,#</td>
<td>wait 3 seconds OR for key &quot;n&quot; or &quot;#&quot;</td>
</tr>
</tbody>
</table>
```

When the tag is a number (-pause N-) execution of the program pauses for "N" seconds. After that time has finished, the program "breaks through" the pause, execution continues with the next line, and the system variable `zkey` is set to zk(timeup). If "N" is 0 (or too small to permit accurate timing), execution continues immediately. If "N" is negative, the -pause N- is converted into a blank-tag pause.

When the tag is blank or "keys=all", execution waits for the user to press a key. Execution then continues with the next line and `zkey` is set to the value of the key pressed: `zkey = zk(key pressed)`.

When the tag contains a key list starting with "keys=", it specifies what types of inputs will be noticed. The program waits for one of the listed inputs and sets `zkey = zk(key pressed)`. If both a number and a keyword list are given, the program pauses until either the time has elapsed or one of the specified keys is pressed.

The key list may contain any alphanumeric character (a X %) or any of these words: timeup, next, back, erase, cr, space, tab, ext, touch (and touch variants). Entries in the key list are separated by commas.

Time, keyset, and mouse events can be combined. The following statement will wait for "k", "m", mouse left button pressed down or mouse moved, or for three seconds to elapse:

```
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pause 3,keys=k,m,touch(left: down,move)</td>
<td>pause one-half second OR for key &quot;n&quot; OR &quot;#&quot;</td>
</tr>
</tbody>
</table>
```

Examples:

```
unit xpause1 $$ using -pause- for timing
i: x
loop x := 50, 300, 25 $$ box moves across screen
  box x, 50; x+75,150
  pause 0.5 $$ pause one-half second
  erase $$ erase entire display
endloop
*
unit xpause2 $$ using -pause- to accept keyset inputs
at 50,50
write press a number
pause keys=0,1,2,3,4,5,6,7,8,9
at 100,100
show zchar(zkey) $$ shows character whose numerical code is "zkey"
at 115,100
show zkey $$ shows ASCII key value (numerical code)
* 
```
See Also:
enable Allowing Mouse Input (p. 132)
The Keyname Function: zk() (p. 204)

pause: Mouse Inputs

Mouse clicks (i.e., where you "touch" the display) are collected with a -pause- command.

\[
\text{pause keys = touch} \quad \text{\$\$ notices left button down}
\text{pause keys = touch(left:down,up,move)}
\text{pause keys = touch(left:down; right:down)}
\]

The simplest form, -pause keys=touch- recognizes only "left button down" mouse clicks. Other mouse clicks are ignored. To recognize other button presses, you must explicitly list them in parentheses after the "keys=touch". The argument (left: down,up) means "notice when the left button goes down or when the left button comes up. This \textit{implies} "ignore movement while the left button is down." (The ancestors of cT, the TUTOR and MicroTutor languages, ran on systems that had touch-sensitive display devices, so that when the user was asked to indicate a point on the screen, the user actually \textit{touched} the screen.)

The six different mouse actions yield these values for zkey, which can be used to compare with the zk function:

\[
\begin{align*}
\text{zkey} = \text{zk(left: down)} & = 4096 = \text{zk(touch)} \\
\text{zkey} = \text{zk(left: up)} & = 4097 \\
\text{zkey} = \text{zk(left: move)} & = 4098 \\
\text{zkey} = \text{zk(right: down)} & = 4099 \\
\text{zkey} = \text{zk(right: up)} & = 4100 \\
\text{zkey} = \text{zk(right: move)} & = 4101
\end{align*}
\]

On systems with a one-button mouse, the "right" button is obtained by holding down the shift key on the keyset while clicking with the mouse button.

The x,y position of the mouse when a click is entered is given by three sets of system variables, one for each of the coordinate systems that were in effect at the time of the -pause-:

\[
\begin{align*}
\text{ztouchx, ztouchy} & \quad \text{\$\$ absolute} \\
\text{zgtouchx, zgtouchy} & \quad \text{\$\$ graphing} \\
\text{zrtouchx, zrtouchy} & \quad \text{\$\$ relative}
\end{align*}
\]

These variables are all zero unless the most recent user input was a mouse click.

It is also possible to determine the current mouse coordinates, independent of whether the most recent input was from the mouse:

\[
\text{zmousex, zmousey} \quad \text{\$\$ current mouse position}
\]

Special note: the current mouse position given by zmousex/zmousey may be outside the cT window due to dragging, and the reported position is affected by -fine- and -rescale-.

The system variables zleftdown and zrightdown are TRUE if the left or right button is currently held down, independent of the current value of zkey (which refers to the most recent -pause- or -getkey- event).
If the time between two down events is less than the time given by system variable `zdblclick`, and the two clicks are near each other on the screen (within two pixels, say), you may want to consider the second click to be a "double click." Some computer systems allow the user to specify the double-click time, in which case `zdblclick` is set to the user-specified time. You can identify double-click events by keeping track of the time between down events (using `zclock`).

**NOTE:** Mouse clicks that cause a menu to appear are reserved by the operating system and cannot be detected by a C1 program. You can create or remove a menu item, but you cannot detect or change the screen position of the menu display.

Time, keyset, and mouse events can be combined. The following statement will wait for "k", "m", mouse left button pressed down or mouse moved, or for 3 seconds to elapse:

```c1
pause 3,keys=k,m,touch(left: down,move)
```

**Examples:**

In examples "xmouseclick" and "xmouseclick2" -pause- accepts mouse inputs and uses `ztouchx` and `ztouchy` to position a -write- statement:

```c1
unit xmouseclick  $$ mouse inputs
at 10,10
write Click the left mouse button at various positions.
Select "Quit running" from the menu to exit.
loop
  pause keys=touch
  at ztouchx, ztouchy
  write HI!
endloop
*
unit xmouseclick2  $$ left & right buttons
at 10,10
write Click the left button for HI!
Click the right button for BYE!
(Hold down the shift key if your
mouse has only one button.)
loop
  pause keys=touch(left:down; right:down)
  at ztouchx, ztouchy
  if zkey = zk(left:down)
    write HI!
  else
    write BYE!
  outloop $$ quit
endif
endloop
*
```

The example "xmousearea" looks for a touch inside an area (see the -touch- and -button- commands for a related capability):

```c1
unit xmousearea  $$ click in an area
box 75,35; 225,70  $$ "white" box
fill 75,105; 225,140  $$ black box
```
pause  keys = touch
at  75,175  $$ position for -write-s
if  75 < ztouchx < 225 & 35 < ztouchy < 70
    write  You clicked the white box.
elseif  75 < ztouchx < 225 & 105 < ztouchy < 140
    write  You clicked the black box.
else
    write  You did not click in a box.
endif
*

The example "xmousedraw" draws a continuous line by initiating a line when the left button goes down, noticing mouse movement when the left button is held down, and exiting when the left button is released:

unit  xmousedraw  $$ draw a "continuous" line
at  10,10
write  To start a line, press down the left button.
To draw a continuous line, move the mouse
while holding down the button.
To stop, release the mouse button.
pause  keys=touch(left: down)
at  ztouchx, ztouchy
loop
    pause  keys=touch(left: move, up)
    draw  ; ztouchx, ztouchy
outloop
    zkey = zk(left: up)
endloop
*

Example "xmousedraw2" also draws a continuous line but uses -inhibit startdraw- :

unit  xmousedraw2  $$ draw a "continuous" line
at  10,10
write  To start a line, press down the left button.
To draw a continuous line, move the mouse
while holding down the button.
To stop, release the mouse button.
inhibit  startdraw
loop
    pause  keys=touch(left: move, up)
    draw  ; ztouchx, ztouchy
outloop
    zkey = zk(left: up)
endloop
*

Example "xrubberband" lets you adjust a vector by "rubber-banding":

unit  xrubberband
f: x1, y1, x2, y2
at  10,20
write  Click and drag the mouse
to "rubber-band" a vector:
pause  keys=touch
calc  x1 := x2 := ztouchx
y1 := y2 := ztouchy
mode xor
loop
  vector x1,y1; x2,y2  $$ draw
  pause keys=touch(left: move, up)
  outloop zkey = zk(left: up)
  vector x1,y1; x2,y2  $$ erase
  calc x2 := ztouchx
    y2 := ztouchy
endloop
*

Example "xmousedrag" lets you drag a box around on the screen (also see "get and put Portions of Screen"):

unit xmousedrag
  f: x, y, W=30, H=20
  at 10,20
  write Click and drag the mouse
to drag a box around:
  pause keys=touch
calc x := ztouchx
  y := ztouchy
mode xor
loop
  box x,y; x+W,y+H  $$ draw
  pause keys=touch(left: move, up)
  outloop zkey = zk(left: up)
  box x,y; x+W,y+H  $$ erase
  calc x := ztouchx
    y := ztouchy
endloop
*

See Also:
button Buttons to Click (p. 142)
touch Touch Regions (p. 129)
Mouse Status (p. 325)
The Keyname Function: zk() (p. 204)
zkey Last User Input (p. 333)
inhibit/allow startdraw (p. 66)
get and put Portions of Screen (p. 89)

**touch: Touch Regions**

With the -touch- command you can establish a "touch rectangle" for which associated mouse events are automatically processed:

define touch: mytouch $$ define a "touch" variable
...
touch mytouch; 10,20;100,120; left:down;xDown;
left:move,up;xMore(15); $$ can continue onto following lines
left: double; xDouble $$ process double-clicks
upmove;xWhileUp  $$ process moves with no button down
This -touch- command establishes that if there is a left-button-down event inside the rectangle (specified by 10,20;100,120), unit "xDown" will be executed, and if the mouse is moved or the left button is released, unit "xMore(15)" will be executed. Note that a unit can have a single numeric (integer or float) pass-by-value argument, as in "xMore(15)".

The "upmove" option does the associated unit whenever the mouse is moved, even though no button is depressed.

After establishing a touch rectangle, you don't need (and can't use) explicit -pause- commands in order to process mouse events associated with that rectangle. A -pause- command is affected only by mouse events outside any touch rectangle(s), buttons, sliders, or edit panels.

For any of the -touch- units to be done, the first down event must be within the rectangle, even if you don't specify a unit for down events. Once the mouse button has been pressed down, the following move events (if any) and the final up event cause the move and up units to be done, even though the mouse may be moved outside the touch rectangle. Which -touch- command "owns" the mouse events is determined by the initial down event, not by later moves.

While a specified touch unit is processing an event, other mouse events are locked out and do not get processed until the specified unit is completed (or there is a -jump-).

The following statement destroys the associated touch region, so that the mouse events are no longer processed automatically:

```
touch mytouch $$ destroy the touch region
```

The following statement first destroys the touch region originally associated with the touch variable "mytouch" and then creates a new touch region:

```
touch mytouch; 200,200; 220,220; left:down;Other $$ change the touch region
```

The touch event options include those available with the -pause- command (left:down, left:up, left:move, right:down, right:up, right:move). You can even specify both left-button and right-button options with the same unit:

```
touch mytouch; 10,20;100,120; left:down,up;right:down,move;Action
```

You can also specify double-click events as "left:double" or "right:double". A double-click is defined as a rapid sequence of down-up-down, and the associated unit is executed when the user releases the mouse button (the second up event in the sequence). Note that in a rapid sequence of down-up-down-up, the first down and up events are processed in the normal way. It is only when the second down event occurs that cT recognizes a double-click event, in which case it does not do the "down" unit but waits for the final up event and does the "double" unit.

For cT to identify a double-click event, the time between the two down events has to be less than the time given by system variable zdblclick, and the two clicks have to be near each other on the screen (within two pixels). Some computer systems allow the user to specify the double-click time, in which case zdblclick is set to the user-specified time. If you don't specify "double" on a -touch- command, you can identify double-click events yourself by keeping track of the time between down events (using zclock).

The system variables zleftdown and zrightdown are TRUE if the left or right button is currently held down, independent of what event triggered the execution of the touch unit. For example, if the unit was triggered by a down event, there might be a situation where you do things differently in this unit, depending on whether the mouse button has already been released.
There is a "priority" option to distinguish among overlapping rectangles, and this option must be the last thing specified in the statement:

```
touch mytouch; 10,20;100,120; left:up;right:up;Process(44); priority,100
```

If priorities on intersecting touch regions are set, a mouse event in an overlap region "belongs" to the region with the highest priority. If priorities are not set, it is undefined as to which touch command is active.

The cT sample program *BigForty.t*, a solitaire card game, is an example of a program that uses -touch- commands extensively and has no explicit -pause- commands at all.

**Example:**

```
unit xtouch
    touch: bigbox, smallbox
    box  10,20;100,120  $$ big box
    box  160,50;220,80  $$ small box
    touch bigbox; 10,20;100,120; left:down;xDown;
        left:move;xMove; left:up;xUp
        left:double;xDouble
    touch smallbox; 160,50;220,80; left:down;xSmall
    at  15,150  $$ position for writing
    pause  $$ don't stop executing even if not in "Run" mode
* unit xDown  $$ handle down events
    write (  
        do xScreenCheck  
* unit xMove  $$ handle move events
    write -  
        do xScreenCheck  
* unit xUp  $$ handle up events
    write )  
        do xScreenCheck  
* unit xDouble  $$ handle double-clicks
    write D  
        do xScreenCheck  
* unit xSmall  $$ down in small box
    write s  
        do xScreenCheck  
* unit xScreenCheck  $$ check for screen wrap
    if zwherex > zxmax-20
        at 15,zwherey+20
    endif

See Also:

- button Buttons to Click (p. 142)
- pause Mouse Inputs (p. 126)
enable: Allowing Mouse Input

The `enable` command is a rather specialized command that prepares an `arrow` command or a blank `pause` command to receive inputs from "external devices." These commands are not needed for normal mouse interactions. An external device is anything other than the keyset; usually it is the mouse. The `disable` command turns off reception of external inputs.

```
enable touch $$ refers to left:down
disable touch
enable touch(left:move; right:down,move)
```

The `enable touch-` and `-disable touch-` commands affect only left-button-down mouse clicks. Enabling other mouse actions requires explicit arguments for the `touch` keyword:

```
enable touch(left:up; right:up)
enable touch(left:down; right:up,down,move)
disable touch(left: up,down,move)
disable touch(left:move; right:move)
```

An `enable touch` followed by a blank `pause` is the equivalent to `-pause keys = touch,all-`. Either a left mouse click or a keypress will break the pause.

When `enable touch` is in effect at an `arrow`, mouse input inside the bounding box defined by the `arrow` command is used to edit the response. Outside that area, a mouse click initiates judging. In that case, `zkey = zk(touch)` instead of `zkey=zk(next)`, as is normally the case when judging is initiated by pressing ENTER or selecting (Enter Response) from the menu.

An `enable touch` also allows a mouse click to move the user to the next unit. When `enable touch` is in effect, pressing the left mouse button at the end of a main unit is equivalent to selecting (Next Page) from the menu.

Example:

Any mouse click outside the arrow area (shown by a box) initiates response judging and matches the `-no zkey = zk(touch)-`. If the click falls inside the hint box, the hint is given (a more elegant way to do this is with a `-button- command). The `-judge exit- then causes the mouse click to be ignored, so that the user can continue entering her response.

```
unit  xenable
at 50,50
write Who was the Great Emancipator?
enable touch $$ allow touch input at arrow
do clickhere $$ display "hint box"
box 50,100; 240,120; 2 $$ show arrow region
arrow 50,102; 240,120
answer [Abraham Abe] Lincoln
no zkey = zk(touch) $$ click outside arrow region
if 100<ztouchx<225 & 150<ztouchy<220
do showhint
endif
judge exit $$ wait for more input
no endarrow
```
He spent much of his childhood in Kentucky.

See Also:
button Buttons to Click (p. 142)
judge Changing the Judgment (p. 182)

press: Force a Keypress

The -press- command simulates keypresses by the user. Input generated by -press- appears to the program as though it were input entered by the user.

press zk(a) $$ press an "a"
press zk(erase)
press zk(right:up),123,456

Note that press-ed inputs are processed by -pause-, -arrow-, and -getkey- but are not passed to graphics objects (edit panels, buttons, and sliders). It is often possible to simulate input to these graphics objects by -do-ing the units associated with these objects or by using the "reset" option to control the object.

The tag of a -press- command is the numeric value of an input the user might make, such as typing "a" or clicking the mouse. This value is expressed as zk(keyname) or as a number. When -press-ing a mouse input, the mouse position must be given.

Examples:

This is a rather contrived example. It would be more typical to use -jump xpress1a- in this situation.

unit xpress1 $$ Use Run from Selected Unit!
next xpress1a
at 50,50
write Click the mouse
in the box to continue:
box 60,150; 190,225
loop
pause keys=touch $$ wait for a left click
$$ if click is in box, exit from loop
outloop (60 < ztouchx < 190) & (150 < ztouchy < 225)
endloop
press zk(next)
* 
unit xpress1a
at 50,100
write Great! You got to the next unit.
*

In this example, the -press- command supplies a "mouse press" that breaks through the -pause-. Note that when a "touch" is pressed, the position of the touch must be indicated.

```
unit xpress2
press zk(right:up),123,456 $$ touch & position
pause keys=touch(right:up)
at 50,50
write zkey = <|s,zkey|> $$ type of input
ztouchx = <|s,ztouchx|> $$ mouse position x
ztouchy = <|s,ztouchy|> $$ mouse position y
*
```

The third example simulates the action at an arrow by filling the arrow buffer (zarrowm) and then pressing a key (zk(next)) to initiate judging.

```
unit xpress3
at 50,50
write What animal says "meow"?
arro 100,100
calc zarrowm := "cat"
press zk(next)
answer cat
write meow!
endarrow
*
```

See Also:
The Keyname Function: zk() (p. 204)

getkey: Check for Input

The -getkey- command collects a single key from the user-input buffer and places it into the system variable zkey. This is a very specialized command; it is not often used.

```
getkey $$ always has a blank tag
```

Normal processing of user inputs occurs only when the program is in a "waiting" state: at an -arrow-, a -pause-, or the end of a main unit. All keys are stored in the "user-input buffer" until the program is ready to process them. The -getkey- command is used to retrieve keypresses when there is no -arrow- or -pause-, such as in a tight calculational loop or during an animation. The -getkey- command removes the first key from the input buffer and places its value in zkey. If there are no keys waiting in the input buffer, zkey is set to -1.

Example:

```
unit xgetkey $$ escape from a tight loop
f: N
calc N := 1 $$ initialize counter
at 50,50
write Press "s" to stop the counter.
```
loop
  .  calc  N := N + 1 $\text{increment counter}$
  .  erase  100,100; 150,150
  .  at  100,100
  .  show N $\text{display counter}$
  .  getkey $\text{get key from input buffer}$
  .  outloop zkey=z(k(s)) $\text{exit if key is } ^{\text{\textbackslash s}}$
  endloop
at 50,150
write after the endloop
*

**clrkey: Clear the Input Buffer**

All inputs, whether from the keyboard, from the mouse, or from -press- commands, are stored in the "user-input buffer" until the program is ready to use them. When keys are pressed rapidly, several keys may be pressed before the first key is processed. Sometimes it is not appropriate to allow several keys to accumulate in the buffer. The -clrkey- command clears the user-input buffer. That is, it throws away all pending key presses.

    clrkey $\text{always has a blank tag}$

*Example:*

Normally it is possible to "stack up" many keys while waiting for the loop to finish. The -clrkey- enforces the instructions; only keys typed after the loop is finished are seen by the -arrow-. Try this example with and without the -clrkey- command.

    unit xclrkey
    f: x
    icons zicons
    at 22,21
    write Type "hello" when the ball reaches the edge of the screen.
    inhibit startdraw $\text{prevent extra icon}$
    loop x := 50, zxmax
        move icons: ; x, 60; zk(D)
    endloop
    clrkey $\text{empty user-input buffer}$
    arrow 73, 96
    answer hello
    endarrow
*

*See Also:*

    inhibit/allow startdraw (p. 66)
Pull-down Menus

Summary of Menu Formats

The -menu- command creates a pull-down menu item. The tag of -menu- is in two major parts. The first part describes what will appear on the pull-down menu and may specify ordering. It is terminated with a colon. The second part names the unit that corresponds to this menu item. The unit name may send one numeric (integer or float) pass-by-value parameter.

```
menu  itemtitle: unitname
menu  cardname; itemtitle: unitname
menu  Special; Choice <|s,nn|>: xterm1  $$ example with embedded -show-
menu  itemtitle: unitname(parameter)
menu  card; title: unitname(parameter)
menu  itemtitle,M: unitname  $$ M is position info
menu  cardname, N; itemtitle: unitname
menu  card,N; title,M: unitname(parameter)
```

Items may be deleted individually, an entire menu card may be deleted, or all program-created items can be deleted at once.

```
menu  cardname:AnItem   $$ delete item
menu  cardname          $$ delete card
menu  $$ cancel all -menu- items
```

The unit name can be in a marker variable:

```
unit  mtest
    marker: sub1
    calc  sub1 := "TryIt"
...  
menu  Hello: (sub1)   $$ equivalent to -menu   Hello: TryIt-
```

Note that a -button- command essentially provides an "on-screen" version of a menu option.

See Also:
Simple Menu Commands (p. 136)
Ordering Menu Items and Cards (p. 138)
Passing Menu Parameters (p. 139)
Menu Like -do- (p. 140)
button       Buttons to Click (p. 142)

Simple Menu Commands

The -menu- command creates a "pull-down" menu item. When the user selects the menu item, the associated unit is executed as if it were executed as a subroutine with a -do- command. Once an item has been added to the menu, it is always available until it is explicitly deleted.
When testing -menu- commands, you must use "Run from Selected Unit" or "Run from Beginning", unless a -pause- command blocks completion of execution.

```
  menu itemtitle: unitname
  menu cardname; itemtitle: unitname
  menu Special; Choice <{s,nn}>: xterm1 $$ can use embedded -show-
```

The "itemtitle" is the word or phrase that appears on the menu. The simplest form of -menu- puts an item on the first menu card. For example, to make available a routine for resetting the scales on a graph, you might have:

```
  menu Reset Graph Scales: setscales
```

If a "cardname" appears, it causes the menu item to appear on a menu card whose title is the "cardname." A semicolon separates "cardname" from "itemtitle." For example, if you had several options for manipulating a graph, it would be nice to have them collected on a separate menu card. The following commands create a new card labeled "Graph Options" that has two items, "Reset Scales" and "Show Grid":

```
  menu Graph Options;Reset Scales: scales
  menu Graph Options;Show Grid: grid
```

A menu item is deleted by giving only its description as the tag of the command, with no colon and no unitname.

```
  menu itemtitle $$ delete item from the standard Options menu
  menu Options; itemtitle $$ delete item from Options menu
  menu cardname; itemtitle $$ delete item from another menu
```

To delete an entire menu card and all of its items, use only the cardname in the tag, with no itemtitle, colon or unitname:

```
  menu cardname $$ delete card
```

To delete all of the program-created menu items, use a -menu- command with a blank tag.

```
  menu $$ delete all menus
```

Some menu items are supplied automatically by the system. These menu items are not controlled by the -menu- command and are not deleted by a blank -menu- command. For example, (Back) appears on the first menu card whenever a -back- command is active.

The unit name can be in a marker variable:

```
  unit mtest
  marker: sub1
  calc sub1 := "TryIt"
  ...
  menu Hello: (sub1) $$ equivalent to -menu Hello: TryIt-
```

Note that a -button- command essentially provides an "on-screen" version of a menu option.
Example:

This example creates a new menu card named "Displays" and puts two items on it. Since no card name is specified, the item "Clear Screen" goes on the first card. If "Draw a Triangle" is selected, -unit triangle- is executed.

Notice that after the -write- statement, execution has reached the end of the unit and is waiting for the user to take an action. After a triangle or square is drawn, the program returns to the point just after the -write- statement and again waits for the user to take action. Thus, in this example, the user could select the triangle and then the square.

Example:

```
unit xmenu $$ "Run from Selected Unit"
next xmenu
menu Displays; Draw a Triangle: triangle
menu Displays; Draw a Square: square
menu Clear Screen: clear
at 50,50
write You can use the menu
to select a drawing.
*
unit triangle
draw 25,150; 100,100; 150,200; 25,150
*
unit square
box 175,100; 250,175
*
unit clear
erase
```

See Also:
- Ordering Menu Items and Cards (p. 138)
- Passing Menu Parameters (p. 139)
- Menu Like -do- (p. 140)
- do Calling a Subroutine (p. 224)
- next Moving Ahead (p. 237)
- button Buttons to Click (p. 142)

Ordering Menu Items and Cards

The first part of the -menu- command (before the colon) may have four pieces. Only the "title" is required. Usually items appear on the menu in the order in which they are created. Items or cards can be forced into a particular order by following the name with a comma, then a precedence value (a number).

```
menu card,N; title,M: unitname
```

In the line above,
- **card** is the label on a user-created menu card
- **N** is the menu card's relative position
- **title** is the phrase that appears on the menu
- **M** is **title**'s relative position on the card
Items (or cards) with small precedence values appear before items (or cards) with large values. The values must be between 0 and 100; they need not be consecutive. If two items (or cards) are given the same precedence value, the one that is mentioned first will appear first. To place an item at the bottom of a card, the command might be

```
menu Reset Graph Scales,100: setscales
```

When deleting items or cardnames, it is not necessary to include the precedence values. A menu item given by

```
menu MenuCard, 70; LastItem,100: someunit
```

could be deleted with

```
menu MenuCard, LastItem $$ delete item
menu MenuCard $$ delete card
```

In order to control the placement of program-created menu items with respect to system-created menu items, you need to know the precedence values assigned by the system:

```
20 Paste
25 Cut
30 Copy
35 Replace
40 Enter Response -- at an -arrow-
96 (Next Page) -- end of unit; -next- active
97 (Proceed) -- at a -pause-
98 (Back) -- command -back- active
99 Quit Running
```

*For the Macintosh ONLY:* An item title that begins with a dash always puts a line of dashes on the menu. No other menu-creation metacharacters do anything.

```
menu -dashedline; dummyunit
```

**See Also:**
Simple Menu Commands (p. 136)
Passing Menu Parameters (p. 139)
Menu Like -do- (p. 140)

### Passing Menu Parameters

An optional parameter may be given in parentheses after the unitname. This parameter is evaluated and stored at the time the `-menu- command is executed. When the item is chosen from the menu, this saved parameter value is passed by value to the unit. Only one parameter is allowed, and it must be integer or float (not a marker or a file).

```
menu Draw a Circle: figures(1)
menu Draw a Square: figures(2)

... unit figures(N)
i: N
```
Because the parameter is evaluated at the time the -menu- item is added to the menu, you cannot have a dynamic parameter that passes different values to the executed unit depending on user actions. If you want a parameter that changes with user actions, you must use a global variable and keep updating its value.

Example:

This example is another version of the example given with "Simple Menu Commands." Instead of using a separate unit for each menu selection, a parameter is used to select the action from one general-purpose unit.

```
unit xmenupp     $$ use "Run from Selected Unit"
next xmenupp
menu Displays; Draw a Triangle: drawit(1)
menu Displays; Draw a Square: drawit(2)
menu Clear Screen: drawit(3)
at 50,50
write You can use the menu to select a drawing.
*
unit drawit(choice) $$ make drawings
  f: choice
case choice
  1    $$ triangle
       . draw 25,150; 100,100; 150,200; 25,150
  2    $$ square
       . box 175,100; 250,175
  3    $$ clear screen
       . erase
endcase
*
```

See Also:
Simple Menu Commands (p. 136)
Menu Like -do- (p. 140)

Menu Like -do-

When a menu item is selected, the associated unit is executed as it would be with a -do- command. Control of the flow of execution is not changed. If the program was awaiting a user response, it is still waiting after the menu unit has been executed. If the program was waiting at a -pause-, it returns to wait at the -pause-.

Sometimes menu items are used to select an entirely different part of the program. In that case, a -jump- is needed instead of a -do-. This must be faked by executing an intermediary unit that executes a -jump-.

```
menu Section 1: jsection1
menu Section 2: jsection2
.
unit jsection1
jump section1
*
unit jsection2
jump section2
```
When the user selects "Section 1", the unit "jsection1" is executed with a -do-. However, the -jump- command causes control to move to unit "section1" and cancels all previous connections.

The same effect can be achieved more compactly by using -menu- commands with parameters:

```
menu Section 1: jumper(1)
menu Section 2: jumper(2)
menu Section 3: jumper(3)
```

For more compact, use -menu- commands with parameters:

```
menu <|s,card|>,pg;<|s,item|>,pos: (unitname)(4)
```

The -menu- above is equivalent to

```
menu Graphing,3;Reset X-Scale,25: opts(4)
```

The variables for card name and item title must be embedded because the -menu- command is expecting text in those positions. Variables can be combined to form the item title or card name:

```
menu <|s,item|> to <|s,pos|>: scale25
menu <|s,card|> page <|s,pg|>: Bigger: big
```

These commands make an item "Reset X-Scale to 3" on the top menu, and an item "Bigger" on a card labeled "Graphing page 3".

See Also:
- Embedding Variables in Text (p. 50)
- Using Embedded Marker Variables (p. 251)
Buttons, Dialog Boxes, Sliders, & Edit Panels

button: Buttons to Click

The -button- command creates a button object that the user can click, and this does a specified unit. The -button- command is rather like an "on-screen" -menu- command. The -touch- command is somewhat similar to a -button- command but does not display anything itself.

Before a -button- command may be used, a corresponding button variable must be defined. This variable identifies the button in subsequent commands:

```
define button: mybutton
```

Like file variables, button variables can be passed by address to subroutines. Two button variables can be compared with "=" or "~=".

The basic form of the -button- command is

```
button variable; rectangle; unit; text; keyword; keyword; ...
```

```
button mybutton; 65,90; 130,115; Proceed; "Go On"; radio
```

The "rectangle" defines the area of the screen containing the button; "unit" specifies the unit to be done when the button is hit (this unit may have a single integer or float pass-by-value argument, just like the unit in a -menu- command); and "text" is a string to be displayed in or with the button. There are also -rbutton- and -gbutton- commands that use relative or graphing coordinates.

If no unit is associated with the button, the unit name must be given as x:

```
button mybutton; 10,20; 150,60; x; "Click Here"
```

Normally the text in a button is displayed in a standard system font, and styles such as subscript or italic are ignored, but with -inhibit buttonfont- in effect, the button text is displayed in the current font set by a -font- or -fontp- command, and the text can contain styles. It can even contain pasted-in images.

Keyword options

The keywords that specify a button's basic properties are

```
check a check box
radio a "radio" button
3d three-dimensional appearance
value "value,TRUE" initializes zvalue() to TRUE
erase indicates that the button should be erased when destroyed
```

If no button style is specified, the button will look like a standard button on that particular computer (and therefore may look different on different computers).

For example, the following puts up a button that has "Press Me" next to a check box, and specifies unit "Action" as the unit to be done when the button is pressed.

```
define button: me

button me; 30,40; 120,70; Action; "Press Me"; check
```
Any of the keywords can be accompanied by an expression that is TRUE or FALSE, to enable or disable that feature. For example, in the following -button- command the erase option will be activated if the variable named "EraseVar" is TRUE:

```
button me; 30,40; 120,70; Action; "Press Me"; erase, EraseVar
```

**zbutton and zvalue**

When a button is clicked on by the user, the system variable `zbutton` is set to be equivalent to the corresponding button variable; `zbutton` doesn't change until another button is clicked on, or the button is destroyed.

Each button has a value associated with it that is returned by the "zvalue()" function: for example, zvalue(mybutton) or zvalue(zbutton). By default this value is FALSE when the button is created, and toggles between FALSE and TRUE each time the button is hit. The value can be initialized to TRUE with "value,TRUE".

You can change the button value with a "reset" version of the -button- command, as in the following example:

```
button reset, me; value, FALSE
```

**Destroying a button**

A button may be destroyed by a -button- command with no keywords:

```
button me
```

Buttons are also destroyed by -jump-, a new main unit, or when a local button variable becomes invalid upon leaving a subroutine. If the button was created with the "erase" keyword, the button is automatically erased when it is destroyed, and that area of the screen is cleared to the window color that was in effect at the time the button was created. If the "erase" keyword is not used, a button is not erased when it is destroyed (but it stops accepting mouse clicks).

**Locking out additional events**

When a specified -button- unit begins executing, cT does not process any other -button-, -slider-, and -edit- events until the end of that unit is reached, or until there is a -pause-, -getkey-, -arrow-, or -jump- command. This blocking of other graphics objects prevents possible confusion from a second event interrupting the unit and do-ing the same unit, although such duplication can occur if you have a -pause-, -getkey-, -arrow-, or -jump- command in the do-ne unit. You may need to set and check a global variable to guard against duplication. Note also that -menu- events can take place during the processing of -button-, -slider-, and -edit- events.

**If unable to create a button**

If not all of a button would be visible due to part of the button being off-screen or clipped, the button is not created, and `zreturn` is set to 2. Once a button has been successfully created, it is unaffected by later -clip- commands: clicking a button temporarily removes the clip in order to highlight the button.

If a button object can be created successfully, the system variable `zreturn` is set to TRUE (-1). If it cannot be created due to lack of computer memory, `zreturn` is set to 1. An attempt to reset the value of an inactive button sets `zreturn` to 3.

Fake mouse clicks generated by a -press- command are ignored by buttons.
Examples:

```
unit xbutton
  button: CircleButton, BoxButton
button CircleButton; 30,40; 120,70; xDrawCircle(15); "Circle"
button BoxButton; 65,90; 130,115; xDrawBox; "Box"; radio
pause
*
unit xDrawCircle(r)
  i: r
mode xor
at 150,55
circle r
*
unit xDrawBox
mode xor
box 60,85; 135,120
*
```

The next example shows how to wait at a -pause- for a button to be clicked, rather than having a unit be do-ne.
You could also get out of the loop with -outloop zvalue(waitb)-, since zvalue(waitb) is initially FALSE when
the button is created and becomes TRUE when the user clicks on the button.

```
unit xbutton2
  button: waitb
at 10,10
write Waiting.....
button waitb;40,45;90,60; x; "Next"
loop
  pause .1 $$ wait a short time
  outloop zbutton = waitb
endloop
at 10,85
write After the click.....
*
```

See Also:

dialog Dialog Box (p. 144)
touch Touch Regions (p. 129)
Scrolling Text Panels (p. 149)
slider Slider or Scrollbar (p. 146)
Summary of Menu Formats (p. 136)
inhibit/allow buttonfont (p. 69)
clip Limiting the Display Area (p. 63)

dialog: Dialog Box

The -dialog- command brings up a "dialog box" for the user to interact with. There are three forms, specified by
the initial key phrases "ok", "yes no cancel", or "input":

```
dialog ok, "Search completed." $$ displays text and an "Ok" button
dialog yes no cancel, "Continue?" $$ Yes/No/Cancel buttons
dialog input, "Add last name: ", "Elvis ", marker $$ get input
```
dialog page setup $$ choose portrait/landscape printing
dialog print $$ choose pages to print, etc.

The "ok" form brings up a simple dialog box with the specified text (which can be a marker expression) and an "Ok" button. The system variable \texttt{zretinf} is set to 1 if the dialog executes successfully.

The "yes no cancel" form brings up a dialog box with the specified text and three buttons for "Yes", "No", and "Cancel". The system variable \texttt{zretinf} is set to 1 for yes, 2 for no, and 3 for cancel.

The "input" form brings up a dialog box with the specified text, an edit panel into which the user can enter text, and "Ok" and "Cancel" buttons. The third argument of the tag is initial text to appear in the edit panel. If you want the edit panel to be blank, just specify \texttt{zempty} as the third argument. The last argument is a marker variable that will be set to the user input (including the initial text, if any). The system variable \texttt{zretinf} is set to 1 for ok, 2 for cancel.

The print-oriented dialog boxes behave in a similar way. If during this session there has not been a -dialog page setup- statement execute, -dialog print- automatically brings up the page setup dialog box before bringing up the regular print dialog box. For more information, see the section on printing.

For all -dialog- commands, \texttt{zreturn} is set as for -file- commands. The possible failures are not enough memory (\texttt{zreturn} = 18), window not big enough (\texttt{zreturn} = 19), and window not being the forward-most window (\texttt{zreturn} = 21). Concerning the latter situation, the system variable \texttt{zforeground} is TRUE if the execution window is fully visible, in front of all other windows.

Styles in a marker expression in a dialog command are ignored (just as in a -button- command).

\textit{Example:}

\begin{verbatim}
unit xdialog
marker: name
dialog ok, "Click ok to continue"  $$ "ok" is only option
if ~zreturn
  at 10,10
  write Need larger window.
endif
at 10,10
write You clicked ok.
dialog yes no cancel, "Are you ready?"
at 10,30
write \zetinf-2\yes\no\cancel
* \texttt{zempty} in the following means that the initial text is empty:
dialog input, "Type your name:", \texttt{zempty}, name
at 10,50
write \zetinf-2\ok\cancel
at 10,70
show name
*
\end{verbatim}

\textit{See Also:}
button Buttons to Click (p. 142)
Printing (p. 14)
MOUSE & KEYSET INTERACTIONS

**slider: Slider or Scrollbar**

The -slider- command creates a slider or scrollbar object that the user can adjust, and this does use a specified unit, which can use the value corresponding to the new slider position. Before a -slider- command may be used, a corresponding slider variable must be defined. This variable identifies the slider in subsequent commands:

```plaintext
define slider: myslider
```

As with file variables, slider variables can be passed by address to subroutines. Two slider variables can be compared with "=" or "~=".

The form of the -slider- command is

```plaintext
slider var; rectangle; unit; keyword, arg(s); keyword, arg(s); ...
```

```plaintext
slider myslider; 65,90; 130,115; MoveIt
```

The "rectangle" defines the area of the screen containing the slider, and the unit specifies a unit to be done when the slider is adjusted (this unit may have a single integer or float pass-by-value argument, just as with the unit in a -menu- command). There are also -rslider- and -gslider- commands that use relative or graphing coordinates.

If no unit is associated with the slider, the unit name must be given as x:

```plaintext
slider myslider; 10,20; 150,60; x; horizontal
```

**Keyword options**

The keywords that specify a slider's basic properties are currently:

- **horizontal** horizontal rather than vertical slider
- **value** specify initial value: e.g., value,15
- **range** specify range: e.g., range,10,20
  (default range is 0 to 100)
- **page** specify the change in value corresponding to "page" scrolling; e.g., page,25
- **line** specify the change in value corresponding to "line" scrolling; e.g., line,2
- **erase** indicates that the slider should be erased when destroyed

The keywords page and line refer to the change in value reported by the slider when the user clicks in the scrollbar to move one "page" (clicking above or below the sliding box) or one "line" (clicking in the end of the scrollbar).

The keyword "erase" can be accompanied by an expression that is TRUE or FALSE, to enable or disable that feature. For example, in the following -slider- command the erase option will be activated if the variable named "EraseVar" is TRUE:

```plaintext
slider myslider; 10,20; 150,60; x; erase, EraseVar
```

**zslider and zvalue**

When a slider is adjusted by the user, the system variable zslider is set to be equivalent to the corresponding slider variable; zslider doesn't change until another slider is adjusted, or the slider is destroyed.

Every time the user adjusts the slider (by dragging with the mouse), the associated unit is done, and the zvalue() function provides the current slider value. For example, zvalue(myslider) or zvalue(zslider) is the
current numerical value of the position of the specified slider. Slider values run from bottom to top of a vertical slider, and left to right for a horizontal slider.

You can change any property of an existing slider with a "reset" version of the -slider- command, as in the following example:

```
slider reset, myslider; value, x+10y
```

**Destroying a slider**

A slider may be destroyed by a -slider- command with no keywords:

```
slider me
```

Sliders are also destroyed by -jump-, a new main unit, or when a local slider variable becomes invalid upon leaving a subroutine. If the slider was created with the "erase" keyword, the slider is automatically erased when it is destroyed, and that area of the screen is cleared to the window color that was in effect at the time the slider was created. If the "erase" keyword is not used, a slider is not erased when it is destroyed (but it stops accepting mouse inputs).

**Locking out additional events**

When a specified -slider- unit begins executing, cT does not process any other -button-, -slider-, and -edit- events until the end of that unit is reached, or until there is a -pause-, -getkey-, -arrow-, or -jump- command. This blocking of other graphics objects prevents possible confusion from a second event interrupting the unit and doing the same unit, although such duplication can occur if you have a -pause-, -getkey-, -arrow-, or -jump- command in the do-ne unit. You may need to set and check a global variable to guard against duplication. Note also that -menu- events can take place during the processing of -button-, -slider-, and -edit- events.

The system variable `zleftdown` or `zrightdown` is TRUE if the left or right mouse button is currently held down, and this makes it possible to wait to perform some action until the user has finished moving the slider and released the mouse button.

**If unable to create a slider**

If not all of a slider would be visible due to part of the slider being off screen or clipped, the slider is *not* created, and `zreturn` is set to 2. Once a slider has been successfully created, it is unaffected by later -clip- commands: clicking a slider temporarily removes the clip in order to manipulate the slider.

If a slider object can be created successfully, the system variable `zreturn` is set to TRUE (-1). If it cannot be created due to lack of computer memory, `zreturn` is set to 1. An attempt to reset the value of an inactive slider sets `zreturn` to 3.

Fake mouse events generated by a -press- command are ignored by sliders.

**Example:**

```
unit xSlider
    slider: SpeedAdjust
    f: angle
    at 3,3
data Adjust the rotation speed:
    slider SpeedAdjust; 10,20; 25,140; x; range, -20,+20; value, 10
    rorigin 110,80
    rotate angle := 0  $$ initialize angle
    mode xor
loop
```
rdraw 60,0; -40,-20; -40,20; 60,0
pause 0.1
rdraw 60,0; -40,-20; -40,20; 60,0
rotate angle := angle+zvalue(SpeedAdjust)
endloop
*

See Also:
button Buttons to Click (p. 142)
Scrolling Text Panels (p. 149)
clip Limiting the Display Area (p. 63)
Scrolling Text Panels

edit: Creating a Text Panel

The -edit- command displays editable text that the user can manipulate (scroll through the text, copy parts of the text, etc.). Before an -edit- command may be used a corresponding edit variable must be defined. This variable identifies the edit panel in subsequent commands:

```
define edit: myedit
```

Just as with file variables, edit variables can be passed by address to subroutines. Two edit variables can be compared with "=" or "-=".

The form of the -edit- command is

```
edit variable; rectangle; unit; text; keyword; keyword; ...
```

```
edit myedit; 35,135; 120,235; DoHot; m1; vscroll; editable
```

The "rectangle" defines the area of the screen containing the text panel, the specified unit is associated with "hot text" (discussed later), and "text" is the string to be displayed in the edit panel. (The unit may have a single integer or float pass-by-value argument, just as with the unit in a -menu- command). There are also -redit- and -gedit- commands that use relative or graphing coordinates.

If no unit for treating "hot text" is associated with the edit panel, the unit name must be given as x:

```
edit myedit; 10,20; 150,60; x; "Click Here"; hscroll
vscroll; single select  $$ can be continued onto a following line
```

Keyword options

The keywords that specify an edit panel's basic properties are:

- `hscroll`: edit panel should have a horizontal scroll bar
- `vscroll`: edit panel should have a vertical scroll bar
- `editable`: text in edit panel may be altered by the user (the edit marker must be changeable)
- `frame`: draws a box around the edit panel (normally TRUE)
- `erase`: edit panel should be erased when destroyed
- `visible`: specify portion of text to be visible in the panel; e.g., visible,m1
- `select`: specify portion of text to be selected; e.g., select,m2
- `tab`: set tab width (tab,5 means 5-pixel tabs)
- `newline`: set newline height (newline,30 for example)
- `supsub`: set super/subscript shift (supsub,10 for example)
- `supsubadjust`: FALSE is like -inhibit supsubadjust-
- `leftmar`: set width of margin from left edge (default 3 pixels)
- `rightmar`: set width of margin from right edge (default 3 pixels)
- `single select`: TRUE causes a single click to act as a double-click
- `focus click`: FALSE causes the first click of an edit panel which does not currently have the focus to go directly to the panel, without having to click twice (once to focus, again to select)
- `highlight`: FALSE causes the edit panel not to highlight selected text
- `before event`: process events before they are sent to the edit panel (see below)
after event process events after they are sent to the edit panel (see below)

Any of the keywords other than "visible", "select", "before event", and "after event" can be accompanied by an expression that is TRUE or FALSE to enable or disable that feature, or a number for tab in absolute (-edit-), graphing (-redit-), or relative coordinates (-redit-). For example, in the following -edit- command the frame option will be deactivated:

```plaintext
edit myedit; 10,20; 150,60; x; "Click Here"; frame, FALSE
```

For the visible and select options, you specify a marker on the text displayed in the edit panel. The select option scrolls the text so that the selected region is visible in the edit panel, and if the edit panel is in focus (see the -focus- command) the selected text is highlighted. For the visible option, the first line showing in the edit panel will be the line that contains the start of the associated marker. Here is an example of an -edit- command that uses the visible option to make the last line of the text be visible in the edit panel:

```plaintext
edit myedit; 10,20; 150,60; x; m1; visible,zlast(m1)
```

You can change any property of an existing edit panel with a "reset" version of the -edit- command, as in the following example:

```plaintext
edit reset, me; select, marker12  $$ change selected text
```

Also, you can change the unit that handles "hot" text:

```plaintext
edit reset, me; hot, SomeUnit(3)  $$ change hot-text-handling unit
```

When an edit panel is active, the system variable `zedit` is set to be equivalent to the corresponding edit variable; `zedit` doesn't change until another edit panel is made active, or the edit panel is destroyed.

**Marker functions for regions of the text**

The "zeditsel()" function returns a marker on the current text selected by dragging the mouse: for example, `zeditsel(myedit)` or `zeditsel(zedit)`. 

The "zeditvis()" function returns a marker on the text currently visible within an edit panel: for example, `zeditvis(myedit)` or `zeditvis(zedit)`. 

The "zedittext()" function returns a marker on the entire text associated with an edit panel: for example, `zedittext(myedit)` or `zedittext(zedit)`. 

You cannot store into a region specified by `zeditsel(editvar)`, `zeditvis(editvar)`, or `zedittext(editvar)`. If you want to change such a region, first use one of these functions to obtain a marker variable bracketing the region:

```plaintext
calc m := zeditsel(zedit)
replace m, "hello"
```

**Changing text in the edit panel**

Changing the actual displayed text does not require using the "reset" option. Whenever you make a change in the edit panel's marker by using -replace, -append-, or -style-, the edit panel notices this and updates its display accordingly. If, however, you use -calc- or -string- to place the original marker around new text, the marker is no longer associated with the edit panel, and you can no longer affect the text displayed in the edit panel by executing -replace, -append-, or -style- on the original marker.
Which edit panel gets the input?
There is only one keyset but there can be several edit panels active on the screen. Which edit panel receives the keyset information can be switched by the user clicking in another panel, or by the program executing a -focus- command. If no edit panel has been selected for input, keyset inputs go to a -pause- or -arrow- command. Often it is appropriate to execute -focus myedit- right after the -edit- command, to make sure that the edit panel is already selected for input. See the topic "focus Focus on a Text Panel".

Hot text
See the topic "Hot Text" for an explanation of how to construct "hypertext" links, in which double-clicking a word or phrase in an edit panel triggers the execution of a unit.

Processing events yourself
See the topic "Before and After Edit-Panel Events" for details on how to intercept keyset and mouse events before and/or after an edit panel handles the events.

Destroying an edit panel
An edit object may be destroyed by an -edit- command with no keywords:

```
edit myedit
```

Edit panels are also destroyed by -jump-, a new main unit, or when a local edit variable becomes invalid upon leaving a subroutine. If the edit panel was created with the "erase" keyword, the edit panel is automatically erased when it is destroyed, and that area of the screen is cleared to the window color that was in effect at the time the edit panel was created. If the "erase" keyword is not used, an edit panel is not erased when it is destroyed (but it stops accepting inputs).

Locking out additional events
When a specified hot-text unit begins executing, cT does not process any other -button-, -slider-, and -edit- events until the end of that unit is reached, or until there is a -pause-, -getkey-, -arrow-, or -jump- command. This blocking of other graphics objects prevents possible confusion from a second event interrupting the unit and do-ing the same unit, although such duplication can occur if you have a -pause-, -getkey-, -arrow-, or -jump- command in the do-ne unit. You may need to set and check a global variable to guard against duplication. Note also that -menu- events can take place during the processing of -button-, -slider-, and -edit- events.

If unable to create an edit panel
If not all of an edit panel would be visible due to part of the panel being off-screen or clipped, the panel is not created, and zreturn is set to 2. Once an edit panel has been successfully created, it is unaffected by later -clip- commands: clicking an edit panel temporarily removes the clip in order to manipulate the text.

If an edit-panel object can be created successfully, the system variable zreturn is set to TRUE (-1). If it cannot be created due to lack of computer memory, zreturn is set to 1. An attempt to reset the value of an inactive edit panel sets zreturn to 3.

Fake mouse clicks generated by a -press- command are ignored by edit panels.

Examples:

```
unit xEditText
edit: Gettysburg
m: Gtext
at 6,6
write Select part of the text with the mouse:
string Gtext
```
Fourscore and seven years ago, our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

```
edit Gettysburg; 20,30; 200,140; x; Gtext; vscroll; editable
loop
    pause 0.5
    if zeditsel(Gettysburg) ~= zempty
    erase 35,160;200,245
    text 35,160;200,245
<s,zeditsel(Gettysburg)></s>
    endif
endloop
```

* This example uses the visible and select options.

```
unit xScrollText
i: wc
edit: testedit
m: edittxt,word

* Create edit panel, make 10th word visible at top of panel:
edit testedit; 10,10; 125,75; x; edittxt; vscroll; visible,word
focus testedit
pause 2
calc word := zstart(edittxt)
loop wc := 1, 10
    calc word := znextword(word)
endloop
style word; italic; blue $$ make 10th word italic and blue
```

See Also:
- Focus on a Text Panel (p. 153)
- Hot Text (p. 153)
- Before and After Edit-Panel Events (p. 154)
**focus: Focus on a Text Panel**

There is only one keyset but there can be several edit panels active on the screen. Which edit panel receives the keyset information can be switched by the user clicking in that panel. You can also force keyset information to go to a particular panel by executing a `focus` command.

```plaintext
define edit: myedit1, myedit2
...
edit myedit1; ...
edit myedit2; ...
...
focus myedit1 $$ send keyset input to myedit1 edit panel
...
focus $$ blank-tag means don't send keyset input to
$$ an edit panel (send to -pause- or -arrow-)
```

See Also:
- `edit` Creating a Text Panel (p. 149)

**Hot Text**

A powerful feature of edit panels is "hot text." It is possible to specify that a word or phrase appearing in an edit panel is "hot text," and when a user double-clicks on that word or phrase the unit specified in the `-edit-` command is done (a single click is sufficient if the "single select" option is specified with the `-edit-` command). This makes it possible to link together text, diagrams, and even video segments as an interconnected "hypertext."

To make a word or phrase be hot text, select the text in the program (typically in a `-string-` or `-calc-` command), just as though you were going to make that text be italic or bold. Then choose "Hot" from the Styles menu. You will be prompted to type a string to be associated with this hot text (at the present time, you cannot use Command-= when entering this text on the Macintosh). If the text is displayed with an `-edit-` command, the hot unit will be done whenever the user double-clicks on the hot text. A hot style and associated string can also be created with a `-style-` command. Currently the associated string must not exceed 80 characters.

The system function `zhotsel(myedit)` gives a marker on the hot text, which might include several words, whereas `zeditsel(myedit)` typically brackets just one word of the hot text that was double-clicked on.

The system function `zhotinfo(myedit)` returns the information associated with the last click of hot text in edit panel "myedit". You can also use `zhotinfo(anymarker)`, which will return the string associated with that marker, which need not be in an edit panel. If you use `zhotinfo(anymarker)` on a marker that has no hot text, you get `zempty`. You can also use `zhasstyle(anymarker, hot)` to check whether there is any hot text associated with a marker. If you use `zhotinfo(anymarker)` on a marker that has more than one piece of hot text, you get all the information, concatenated into one string.

You can change the unit that handles "hot" text:
Example:

```
unit xHotExample
edit: textvar
marker: sometext, double
string sometext
Here is some text displayed by the -edit- command. Double click the hot text and see what happens!
\calc
double := zsearch(sometext, "Double click")
\style
double; italic; red; hot,"This is powerful!"
\edit
textvar; 10,10; 220,150; HandleIt; sometext
\pause
*unit HandleIt
erase 10,160; 400,250
at 10,160
write The mouse selection is "<s,zeditsel(zedit)>",
the hot text itself is "<s,zhotsel(zedit)>",
and the associated string is "<s,zhotinfo(zedit)>".
*
```

See Also:
Creating a Text Panel (p. 149)

Before and After Edit-Panel Events

You can do your own special processing of keyset and mouse events before or after they are processed by the edit panel. Consider the following -edit- command:

```
edit myedit;10,10;100,100; x; text; before event,left down,Ebefore;
after event,key,Eafter
```

When there is a mouse "left down" event in the edit panel, unit "Ebefore" is done before the mouse event is sent on to the edit panel. When a key on the keyset is pressed, unit "Eafter" is done after the key has been processed by the edit panel.

With "before event" and "after event" you can specify "key", "left down", "left up", "right down", "right up", or "down move". A "before" or "after" unit can optionally have one integer or float pass-by-value argument, as in "Ebefore(5)".

The system variable `zeditkey` is set when a "before" or "after" unit is done, and it has the same interpretation as `zkey`, which is not reset by events that are sent to an edit panel.

The system variables `zeditx` and `zedity` give the current screen location of the cursor in the edit panel that currently is in focus.

The function `ztextat(edit-panel, x-position, y-position)` returns a marker on the character in the text of the specified edit panel within which is found the specified x,y position. If you specify an x,y position that is
outside of the edit panel, you will get a marker at the beginning or end of the line nearest the specified x,y position.

In a unit processing "before events" sent to an edit panel, a -cancel- command throws away the event, so that it is not sent to the edit panel:

```
cancel $$ blank tag; cancel edit-panel event
```

You can change the specifications for how to handle before or after events:

```
edit reset, myedit; before event,right down,NewUnit
```

In the sample programs distributed with cT, japan.t uses "before events" to let you type Japanese "Kanji" characters.

**Example:**

```
unit xEditEvents
  edit: ed1, ed2
  m: m1, m2
string m1
Click in either of these edit panels and observe the effects.
\string m2
The "Before" unit processes "left down" events.
\edit ed1; 15,30;150,80; x; m1; frame
  before event, left down, Before
\edit ed2;40,100;173,151; x; m2; frame
  before event, left down, Before
*
unit Before
  m: mtemp
  calc mtemp := ztextat(zedit,zeditx,zedity)
  style mtemp; bold; red
  edit reset, zedit; select, zstart(mtemp)
cancel *
```

**See Also:**
- Creating a Text Panel (p. 149)
- Focus on a Text Panel (p. 153)

**A File Editor Application**

Here is a basic file editor that uses the -edit- command and file commands to create, inspect, or modify styled text files. When editing a file you can apply bold and italic styles, and you could easily add more features. On Macintosh or Windows you can even paste in an image copied from another application. One useful change would be to fix up the editor to open several files at the same time, so you can copy and paste between two files.

An existing file may contain several sections of styled text if the file was generated by several -dataout-s of markers. The file editor shown below just reads the first section of such a file, but in the sample programs
distributed with cT there is a more complex version called editfile.t that keeps track of multiple sections by wrapping a marker around each of them, and when you save the file it writes the sections out one at a time so as to preserve the original structure of the file.

* The -define- must be in the IEU (initial entry unit),
* before the first -unit- command:

define
  group,files:
    file: fd $$ the file that is read and written
    m: text $$ the full text of the file
  group,styles:
    i: BOLD=1, ITALIC=2 $$ for unit xSetStyles

unit
  xInitial

menu File; Open file: xOpenFile

menu File; New file: xNewFile

at 10,10

write Choose Open file or New file on the File menu.

**********

unit xEdit

merge,files:

merge,styles:
  edit: display
  i: nn

menu File; Save file: xSaveFile

menu Styles; Bold: xSetStyle(BOLD)

menu Styles; Italic: xSetStyle(ITALIC)

edit display; 0,0;zxmax,zymax; x; text; hscroll; vscroll; editable

focus display

**********

unit xSetStyle(style)

i: style

merge,styles:

m: selection

calc selection := zeditSel(zedit)

case style

  BOLD
    style selection; bold

  ITALIC
    style selection; italic

dcase

**********

unit xOpenFile

merge,files:

i: nn

m: temp, last

setfile fd; zempty; rw; styled $$ open styled file

if zreturn
  calc text := zempty $$ delete all text

  datain fd; text

  jump xEdit
endif

**********

unit xNewFile

merge,files:
addfile fd; zempty; styled
if zreturn
    calc text := zempty $$ delete all text
    jump xEdit
endif

**********
unit xSaveFile
merge,files:
i: nn
reset fd; empty
dataout fd; text
*

See Also:
edit Creating a Text Panel (p. 149)
style Assigning Styles to Markers (p. 262)
addfile Create a File (p. 289)
setfile Select a File (p. 291)
datain Read Data from a File (p. 294)
dataout Write Data to a File (p. 298)
Sample Programs (p. 28)
Word & Number Input

Overview of Word & Number Input

Typed words and numbers can be input using an -arrow- command, and there is a suite of "response-judging" commands and options to analyze or evaluate the user input.

The response-judging commands do not stand alone. They all interact and the order of the commands is important. The basic structure is

```
arrow
  response-judging commands
endarrow
```

The -arrow- command displays a pointy symbol (">") to indicate that a response is expected. The user enters a response and presses ENTER. The response-judging commands interpret the user's response and report whether the response is "ok". The user may not proceed past the -endarrow- until his/her response is in a form judged acceptable by the program. However, the user can, at this point, select any of the options provided on a menu, such as Review or Get Help. It is up to the programmer to provide some means for the user to escape from an -arrow- if no response is accepted.

The -answer- and -wrong- commands are used when a simple phrase or sentence response is expected. These commands provide for synonymous or ignorable words.

The -ansv- command is used when the response is numerical or algebraic. When the response involves a number to be stored or a mathematical expression to be evaluated, the -compute- command is used.

The default rules for acceptable input may be modified with a -specs- command. For example, executing -specs okspell- allows misspelled input to be considered valid.

The suite of response-judging commands associated with -arrow- do not attempt to "understand" user input in the sense of artificial intelligence. Rather, they try to make it possible for the program author to interpret or evaluate a wide range of user inputs, assuming that the user is trying to make a sensible input.

The -arrow- and response-judging commands are useful for checking the validity of typed input, giving feedback about invalid input, and automatically looping until a valid input is obtained. For example, is an input number within a specified range? If not, remind the user about the limits on the input, and let the user change the number. In a language drill, did the user type the correct form of the past tense? If not, tell the user what's wrong and allow him or her to correct the input.

There are some situations where the -edit- command provides a more flexible way to enter keyset information, but without the built-in feedback and looping mechanisms provided by the -arrow- command.

See Also:

- arrow Soliciting a Response (p. 159)
- answer Expected Responses (p. 162)
- ansv Numerical Responses (p. 163)
- compute Storing and Evaluating Inputs (p. 165)
- Modifying Judging Defaults (p. 169)
- Scrolling Text Panels (p. 149)
Basic Judging Commands

arrow: Soliciting a Response

The -arrow- command is used to handle keyset inputs, and it causes several actions. It displays an arrow (a pointy symbol like this: >) on the screen to indicate to the user that a response is expected, prepares the program to receive the response, and places the choice (Enter Response) on the Option menu. The -endarrow- command marks the end of the program segment that is influenced by the -arrow-. Every -arrow- must be balanced by an -endarrow-.

```
arrow beginx,beginy
arrow beginx,beginy; endx,endy
arrow $$ may have blank tag
```

```
endarrow $$ never has a tag
```

When the user enters a response to an -arrow- command, it appears to the right of the arrow. While the user is entering a response, he/she can use the mouse to edit the input, to select a special region on the display, or to choose a menu item.

The tag of -arrow- is a screen position that specifies where the arrow will appear. The blank tag form of -arrow- plots the arrow at the current screen position. If the tag starts with a semicolon, the current screen position is used.

The tag is treated just like the tag of an -at- command in terms of setting margins for input (but the -arrow- resets the lower-right margin to the lower-right corner of the window for output). If the tag includes two positions, the two points together specify a bounding box for the space allowed for the user's response. The arrow always appears at the upper-left corner of the bounding box. The two-coordinate form should usually be used; otherwise, the automatic erasing associated with -arrow- may erase too large an area.

The display of the arrow can be suppressed with -inhibit arrow-. This inhibits only the visible prompt; it does not affect the other -arrow- actions.

A new -arrow- may be nested within a current arrow. The inner arrow must be completed before returning to the outer arrow.

The user's input is available as the system marker variable zarrown, and any portion of the input that has been selected with the mouse is available as the system marker variable zarrwsel.

Associated with every -arrow- is at least one "response handling" command. These commands determine whether the user's input is acceptable. The user cannot continue until a satisfactory input is given. In the example below, we will discuss in detail the -arrow- and its response handling commands.

Note: Another way to handle keyset inputs is with an -edit- command, especially if you aren't using -answer-, -ansv-, or similar response-handling commands. To handle mouse as well as keyset inputs at an -arrow-, you must use a -button- command, or execute -enable touch- before the -arrow- and limit the screen area of the -arrow- with the four-argument form, since mouse clicks inside the -arrow- region are taken to mean that the user is editing a response with the mouse.

Example:

```
unit xarrow
at 50,50
```
The user types his/her response. The response can be edited as normal text. When the response is complete, the user presses ENTER.

For the response "green," the comment "Excellent!" appears three lines below "green" and an "ok" is printed on the same line and following the 'green'. The user has given an acceptable response, so execution immediately continues after the -endarrow- and prints "after the -endarrow-".

> green ok

Excellent!

For the response "blue," the comment "That's the sky!" is printed three lines below "blue" and the word "no" is displayed after "blue". Program execution then waits for the user to correct the response by erasing "blue" and putting in another response.

For the response "yellow," there is no explicit treatment specified. The program searches as far as the -endarrow- but does not find "yellow," so "no" is displayed and the program waits for a user action.

After a response is marked "no," the user can correct it by pressing ENTER to erase the entire response, or by editing the previous response. As soon as the user takes an action, the comment about the response (the "response-contingent display") is automatically removed.

Execution will not proceed past the -endarrow- until the arrow is "satisfied," that is, until the response is "ok".

The -arrow- might be thought of as initiating a series of "if" statements:

```python
if the response is "green,"
    display "Excellent!"
    print "ok"
    continue after the -endarrow-
else if the response is "blue,"
    display "Try again."
    print "no"
    wait for user action
else
    print "no"
    wait for user action
```

Examples:

This example allows the user to type anything at all. (The default rules for an arrow do not allow blank responses.)
BASIC JUDGING COMMANDS

The example collects a number and uses it later. The compute command evaluates the response and stores it in "x". The system variable \texttt{zreturn} is TRUE if the response can be evaluated. The response-contingent write uses an "embedded show command".

This example shows a nested arrow. The second question is asked only if the user's first response is "two bits."

\textit{See Also:}

- \texttt{zarrowm}: Markers at an -arrow- (p. 256)
- \texttt{zarrowsel}: Selected Text at an -arrow- (p. 258)
- \texttt{zreturn}: The Status Variable (p. 332)
- \texttt{enable}: Allowing Mouse Input (p. 132)
- Pull-down Menus (p. 136)
- Judging System Variables (p. 330)
answer: Expected Responses

The -answer- command is a response-judging command that gives a response that is "acceptable" at an -arrow-. The -wrong- command gives a specific incorrect response. These commands must be between an -arrow- and an -endarrow-. One -arrow- may have many -answer- and -wrong- commands.

The (optional) indented lines that follow response judging commands are called "response-contingent" commands. The indented commands are executed only if the user's response matches the tag of the previous response-judging command.

```
answer      <I see a> [big large] dog
 .         write    woof-woof
wrong      <I see a big little> cat
wrong      the <|s,num|> bears $$ can use embeds
```

The -answer- command specifies an expected response. Ignorable words are enclosed in angle braces <>. Synonyms are enclosed in square braces []. Punctuation is optional unless special qualifications are given (see -specs-). Thus the -answer- command above is matched by any of these phrases:

```
I see a big dog!
big dog
```

Position is important, so that "dog big" is not correct. However, ignorable words may appear any place in the response, so these phrases would be correct:

```
a large I see a dog!
big, I see a dog!
```

Silly possibilities such as those above are often tolerated, because it is so important to allow flexibility in the user's response. The author must find a compromise between flexibility and silly responses that is appropriate for his or her program.

See the section on "Modifying Judging Defaults" for ways to specify exactly how the -answer- command should work. In particular, the -specs- command lets you specify that extra words are allowed, that spelling mistakes or different word order don't matter, that capitalization is optional, etc.

Examples:

The incorrect response of cat, puss, feline, or kitten gets an explicit comment. Any other unrecognized response is simply marked "no".

```
unit      xanswer   $$ use "Run from Selected Unit
next      xanswer   $$ ENTER to try this unit again
at        50,50
write     What animal says bow-wow?
arrow     50,100
answer    <the a> [dog hound canine]
```
. write Excellent!
wrong <the a> [cat puss feline kitten]
. write Cats say meow!
endarrow
*

The next example uses conditional commands (-write- and -answer-) to build a 5-question "drill."

unit xanswer2 $$ use "Run from Selected Unit"
i: Q
next xanswer2 $$ can repeat this unit
randu Q, 4 $$ select one of 4 questions
at 50,50
write \Q-2\ Where would you spend a "zloty"?
\What coin is "two bits"?
\What is the capital of South Dakota?
\Who wrote about Miss Marple?
arrow 90,90; 300,120
answer \Q-2\Poland\quarter\Pierre\<Agatha> Christie
endarrow
*

See Also:
Modifying Judging Defaults (p. 169)
Judging System Variables (p. 330)
ok Unexpected Responses (p. 164)
enable Allowing Mouse Input (p. 132)
Logical Operators (p. 201)
Conditional Commands (p. 18)

ansv: Numerical Responses

The -ansv- and -wrongv- commands analyze numerical responses at an -arrow-. (For evaluating and storing a numerical or algebraic input, rather than checking for specific values, see the -compute- command.)

ansv 29
ansv 50,5%
wrongv 50,10

The first argument of the tag is the value of the expected response. Any arithmetic expression that evaluates to the expected value is acceptable. The expression can be algebraic and involve user variables (variables defined with -define user:-).

The second tag is the tolerance -- that is, how much the user's response may vary from the expected response and still be acceptable. The tolerance may be expressed in percent or as a number.

Encountering an -ansv- or -wrongv- causes the program to try to numerically evaluate the user's response. If the response cannot be evaluated, the system variable zreturn is set to a number that tells why the response could not be evaluated, and execution continues to search for a matching response-judging command. (Refer to "zreturn The Status Variable"). If no command is found that matches the response, the response is marked "no".
The -ansv- and -wrongv- commands are affected by -specs noops- (no operators allowed, such as +), -specs novars- (no user variables allowed), and -specs okassign- (without this, assignment into user variables is not allowed).

Example:

```
unit xan sv $$ use "Run from Selected Unit"
xan sv $$ press ENTER to try this unit again
at 50,50
write How many states in the U.S.A.?
arow 100,100
ansv 50
wrongv 50,1
. write You're off by 1.
wrongv 50,10%
. write You're within 10%.
no not(zreturn) $$ could not evaluate the response
. write Please enter only a number.
no
. write Sorry, you're way off.
endarrow
*
```

See Also:
- zreturn The Status Variable (p. 332)
- compute Storing and Evaluating Inputs (p. 165)
- compute Computing with Marker Variables (p. 254)
- ok Unexpected Responses (p. 164)
- specs Specifying Special Options (p. 169)
- Plot Two User Functions Simultaneously (p. 281)

**ok: Unexpected Responses**

The -ok- and -no- commands are used when any response is correct (or incorrect, respectively). The -ok- command is frequently used when soliciting information such as "What is your name?" The -no- command can serve as a catch-all for responses that did not match any previous -answer- or -ansv- commands.

```
ok (x=7) | (zntries>3)
ok
no
```

The -ok- and -no- usually have blank tags. When a tag is present, it must be a true/false expression. If the expression evaluates to TRUE the command is matched.

Example:

```
unit xok $$ use "Run from Selected Unit"
f: age
next xok $$ "next" repeats this unit
at 50,50
write Enter your age:
arow 100,100
```
compute age
no zreturn ! = TRUE $ $ can't evaluate response
. write Please enter just a number.
ok 4 < age < 99
no
. write I think you are trying to fool me.
endarrow
*

See Also:
Logical Operators (p. 201)
Conditional Commands (p. 18)

**compute: Storing and Evaluating Inputs**

The -compute- command stores a number or numerically evaluates a mathematical expression. The number or expression may be one entered by the user, or it may be a string of characters generated by the program. Any variables in the expression must be defined as "user" variables at the start of the program.

```
compute variable $$ integer or floating-point variable
compute variable, string
```

The -compute- with only one argument evaluates the string in the input buffer zarrown, that is, the last response that the user typed. The result is placed in the variable named in the tag. The -compute- with two arguments is discussed in the section on Marker Variables. (Refer to "compute Computing with Marker Variables".)

If the string contains variables, these must have been defined as "user" variables (define user:) in the IEU of the same file (initial entry unit preceding the first -unit- command).

The program must be able to detect when the string given to the -compute- command cannot be evaluated. For example, "(3+4 /5" is not a coherent arithmetic statement because every opening parenthesis, "(", needs a closing parenthesis, ")". For this purpose, the -compute- command sets the system variable zreturn. If the string is well formed, zreturn has the value "TRUE". If the string is not acceptable, zreturn has a positive numeric value that specifies why the string cannot be evaluated. (Refer to "zreturn The Status Variable".)

If you want to allow the user to assign values to user variables you must explicitly say so with -specs okassign-. That is, if there is a user variable "z" the input "z := 5" will be given a bad zreturn value unless -specs okassign- is in effect. Note too that -specs noops- and -specs novars- can be used to disallow the use of other operators (such as +) or of user variables (such as z). All -specs- options are cleared by an -arrow- command but are maintained across main units (including execution of -jump-).

**Examples:**

In both examples, the first -compute- merely sets zreturn; at that point we only care about whether it is possible to evaluate the response. If the user has entered a function that cannot be evaluated, the response is marked "no".

```
unit xcompute $$ use "Run from Selected Unit"
f: result
next xcompute $$ ENTER repeats this unit
at 35,66
write Enter an even number.
```
In order for the second example to work, the variable "x" must be defined as a user variable with a -define-statement at the beginning of the program:

define user: $$ these two lines belong
f: x $$ in the IEU

unit xcompute2 $$ graph a user's function
merge.global:
f: y
$$ question & instructions:
at 20,20
write Enter a function of x:
at 50,70
write Try something like: 3sin(xDEG) + 2cos(2xDEG)
$$ prepare for graphing:
gorigin 80,190 $$ show the graph axes & labels
axes 0,.95; 300,95 $$ describe axes
scalex 360 $$ 360 at right end of x-axis
scaley 5 $$ 5 at top of y-axis
labelx 90,15 $$ label x-axis every 90
labely 1 $$ label y-axis every 1
$$ collect the function:
arrow 150,20; 430,50 $$ ask for a function
compute y $$ is function a legal one?
ok zreturn $$ if legal, continue past endarrow
endarrow
inhibit startdraw $$ fixes initialization of line
loop x := 0,360,5 $$ x = 0, 5, 10, 15, etc.
compute y $$ function value for each x
gdraw ;x,y $$ previous point to x,y
endloop
*

See Also:
zreturn The Status Variable (p. 332)
compute Computing with Marker Variables (p. 254)
znumeric Extract Number from Marker (p. 272)
Defining Variables (p. 190)
Basic Calculational Operations (p. 200)
User Variables (p. 198)
inhibit/allow startdraw (p. 66)
specs Specifying Special Options (p. 169)
Algebraic Responses

Algebraic responses can be treated in a clever way that makes the program appear "intelligent," even though it is not. The trick is evaluation of the user's response for some generated values of the variables.

Example:

In this example, random values are assigned to a and b. The user's response is evaluated; if its value is the same as \((a+b)/2\), which is the average of a and b, then the user's expression is correct. If the user's expression cannot be evaluated, the system variable \(z\text{return}\) is not TRUE. The system variable \(z\text{ntries}\) is used to give the user help after the third unsuccessful attempt.

The system variable \(z\text{opcnt}\) counts the number of arithmetic operations. If there are more than two operations (+ and /), a special comment is given. Note the response "0.5(a+b)" also has two operations, a plus and an implied multiplication. You can also use the system variable \(z\text{varcnt}\) to find out how many references to user variables there are.

These lines, which declare "user variables," must be placed at the beginning of your program:

\[
\begin{align*}
\text{define user:} & \quad \text{\$\$ these 2 lines belong} \\
\text{f: a,b} & \quad \text{\$\$ in the I EU} \\
\text{unit} & \quad \text{xalgebra} \\
\text{next} & \quad \text{xalgebra} \\
\text{text} & \quad 31,27;367,300 \\
\text{Write an expression for the} \\
\text{average of "a" and "b".} \\
\text{at} & \quad 50,80 \\
\text{write} & \quad \text{average =} \\
\text{arrow} & \quad \text{zwhere x,zwhere y; 357,400} \\
\text{randu} & \quad \text{a} & \quad \text{\$\$ assign random values to a & b:} \\
\text{randu} & \quad \text{b} & \quad \text{(0 < \text{a} < 1); (0 < \text{b} < 1)} \\
\text{ansv} & \quad \text{(a+b)/2} \\
\text{if} & \quad \text{z\text{opcnt} > 2 \quad \$\$ system variable} \\
\text{endif} & \quad \text{write} \\
\text{wrongv} & \quad \text{a*b/2} & \quad \text{The average does not} \\
\text{wrongv} & \quad \text{write} & \quad \text{require multiplication.} \\
\text{wrongv} & \quad \text{(a-b)/2} & \quad \text{The numbers should be} \\
\text{write} & \quad \text{write} & \quad \text{added, not subtracted.} \\
\text{no} & \quad \text{not(z\text{return})} & \quad \text{\$\$ can't evaluate} \\
\text{write} & \quad \text{write} & \quad \text{Use only the variables} \\
\quad \text{\"a\" and \"b\".} \\
\text{no} & \quad \text{zntries > 2} & \quad \text{\$\$ three tries, give answer} \\
\text{write} & \quad \text{write} & \quad \text{One correct answer} \\
\quad \text{is (a+b) / 2 .} \\
\text{endarrow} & \quad *
\end{align*}
\]
In the next example the user is asked to simplify an equation, and we must guard against identities such as "2+2=4". We treat "a" as the independent variable and calculate the correct corresponding value for "b". If the result is TRUE, we alter "b" and test again: an identity such as "2+2=4" will continue to be TRUE, but a correct equation will now be FALSE.

```
define user: $\text{these 2 lines belong}\\$ f: a,b $\text{in the IEU}$

unit xidentity $\text{use } "Run from Selected Unit"$
next xidentity $\text{"next" to repeat this unit}$
at 31,27
write Simplify the equation
3a-2b+5-a = 4b+8
* Possible answer is 2a-6b = 3 (4 operations)
arrow 50,80; 357,400
randu a $\text{$assign random values to a}$
calc b := (2a-3)/6 $\text{$correct value for b}$
ansv TRUE
randu a $\text{$change value of a}$
wrongv TRUE
write That is an identity.
if zopcnt > 4 $\text{$system variable}$
write That is not the simplest form.
judge wrong
endif
wrongv FALSE
write Your equation is false.
no not(zreturn) $\text{$can't evaluate}$
write Use only the variables "a" and "b".
no
write You must write an equation.
endarrow *
```

See Also:
- zreturn The Status Variable (p. 332)
- Defining Variables (p. 190)
- User Variables (p. 198)
- Logical Operators (p. 201)
Modifying Judging Defaults

Response Judging Defaults

Many default assumptions are provided with the response judging commands. All of these defaults can be modified to allow the author to customize all or some of the -arrow-s in the program.

The -arrow- assumes that a matching response must be found before execution can continue after the -endarrow-. The -judge- command modifies this requirement. The -arrow- and -judge- commands are used to modify -arrow- defaults in a systematic way.

In order to be "correct," the user's response must exactly match one of the responses provided by the author, except for punctuation. Any punctuation (. , ! ?) not explicitly mentioned by the author is optional. The -punc- command modifies this assumption. Other -specs- options permit misspelled words, extra words, optional capitalization, and free ordering of words.

Blank responses are ignored. That is, if the user selects Enter Response from the menu or presses ENTER when there is nothing typed after the arrow prompt, the request for judging is simply ignored. This can be modified with -allow blanks-.

Because the square and pointy brackets ([ ] < >) are used as part of the syntax of the -answer- command, those symbols cannot be part of the expected response in an -answer-. The -exact- command can be used if the brackets must be identified.

Response "markup" is provided on all responses judged by -answer- and -wrong-. The -nomark- command suppresses this markup. Other -specs- options, such as "okspell" modify the interpretation of the response and thus also modify the markups.

The markups show how the user's response differs from the -answer- or -wrong- command that it most nearly matches. In this respect, -wrong- commands must be used with some caution, because a user may succeed in "correcting" a response, only to find that it is a correctly entered incorrect response.

A word that is incorrectly spelled is changed to italics:
  a big grey elephant
An unrecognized word is changed to bold:
  a big grey skizzle elephant
A missing word is indicated by empty brackets:
  a [ ] grey elephant
A word out-of-order is preceded by a < - symbol:
  a [ ] grey <big elephant
A missing capital letter is marked by a ^ symbol:
  George ^washington

Word order is assessed left-to-right, so an out-of-order symbol is always paired with a missing-word symbol to its left.

specs: Specifying Special Options

The -specs- command is used to modify the default behavior of the response-judging commands.

  specs okspell, nookno $$ may combine tags
  specs clear $$ cancels previous -specs-
Here is a summary of the -specs- tags. Each tag is illustrated with an example below.

- **clear**: cancel any previous -specs-
  blank tag is synonymous to -clear-
- **nomark**: do not show errors in response
- **nookno**: do not display "ok" or "no"
- **noops**: no arithmetic operators allowed
- **nospell**: turn off spelling checker
- **novars**: no variables allowed
- **okassign**: permits user to use assignment (:=)
- **okcaps**: capitalization is optional
- **okextra**: extra words are okay
- **okorder**: words in response may be in any order
- **okspell**: consider misspelled words as correct
- **punc**: punctuation must be exactly as given

The -specs- command normally appears after an -arrow- command, because -arrow- resets all -specs- options. However, the **okassign**, **noops**, and **novars** options can be used to control -compute- in the absence of an -arrow-, and -specs- options are not reset across a change in main unit (e.g., with -jump-).

**Examples:**

In -unit xspecs1-, the tags **okextra** and **okcaps** allow extra words and make capitalization optional. Acceptable responses include

george washington
G. Washington
Washington George it was.

Another acceptable response, which illustrates the danger of using **okextra**, is: "It was not George Washington."

Multiple -specs- commands are cumulative. The two -specs- lines below could have been combined into one line.

```
unit xspecs1
at 50,50
write Who was the first president of the U.S.A.?
arrow 75,100
specs okextra $$ extra words ok
specs okcaps $$ capitalization optional
answer [george g] washington
endarrow *
```

The -specs- command does not have to appear before the first judging command. It may be inserted in the middle of judging, to change the judging requirements. In -unit xspecs2-, the exact spelling is looked for first. If that is not found, an approximate spelling is accepted.

```
unit xspecs2
at 50,50
write What is the chemical name for aspirin?
arrow 100,100
```
answer acetylsalicylic acid
write Excellent!
specs okspell
answer acetylsalicylic acid $$ arrive here if misspelled
write You've got the right idea.
The correct spelling is
\textit{acetylsalicylic acid}.

In general, the -answer- command does not pay any attention to punctuation marks added by the user. However, punctuation marks included by the author \textit{must} be included in the response. The \texttt{punc} tag requires the user's punctuation to exactly match the author's punctuation.

The \texttt{nomark} tag prevents the normal markups. In -unit xspecs3-, the words of the desired response are shown, so we assume that clues about spelling, order, and word order are not required. Since an exact response with punctuation is required, we don't want to clutter the response line with an "ok," therefore the -specs- also includes \texttt{nookno}.

\begin{verbatim}
unit xspecs3
at 50,50
write Put correct punctuation and
capitalization into this sentence:

\textit{stop do you have a ticket}

\end{verbatim}

\begin{verbatim}
arrow 80,120
specs punc, nomark, nookno
answer Stop! Do you have a ticket?
write Excellent!
specs clear $$ cancel previous -specs-
wrong Stop Do you have a ticket
write Your punctuation is incorrect.
endarrow *
\end{verbatim}

The \texttt{okorder} tag allows the words of the response to appear in any order.

\begin{verbatim}
unit xspecs4
at 50,50
write Name the five Great Lakes.
arrow 75,100
specs okorder
answer Erie Huron Michigan Superior Ontario
endarrow *
\end{verbatim}

The \texttt{noops} tag forbids the use of arithmetic operators (+ - * /) in a response. If an arithmetic operator is used, the -specs noops- causes evaluation to fail and \texttt{zreturn} is set. Special feedback for arithmetic operators in responses also may use \texttt{zopcnt}.

The \texttt{novars} tag forbids the use of variables in the response. If a variable is used, the -specs novars- causes evaluation to fail and \texttt{zreturn} is set. Special feedback for variables in responses also may use \texttt{zvarcnt}. 

171
The -compute- command in the following example is necessary, because it triggers evaluation of the user's response and sets the \texttt{zreturn}, \texttt{zvarcnt}, and \texttt{zopcnt} values. The -ansv- would also trigger this evaluation, but it needs to come last, after all the error cases have been eliminated. In "xspects5", the variable "temp" is used as a dummy variable because -compute- must have a place to store its result.

```
unit xspecs5 $$ use noops & novars
i: temp
at 100,50
write Suppose a =3

arrow 100,120
specs noops, novars
compute temp $$ evaluate response
zreturn = 0 $$ operators used
write You must not use any arithmetic operators.
no zreturn = 10 $$ -specs novars-
write You must not use any variables.
no not(zreturn) $$ catch other errors
write Cannot evaluate your response.
ansv 3
endarrow
```

The next example assumes that you have already defined user variables at the very beginning of the program:

```
define user: these lines belong
f: x, y, z $$ in the IEU
```
When drawing a graph or making calculations, the user may need to set certain values into variables. The assignment operation is not legal with -ansv- or -compute- unless the -specs- tag okassign is used. The -compute temp- is required to trigger evaluation of the user's response, so that zreturn will have meaningful information.

```
unit xspecs6 $$ previously defined: x, y, z
merge,global:
f: temp
next xspecs6 $$ "Run from Selected Unit"
calc x := y := z := 5
do CurrentValues $$ display x, y, z
at 50,100
write Enter new value (as, y:=9)
```

Enter "Q" to quit.
```
arrow 50,150
specs okassign $$ allow assignments (=)
anwer Q
compute temp $$ evaluate response
ok zreturn
do CurrentValues
judge ignore $$ cycles back to arrow
write Cannot evaluate
your response.
endarrow
```
```
unit CurrentValues
erase 50,50; 250,90
at 50,50
write Current values:
x=<s,x>; y=<s,y>; z=<s,z>
```

```
See Also:
Judging System Variables (p. 330)
Logical Operators (p. 201)
Conditional Commands (p. 18)
```

**iarrow: Arrow Initializations**

The -iarrow- command specifies a unit that will be do-ne immediately after each -arrow- command in the current main unit.

```
iarrow arrowunit
iarrow q $$ cancel any previous setting
```

This command is usually used only in specialized situations where the normal arrow defaults are not appropriate. It is often placed in the -imain- unit.

When -iarrow- is in effect, the execution of an arrow behaves as if the program were written like this:

```
arrow 100,100
    do arrowunit
```
The indented commands are done immediately after the arrow is encountered. When the user has entered a response, execution starts with the first nonindented line. If the first response was incorrect and a second response is tried, then execution again starts with the first nonindented line.

Example:

This example uses -iarrow- to write a reminder to the user under each arrow. The -ijudge- supplies a -specs okcaps- for each arrow. (This example would undoubtedly be poor design for a lesson, but it shows the sequence of execution.) To execute this example, select unit "Atitle" and "Run from Selected Unit".

```
unit Atitle
next Questions $$ press ENTER for next unit
text 0,50

A Title Page

Press ENTER to continue

\
imain mainunit
*
unit Questions
at 50,50
write What animal is "man's best friend?"
arrow zwherex+10,zwherey; 250,80
answer [dog hound puppy mutt]
endarrow $$ end of first arrow
at 50,150
write How many stars
  on the U.S. flag?
arrow zwherex+10,165; 250,180
answer fifty $$ answer with a word
ansv 50 $$ or with a number
endarrow $$ end of second arrow
*
unit mainunit
at 0,0
write using -iarrow- and -ijudge-
iarrow arrowsetups
ijudge judgesetups
*
unit arrowsetups
at 70, zwherey+25 $$ below arrow
write Please answer briefly!
*
unit judgesetups
specs okcaps
at 100,zwherey
write [now judging response]
*```
MODIFYING JUDGING DEFAULTS

See Also:

ijudge Judge Initializations (p. 175)
ifmatch After the Response (p. 181)
imain Modifying Every Unit (p. 232)

ijudge: Judge Initializations

The -ijudge- command specifies a unit that will be done each time the user presses a key that initiates response judging.

ijudge judgeunit
ijudge q $$ cancel any previous setting

This command is usually used only in specialized situations where the judging defaults are not appropriate. It is often placed in the -imain- unit.

When -ijudge- is in effect, the execution of an arrow behaves as if the program were written like this:

=> arrow 100,100
do judgeunit
specs nookno
answer hello

The pointer indicates where the -ijudge- unit is done as though there were a -do- command.

The example for -ijudge- is with the -iarrow- discussion.

See Also:

iarrow Arrow Initializations (p. 173)
ifmatch After the Response (p. 181)
imain Modifying Every Unit (p. 232)

eraseu: Erasing after a Response

The -eraseu- command specifies a unit that will be done when the user presses any key after a response has been judged "no".

eraseu someunit
eraseu q $$ cancel any previous setting

The -eraseu- command is used when the default erasing of the last response-contingent text is not sufficient. The -eraseu- may be placed in any position in the unit, so long as it is executed before the erasing is needed. The "q" tag cancels any previously set -eraseu- unit. The -eraseu- option is cleared at the end of a main unit.

Example:

In this example, if the user makes an error, a little diagram is drawn. The -eraseu- removes the diagram. Note that the erasing is not done until the user presses a key.

unit xeraseu
draw 100,25; 100,150; 200,150; 100,25
at 25,200
write What is this figure?
arrow \text{zwherex}+10,\text{zwherey}
answer right triangle
wrong triangle \text{\$ incomplete response}
. eraseu zapit \text{\$ set special erase}
. at 55,239
. write Your answer is not
. specific enough.

. What is the indicated angle?
. do diagram \text{\$ show angle}
endarrow

* unit zapit \text{\$ remove diagram}
mode erase \text{\$ write with "white dots"}
do diagram \text{\$ redisplay diagram}
mode write \text{\$ return to "black dots"}
* unit diagram \text{\$ right angle illustration}
draw 101,133; 117,133; 117,149
draw 101,134; 116,134; 116,149
vector 24,176; 93,155
* 

See Also:
Logical Operators (p. 201)
Conditional Commands (p. 18)
Inhibit and Allow in Judging

-inhibit- and -allow- in Judging

The -inhibit- and -allow- commands modify various default behaviors. The tag is a keyword naming the behavior to be modified.

```
inhibit  startdraw
inhibit  arrow, blanks  $$ combined keywords ok
```

```
inhibit  $$ blank tag; reset defaults
allow   $$ blank tag; reset defaults
```

There may be several -inhibit- and/or -allow- commands in one unit; the results are cumulative. Several keywords may be combined in one tag. The following keywords are available as tags for -inhibit- with respect to judging (in addition to the inhibit options startdraw, display, update, objdelete, supsubadjust, and degree for graphics):

```
ansererase  $$ do not erase answer comment
arrow      $$ do not display the arrow symbol
blanks     $$ do not allow blank arrow inputs
```

When these tags are used with -allow- all of the "do nots" are changed to "do". The effect of an -inhibit- or -allow- lasts until it is canceled by one of these actions:

1) a blank-tag -inhibit- or -allow-
2) a specific -inhibit- or -allow- command
3) the beginning of a new main unit

The blank-tag -inhibit- and the blank-tag -allow- have the same effect. All of their options are changed back to the default status, as if these commands had been executed:

```
inhibit  blanks
allow   ansererase, arrow, erase, startdraw, display
allow   update, objdelete, supsubadjust, degree
```

The keyword reset can be used to reset to the default options:

```
inhibit  reset, arrow  $$ set to defaults, then -inhibit arrow-
```

See Also:
-inhibit- and -allow- in Graphics (p. 65)

inhibit/allow ansererase

This command modifies the treatment of response-contingent writing. The keyword ansererase ("answer-erase") refers to the automatic erasure of comments, which normally happens when the user modifies her or his response. The default status is -allow ansererase-.

When the user enters a response at an -arrow-, the response is judged and any processing associated with the matched response (response-contingent commands) is executed. Typically, this is a comment, such as "No. You
multiplied instead of adding.” As soon as the user starts to modify her/his response, the last response-contingent writing is automatically erased. The `-inhibit anserase- prevents this automatic erasure.

**Example:**

```
unit xanserase
at 50,50
write How many stars
on the U.S. flag?
arrow 50,100; 200,120
inhibit anserase
answer fifty
ansv 50
no
write The number of stars is the same
as the number of states.
endarrow *
```

**See Also:**
- Overview of Word & Number Input (p. 158)
- arrow Soliciting a Response (p. 159)
- eraseu Erasing after a Response (p. 175)

**inhibit/allow arrow**

This command modifies the `-arrow- display. The keyword `arrow` refers to the pointy ("arrow") symbol displayed by the `-arrow- command to indicate to the user that a response is required. The `-inhibit arrow- prevents display of the arrow symbol. It does not affect the operation of the `-arrow- command; only the symbol is affected. The default status is `-allow arrow-.

**Example:**

In this example, the normal arrow symbol is suppressed and instead a question mark is displayed at the position where the arrow would have appeared. The `-arrow- command allows space for its symbol and a blank space before the user's typing appears. Thus, the `-at- and the `-arrow- are positioned at the same place.

```
unit xInhibitArrow
at 100,75
write Please tell me your name.
at 100,100 $$ position of "?"
write ?
inhibit arrow $$ don't display ">
arrow 100,100
specs nookno $$ suppress the "ok"
ok
write Thank you.
endarrow *
```
**inhibit/allow blanks**

This command changes the requirement for user input at an -arrow-. The keyword **blanks** refers to the treatment of blank responses from the user. The default status is -inhibit blanks-.

Normally, just pressing ENTER is ignored if no response has been entered. An -allow blanks- lets the user initiate judging even though no response has been entered.

The -allow blanks- is very useful when preparing a series of numerical values, since "blank response" can be treated as "do not change the current value."

*Example:*

In this example, the -compute- tries to evaluate the response and sets **zreturn**. If the system variable **zjcount** is zero, indicating a blank response, the judgment is always "ok".

If **zjcount** is greater than zero and **zreturn** is TRUE, the response is a number and the new value is passed back to N(i). If **zreturn** is FALSE, judging reaches the -endarrow- without having found an "ok," so the response is automatically "no".

```plaintext
unit xInhibitBlanks
f: i $$ index for array
f: N(5) $$ array of 5 values
i: exitflag $$ used to exit from loop
set N := 1,2,3,4,5 $$ set "current" values
calc exitflag := FALSE $$ "don't exit"
loop $$ cycles over and over
  do display( ;N) $$ show current values
  loop i := 1, 5 $$ get 5 values
    do getnumber(i; N(i), exitflag)
  endloop
outloop exitflag = TRUE $$ user chose "Q"
endloop
erase 50,200;242,267
at 120,200
write Finished.
*
unit getnumber(k; value, flag)
i: k $$ index of array
f: value, temp $$ "quit"
i: flag $$ exit flag
at 186,76
arrow 175,90+15k; 250,120+15k
allow blanks
specs nookno
answer [q, Q, quit] $$ "quit"
calc flag := TRUE
ok zjcount = 0 $$ blank response
compute temp $$ try to find new value
ok zreturn $$ new value ok
calc value := temp
no at 140,110+15k
```
write I don't understand.

endarrow
*

unit
display( ;nn)
i: k
f: nn(*) $$ note index is given as *
erase $$ clear display
box
at 50,30
write Type the new number or
press ENTER to leave the value unchanged.
at 50,75
write Current Values   New Values
loop k := 1,5 $$ show current values
at 100,90+15k
show nn(k)
endloop
at 50,220
write Enter "Q" when you are finished.
*

See Also:
zreturn The Status Variable (p. 332)
judge Changing the Judgment (p. 182)
Logical Operators (p. 201)
Using Arrays(p. 208)
**Specialized Judging Commands**

**exact: Exact Responses**

The -exact- and -exactw- are response-judging commands. They are used when no flexibility is allowed in the user's response. To match an -exact- or -exactw-, the user's response must *exactly* match the given tag, including spaces and punctuation marks.

```
exact 3A 427/9/5 abc
exactw 3a 427/9/5 ab
```

The -exact- and -exactw- commands do not give any "response markup" feedback. It can be very frustrating to receive a "no" response with no clue about why it is incorrect. In most circumstances, the -exact- and -exactw- are too restrictive.

*Example:*

```
unit xexact
at 50,50
write Type this string:
     Sally sells seeshells at the seashore.
arrow 75,140
exact Sally sells seeshells at the seashore.
endarrow
*
```

**ifmatch: After the Response**

The -ifmatch- command provides a way to do special processing *after* a response is judged but *before* the -endarrow- is reached. It is an exception to the rule "the indented commands after a matched response are executed, and then execution *immediately* proceeds after the -endarrow- or returns to the -arrow- for another attempt." When a response is matched (whether correct or incorrect), the indented commands following that response are executed, *any indented commands following the -ifmatch- are executed*, and only then does execution return to the -arrow- or continue after the -endarrow-.

```
ifmatch $$ has no tag
```

The -ifmatch- and the indented commands that follow it must come just before the -endarrow-. The -ifmatch-command has no tag. Only one -ifmatch- may appear in an -arrow-/endarrow- sequence.

Note that an -ok- or a -no- assures that *every* input is "matched."

*Example:*

In this example, rather than typing the comment after both the -answer- and the -ansv-., the comment is entered only once, after the -ifmatch-.

```
unit xifmatch
at 50,40
write How many states are in the USA?
arrow 50,90
```
See Also:

judge Judge Initializations (p. 175)
arrow Arrow Initializations (p. 173)
main Modifying Every Unit (p. 232)

judge: Changing the Judgment

The -judge- command allows the author to change the judgment given to a user's response. This is convenient when the response needs to be subjected to some analysis before a decision about its correctness is made. The system variable \zjudged gives the current status of response judging.

\begin{verbatim}
judge no
judge unjudge
judge \zreturn\ok\no
\end{verbatim}

The tag of -judge- is a keyword; possible keywords are as follows: no, ok, wrong, unjudge, exit, ignore, quit, okquit, noquit, exdent, rejudge.

The system word \zjudged is set by the response judging commands such as -answer- and -wrong-. It can be modified by the -judge- command. The \zjudged system variable has four possible values:

\begin{verbatim}
-1 after any response judged "ok"
0 after any response judged "wrong"
1 after any response judged "no"
2 when no response has been matched
and neither -endarrow- nor -ifmatch- has yet been reached
\end{verbatim}

The "no" and "wrong" responses give different \zjudged values because sometimes the author must distinguish between anticipated incorrect responses and unanticipated incorrect responses.

The following four tags for the -judge- command affect only the value of \zjudged:

\begin{verbatim}
ok sets \zjudged = -1
wrong sets \zjudged = 0
no sets \zjudged = 1
unjudge sets \zjudged = 2
\end{verbatim}

The following two tags cause processing to stop, set \zjudged=2, and return to the -arrow- to wait for further action from the user:

\begin{verbatim}
exit
ignore also erases the user's response
\end{verbatim}
The next three tags cause an immediate branch to the -ifmatch- command. Commands following the -ifmatch-
are then executed:

- **quit** after the -ifmatch-, continues to the
  next command after the -endarrow-
  even if the response was not "ok".
  zjudged is unchanged

- **okquit** sets zjudged=-1, continues after the
  -endarrow-

- **noquit** sets zjudged=1, returns to the -arrow-
  after the -ifmatch- is completed and
  waits for action from the user

The final two tags set **zjudged=2**, stop processing of response-contingent commands, and continue with the
next nonindented command:

- **exdent**
- **rejudge** initializes answer-judging system
  variables such as zntries and zanscnt

**Examples:**

```
unit xjudge
  f: number
  at 50,34
  write Enter an even number
  between 0 and 500.
  arrow 70,84
  compute number
  if not(zreturn)
    . write I do not understand.
    . judge no
  elseif number >500
    . write Your number is too big.
    . judge no
  elseif number <0
    . write Your number is too small.
    . judge no
  elseif frac(number/2) != 0
    . write That is not an even number.
    . judge wrong
  else
    . write Good!
    . judge ok
  endif
  endarrow
```

**See Also:**
- Judging System Variables (p. 330)
- Basic Judging Commands (p. 159)
- Conditional Commands (p. 18)
## Typing Non-English Text

To enter the following non-English characters press the "compose" key (currently, Command-= on Macintosh, and CTRL-z on Windows and Unix), followed by the two-character compose sequence. Your machine may already provide additional ways to type some of these characters. **Note that some of these characters are not available on a Macintosh unless the special "ISO" fonts distributed with cT are installed.**

<table>
<thead>
<tr>
<th>code</th>
<th>compose</th>
<th>regular font</th>
<th>zsymbol font</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>sp sp</td>
<td>No-Break Space</td>
<td>No-Break Space</td>
</tr>
<tr>
<td>161</td>
<td>!!</td>
<td>Inverted !</td>
<td>Capital Upsilon</td>
</tr>
<tr>
<td>162</td>
<td>c/</td>
<td>Cent sign</td>
<td>Long apostrophe</td>
</tr>
<tr>
<td>163</td>
<td>L-</td>
<td>Pound sign</td>
<td>Less than or equal, &lt;=</td>
</tr>
<tr>
<td>164</td>
<td>XO</td>
<td>Currency sign</td>
<td>Slash</td>
</tr>
<tr>
<td>165</td>
<td>Y-</td>
<td>Yen sign</td>
<td>Infinity</td>
</tr>
<tr>
<td>166</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>167</td>
<td>SO</td>
<td>Section sign</td>
<td>Club suit in cards</td>
</tr>
<tr>
<td>168</td>
<td>&quot;</td>
<td>Dieresis</td>
<td>Diamond suit in cards</td>
</tr>
<tr>
<td>169</td>
<td>co</td>
<td>Copyright sign</td>
<td>Heart suit in cards</td>
</tr>
<tr>
<td>170</td>
<td>a_</td>
<td>Fem. ordinal (Span.)</td>
<td>Spade suit in cards</td>
</tr>
<tr>
<td>171</td>
<td>&lt;&lt;</td>
<td>Left angle quotation</td>
<td>Two-ended hor. arrow</td>
</tr>
<tr>
<td>172</td>
<td>-</td>
<td>NOT sign</td>
<td>Left arrow</td>
</tr>
<tr>
<td>173</td>
<td>--</td>
<td>Soft hyphen</td>
<td>Up arrow</td>
</tr>
<tr>
<td>174</td>
<td>RO</td>
<td>Registered trade mark</td>
<td>Right arrow</td>
</tr>
<tr>
<td>175</td>
<td>^</td>
<td>Macron</td>
<td>Down arrow</td>
</tr>
<tr>
<td>176</td>
<td>0^</td>
<td>Ring above</td>
<td>degree sign</td>
</tr>
<tr>
<td>177</td>
<td>+</td>
<td>Plus-minus sign</td>
<td>Plus-minus sign</td>
</tr>
<tr>
<td>178</td>
<td>2</td>
<td>Superscript two</td>
<td>Right quote marks</td>
</tr>
<tr>
<td>179</td>
<td>3</td>
<td>Superscript three</td>
<td>Greater than or equal, &gt;=</td>
</tr>
<tr>
<td>180</td>
<td>&quot;</td>
<td>Acute accent</td>
<td>Multiplication sign</td>
</tr>
<tr>
<td>181</td>
<td>/u</td>
<td>Micro sign</td>
<td>Proportional to</td>
</tr>
<tr>
<td>182</td>
<td>P!</td>
<td>Paragraph sign</td>
<td>Partial derivative</td>
</tr>
<tr>
<td>183</td>
<td>^</td>
<td>Middle dot</td>
<td>Bullet</td>
</tr>
<tr>
<td>184</td>
<td>,</td>
<td>Cedilla</td>
<td>Quotient sign</td>
</tr>
<tr>
<td>185</td>
<td>1</td>
<td>Superscript one</td>
<td>Not equal, &lt;&gt;</td>
</tr>
<tr>
<td>186</td>
<td>o_</td>
<td>Masc. ordinal (Span.)</td>
<td>Equivalent; triple equal sign</td>
</tr>
<tr>
<td>187</td>
<td>&gt;&gt;</td>
<td>Right angle quotation</td>
<td>Approximately equal</td>
</tr>
<tr>
<td>188</td>
<td>14</td>
<td>Fraction one quarter</td>
<td>…</td>
</tr>
<tr>
<td>189</td>
<td>12</td>
<td>Fraction one half</td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>34</td>
<td>Fraction three quarters</td>
<td>Long horizontal bar</td>
</tr>
<tr>
<td>191</td>
<td>??</td>
<td>Inverted question mark</td>
<td>Enter; vertical bar + left arrow</td>
</tr>
<tr>
<td>192</td>
<td>A`</td>
<td>A + grave accent</td>
<td>Aleph</td>
</tr>
<tr>
<td>193</td>
<td>A'</td>
<td>A + acute accent</td>
<td>Hebrew math</td>
</tr>
<tr>
<td>194</td>
<td>A^</td>
<td>A + circumflex accent</td>
<td>Hebrew math</td>
</tr>
<tr>
<td>195</td>
<td>A~</td>
<td>A + tilde</td>
<td>Hebrew math</td>
</tr>
<tr>
<td>196</td>
<td>A&quot;</td>
<td>A + dieresis</td>
<td>Multiply in a circle</td>
</tr>
<tr>
<td>197</td>
<td>A*</td>
<td>A + a ring above</td>
<td>Add in a circle</td>
</tr>
<tr>
<td>198</td>
<td>AE</td>
<td>AE</td>
<td>O with slash through it</td>
</tr>
<tr>
<td>199</td>
<td>C,</td>
<td>C + cedilla</td>
<td>Intersection; upside-down U</td>
</tr>
<tr>
<td>200</td>
<td>E`</td>
<td>E + grave accent</td>
<td>Union; U</td>
</tr>
<tr>
<td>201</td>
<td>E'</td>
<td>E + acute accent</td>
<td>Contains</td>
</tr>
<tr>
<td>Code</td>
<td>Symbol</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>------------------------------------</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>E^</td>
<td>E + circumflex accent Contains and equal</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>E&quot;</td>
<td>E + dieresis Not contained in</td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>I`</td>
<td>I + grave accent Contained in</td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>I'</td>
<td>I + acute accent Contained in and equal</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>I^</td>
<td>I + circumflex accent Exists</td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>I&quot;</td>
<td>I + dieresis Not exists</td>
<td></td>
</tr>
<tr>
<td>208</td>
<td>D-</td>
<td>ETH (Icelandic) Angle</td>
<td></td>
</tr>
<tr>
<td>209</td>
<td>N~</td>
<td>N + tilde Del</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>O`</td>
<td>O + grave accent Register mark</td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>O'</td>
<td>O + acute accent Copyright</td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>O^</td>
<td>O + circumflex accent Trademark</td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>O~</td>
<td>O + tilde Pi or product</td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>O&quot;</td>
<td>O + dieresis Square root</td>
<td></td>
</tr>
<tr>
<td>215</td>
<td>xx</td>
<td>Multiplication sign Multiplication dot</td>
<td></td>
</tr>
<tr>
<td>216</td>
<td>O/</td>
<td>O + oblique stroke Horizontal line with hook</td>
<td></td>
</tr>
<tr>
<td>217</td>
<td>U`</td>
<td>U + grave accent Intersection; upside-down V</td>
<td></td>
</tr>
<tr>
<td>218</td>
<td>U'</td>
<td>U + acute accent Union; V</td>
<td></td>
</tr>
<tr>
<td>219</td>
<td>U^</td>
<td>U + circumflex Hor. two-headed double arrow</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>U&quot;</td>
<td>U + dieresis Left double arrow</td>
<td></td>
</tr>
<tr>
<td>221</td>
<td>Y`</td>
<td>Y + acute accent Up double arrow</td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>TH</td>
<td>THORN (Icelandic) Right double arrow</td>
<td></td>
</tr>
<tr>
<td>223</td>
<td>ss</td>
<td>German ss Down double arrow</td>
<td></td>
</tr>
<tr>
<td>224</td>
<td>a`</td>
<td>a + grave accent Open diamond</td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>a'</td>
<td>a + acute accent Left parenthesis</td>
<td></td>
</tr>
<tr>
<td>226</td>
<td>a^</td>
<td>a + circumflex accent Register mark</td>
<td></td>
</tr>
<tr>
<td>227</td>
<td>a~</td>
<td>a + tilde Copyright</td>
<td></td>
</tr>
<tr>
<td>228</td>
<td>a&quot;</td>
<td>a + dieresis Trademark</td>
<td></td>
</tr>
<tr>
<td>229</td>
<td>a*</td>
<td>a + a ring above Summation - sigma</td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>ae</td>
<td>æ</td>
<td></td>
</tr>
<tr>
<td>231</td>
<td>c,</td>
<td>c + cedilla</td>
<td></td>
</tr>
<tr>
<td>232</td>
<td>e`</td>
<td>e + grave accent</td>
<td></td>
</tr>
<tr>
<td>233</td>
<td>e'</td>
<td>e + acute accent</td>
<td></td>
</tr>
<tr>
<td>234</td>
<td>e^</td>
<td>e + circumflex accent</td>
<td></td>
</tr>
<tr>
<td>235</td>
<td>e&quot;</td>
<td>e + dieresis</td>
<td></td>
</tr>
<tr>
<td>236</td>
<td>i`</td>
<td>i + grave accent</td>
<td></td>
</tr>
<tr>
<td>237</td>
<td>i'</td>
<td>i + acute accent</td>
<td></td>
</tr>
<tr>
<td>238</td>
<td>i^</td>
<td>i + circumflex accent</td>
<td></td>
</tr>
<tr>
<td>239</td>
<td>i&quot;</td>
<td>i + dieresis</td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>d-</td>
<td>eth (Icelandic)</td>
<td></td>
</tr>
<tr>
<td>241</td>
<td>n~</td>
<td>n + tilde</td>
<td></td>
</tr>
<tr>
<td>242</td>
<td>o`</td>
<td>o + grave accent Integral sign</td>
<td></td>
</tr>
<tr>
<td>243</td>
<td>o'</td>
<td>o + acute accent</td>
<td></td>
</tr>
<tr>
<td>244</td>
<td>o^</td>
<td>o + circumflex accent</td>
<td></td>
</tr>
<tr>
<td>245</td>
<td>o~</td>
<td>o + tilde</td>
<td></td>
</tr>
<tr>
<td>246</td>
<td>o&quot;</td>
<td>o + dieresis</td>
<td></td>
</tr>
<tr>
<td>247</td>
<td>-:</td>
<td>Division sign</td>
<td></td>
</tr>
<tr>
<td>248</td>
<td>o/</td>
<td>o + oblique stroke</td>
<td></td>
</tr>
<tr>
<td>249</td>
<td>u`</td>
<td>u + grave accent</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>u'</td>
<td>u + acute accent</td>
<td></td>
</tr>
<tr>
<td>251</td>
<td>u^</td>
<td>u + circumflex accent</td>
<td></td>
</tr>
<tr>
<td>252</td>
<td>u&quot;</td>
<td>u + dieresis</td>
<td></td>
</tr>
</tbody>
</table>
Typing Special Function Keys

Not all keysets provide all the function keys available on large keysets. cT offers a way to compose the missing keys. First type the compose sequence (currently Command-= on a Macintosh, Control-z on other machines), then a semicolon (;), and finally a three-letter code that specifies the key:

- **fd-DEL**: Delete forward
- **bk-TAB**: KBACKTAB
- **esc**: Escape
- **???**: Help
- **fna**: Function key A
- **fnb**: Function key B
- **fnc**: Function key C
- **fnd**: Function key D
- **trn**: Transpose neighboring letters
- **lf-**: Left (and move selection)
- **lfe**: Left and extend selection
- **rt-**: Right (and move selection)
- **rte**: Right and extend selection
- **up-**: Up (and move selection)
- **upe**: Up and extend selection
- **dn-**: Down (and move selection)
- **dne**: Down and extend selection
- **bl-**: Beginning of line (and move selection)
- **ble**: Beginning of line and extend selection
- **el-**: End of line (and move selection)
- **ele**: End of line and extend selection
- **bp-**: Beginning of page (and move selection)
- **bpe**: Beginning of page and extend selection
- **ep-**: End of page (and move selection)
- **epe**: End of page and extend selection
- **bf-**: Beginning of file (and move selection)
- **bfe**: Beginning of file and extend selection
- **bfs**: Beginning of file and don't change selection
- **ef-**: End of file (and move selection)
- **efe**: End of file and extend selection
- **efs**: End of file and don't change selection
- **pu-**: Page up (and move selection)
- **pue**: Page up and extend selection
- **pus**: Page up and don't change selection
- **pd-**: Page down (and move selection)
- **pde**: Page down and don't change selection
- **pds**: Page down and don't change selection
- **cut**: Cut
- **cpy**: Copy
- **pst**: Paste
- **und**: Undo
5. Calculations

Calculation Introduction

The following introduction to cT calculations is very important if you have not written computer programs before, and it can usefully be skimmed by experienced programmers to see the basic syntax of cT numerical calculations.

The most important concepts involved in doing calculations are defining variables, using variables in calculations, using variables in repetitive loops, and making decisions based on variables. A variable is a region of computer memory in which is stored a number, a string of characters, or other information.

The -define- command declares variables; that is, it reserves regions of computer memory for storing information and gives those regions names of your choice, so that you can use meaningful names to refer to the information stored in those regions.

The -calc- command assigns a value to a variable; that is, this value is stored in the named region of memory. The value may be calculated from the current contents of other variables.

The -loop- command executes the same instructions repeatedly, with changing values of some variables.

The -if- command lets you conditionally execute some statements for particular values of variables.

Here is a simple example of these three basic concepts in action. Copy this into the beginning of your program window (because the -define- command must precede the first -unit- command) but after the $syntaxlevel line. Then choose "Run from beginning" from the Option menu. After running the program, study the comments below.

* Define "global" variables, accessible to all units:
define integer: Ncows, Nhorses, Nanimals
*
unit xcalcintro
merge.global: $$ refer to "global" variables
integer: index $$ "local" variable
calc Ncows := 3 $$ ":=" means "assign value"
Nhorses := 5
Nanimals := Ncows+Nhorses $$ 3+5
at 10,10
show Nanimals $$ this variable contains "8"

* Do repetitive loop for index := 1, 2, 3, ..., 10:
loop index := 1, 10
at 10,20+15*index
* Show number with 1 digit before
* the decimal point and 3 after:
showt sqrt(index),1.3
if index = 7 $$ if index equal to 7
write $$ Root of seven
endif
endloop
at 10,200
write Finished the loop.
*
The -define- command reserves three regions of memory (named "Ncows", "Nhorses", and "Nanimals") to hold integer values, which are negative or positive whole numbers ("define float:" is used to define "floating-point" variables that can contain fractional values). These are called "global" variables because they can be used in all the units in the program. The first line of the unit ("merge global:" ) declares that this unit will refer to the global variables created by the -define- command, and the second line declares a "local" variable named "index" that is accessible only to this unit. Other units in the program can refer to the global variables, but only unit "xcalcintro" can refer to its local variable "index". The variable definitions following the -unit- command are considered to be extensions of the -unit- command, and the command part of the lines is left blank.

The -calc- command assigns values to the global variables "Ncows" and "Nhorses", then uses the numbers stored in those regions of computer memory to calculate the sum of the two values, and assign that sum to "Nanimals". The sum is displayed with a -show- command, which displays a numerical value. (It is permissible to omit the command name "calc" on successive lines, and this may improve readability.)

The -loop- command lets you do repetitive operations with varying values of variables. In the example above, the local variable "index" is initially assigned the value 1, and the contents of the loop (the indented statements bracketed by the -loop- and -endloop- commands) are executed with this value. When the computer reaches the -endloop-, the program goes back to the top of the loop and "index" is incremented by 1, so the region of computer memory referred to by "index" now has the value 2. A check is made to see whether the current value of "index" is greater than 10, the ending value specified on the -loop- command. Since "index" is equal to 2, which is less than 10, the computer again executes the indented statements.

After 10 times through the loop, "index" is incremented to 11, which is greater than the specified ending value, and the computer proceeds immediately to the statements following the -endloop-. There are additional forms of the -loop- command for dealing with other kinds of repetitive situations, and there are -reloop- and -outloop- commands for altering the sequence of operations.

The -showt- command displays a numerical value in tabular form. In this case, it displays the square root of the variable "index" with 1 digit before the decimal point and 3 digits after the decimal point.

The -if- statement inside the loop checks whether the logical expression "index = 7" is true or not. If it is true (that is, if the current value of "index" is equal to 7), the indented statements bracketed by the -if- and -endif- commands are executed; otherwise the bracketed statements are skipped. There is an important difference between "index = 7" (a logical expression that is true or false) and "index := 7" (assign the numerical value 7 to the region of computer memory referred to by "index"). The -elseif- and -else- options provide additional control, and the -case- command provides an important alternative to the -if- command.

Note the difference between the following two statements:

<table>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>show</td>
<td>Nanimals</td>
<td>displays the number &quot;8&quot;</td>
</tr>
<tr>
<td>write</td>
<td>Nanimals</td>
<td>displays the text &quot;Nanimals&quot;</td>
</tr>
</tbody>
</table>

You can mix text and numbers using "embedded" -show- commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>There are &lt;show,Nanimals&gt; in the field</td>
<td></td>
</tr>
</tbody>
</table>

Also note the following peculiar-looking assignment statement:

```
calc N := N+3
```

This means "get the current value of N, add 3 to it, and store the sum back into N." The effect is to increment N by 3.
Global vs. local variables: In a large, complex program it is advisable to use "local" variables or "group" variables (a named set of global variables) whenever possible, to avoid problems from different units making conflicting assignments to the same variables.

This introduction has discussed the most important aspects of cT calculations. Further details are provided in the following sections.
Defining Variables

Summary of Variable Definitions

Here is a summary of the forms available for declaring variables. These tags may be used with a -define-command (for global or group variables) or on the lines immediately after a -unit- command (for local variables).

The name of a variable or constant must start with a letter. The rest of the name may contain letters, numbers, and underscore. It cannot contain a space. A variable name may be 30 characters long. Variables are case sensitive: "zip" is distinct from "Zip".

integer: i, j, k
i: cows, horses
i: dim1=5, dim2=3

byte: b1, b2, b3
b: byte1, byte2
b: AllOnes = 255
b: SAME=2#11111111
b: Again = 16#ff

float: a, b, c
f: tempX, tempY
f: C = 3*10E5

button: GoOn, Stop
edit: Text1, Speech
slider: AdjustSpeed
screen: save1, save2
touch: treg1, tch0

file: fd1, fd2

marker: m1, m2
m: name, formula

arrays may be of any variable type:
i: AnArray(50)
f: points(dim1,dim2)
f: GNP(1960:1980, 5)

numeric arrays (integer, float, byte) may be dynamically allocated:
i: RData(*,*)

You can define "groups" of variables:
group, myset:
merge, global:
merge, myset:
"user" and "author" can be used only in the IEU:

user: $$\begin{array}{l}
\text{f: x,y,z} \\
\text{author:} \\
\end{array}$$ variables available also to user

$$\begin{array}{l}
\text{f: x,y,z} \\
\text{author:} \\
\end{array}$$ variables available also to user

See Also:
define Global Variables and Groups (p. 193)
Basic Calculational Operations (p. 200)
Defining Marker Variables (p. 248)

Types of Variables

The types of cT variables are byte, integer, float, marker, button, edit, slider, screen, touch, and file.

Quick Summary: (more details come later)

Floating-point variables contain values that are not integers, such as 0.00039 or -5287.31. The range and accuracy available depend on the particular machine you are using. In cT, if a floating-point variable is used or stored as an integer, the value is rounded to the nearest integer.

There are two special floating-point values that may be displayed by a -show- command: INF (infinity) is displayed when a nonzero value has been divided by zero; NAN ("Not A Number") is displayed when a zero value is divided by zero, which might also be considered an "indefinite" result.

Integer variables are signed 32-bit integers. These variables use less storage space than floating-point numbers and give much faster execution speed. Integer values range from $-2^{31}$ to $+2^{31}$, which is approximately $-2*10^9$ to $+2*10^9$.

The value $2*10^9$ (two billion) is not such a large number. If you multiply 2 billion times 2, the result, 4 billion, is too large to fit into an integer. The integer variable overflows and the result is nonsense. No warning is given when such an overflow occurs. Any calculation that might result in large numbers should always be done with floating-point variables.

Bytes are unsigned 8-bit quantities suitable for storing positive numbers in the range 0 to 255. They use only one-fourth as much space as integer variables.

Constants contain values that do not change during the program. Attempting to assign a value to a constant (such as CONSTANT := 3*x+4*y) gives a compile error, because the value of a constant can't change. Many programmers make a habit of naming constants with all capital letters (i.e., BASE=47). That is not required, but it is a convenient convention. There are four system-defined constants: TRUE, FALSE, PI, and DEG (radians per degree = 2PI/360).

A marker variable brackets a string of characters. That is, the marker shows both where the string of interest starts and where it ends. Another marker can mark a subsection of the same string.

A file descriptor variable is actually a group of variables that describe a file. These variables cannot be manipulated directly; they are modified with commands such as -addfile-, -setfile-, and -reset-.

When file descriptors are passed to subroutines, they must be passed by address. If a locally defined file descriptor is used to -addfile- or -setfile-, the file is automatically closed upon exit from the unit. Similarly, a -jump- command closes all files that were associated with local file descriptors, leaving active only those files associated with global file descriptors.
Graphics objects and screen areas are referred to by their own variable types: edit variables, button variables, slider variables, screen variables, and touch variables. Like file variables, they must be passed by address, and memory associated with local variables of these kinds is released upon exit from the unit. It is possible to compare these variables (of the same kind) for equality or inequality.

Arrays of integer, floating, and byte variables can be dynamically allocated with the -alloc- command: dynamic arrays can grow or shrink as needed during execution.

More details about numerical variables and constants:

**Floating-point variables**, or "real numbers," are stored in a computer in four pieces: the mantissa, the exponent, the sign of the mantissa (+ or -), and the sign of the power. The value of a number is derived from these pieces with some sleight-of-hand, but essentially it is

\[(\text{sign})\text{mantissa} \times 2^{(\text{sign})\text{power}}\]

The accuracy of the number is determined by the number of bits used for the mantissa. The range of numbers that can be represented is determined by the number of bits used for the power. All cT floating-point variables and constants use "double precision."

Many modern computers use the IEEE (Institute of Electrical and Electronic Engineers) standard for floating-point numbers. Double precision uses 64 bits for a floating-point number: 51 bits for the mantissa, 11 bits for the power, and 2 bits for the signs.

Many machines use a 64-bit IEEE format. This gives an accuracy of about 16 decimal digits. The range of numbers is approximately:

- Largest positive: \(2 \times 10^{308}\)
- Smallest positive: \(2 \times 10^{-308}\)
- Smallest negative: \(-2 \times 10^{-308}\)
- Largest negative: \(-2 \times 10^{308}\)

**Floating-point constants** may be written in so-called "scientific notation" using either E+ or e+ notation:

\[2328756 = 2.328756\times10^5\]
\[-.00029 = -2.9\times10^{-4}\]

There must not be any spaces in the E format, nor in the # format discussed below.

**Integer constants** may be written in any number base from 2 to 16:

\[(\text{sign})\text{base}\#\text{number}\]

- \(2\#1011\) is 11 in binary
- \(5\#21\) is 11 in base 5
- \(-8\#13\) is -11 in octal
- \(#b\) is 11 in hexadecimal

If "base" is omitted, the number is in hexadecimal (base 16). Other commonly used bases are binary (base 2) and octal (base 8). For these bases there are specialized display commands: -showb-, -showo-, and -showh-.
Integer variables are stored in a computer in two pieces: the sign and the number. The sign is stored in the leftmost bit and is 0 for positive and 1 for negative.

Most modern computers represent negative integers internally with a technique called "two's complement." In two's complement, in order to make a negative integer, all of the bits of the corresponding positive integer are complemented (1's turn into 0's and vice-versa) and then 1 is added to the result. This example shows (in binary) the representation of -11:

- Start with +11: 0000 . . . 0001011
- Flip all the bits: 1111 . . . 1110100
- Add 1, make -11: 1111 . . . 1110101

Integer variables may range from -2147483648 (#80000000) to 2147483647 (#7fffffff).

See Also:
- Basic Calculational Operations (p. 200)
- Buttons, Dialog Boxes, Sliders, & Edit Panels (p. 142)
- get and put Portions of Screen (p. 89)
- Files, Serial Port, & Sockets (p. 285)

**define: Global Variables and Groups**

The `define` command specifies variables for use internally within the program, variables for the user of the program, and constants. Variables and constants created by `define` in the IEU (initial entry unit) are global variables that may be used anywhere in the program, and such variables can be gathered into groups that can be referred to with the "merge" option.

```
define float: a, b, c $$ floating point; begin global definitions
f: TArray(8,4,7) $$ same as float:
f: Vec(-50:50) $$ array index runs from -50 to 50
f: Ka = 23.57 $$ can use Ka just like 23.57
integer: horses, cows $$ integer
i: pig, hog $$ same as integer:
byte: byte1, byte2 $$ 8-bit byte
b: B(10,20) $$ same as byte:
file: fd1 $$ file descriptor
screen: s1, s2, s3 $$ screen area for -get- and -put-
edit: ev $$ an edit panel
button: bv $$ a button to click
slider: sv $$ a slider to adjust
touch: tch $$ a touch region
```

*numeric arrays (integer, float, byte) may be dynamically allocated:*
```
i: RData(*,*) $$ two-dimensional dynamic array; use -alloc-
```

*You can define "groups" of variables:*
```
group, alpha:
i: A=1, B=2, C=3 $$ group alpha consists of A, B, C
```
```
group, beta:
merge, alpha:
i: Bx, By, Bz $$ beta consists of A, B, C, Bx, By, Bz
```
CALCULATIONS

user: \( $$ \) available to users during execution
f: x, y, z
author: \( $$ \) not available to users; part of global group
f: m1, m2, m3

.....
unit test
* use global variables (including user variables):
  merge.global:
  merge.alpha: \( $$ \) use alpha group variables
  i: ii, jj, kk \( $$ \) local variables used only in unit test
calc jj := pig+B \( $$ \) local := global + beta definitions

The -define- command must appear in the IEU (initial entry unit preceding the first -unit- command). "user" variables must be defined as global variables; see "User Variables" below. Several variables may be defined on one line, but each new variable type must be introduced on a new line. The variable type (such as "f:" ) does not need to be repeated on subsequent lines.

The name of a variable or constant must start with a letter. The rest of the name may contain letters, numbers, and underscore. It cannot contain a space. A variable name may be 30 characters long. Variables are case sensitive: "zip" is distinct from "Zip".

You can define variable (or constant) names even though there exist system variables and functions with the same names. Your definitions override the system definitions. All system variables are prefixed with "z" so that you can easily avoid such a conflict. The system-defined constants are TRUE, FALSE, PI, and DEG (radians per degree = 2PI/360).

The group and merge options are useful for identifying separate sets of variables for use by separate parts of your program. Also, you can have "define group.gamma" in a -use- file and refer to it with a "merge, gamma:" in your main file. This provides a convenient way to share information between two files.

When you run your program (or perform "Make Binary"), cT first compiles the IEU in your main program (including the global -define-s), then it compiles the IEUs (and global -define-s) of each -use- file, in the order in which you have listed your -use- commands.

A variable defined in a group cannot be used in the IEU. If you want to initialize some group variables in the IEU, use a -do- of a unit that initializes the variables.

You can say "merge, gamma:" in your definitions even if the "group, gamma:" information is present later in the file, or is in a -use- file. However, your own -define- statements cannot themselves reference definitions that are compiled later. Here is what you can and can't do:

\[
\begin{align*}
\text{define} & \quad \text{group, main:} \\
& \quad i: \text{One, Two} \\
& \quad \text{merge, gamma:} \\
& \quad i: \text{Array(NN)} \quad $$ \text{would give compile error; NN not defined yet} \\
& \quad \text{group, gamma:} \quad $$ \text{could be in -use- file} \\
& \quad i: \text{NN=35} \\
\end{align*}
\]

unit First
merge, main:
show NN $$ show 35 because "gamma" included in "main"
Local Variables

Local variables are variables that are defined for only one unit. The format for specifying local variables is the same as for a -define- statement, except that "user" variables cannot be local. Local variables must be indented and must follow immediately after the -unit- command line:

```plaintext
unit someunit
merge,diagram: $\$$ merge in group named "diagram"
f: local1, local2, local3
f: tx, ty, tz
```

The example above refers to the "diagram" group of variables and has six local variables: local1, local2, local3, tx, ty, and tz. Local variables are usually used when values are passed to a unit. By using local variables, a unit can do calculations without disturbing the values of global variables. This is important when the unit is accessed from many different places.

```plaintext
unit someU2(tx, ty, tz)
f: tx, ty, tz, index1, index2
```

If the unit needs to have access to global variables (those declared without a group name in a -define- at the beginning of the program), the local variable declaration must include "merge,global:" as its first line, and it can also have other merge operations to make reference to group variables in this file or in a -use- file:

```plaintext
unit someU3(myvar1, myvar2)
merge,global: $\$$ merge global set of definitions
f: myvar1, myvar2, myvar3
merge,gamma: $\$$ also merge the group named gamma
```

Every time a unit is executed with a -do- a new set of local variables is created. These local variables are maintained until the entire unit is finished, even though other units (with their own local variables) are done along the way. This makes it possible to do recursive problems.

**NOTE:** The local variable definitions are part of the -unit- command; they are "continued lines" of the -unit- command. There **must not** be any other lines of code until after the local defines are finished.

**Examples:**

In this example, the figure is drawn at three different positions with three different vector lengths. The screen position and vector length are passed to unit `xfigure` as arguments in each of the three -do- commands.

```plaintext
unit xlocal1
do xfigure(50,100, 50)
do xfigure(200,100, 100)
do xfigure(75,220, 70)
```
CALCULATIONS

* unit xfigure(tx, ty, tlength)
  f: tx, ty, tlength, angle
rorigin tx, ty
rat 0,0
circle tlength
loop angle := 0, 315, 45
  rotate angle
  rvector 0,0; tlength,0
endloop
rotate 0

The next example shows an interesting design created through "recursion," a technique in which a subroutine calls itself. In this case, note how unit FractArrow calls itself.

unit fractals $$ recursion example
do FractArrow(175,50, 1, 0, 0)
*
unit FractArrow(rx, ry, scale, rotate, depth)
f: scale, rotate
i: rx, ry, depth
i: maxdepth = 8 $$ maximum depth
i: angle = 70 $$ try \ 30 < \text{angle} < 140
i: length = 100 $$ length of first line
f: ff = .65 $$ scale reduction factor
outunit maxdepth < depth $$ exit condition
at rx, ry $$ set screen position
rorigin $$ origin at current screen position
size scale
rotate rotate
rdraw ; 0,length $$ draw from current position
calc rx := zwherex $$ save current screen x
  ry := zwherey $$ save current screen y
do FractArrow (rx ,ry, ff*scale, rotate+angle, depth+1)
do FractArrow (rx ,ry, ff*scale, rotate-angle, depth+1)

See Also:
define Global Variables and Groups (p. 193)
Combining Global and Local Variables (p. 196)
do Calling a Subroutine (p. 224)

Combining Global and Local Variables

The "merge,global:" keyword is used with local variable definitions to combine ungrouped global variables (both author and user) with local variables. The "merge,global:" must appear on the first line after a unit command and must be indented. You can also merge named groups as well:

define i: G1, G2
group,one:
i: ONE=1
group,two:
i: TWO=2

.....
DEFINING VARIABLES

There are four possible situations in a program with respect to variables:

1) no variables at all
2) global variables only
3) local variables only
4) both global and local variables

In case 1, there are no variables and thus no need for rules of precedence.

In case 2, the global variables (specified with a -define- statement) are available everywhere in the program. Those variables specified as "user" variables are available both to the user and to the program.

In case 3, the local variables are available within each individual unit, but there is no carry-over of variables from one unit to another. Local variables can, however, be passed from one unit to another either with pass-by-value or pass-by-address:

```
unit one
  f: a,b,c
do two(a, 9; b)
  ...
  ...
unit two(t1, t2; t3)
  f: t1, t2, t3
  ...
```

In case 4, the global variables are available everywhere in the program unless local variables have been specified. In a unit that specifies local variables, only the local variables are available, unless group variables are merged. Sometimes it is desirable to use both global and local variables. The `merge,global:` option explicitly allows both global and local variables to be active in the same unit.

```
define f: x, y, z
* unit start
  * the global variables x, y, and z are available
* unit later
  f: x, y, z
  * the local variables x, y, and z are available
  * they have no relationship to the global x, y, z
* unit combine
  merge,global:
    f: a, b, c
    * the global x, y, and z are available
    * the local a, b, and c are available
*```

Note that local variables may have the same names as global variables, if there is no `merge,global:`. However, this is not good programming practice!
User Variables

Ordinarily, when the user enters a response ("heat the mixture"), it is treated as a phrase -- a sequence of letters. If the user enters an algebraic expression ("x+4"), the program must be given special instructions so that the "x" is treated as a variable instead of simply as a letter in a phrase. The "user:" keyword is used to specify that the variables that follow in the -define- statement are available to the user (and to the program). The "author:" keyword specifies that the variables that follow it are available only to the program.

\[
\text{define user: } \quad x, y \quad \text{available to user & author}
\]
\[
\text{author: } \quad z \quad \text{available only to author}
\]

The only commands that treat "user" variables specially are -compute-, -ansv-, and -wrongv-. They use the "user" definitions made in the same file.

The "user:" and "author:" options refer only to the ungrouped global variables and cannot be used in the definition of a named group of variables. However, the "user:" definitions can be made in terms of a variable group, which provides a way to share access to user variables. Similarly, if definitions of constants are used in both the "author" and "user" defines, the constant must be merged from a separate group, like this:

\[
\text{define group, length:}
\]
\[
i: \text{NN}
\]
\[
\text{author: } \quad xx(NN)
\]
\[
\text{user: } \quad ux(NN)
\]
\[
\text{merge, global: } \quad z \quad \text{merges both author and user defines}
\]

Example:

\[
\text{define user: } \quad x, y \quad \text{in IEU, before the first -unit- command}
\]
\[
\text{unit xUserVariables}
\]
\[
\text{merge, global: } \quad z
\]
\[
\text{calc } \quad x := 5
\]
\[
\text{at } \quad 50, 30
\]
\[
\text{write } \quad x = 5
\]
\[
\quad y = 7
\]
Type an expression using \( x \) and \( y \).

\[ \text{arrow 100,120; 500,300} \]
\[ \text{compute } z \]
\[ \text{ok} \]
\[ \text{return} \]
\[ . \]
\[ \text{write} \quad \text{Your expression evaluates} \]
\[ . \]
\[ \text{to } \langle s, z \rangle. \quad $$ \text{embedded -show-} \]
\[ \text{no} \]
\[ . \]
\[ \text{write} \quad \text{Sorry, I don't understand.} \]
\[ \text{endarrow} \]

* 

See Also:

- ansv Numerical Responses (p. 163)
- compute Storing and Evaluating Inputs (p. 165)
- compute Computing with Marker Variables (p. 254)
- Algebraic Responses (p. 167)
- specs Specifying Special Options (p. 169)
- Plot Two User Functions Simultaneously (p. 281)
- Plotting Parametric Equations (p. 282)
Basic Calculational Operations

calc: Assigning a Value to a Variable

The -calc- command is used to assign a value to a variable. All variables must have been previously -define-d, either in the -define- set at the beginning of the program or as local variables within a unit.

\[
\text{calc} \quad \text{result} := x^2 + \sin(x) + 7
\]

The assignment symbol is ":=". Only one assignment may be made per line, unless the assignments cascade:

\[
\begin{align*}
\text{calc} & \quad y := x := z := 0 \quad \text{\$ this is okay} \\
\text{calc} & \quad y := 5; \quad x := 11 \quad \text{\$ this is NOT okay}
\end{align*}
\]

You may use "scientific notation" with either upper- or lowercase E. The statement "x becomes 4.2837 times 10^{15}" is written:

\[
\text{calc} \quad x := 4.2837E+15
\]

You may use parentheses (), braces {}, or brackets [] to enclose pieces of an expression. Implied multiplication is allowed except where two variables are involved.

\[
\begin{align*}
\text{calc} & \quad x := 3(\alpha + 7\beta) \quad \text{\$ this is okay} \\
\text{calc} & \quad x := \{\alpha+\beta\}[3y+7] \\
\text{calc} & \quad x := \alpha\beta \quad \text{\$ this is NOT okay}
\end{align*}
\]

When the program is executed, the user is permitted to use implied multiplication of "user" variables (created with the -define- command) when processed by -compute- or -ansv-. Also see "User Variables".

See Also:
- Defining Variables (p. 190)
- System Variables (p. 318)
- calc Simple Marker Calculations (p. 249)
- compute Computing with Marker Variables (p. 254)
- compute Storing and Evaluating Inputs (p. 165)
- ansv Numerical Responses (p. 163)
- User Variables (p. 198)

Arithmetic Operators

The symbols used for simple arithmetic operations are

\[
+ \quad - \quad * \quad / 
\]

Exponentiation may be expressed with a caret (^) or with two asterisks (**):

\[
x^y \text{ is } x^\text{y} \text{ or } x^{\text{**y}}
\]

The $divt$ and $divr$ operators perform integer division, either truncating or rounding the result. Integer division is much faster than the "floating division." The standard division operator ("/") always means floating division, even if the quantities must be converted (internally) into floating-point format.
11 $divt$ 4 is 2
11 $divr$ 4 is 3
11/4 is 2.75

Integer, floating-point, and byte variables may be mixed in expressions. For example, it is perfectly legal to add "f+i+b", in which case the integer and byte variables are changed (internally) to floating-point variables before a floating addition is performed.

See Also:
Defining Variables (p. 190)
Logical Operators (p. 201)

Logical Operators

The following logical operations are available, as well as the system-defined constants TRUE and FALSE:

- equal
- <= less than or equal
- >= greater than or equal
- <> inverse of A
- not(A)
- & logical intersection
- $and$
- | logical union
- $or$

Logical operations, or boolean operations, involve comparisons that are either "true" or "false". For example, the expression "a>b" means "a is greater than b". That statement is either true or false. The expression "A<B<C" is treated as though it were written "(A<B) & (B<C)" and is true only if A is less than B and B is less than C.

With markers (strings of characters), comparisons such as "greater than" (>) are terms of alphabetic order, so that "z" is greater than (comes after) "a", and "a" is greater than (comes after) "Z".

With edit, button, slider, and screen variables, the only legal comparisons are equal or not equal.

An -if- statement always expects a logical expression:

if a > b
  write a is greater than b
else
  write a is not greater than b
endif

Commands (such as -if-, -reloop-, -ok-) that require true/false expressions in their tags regard all negative values (after rounding to an integer) as "true" and all positive values as "false."

A "true" expression evaluates to -1 and a "false" expression evaluates to 0. These values may be used in calculations. The system-defined constants TRUE and FALSE may also be used in calculations:
\( x := 14 \cdot (a<b) \)
\( \text{status} := \text{TRUE} \)

The variable "x" becomes either -14 (a is less than b) or 0. The variable "status" is -1.

In expressions involving & ($\text{and}$) and | ($\text{or}$), the second part of the expression is evaluated only if the first part has not determined the truth of the statement. For example, in the expression "X & Y", if X is false, the entire expression is false, and Y is not evaluated. Similarly, in X | Y, there is no need to evaluate Y if X is true. Here is an example of the advantage of this treatment:

\[
\begin{align*}
\text{define} & \quad f : X, Y(10) \\
\text{...} & \\
\text{if} & \quad X<1 \mid Y(X) \\
\end{align*}
\]

If X is zero, evaluating Y(0) would give an execution error. The prior check for X<1 prevents the evaluation of Y.

In order to compensate for roundoff errors, cT uses a "fuzzy zero." When making logical comparisons, \( X = Y \) if \( \text{abs}(X-Y)/X < 10^{-11} \), or if \( \text{abs}(X-Y)<10^{-9} \).

\textbf{See Also:}
- Defining Variables (p. 190)
- Arithmetic Operators (p. 200)

\textbf{Trigonometric Functions}

These trigonometric functions are available:

\[
\begin{array}{cccccc}
\text{sin} & \text{csc} & \text{arcsin} & \text{arccsc} & \text{sinh} \\
\text{cos} & \text{sec} & \text{arccos} & \text{arcsec} & \text{cosh} \\
\text{tan} & \text{cot} & \text{arctan} & \text{arccot} & \text{tanh}
\end{array}
\]

The arguments of the trigonometric functions are given in radians. The results returned from the inverse trigonometric functions are also in radians. (Angles used with the \text{-rotate-} and \text{-polar-} commands are normally in degrees, but you can use \text{-inhibit degree-} to change to using radians, for consistency with the trigonometric functions.) The functions \text{sinh}, \text{cosh}, \text{tanh} are hyperbolic sine, cosine, and tangent.

The \text{arctan} function has two formats. When there is one argument, \text{arctan}(p) returns the radian measure of the angle whose tangent is "p". The angles range from \(-\pi/2\) to \(+\pi/2\). When there are two arguments, \text{arctan}(p,q) returns the radian measure of the angle whose tangent is "p/q". The angles range from \(-\pi\) to \(+\pi\).

The system-defined constant DEG (radians per degree) is useful for converting degrees into radians: \( \cos(60\text{DEG}) \) is 1/2, and \( \text{arctan}(1)\text{DEG} \) is 45.

\[
\begin{align*}
\text{PI} & = 3.14159..... \\
\text{DEG} & = 2\text{PI}/360 \quad (\text{the radian measure of one degree})
\end{align*}
\]

\textbf{Example:}

This example shows arctangent values for the two formats of \text{arctan}. Notice how two consecutive \text{-write-} statements are used to make one line of display. Then the embedded carriage return prepares for the next line.
BASIC CALCULATIONAL OPERATIONS

unit xarctan $$ show arctan values
at 40,50
write Arctan(X)<|tab|>Radians<|tab|>Degrees
at 40,70
do atans1(100) $$ 1st quadrant
do atans1(1) $$ 1st quadrant
do atans1(-1) $$ 4th quadrant
do atans1(-100) $$ 4th quadrant
at 40,150
do atans2(-1,-1) $$ 3rd quadrant
do atans2(-1,1) $$ 4th quadrant
do atans2(1,1) $$ 1st quadrant
do atans2(1,-1) $$ 2nd quadrant
*
unit atans1(q) $$ range: -PI/2 to PI/2
f: q
write <|t,q,4|><|t,arctan(q) ,6,3|>
write <|t,arctan(q)/DEG ,6,3|><|cr|>
*
unit atans2(q,r) $$ range: -PI to +PI
f: q,r
write <|t,q,2|>,<|t,r,2|><|t,arctan(q,r), 6,3|>
write <|t,arctan(q,r)/DEG, 6,3|><|cr|>
*

See Also:
Other Mathematical Functions (p. 203)
inhibit/allow degree (p. 70)

Other Mathematical Functions

These mathematical functions are available:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>absolute value of x</td>
</tr>
<tr>
<td>frac(x)</td>
<td>fractional part of x</td>
</tr>
<tr>
<td>int(x)</td>
<td>integer part of x</td>
</tr>
<tr>
<td>round(x)</td>
<td>integer part of (x+0.5)</td>
</tr>
<tr>
<td>sign(x)</td>
<td>-1, 0, or +1 for x&lt;0, x=0, x&gt;0</td>
</tr>
<tr>
<td>mod(x,y)</td>
<td>modulo function (remainder of x/y)</td>
</tr>
<tr>
<td>comp(n)</td>
<td>bit complement</td>
</tr>
<tr>
<td>bitcnt(n)</td>
<td>number of bits set in n</td>
</tr>
</tbody>
</table>

The functions int and frac are defined so that int(x)+frac(x) = x. Note the behavior of int and frac with negative numbers: int(-3.7) is -3; frac(-3.7) is -0.7.

The mod function is defined as mod(x,y) = y*frac(x/y)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log(x)</td>
<td>logarithm of x (base 10)</td>
</tr>
<tr>
<td>ln(x)</td>
<td>logarithm of x (base e)</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root of x</td>
</tr>
<tr>
<td>alog(x)</td>
<td>10^x</td>
</tr>
<tr>
<td>exp(x)</td>
<td>e^x</td>
</tr>
<tr>
<td>factorial(x)</td>
<td>(x)(x-1)(x-2)....(1)</td>
</tr>
</tbody>
</table>
CALCULATIONS

\[
\text{combin}(x,y) = \frac{\text{factorial}(x)}{\text{factorial}(y) \times \text{factorial}(x-y)}
\]
\[
\text{gamma}(n)
\]

For positive integer values of \(n\), the function \(\text{gamma}(n)\) is equivalent to \(\text{factorial}(n-1)\).

System-defined constants are written in capital letters:

\[
\begin{align*}
\text{TRUE} &= -1 \\
\text{FALSE} &= 0 \\
\text{PI} &= 3.14159..... \\
\text{DEG} &= \frac{\text{PI}}{180} \text{ (the radian measure of one degree)}
\end{align*}
\]

See Also:

- Trigonometric Functions (p. 202)
- Bit Manipulations (p. 206)

**The Keyname Function: zk()**

The \(zk()\) function gives the numeric value of a character or other input.

\[
\text{zk(keyname)}
\]

The *keyname* for the \(zk()\) function may be any alphanumerical character such as \(zk(B)\) or \(zk(7)\), or any of these keywords:

- back     cr
- erase    ext
- next     space
- tab      timeup
- touch (and touch variants)

- copy     cut     paste     undo
- escape    help    transpose
- fa        fb       fc
- left_arrow left_arrow_extend
- right_arrow right_arrow_extend
- up_arrow  up_arrow_extend
- down_arrow down_arrow_extend
- begin_line begin_line_extend
- end_line  end_line_extend
- begin_page begin_page_extend
- end_page  end_page_extend
- begin_file begin_file_extend
- end_file  end_file_extend
- page_up   page_up_extend
- page_down page_down_extend
- tab_back  erase_fwd

The touch variants are covered in "pause Mouse Inputs".
The system function \texttt{zks()} generates a character string corresponding to a numeric key value. For example, \texttt{zks(zk(next))} on a Macintosh generates the character string "return" and on a PC generates "enter". The \texttt{zks()} function provides a machine-independent way to display on the screen the name of the key or keys that the user would press in order to achieve a particular effect.

\textbf{Macintosh} keyboard mappings are

- copy, cut, paste, undo: edit menus or key equivalents of menus or function key equivalents
- escape: \texttt{esc}
- help: \texttt{help}
- transpose: \texttt{control t}
- \texttt{fa, fb, fc, fd: f5, f6, f7, f8}

\begin{align*}
\text{left\_arrow, (left\_arrow\_extend): leftarrow, (shift leftarrow)} \\
\text{right\_arrow, (right\_arrow\_extend): rightarrow, (shift rightarrow)} \\
\text{up\_arrow, (up\_arrow\_extend): uparrow, (shift uparrow)} \\
\text{down\_arrow, (down\_arrow\_extend): downarrow, (shift downarrow)} \\
\text{begin\_line, (begin\_line\_extend): cmd leftarrow, (cmd-shift leftarrow)} \\
\text{end\_line, (end\_line\_extend): cmd rightarrow, (cmd-shift rightarrow)} \\
\text{begin\_page, (begin\_page\_extend): cmd uparrow, (cmd-shift uparrow)} \\
\text{end\_page, (end\_page\_extend): cmd downarrow, (cmd-shift downarrow)} \\
\text{begin\_file, (begin\_file\_extend): cmd home, (cmd-shift home)} \\
\text{begin\_file\_scroll: home} \\
\text{end\_file, (end\_file\_extend): cmd end, (cmd-shift end)} \\
\text{end\_file\_scroll: end} \\
\text{page\_up, (page\_up\_extend): cmd pageup, (shift-cmd pageup)} \\
\text{page\_up\_scroll: pageup} \\
\text{page\_down, (page\_down\_extend): cmd pagedown, (shift-cmd pagedown)} \\
\text{page\_down\_scroll: pagedown} \\
\text{tab\_back: cmd tab} \\
\text{erase\_fwd: del}
\end{align*}

\textbf{Windows} keyboard mappings are

- copy: \texttt{control c} or \texttt{shift insert}
- cut: \texttt{control x} or \texttt{shift del}
- paste: \texttt{control v} or \texttt{insert}
- ?? undo: \texttt{control u}
- escape: \texttt{esc}
- help: \texttt{f1}
- transpose: \texttt{control t}
- \texttt{fa, fb, fc, fd: f5, f6, f7, f8}

\begin{align*}
\text{left\_arrow, (left\_arrow\_extend): leftarrow, (shift leftarrow)} \\
\text{right\_arrow, (right\_arrow\_extend): rightarrow, (shift rightarrow)} \\
\text{up\_arrow, (up\_arrow\_extend): uparrow, (shift uparrow)} \\
\text{down\_arrow, (down\_arrow\_extend): downarrow, (shift downarrow)} \\
\text{begin\_line, (begin\_line\_extend): home, (shift home)} \\
\text{end\_line, (end\_line\_extend): end, (shift end)} \\
\text{begin\_page, (begin\_page\_extend): ctrl home, (shift-ctrl pageup)} \\
\text{end\_page, (end\_page\_extend): ctrl end, (shift-ctrl end)} \\
\text{?? begin\_file, (begin\_file\_extend): ctrl home, (shift-ctrl home)}
\end{align*}
CALCULATIONS

end_file, (end_file_extend): ctrl end, (shift-ctrl end)
page_up, (page_up_extend): pageup, (shift pageup)
page_down, (page_down_extend): pagedown, (shift pagedown)
tab_back: shift tab
erase_fwd: delete

On any machine the function keys can be composed. This allows all the function keys to be accessed even on a machine that doesn't have the physical keys we expect. See "Typing Special Function Keys".

Example:

unit xzk
loop
  at 50,50
  write Press the TAB button or the spacebar.
  pause
  erase at 100,100
  if zkey = zk(tab) | zkey = zk(space)
     write \zkey=zk(tab)
     \You pressed TAB!
     \Spacebar pressed.
     outloop
  else
     write You typed <|s,zks(zkey)|>.
     Please press TAB or Spacebar
  endif
endloop
at 10,250
write After the -endloop-.

See Also: Single Key & Timed Pause (p. 125)
Mouse Inputs (p. 126)
Typing Special Function Keys (p. 184)

Bit Manipulations

Bit manipulation operations allow the manipulation of individual bits within a variable. These operations may be used with byte variables and integer variables.

\n```plaintext
\n\nc \$lsh\$ N $$ left shift c by N positions
\nc \$rsh\$ N $$ right shift c by N positions
\nm \$mask\$ n $$ bit-wise "and"
m \$union\$ n $$ bit-wise "or"
m \$diff\$ n $$ bit-wise "exclusive or"
\ncomp(c) $$ bit complement of c
bitcnt(c) $$ count number of 1's bits in c
\n```
All bit manipulations are defined in terms of the binary expression of a number. Moving one position means moving one binary position.

**Left shift** moves every bit one position to the left. The right-hand bits become zero and any ones at the left are "pushed off" the end.

**Right shift** moves every bit one position to the right. The right shift operation ($rsh$) is loosely defined. On some machines the left-most ("sign") bit is extended (copied) into neighboring positions during the shift, while on other machines this bit may not be extended. Bits shifted off the right end are discarded in any case. This loose definition corresponds to the definition of right shift operations in the C language.

In $m \text{ $mask$ } n$, every bit that is 1 in both $m$ and $n$ becomes 1. All other bits become 0.

In $m \text{ $union$ } n$, every bit that is 1 in either $m$ or $n$ becomes 1. The resulting bit is 0 only if the corresponding bits in both $m$ and $n$ are 0.

In $m \text{ $diff$ } n$, the resultant bit is 1 if the corresponding bits in $m$ and $n$ are different (i.e., one is 1 and the other is 0). If the corresponding bits are the same (both 1 or both 0), the resultant bit is 0.

The **comp** function changes every 1 bit of the named variable into 0, and every 0 bit into 1.

The **bitcnt** function counts the number of 1 bits in the named variable.

**NOTE:** All arithmetic operations on integer and byte variables are carried out to 32-bit precision, even if some of the data are read from or stored into byte variables.

*Example:*  

Internally, the left shift operation is carried out with 32 bits of working storage. If one of these 32-bit intermediate results is stored into a byte variable, only the lowest 8 bits are saved. All 32 bits can be obtained by storing the result into an integer.

The byte variable B can hold only 8 bits, with numerical values between 0 and 255. The left shift by 4 nevertheless produces a larger number (1600) and stores that larger number in the integer variable N. If it is stored into the byte variable C, only the right-most 8 bits are kept. The other bits are simply lost.

```
unit xbits1
b: B, C
i: N
calc
B := 100
C := B $lsh$ 4 \quad \text{ same as B*16 }
N := B $lsh$ 4
at 50,50
write <|t,B,4|> <|b,B,16|><|cr|>
<|t,C,4|> <|b,C,16|>
<|t,N,4|> <|b,N,16|>
```

See Also:

- Other Mathematical Functions (p. 203)
- Types of Variables (p. 191)
- Defining Variables (p. 190)
Array Operations

Using Arrays

Here is a simple example of how to define and use an array, in this case an array that will contain 31 temperatures for each day of a month:

```
unit xarray
float: Temp(31)  $$ one-dimensional array, 31 elements
calc Temp(12) := 47.3  $$ 47.3 degrees on the 12th day
```

Arrays may be composed of any type of variable. They may be multidimensional, and the lengths of byte, integer, or float arrays can be dynamically allocated with the - alloc- command. The function `zlength(array)` returns the total number of elements in an array. Here is a variety of definitions of one- and multi-dimensional arrays:

```
define f: A(5), B(2,3), C(10:15)
i: D1=2, D2=3, D3=5
 $$ array lengths in terms of defined constants:
b: MultiDim(D1,D2,D3)

m: MyMarker(10)
file: DataFiles(4)
touch: Regions(5,3)

$$ first index is 1960, 1961, 1962:
i: S(1960:1962, 5)

$$ 1st index is -10, -9, -8; 2nd is -5 to 5:
f: T(-10:-8, -5:5)

i: Dyn(*,*)  $$ integer array dynamically allocated with -alloc-
```

The array "A" has five elements: A(1), A(2), A(3), A(4), A(5). The array "B" has 6 elements. In order, they are B(1,1), B(1,2), B(1,3), B(2,1), B(2,2), B(2,3).

The defined constants, D1, D2, and D3, are used to specify the dimensions of "MultiDim".

Unless otherwise specified, array indices start at 1. Arrays can be defined that do not start at 1 by specifying the starting and ending indices separated by a colon. An array defined as "f: Z(k:m) starts with Z(k) and runs through Z(m). It has (m-k+1) elements. The array "C" has 6 elements: C(10), C(11), \ldots C(15).

In the array "S", the first index runs from 1960 to 1962 and the second index runs, in the usual fashion, from 1 to 5. An array may not have a negative length, but if start and end are given, an array can have negative indices, as in the array "T".

When a whole array is passed to a subroutine, it must be passed by address (an individual element of an array can be passed by value). That is, the location of the array is passed, and not the values contained by the array. Individual elements of an array can be passed by value. In pass-by-address, a change made to the named variable by the called unit (the -do- unit) is a change to the original variable.
When a whole array is passed to a unit, only the name of the array is given. That is, no index(es) are specified. The called unit must have a locally defined array whose index(es) are specified with asterisks (*). The asterisk tells the unit that it must set the length of the array dynamically when the array arrives.

You can also define basic arrays to be dynamically allocated, such as Dyn(*,*) in the example above. At present only numeric arrays can be defined in this way (integer, float, and byte), and each index must start at 1. See the -alloc- command for how to use such arrays.

You can use the -zero- command to zero an entire array in one operation, and a -block- command to transfer entire arrays or portions of them from one array to another (or within the same array).

**Example:**

This example fills an array using -set-. The values in the array are displayed. Then three values of the array are changed by unit zonk, and finally the values are displayed again.

```plaintext
unit xarray
i: myarray(3,5) $$ 15-element array
set myarray := 1,2,3,4,5, $$ fill row 1
21,22,23,24,25, $$ fill second row
31,32,33,34,35 $$ fill third row
do showit(0; myarray) $$ display array
do zonk(2,1 ; myarray ) $$ modify array
do showit(100; myarray) $$ display array again
*
unit zonk( p,q ; temp ) $$ modify values in array
i: p, q $$ get first element to change
i: temp(*,*) $$ receive passed array
set temp(p,q) := 2001, 2002, 2003 $$ 3 elements changed
at 30,90
write Array values changed starting at (<|s,p|>,<|s,q|>):
*
unit showit(ty; temp) $$ display the array
i: ty $$ display offset in y-direction
i: i, j $$ size of array
i: temp(*,*) $$ local array
loop i := 1,3 $$ display each row
loop j := 1,5 $$ each element of row
at 40*j, ty+15*i
showt temp(i,j), 4, 0
endloop
endunit
*
See Also:
Defining Variables (p. 190)
set Assigning Values to an Array (p. 210)
block Block Transfer (p. 212)
zero Initializing Arrays (p. 211)
do Calling a Subroutine (p. 224)
alloc Allocating Dynamic Arrays (p. 212)
set: Assigning Values to an Array

The -set- command provides a quick way to assign values to individual elements of an array. When the assignments start with the first value of the array, only the array name needs to be specified.

```
declare f: a(10), b(10)
    f: T(1960:1962, 2)
    marker: names(3)
... set a := 1, 3, 5, 2, 0, 4, 6, 9, 0
set b := 10, 20, 30, 40, 50, 60,
    70, 80, 90, 100  $$ can be continued on next line
set T(1961,1) := 37, 82
set names := "Sally", "Joe", "Beth"
```

The first -set- above is equivalent to:

```
calc a(1) := 1
a(2) := 3
a(3) := 5
... a(10) := 0
```

The -set- is not required to start with the first element of the array. It may be used to fill some small portion of a large array. The second example above is equivalent to

```
calc T(1961,1) := 37
T(1961,2) := 82
```

If the -set- command specifies more elements than are contained in the array, it will generate an error.

You can use the -zero- command to zero an entire array in one operation, and a -block- command to transfer entire arrays or portions of them from one array to another (or within the same array).

Example:

```
unit xset
i: M(8), index
set M := 5, 12, 7, 3, 4, 22, 1, 2
at 50,50
loop index := 1, 8
at 50, 18*index
write M(<|s,index|>) is <|s,M(index)|>
endloop
```

See Also:
- Defining Variables (p. 190)
- Using Arrays (p. 208)
- block Block Transfer (p. 212)
- zero Initializing Arrays (p. 211)
- alloc Allocating Dynamic Arrays (p. 212)
zero: Initializing Arrays

The -zero- command sets some or all of the elements of an array to zero (or zempty in the case of markers).

```plaintext
define i: A(5,3)
m: names(12)

zero A $$ zero entire array
zero A,4 $$ zero first 4 elements
zero A(4,1),3 $$ zero 3 elements from A(4,1)
zero names $$ set all markers to zempty
```

The simplest form of the -zero- command, with only one tag, names an array. Every element of the array is set to zero.

The two-tag -zero- has two variants. The second argument is always an expression that evaluates to some number, N. If the first tag names an array as the first argument, the first N elements of the array are zeroed. If the first argument of the tag is an element of an array, then N elements are zeroed, starting with the named element.

A related command is the -block- command, which you can use to transfer entire arrays or portions of them from one array to another (or within the same array).

Example:

```plaintext
unit xzero0
i: A(5,3), i, j
set A := 1,3,5,7,9,11,13,15,2,4,6,8,10,12,14
at 50,40
loop i := 1,5
   loop j := 1,3
      write <|t,A(i, j), 3|><|cr|>
   endloop
endloop
zero A,4 $$ zero first 4 elements
zero A(4,2),3 $$ zero 3 elements from A(4,2)
at 100,40
loop i := 1,5
   loop j := 1,3
      write <|t,A(i, j), 3|><|cr|>
   endloop
endloop
```

See Also:
- Defining Variables (p. 190)
- Using Arrays (p. 208)
- set Assigning Values to an Array (p. 210)
- block Block Transfer (p. 212)
- alloc Allocating Dynamic Arrays (p. 212)
block: Block Transfer

The -block- command lets you transfer a block of variables from one place to another, without having to write a loop to do it:

```
block A, B $$ move A(1) to B(1), A(2) to B(2), etc.
block A, B, 3 $$ move only the first 3 elements of A
block A(3), B(7), 2 $$ move A(3) to B(7), A(4) to B(8)
```

An execution error occurs if the receiving array is too short to hold all the transferred values.

A related command is the -zero- command, which you can use to set an entire array or a portion of the array to zero.

Example:

```
unit xblock
  i: A(2,5), B(2,7)
set A := 1, 2, 3, 4, 5, 11, 22, 33, 44, 55
set B := 10, 20, 30, 40, 50, 60, 70, 100, 200, 300, 400, 500, 600, 700
at 10,20
show B(2,1)  $$ displays the value "100"
block A(2,1), B(1,7), 5
at 10,50
show B(2,1)  $$ displays the value "22"
```

See Also:
- Defining Variables (p. 190)
- Using Arrays (p. 208)
- set Assigning Values to an Array (p. 210)
- zero Initializing Arrays (p. 211)
- alloc Allocating Dynamic Arrays (p. 212)

alloc: Allocating Dynamic Arrays

The -alloc- command lets you lengthen or shorten dynamic arrays, which are used when you don't know exactly how large the arrays may have to be:

```
unit test
  i: D(*,*) $$ dynamic array (integer, float, or byte)
alloc D(5,3) $$ allocate 5*3 = 15 elements
calc D(4,2) := 402
at 10,20
write <|s,D(4,2)|>; <|s,zlength(D)|> elements
alloc D(8,10) $$ extend to 8*10 = 80 elements
at 10,50
* D(4,2) is unchanged:
write <|s,D(4,2)|>; <|s,zlength(D)|> elements
```

Currently, dynamic arrays are available only for numerical values (integer, float, byte), and each index must start at 1. When a dynamic array is lengthened, the original elements are preserved and the new elements are set to zero.
The `-alloc-` command sets `zreturn` to `-1` (TRUE) if space was found for the array, or to 12 if there was insufficient memory available.

**See Also:**
- Defining Variables (p. 190)
- Using Arrays (p. 208)
- set Assigning Values to an Array (p. 210)
- zero Initializing Arrays (p. 211)
- block Block Transfer (p. 212)
- alloc Allocating Dynamic Arrays (p. 212)

**zlength: Number of Elements in an Array**

The `zlength` function gives the number of elements in an array (including a dynamic array), or the number of characters in a marker variable, or the number of bytes in a file:

```plaintext
unit test
i: A(10), B(3, 5)
f: D(*,*)
marker: M(4)
file: fd
calc M(2) := "hello"
alloc D(4,9)
showt zlength(D),6 $$ shows 4*9 = 36 elements in D
showt zlength(A),6 $$ shows 10 elements in A
showt zlength(B),6 $$ shows 3*5 = 15 elements in B
showt zlength(M),6 $$ shows 4 elements in the M array
showt zlength(M(2)),6 $$ shows 5 characters in M(2)
setfile fd; zempty; ro
showt zlength(fd),6 $$ number of bytes in selected file
```

More technically, `zlength(marker)` gives the number of `znext` operations required to proceed through the marker.

After a file has been opened with `-addfile-` or `-setfile-`, `zlength(file descriptor)` gives the length of the file in bytes.

**Sorting an Array**

Here we offer a simple example of how to sort an array of numbers using a simple "bubble sort," in which larger numbers are repeatedly switched with smaller numbers, so that large numbers move to the end of the array, and small numbers move to the beginning of the array:

```plaintext
unit xTestSort
i: dates(5), nn
set dates := 1945, 1066, 1776, 1492, 800
do Sort(; dates)
loop nn := 1, 5
  at 10,10+20nn
  showt dates(nn), 4
endloop *
```
unit Sort(array) $ simple "bubble" sort
    i: array(*) $ array of integers to sort
    i: temp $ temporary work variable
    i: ii, jj $ loop indexes
loop jj := zlength(array), 2, -1
    loop ii := 2, jj
        if array(ii-1) > array(ii) $ exchange integers so larger number "bubbles" toward end
            calc temp := array(ii)
            array(ii) := array(ii-1)
            array(ii-1) := temp
        endif
    endloop
endloop

See Also:
    Alphabetize a List (p. 278)
**IF, CASE, and LOOP**

**if: IF Statements**

The *if-* family of commands allows conditional execution of parts of the program, depending on whether an expression in the *if-* or *elseif-* is TRUE or FALSE.

```plaintext
if  expression
  .  outif  expression
elseif  expression
  .  outif  $$ may have blank tag
else       $$ always blank tag
endif       $$ always blank tag
```

The "expression" in the tag of the *if-* and *elseif-* may be any expression. If this value is TRUE (negative), the indented commands following the TRUE expression are executed. After the indented commands are processed, execution continues with the command after the *endif*. If the expression is FALSE (zero or positive), the next *elseif-* (if any) is evaluated. An *else-* is always TRUE, so if execution reaches an *else-, the indented tags following it will always be done. An *if-* statement does not require *elseif-* or *else-.*

The indented commands in an *if-* structure must be indented with a TAB. A series of spaces is not equivalent. If you like, a period may be used before the TAB to make the indenting more noticeable.

The *endif-* marks the end of the area influenced by the *if-. The *if-* commands may be nested. That is, another *if-* sequence may be included as part of the code executed by an *if-* condition. Every *if-* must have its matching *endif-*.

The *outif-* command provides a convenient way to exit from an *if-* structure. If the tag of *outif-* evaluates to TRUE or is blank, execution resumes following the *endif* command. To exit from nested *if-*s, the *outif-* may be "exdented" to the indentation level of the *if-* from which you wish to exit. Note that *outif-* is executed only if the preceding statement is executed, regardless of the indent level.

**Examples:**

```plaintext
unit  xif1      $$ very simple *if-
  f:  a, b      $$ a & b get random values:
randu  a, 5       $$ 1 <= a <= 5
randu  b, 4       $$ 1 <= b <= 4
at 50,50
write a = <|s,a|>; b = <|s,b|>
if  a > b
  at 50,80
  write a is bigger than b
endif
*
unit  xif2      $$ nesting, *outif-
  i:  a, b      $$ try different values for a, b
calc  a := 4
calc  b := 6
at 50,50
write a = <|s,a|>; b = <|s,b|>
at 50,80
if  a != b      $$ outer *if-
```
See Also:
Logical Operators (p. 201)
Conditional Commands (p. 18)

**case: CASE Statements**

The `case` family of commands allows conditional execution of parts of the program, depending on the value of its tag. Sometimes a `case` statement is more convenient than an `if` structure. Moreover, if the selector and possible values are integers, a `case` statement can execute much faster than the equivalent `if` statements.

```
case selector
  1
    write one
    outcase expression
      write !
  2
    write 2
  else
    write other
endcase
```

The tag of `-case` is the "selector" (the variable or expression of interest). The possible values of the selector appear on succeeding lines. When the value of the selector matches a listed value, the indented code following the listed value is executed. An `-else` may be used for "all other values." Several values may be listed on one line. The selector must be an integer, float, byte, or marker variable.

```
case myvar
  1
    write myvar is 1
    outcase
      z=12
      write myvar = PI, z not equal to 12
  2,3,7
    write 2 or 3 or 7
  else
    write $ all other values
endcase
```
Every -case- statement must end with an -endcase- command. The -outcase- command provides a convenient way to exit from a -case- structure. If the tag of -outcase- evaluates to TRUE or is blank, execution immediately moves to the command following the -endcase-.

Internally, the operation of the -case- statement is basically the same as that of the equivalent -if- statement. There is a slight speed advantage gained by changing to -case- when there are multiple comparisons on the same line. There is a significant speed advantage if the comparisons are among integers.

Examples:

Here are equivalent examples using -case- and -if-. Note that variable expressions (such as "y+3") are allowed.

```
unit xcase1 $$ version using -case-
  i: y, z
  calc z := 47 $$ set y & z to some values
    y := 18
  at 15,15
  case z $$ z is the variable of interest
    40, 50
      write z is either 40 or 50
    y+3
      write z = y+3
    else
      write z is neither 40, nor 50, nor (y+3)
  endcase
*
unit xcase2 $$ version using -if-
  i: y, z
  calc z := 47 $$ set y & z to some values
    y := 18
  at 15,15
  if z = 40 | z = 50
    write z is 40 or 50
  elseif z = y+3
    write z = y+3
  else
    write z is neither 40, nor 50, nor (y+3)
  endif
```
**loop: Looping and Iterations**

The `-loop` family of commands is used to do repetitive operations. The commands in the body of the loop (the indented commands between `-loop`- and `-endloop-`) are executed over and over until something causes the loop to stop.

The `-outloop-` command is used to exit from a loop. If the tag of `-outloop-` evaluates to TRUE or is blank, execution immediately moves to the command following the `-endloop-`.

When `-reloop-` has a blank tag or when the expression in the tag is TRUE, execution immediately goes back and begins the next cycle through the loop.

There are three types of loops:

1) `-loop-` with no tag
2) `-loop-` with a logical expression
3) `-loop-` with specified iterations

The `-loop-` with no tag cycles until it is interrupted. It will go forever unless an exit is provided. The normal way to exit from such a `-loop-` is with an `-outloop-` command. A `-jump-` might also be used.

```
loop ...
  outloop  z = 8
  ...
endloop
```

The `-loop-` with a logical expression in the tag is repeated while the expression is true. The loop below is repeated as long as "a" is larger than 3. Since the `-randu-` is assigning random values into a, we cannot predict how many times the loop will be executed.

```
calc a := 0
loop a > 3
  ...
  randu a,4
  ...
endloop
```

When `-loop-` has an iterative tag (var := begin,end,increment), the loop is done repetitively while the variable "var" is increased by "increment" until it becomes greater than "end". The loop below is executed exactly 5
times, with count = 2, 4, 6, 8, and 10. The value of the indexing variable (i.e. "count") is NOT guaranteed after
the end of the loop. Its value may differ on different machines.

```
loop     count := 2,10,2
  \...
endloop  $$ at this point, the value
$$ of "count" is not known
```

If the "increment" in an iterative loop is omitted, it is assumed to be 1. The increment may be negative and/or a
fraction. When the increment is negative, "var" decreases from "begin" until it becomes less than "end".

In the case of nested loops, both -outloop- and -reloop- refer to the -loop- at the same indent level. You can
escape more than one level by placing the command directly under an outer -loop-:

```
loop   m:=1,5 $$ outer loop
  loop  s:=1,10 $$ inner loop
    outloop $$ leave inner loop
    outloop $$ leave inner loop
  endloop $$ end inner loop
endloop $$ end outer loop
```

Examples:

The example below draws a series of vertical lines using an iterative loop. The increment is 10, so the lines
appear at 50, 60, 70, etc.

```
unit xloop1
  i: x
  loop x := 50, 200, 10
    draw x,50; x,250
  endloop
```

When -loop- has a blank tag, some means must be provided for exiting from the loop or it will run forever. The
example below exits when count>6 or x>80.

```
unit xloop2 $$ use "Run from Selected Unit"
  i: x, count
  next xloop2
  calc count := 0
  loop
    calc   count := count + 1 $$ skip count = 3
    reloop count = 3 $$ skip count = 3
    outloop count > 6 $$ exit if count > 6
    randu x,100 $$ choose 0 < x < 101
    at 100, 50 + 15*count
    show count
    at 150, 50 + 15*count
    show x
    outloop x > 80 $$ exit if x > 80
  endloop
```

The -loop- in the next example uses an increment that is both fractional and negative.

```
unit xloop3
f: value
i: ypos
calc ypos := 20 $\$ position for 1st line
at 50,ypos
write value
at 120, ypos
write sqrt(value)
loop value := 2, 0, -.25 $\$ value = 2, 1.75, 1.5, 1.25 . . .
calc ypos := ypos + 20
at 50,ypos
showt value, 1, 2
at 120,ypos
showt sqrt(value), 1, 4
endloop
```

*See Also:
Logical Operators (p. 201)*
Random Variables

randu: Random Values

The -randu- command assigns a random value to the named variable.

\[
\begin{align*}
\text{randu} & \quad \text{variable} \quad \text{fraction between 0 & 1} \\
\text{randu} & \quad x, \text{maximum} \quad 1 \leq x \leq \text{maximum}
\end{align*}
\]

When -randu- has only one argument, the value selected is a fraction between 0 and 1. When the tag of -randu- has two arguments, the value selected is an integer ranging from 1 to the second argument.

Examples:

The first example below makes 500 random dots in the box. The second example just multiplies two fractions.

\[
\begin{align*}
\text{unit} \quad & x\text{randu1} \quad \text{-randu- with integers} \\
& \quad i : x,y,\text{count} \\
& \quad \text{box} \quad 0,0; 300,200 \\
& \quad \text{loop} \quad \text{count} := 1, 500 \\
& \quad . \quad \text{randu} \quad x,300 \quad 1 \leq x \leq 300 \\
& \quad . \quad \text{randu} \quad y,200 \quad 1 \leq y \leq 200 \\
& \quad . \quad \text{dot} \quad x,y \\
& \quad \text{endloop} \\
& \quad \text{at} \quad 180,130 \\
& \quad \text{write} \quad \text{all done} \\
& \quad * \\
\end{align*}
\]

\[
\begin{align*}
\text{unit} \quad & x\text{randu2} \quad \text{-randu- with fractions} \\
& \quad f : x, y \\
& \quad \text{randu} \quad x \quad 0 < x < 1 \\
& \quad \text{randu} \quad y \quad 0 < y < 1 \\
& \quad \text{at} \quad 100,50 \\
& \quad \text{write} \quad <s,x> \text{ times } <s,y> = <s, x*y> \\
& \quad * \\
\end{align*}
\]

See Also:
Embedding Variables in Text (p. 50)
Permutations (p. 221)

Permutations

Repeated execution of the -randu- command may choose the same numbers more than once. Sometimes it is useful to choose numbers just once from a list. This can be achieved by forming a "permutation" of the numbers. For example, a permutation of the first five integers is 43152 or 25132. Here is a routine to do this:

Examples:

\[
\begin{align*}
\text{unit} \quad & \text{GetPermutation} \\
& \quad i : \text{array(5)}, \text{index} \\
\text{next} \quad & \text{GetPermutation} \\
\text{do} \quad & \text{Permute( ;array)} \\
\text{at} \quad & 10,20
\end{align*}
\]
CALCULATIONS

| loop | index := 1,5 |
| write | <|s,array(index)|> |
| endloop |

* unit Permute(random) $\$$ make a permutation array
  i: random(*)
  i: size, available, choice, nn
  * For concreteness, suppose the array is 5 long.
  * Initialize the array to 1, 2, 3, 4, 5.
  * Choose a random number from 1 to 5. Suppose it is 2.
  * Swap the contents of element 2 and 5, so now the array holds 1, 5, 3, 4, 2.
  * Next choose a random number from 1 to 4. Suppose it is 3.
  * Swap the contents of element 3 and 4, so now the array holds 1, 5, 4, 3, 2.
  * Continue, reducing number of available slots, till all contents are permuted.
  calc size := zlength(random)
  loop nn := 1, size $\$$ initialize array to 1, 2, 3, 4, 5....
    calc random(nn) := nn
  endloop
  * After each choice, available slots reduced by 1:
  loop available := size, 2, - 1
    * Choose among available slots:
      randu choice, available
    * Save current contents of last slot:
      calc nn := random(available)
    * Move "choice" slot to last slot:
      calc random(available) := random(choice)
    * Move what was in last slot into "choice" slot:
      calc random(choice) := nn
  endloop
  *
6. Connecting Units & Programs

Units -- Program Subdivisions

unit: Basic Building Blocks

Every cT program is divided into subdivisions called "units". Each unit has a unique name, which is the tag of a -unit- command.

```
unit   somename
unit   calculate(a; var1)
  f: a, var1, tx, ty  $$ may have local defines
```

A unit name must begin with a letter and may contain letters, numbers, and the underscore character. It may be 30 characters long. Unit names cannot contain any punctuation marks or spaces. Unit names are case sensitive: "one" and "One" are different names. There is no specific command for "end-of-this-unit." A unit ends when another -unit- command (or end-of-file) is encountered.

A unit may use variables that are local to that unit. Local variables are defined on the lines immediately following the -unit-. The command field is blank; the definitions are a "continuation" of the -unit- specifications. The syntax for local variables is the same as the syntax for global variables (see -define-). If no local variables are used, all global variables are automatically available to the unit. If the unit will use both global and local variables, the first line of the definitions must be "merge,global:“, and named groups of variables may also be referred to

```
unit myunit
  merge,global: $$ use global defines
  f: a, b, c $$ local variables
  merge,mygroup: $$ use a named group of variables
```

A unit receives information from a -do-, -jump-, or -menu- command by specifying arguments to receive the information. A unit may receive 10 pass-by-value arguments plus 10 pass-by-address arguments. The arguments should be defined as local variables.

```
unit somename(arg1; place1, place2)
  f: arg1, place1, place2
```

The variable named in the first argument (arg1) receives information that is "passed by value". The arguments after the semicolon (place1 and place2) receive information that is "passed-by-address". The arguments received by a -unit- must match in number the arguments sent by the -do- or -jump-. For pass-by-address variables, the variable type (i.e., integer, marker) must also match. The discussion of the -do- command includes details about passing arguments.

Example:

```
unit xunit  $$ unit named "xunit"
i: N  $$ local variable
randu N, 10  $$ random integer into N
at 50,50
write  The local variable
       "N" is <|[s,N]|>.
```
See Also:
Moving between Main Units (p. 237)
define Global Variables and Groups (p. 193)
Summary of Variable Definitions (p. 190)
Local Variables (p. 195)
do Calling a Subroutine (p. 224)
IEU The Initial Entry Unit (p. 227)
Pull-down Menus (p. 136)
imain Modifying Every Unit (p. 232)
arrow Arrow Initializations (p. 173)
judge Judge Initializations (p. 175)
eraseu Erasing after a Response (p. 175)
Unit Markers zcurrentu & zmainu (p. 336)

do: Calling a Subroutine

The -do- command executes the unit named in its tag as a subroutine. When a -do- is encountered, the commands in the named unit are executed, and then the commands following the -do- are executed.

\begin{verbatim}
  do thatunit
  do someunit(4.7, lth; zonk, zip)
  do \expression \one \two(a,b) \final
     $$ conditional -do-
  do (marker)(1,2) $$ unit name in a marker; details below
\end{verbatim}

Pass-by-value: The -do- command optionally may pass arguments to the subroutine. The arguments that appear before the semicolon are "passed by value," and the subroutine can use those values but can't change the original variables (for a partial exception, see the topic "do Passing Markers"). Consider the following use of pass-by-value arguments:

\begin{verbatim}
  calc initial := 10
  do ShowSum(initial, 3.2)
  ...
  unit ShowSum(xx, yy)
  float: xx, yy
  show xx := xx+yy $$ does not change "initial"
\end{verbatim}

The effect is as though you had assigned the value "initial" to xx and the value "3.2" to yy:

\begin{verbatim}
  calc xx := initial
  yy := 3.2
  show xx := xx+yy
\end{verbatim}

Pass-by-address: In contrast to the pass-by-value arguments, those arguments that come after the semicolon in the -do- statement are "passed by address": the subroutine is told where in the computer's memory this variable is located, which makes it possible for the subroutine to alter the variable, not just use its original value. For that reason, pass-by-address can be used to get a calculated result back from a subroutine. Consider the following sequence, in which the pass-by-value variable "num" is unaffected but the pass-by-address variable "result" is changed:

\begin{verbatim}
  do Solve(num; result)
  ...
  unit Solve(n; x)
  i: n, x
\end{verbatim}
calc n := n+10  $$ does NOT affect "num"
     x := x+n  $$ equivalent to result := result+num+10

When using pass-by-address, the variable types must match. That is, you cannot pass-by-address an integer to a
float, or a float to an integer. With pass-by-value, however, you can mix integers and floats, and the necessary
conversions are performed automatically (e.g., a float is rounded to the nearest integer). Even with pass-by-value
you can't mix markers with integers or floats, because text and numbers are treated quite differently.

**Number of arguments:** The -do- command can have up to 10 pass-by-value and 10 pass-by-address
arguments. The number of arguments in the receiving -unit- must match the number of arguments in the -do-. If
there are no pass-by-value arguments, an initial semicolon is needed to show that fact:

```
do sub(; pos, speed, time)
```

**Arrays passed-by-address:** An individual element of an array can be passed-by-value to a subroutine, but a
whole array cannot be passed-by-value. Whole arrays can, however, be passed-by-address:

```
unit DataHandler
    i: data(50, 10)
...
do Sum(; data)  $$ just name the array; no indices
unit Sum(; work)
    i: work(*, *)  $$ array bounds will be filled in
    $$ do calculations on "work(i, j)"
```

Note carefully that the -do- statement contains the name of the array *without any indices*, and that the definition
of the "work" array *must have asterisks* to indicate unspecified array lengths. Two asterisks are necessary because
the array "data" has two dimensions.

The `zlength` function tells the total size of an array. In the case above of a 50*10 array, "zlength(data)" or
"zlength(work)" is equal to 500.

**Pass-by-address required:** Other complex variables such as file descriptors and screen variables can be
passed-by-address to subroutines but cannot be passed-by-value.

**Unit names in markers:** The unit name can be in a marker variable:

```
unit mtest
    marker: sub1
calc sub1 := "TryIt"
...
do (sub1)(10,20)  $$ equivalent to -do   TryIt(10,20)-
```

**Examples:**

Unit "xdo1" illustrates pass-by-value. The arguments may be numbers, variables, or expressions.

```
unit xdo1
    f: tmp
calc tmp := .7
at 100,50
do thatunit(3, tmp, 12)  $$ 3 arguments passed
at 100,120
```
do thatunit(sin(2*tmp), -13, 5.6E+4)
*
unit thatunit(a, b, c) $$ receives 3 arguments
  f: a, b, c
write a = <|s, a|>
  b = <|s,b|>
  c = <|s,c|>
*

Units "xdo2" and "xdo3" differ only in the way K is passed to the subroutine. In "xdo2", the value of K is passed, and that value is not affected by what happens in "xdo2a".

unit xdo2
  i: K
calc K := 45
do xdo2A( K) $$ pass value of K
at 100,150
write back in xdo2
  K = <|s,K|> $$ K unchanged
*
unit xdo2A( N)
  i: N
at 50,50
write Press "s" to stop the counter.
loop
  calc N := N+1
  erase 100,100; 150,150
  at 100,100
  show N
  getkey
outloop zkey=zk(s)
endloop
*

In unit "xdo3", the -do- command has a semicolon before K. The address of K is passed, and the value of K is changed by "xdo3a".

unit xdo3
  i: K
calc K := 45
do xdo3A( ;K) $$ pass address of K
at 100,150
write back in xdo3
  K = <|s,K|> $$ K has new value
*
unit xdo3A( ;N)
  i: N $$ N same type (integer) as K
at 50,50
write Press "s" to stop the counter.
loop
  calc N := N+1
  erase 100,100; 150,150
  at 100,100
  show N
. getkey
outloop zkey=zk(s)
endloop
*

See Also:
do Passing Markers (p. 253)
Units -- Program Subdivisions (p. 223)
Local Variables (p. 195)
Logical Operators (p. 201)
Using Arrays (p. 208)
Conditional Commands (p. 18)

IEU: The Initial Entry Unit

The IEU, or initial entry unit, refers to those commands that appear before the first -unit- command. These commands are always executed when the program is entered. After the IEU is finished, execution proceeds to the first (physical unit) in the program. In editing mode, when "Run" or "Execute" from selected or current unit is chosen, execution proceeds from the IEU immediately into the specified unit.

The -define- command (for global variables and groups of variables) must be in the IEU. Other commands that describe the environment of the program are usually placed in the IEU so that the editing environment will be correct when running or executing any unit. The IEU might look like this:

```plaintext
define i: DIM1=50, DIM2=100
      MaxTrials = 5
      f: MyArray(DIM1, DIM2)
font zsans,12
fine 500,500
rescale TRUE, TRUE, FALSE, TRUE
icons "myicons"
imain MyMainUnit
do BasicMenus
do InitializeVariables
color zblack,zcyan $$ establish color environment
wcolor zcyan
erase
*
```

If there are -use- files, their IEUs are executed after the original IEU, and before starting the first main unit in the original file. This makes it possible for each -use- file to do some of its own initializations. In the original IEU you should not -do- a routine in a -use- file, because that file might not yet have executed its own IEU to initialize things.

See Also:
jump Jumping to a New Topic (p. 239)
fine Declaring a Screen Size (p. 35)
rescale Adjusting the Display (p. 37)
font Selecting a Typeface (p. 43)
Main Units

A program can move from unit to unit with sequencing commands such as -next-, -back-, and -jump-. A unit reached through the action of such sequencing commands is a "main unit." The main units are the anchor points around which the flow of the program is built.

Note: Main units and window reshapes or redispays are treated below, under "Restarting main units."

In many programs, a process (such as a payroll calculation) starts at the beginning and works in a straight line until the process is done. cT is designed for programs where the path may not be a straight line. The user may want to repeat one section several times, review previous sections, consult a glossary, or go back to an index and start over. In this situation, the program must be a collection of relatively independent modules linked together for easy mobility.

Program execution starts with the IEU (Initial Entry Unit), the statements preceding the first -unit- command. If there are -use- files, their IEUs are also executed. Then cT starts executing the first physical unit in the original file.

After the first unit, which is always a main unit, the order of execution is completely controlled by sequencing commands. When the last command in a main unit has been executed, execution is suspended and does not resume until the user takes some action. The possible actions (not necessarily all available in every unit) are

- Press ENTER
- Select (Next Page) from menu
- Select (Back) from the menu
- Select Quit from the menu
- Select some other menu item
- Click the mouse

ENTER and (Next Page) are available if a -next- command is active. If -enable touch- is active, clicking the mouse is equivalent to pressing ENTER. (Back) is activated with a -back- command. Quit is always available (shown as Quit running when in programming mode). You can provide other menu items with the -menu- command.

The end of the program is reached when a unit ends and there are no options for the user to "move somewhere else." That is, the user has completed a main unit that contains no sequencing commands and that has no menu choices (except Quit). Pressing ENTER at this point exits from the program. You can also terminate a program by executing a -jumpout- command. Authors frequently store miscellaneous routines toward the end of the file, so that the final unit in the execution of a program is usually not the last unit in a file.

The name of the current main unit is contained in the system variable zmainu.

One unit may execute another unit with a -do- command. A unit reached with a -do- command is a subsidiary unit and is called a "subroutine." At the end of a subroutine, execution continues with the command that follows the -do-. The -imain- command allows a subroutine to be do-ne at the beginning of every main unit. In principle, the same unit could be (at different times during the execution of a program) both a main unit and a subroutine. In practice, a subroutine is usually a small, special-purpose unit that does not contain sequencing commands.

Some things are initialized whenever a main unit is executed; see "Main Unit Initializations." The -imain- command lets you specify a unit to be do-ne at the beginning of every main unit, to perform additional initializations of your own.
**Restarting main units:** When the display needs to be restored (because the window size changed or another window passed in front), execution recommences from the beginning of the current main unit. This means that you must be prepared to start a main unit over at any moment, because you do not know when the user may reshape the window, or bring the window forward from behind some other windows.

In all computer programs on modern machines, not just programs written in cT, if you erase any part of a window (by going to a different application or moving another window in front of the existing window), upon returning to the program the program is told to reexecute itself to restore the window display, because the underlying operating system does not save a copy of the display that it destroyed! For example, when you go from a word processor to another application and back to the word processor, the word processor has to reexecute itself to redisplay the contents of its window. Often this redrawing is less noticeable than in a cT program, because it doesn't erase the entire window first as cT presently does, if only a portion of the window was erased/occluded.

The mechanism that cT provides to you for coping with this issue is associated with "main units." The first unit in the program is a main unit, and any unit reached by -jump- or -next- or -back- (but not by -do-) is a main unit. Any time the user returns to cT, the window is erased and the current main unit is re-executed, from the beginning of that main unit. In trivial cases this automatically restores the screen, if executing from the beginning of the unit is sufficient to restore the display.

However, if in one main unit the user has done many things that affected the display, the user certainly doesn't want to go through all that again to get back to where he or she was. That means you have a problem, unless you can organize the program to -jump- to a new main unit whenever a particular stage in the interaction has been completed. This unfortunately puts a programming burden on you that cT is powerless to handle automatically for you. Essentially, what you must do is keep track of what things the user has done, and use that stored information to restore the screen appropriately.

One reasonable scheme is to use a global variable to track how far through the unit the user has gotten, and use the contents of that variable to decide what graphics to restore to the screen. A related scheme is to take an "object-oriented" approach in which there is a global status variable corresponding to each major graphical element of the screen, and at the beginning of the main unit you use these status variables to decide what to display. Note that these global variables need to be initialized before jumping into the main unit.

One other cT feature that can be useful in managing the window-redrawing problem is that the system variable zreshape is TRUE at the beginning of a main unit if that main unit has been reexecuted due to the window being brought forward after being occluded, or the window being reshaped. You can check the value of zreshape at the beginning of a main unit to see whether this is a reexecution rather than a first execution.

**Examples:**

Use "Run from Selected Unit" to try out this example. Try reshaping the window while you are running. This "program" has no end, because both main units contain sequencing commands.

```cT
unit      xMain     $$ a main unit
next      xMain2    $$ sequencing command
at 38,29
write     Hello! This is "xMain".
do        xdone1    $$ do-ing a subroutine
do        xdone2    $$ do-ing a subroutine
at zxmax - 60, zymax - 35
write     Press <[cr]>ENTER.
back      xunit     $$ unit named "xunit"
*
unit      xdone1    $$ a subsidiary unit
```
Here is an example in which a global status variable (HowFar) keeps track of how far the user got through the main unit. Try reshaping the window at various stages to see what happens.

```plaintext
define group,status:
i: HowFar   $$ 1, 2, 3, 4 for stages 1, 2, 3, 4
i: SHOW=10, ERASE=11  $$ options for displaying or erasing
*
unit xInitialize
merge,status:
calc HowFar := 1   $$ initialize HowFar
jump xFirst $$ make "First" be the main unit, the anchor point
*
unit xFirst      $$ the main unit, will restart here
merge,status:
do xStage(HowFar,SHOW)
* Interact with user, incrementing HowFar as they pass various stages.
* Do not present information inappropriate for user who has passed a stage.
if HowFar = 1
   pause keys=all,touch
   do xStage(HowFar,ERASE)
   do xStage(HowFar := HowFar+1,SHOW)
endif
if HowFar = 2
   pause keys=all,touch
   do xStage(HowFar,ERASE)
   do xStage(HowFar := HowFar+1,SHOW)
endif
if HowFar = 3
   pause keys=all,touch
   do xStage(HowFar,ERASE)
   do xStage(HowFar := HowFar+1,SHOW)
endif
```
unit xStage(nstage,display) $$ display graphics for a stage
merge,status:
i: nstage $$ which stage we're in
i: display $$ = SHOW if should display, ERASE to erase
case display
SHOW
  mode write
ERASE
  mode erase
case nstage
1
  *make initial display
  fill 25,20;130,75
  at 140,30
  write Stage 1
2
  *make display appropriate to user who has passed stage 1
  vector 12,12;140,80
  at 100,35
  write Stage 2
3
  *make display appropriate to user who has passed stage 2
  circle 60,15;85,95
  at 125,50
  write Stage 3
4
  *make display appropriate to user who has passed stage 3
  disk 60,40;160,60
  at 91,74
  write Stage 4
endcase
mode write

See Also:
Main Unit Initializations (p. 232)
Moving between Main Units (p. 237)
Summary of Variable Definitions (p. 190)
Local Variables (p. 195)
IEU The Initial Entry Unit (p. 227)
do Calling a Subroutine (p. 224)
imain Modifying Every Unit (p. 232)
Pull-down Menus (p. 136)
arrow Arrow Initializations (p. 173)
ijudge Judge Initializations (p. 175)
eraseu Erasing after a Response (p. 175)
jumpout Jump to Another cT Program (p. 241)
Main Unit Initializations

Many initializations are done at the beginning of a main unit:

- the screen is cleared
- the mode is set to "write"
- the current screen position is set to 0,0
- the left and right margins are set to the edges of the display
- all variables associated with response handling are initialized
- inhibit/allow defaults

Some modifications, once made, last for the entire program unless they are explicitly changed:

- the selected font and icons
- the cursor icon
- the graph origin and scaling
- the size and rotate for relative commands
- the -fine- and -rescale- settings
- the -imain- command
- the menus
- clipping

NOTE: Because the program is sensitive to the "history" with respect to these commands, it is important to realize that "Execute Selected Unit" may behave differently from "Run from Beginning." If the same parameters will last for the entire program, the -font-, -icons-, -fine-, and -rescale- commands should go in the IEU, the statements preceding the first -unit- command.

See Also:

IEU The Initial Entry Unit (p. 227)
fine Declaring a Screen Size (p. 35)
rescale Adjusting the Display (p. 37)
font Selecting a Typeface (p. 43)
icons Selecting an Icon (p. 93)
imain Modifying Every Unit (p. 232)
arrow Arrow Initializations (p. 173)
ijudge Judge Initializations (p. 175)
Judging System Variables (p. 330)

imain: Modifying Every Unit

The -imain- command causes the unit named in its tag to be executed at the beginning of every subsequent main unit as if it had been inserted into that unit with a -do- command. It is used to insert code that needs to be done for every new main unit, such as data-keeping or a fancy frame around the display. The special tag "q" cancels the effect of any previous -imain- command.

```
imain someunit
imain q $$ cancel previous imain setting
```

The -imain- command typically belongs either at the very beginning (in the IEU) or in a unit that serves as a title-page or index. One imain unit is often appropriate for an entire program.

The imain unit can be changed at any time by an -imain- command, but the effect is not felt until the next main unit. Notice that this can be a bit tricky, because if the screen changes size, the current main unit is reexecuted.
and the *newly specified* main unit is executed. This may not be what you want. In such cases, a sequence such as the one below is required:

```plaintext
imain  firstmain
*
unit  one
next  oneb
* some code in this unit
*
unit  oneb
imain  secondmain
jump  two
*
unit  two
*
```

The unit *firstmain* is the imain unit for unit *one*. If the screen is reshaped, *firstmain* is still the imain unit. When the user presses ENTER to continue to the next unit (unit *two*), control first passes to unit *oneb* which does nothing at all except reset the imain unit and jump to unit *two*. This "protects" the imain unit while the user is in unit *one*, but allows a different imain unit to be executed for unit *two*.

**Example:**

To try this sequence, put the editing cursor in unit "xImainZero", make this the "Selected unit", and use "Run from Selected Unit." Unit "xImainZero" activates the -imain- unit. The -jump- causes the program to go immediately to unit "xImainTitle", so there is no pause between "xImainZero" and "xImainTitle". Note that the -menu- option (Quick Grammar) is active in unit "three" even though the reminder message does not appear.

```plaintext
unit  xImainZero
imain mymain $$ activate imain unit
menu  Quick Grammar: quickgram
      $$ might zero variables here . .
jump  xImainTitle $$ no hesitation here
*
unit  xImainTitle
      $$ as if -do  mymain- were at this point
next  xImainOne  $$ sequencing command
text  0,100

    This is a Fancy Title
\
*
unit  xImainOne
      $$ as if -do  mymain- were at this point
next  xImainTwo  $$ sequencing command
do  message(1, 50)
*
unit  xImainTwo
      $$ as if -do  mymain- were at this point
imain  q  $$ no -imain- in next main unit
next  xImainThree  $$ sequencing
do  message(2, 100)
*
unit  xImainThree
do  message(3, 150)
```
outunit: Exiting from a Unit

The -outunit- command allows premature exit from a unit. If the tag is blank or evaluates to TRUE, execution goes to the end of the unit.

```
  outunit  $$ blank-tag, unconditional exit
  outunit  expression  $$ exit if TRUE
```

The -outunit- command is not the same as a -jump- command. The -outunit- merely moves to the end of the unit, thus skipping all intervening commands. It does not change the main unit and it does not change any sequencing commands (-next- or -back-) that are in effect. The -jump- command is much more drastic; it causes the program to "forget" the current status and to move to an entirely new place.

Code using -outunit- can always be treated more formally by a series of -if-s. However, when a series of -if-s is very deeply nested and/or very lopsided, it may be clearer and more convenient to use -outunit-.

Examples:

In the following example, -outunit- is used to terminate a recursive routine:

```
  unit     xoutunit
  rorigin  150,150
  inhibit  startdraw  $$ prevent first -rdraw- segment
```
do spiral(720, 100, 0.97)
*
unit spiral(angle, length, factor)
  f: angle, length, factor
outunit angle < 10
rotate angle
rdraw ; length, length
do spiral(angle*factor, length*factor, factor)
*

The following units -xoutunit1- and -xoutunit2- do exactly the same thing. They are not equivalent if code is added after the last -endif-. In the second unit, the indenting makes it immediately obvious that there is a deeply indented structure. However, it is very bulky and may be difficult to read. The first unit is very compact, but gives fewer cues about the underlying structure.

unit xoutunit1 $$ example using -outunit-
i: a,b,c,d
calc a := b := TRUE $$ choose some values
c := d := FALSE
at 50,50
write \ a \This is A\ This is not A
outunit not(a)
write \ b \ and B\ but not B
outunit not(b)
write \ c \ and C\ but not C
outunit not(c)
write \ d \ and D\ but not D
*
unit xoutunit2 $$ example using -if-
i: a,b,c,d
calc a := b := TRUE
c := d := FALSE
at 50,50
if a
  . write This is A
  . if b
    . write and B
  . if c
    . write and C
  . if d
    . write and D
  . else
    . write but not D
  . endif
  . else
    . write but not C
  . endif
else
  . write This is not A
endif $$ this is the last command in the unit
See Also:

if           IF Statements     (p. 215)
Logical Operators  (p. 201)
Moving between Main Units

(next: Moving Ahead)

When program execution reaches the end of a main unit, execution stops and waits for the user to take an action. The -next- command gives the name of the unit that will be executed when the ENTER key is pressed. The -next- command also causes the option (Next Page) to appear on the menu when the current main unit is finished.

next someunit
next \expression\negunit\posunit\greater
next \okays=17\review
next \okays=17\x\review $$ x \text{ same as nothing}
next q $$ cancel next setting

The -next- command does not cause an immediate action. Instead, it sets a flag that says "when the end of the unit is reached and the user presses ENTER, then move to unit X." There are three equivalent user actions that will move to the "next" unit:

1 - Press ENTER (or the equivalent button on your machine)
2 - Select (Next Page) from the menu
3 - Click the mouse, if -enable touch- has been activated

The "q" tag cancels any next instruction that was given earlier. Either nothing or an "x" means "do nothing" -- do not change the existing -next- unit.

Every main unit must have some way of specifying "what to do next." This is often a -next- command, or it can be some -menu- option. The end of the program is reached when execution comes to a main unit that has no directions for moving to another unit (no -next-, no -back-, no active -menu-).

The name of the current main unit is contained in the system variable zmainu.

The unit name can be in a marker variable:

unit mtest
  marker: sub1
  calc sub1 := "TryIt"
  ...
  next (sub1) $$ equivalent to -next TryIt-

Examples:

This example moves from unit "xnextOne" to unit "xnextTwo". Unit xnextTwo contains no sequencing instruction, so when the user presses ENTER or chooses (Next Page) on the menu, he or she will leave the program. Use "Run from Selected Unit" to try these examples.

unit xnextOne
next xnextTwo
at 50,50
write This is "xnextOne."
*
unit xnextTwo
at 100,50
In this second set of example units, every (main) unit has a -next- command, so there is no end-of-program. To exit from the program the user must select Quit from the menu.

```
unit xNextA
next xNextB $$ set next unit = "xNextB"
at 50,50
write This is unit xNextA.
do continue $$ instructions
*
unit xNextB
f: temp
next xNextA $$ set next unit = "xNextA"
randu temp,3 $$ choose random #: 1, 2 or 3
next \temp=1\ xNextC \ x $$ modify "next" unit
at 100,50
write This is unit xNextB.
```

The current value of temp is <|s,temp|>.

```
do continue
*
unit xNextC
next xNextA $$ remove this line, and try again
at 50,20
write This is unit xNextC.
```

We arrived in unit xNextC because the expression "temp=1" was true, so that the -next- setting in unit xNextB was changed from "xNextA" to "xNextC".

```
do continue
*
unit continue
text 0,zymax-40; zxmax,zymax
To continue, press ENTER, or select (Next Page) from the menu.
```

See Also:
- Main Units (p. 228)
- imain Modifying Every Unit (p. 232)
- unit Basic Building Blocks (p. 223)
- enable Allowing Mouse Input (p. 132)
- Logical Operators (p. 201)
- Conditional Commands (p. 18)
**back: Reviewing Previous Material**

The -back- command provides a convenient way for the user to review a previous portion of the program. The -back- command causes a *(Back)* choice to appear on the menu. When BACK is chosen, execution moves immediately to the unit named in the argument of the -back- command. The user does not need to be at the end of the unit to choose *(Back)*.

```
back   one
back   q      $$ cancel back setting
```

Frequently the *(Back)* choice returns the user to the previous "page" or topic. The -back- command is optional; the author must decide when it is appropriate to provide for review.

The name of the current main unit is contained in the system variable *zmainu*.

The unit name can be in a marker variable:

```
unit    mtest
        marker: sub1
calc    sub1 := "TryIt"
...
back    (sub1) $$ equivalent to -back    TryIt-
```

**Example:**

Use "Run from Selected Unit" to try this example. Use the menu for *(Back)*.

```
unit    xbackOne
next    xbackTwo    $$ set "next" unit
at      50,50
write   hello
*
unit    xbackTwo
back    xbackOne    $$ set "back" unit
at      150,100
write   hello, again
*
```

**See Also:**
- Main Units (p. 228)
- imain Modifying Every Unit (p. 232)
- unit Basic Building Blocks (p. 223)
- enable Allowing Mouse Input (p. 132)

**jump: Jumping to a New Topic**

The -jump- command provides a way to move immediately to another unit without waiting for the user to select *(Next Page)*, *(Back)*, or some other menu item.

```
jump    someunit
jump    \expression\unitneg\unitzero\unitplus
jump    myunit(a,b,c; zz1,zz2)
```
The -jump- action occurs as soon as the command is encountered, and commands following the -jump- are not executed. After a -jump- has been made, the new unit becomes the "main" unit and the program "forgets" where it came from. All of the usual main unit initializations are done, including erasure of the display and execution of any -imain- unit. Note that -jump- is a much more drastic operation than -do-.

The -jump- command essentially lets you divide your program into several different subprograms, each with its own distinctive displays and interactions, but sharing many of the same subroutines (-do-).

The -jump- command passes pass-by-value arguments the same way as the -do- command. The -jump- command cannot pass arguments by address. (Refer to the -do- write-up for information about passing arguments.) Although the -jump- command allows passing of arguments to main units, this is not advised. If the display is reshaped, execution starts again, but the arguments are not passed again. Numeric local variables but not other kinds of local variables retain the values they had just before the reshape.

The unit name can be in a marker variable:

```
unit mtest
marker: sub1
calc sub1 := "TryIt"
... jump (sub1) $$ equivalent to -jump TryIt-
```

Example:

This rather silly example shows some numbers and then immediately jumps to "xjump2". Notice that there is no user input, and no -next- command involved.

```
unit xjump
i: count
loop count := 1, 10
  . showt count, 3
  . pause 0.1
endloop
jump xjump2
*
unit xjump2
at 100,100
write after the -jump-
*
```

See Also:
- Main Units (p. 228)
- imain Modifying Every Unit (p. 232)
- do Calling a Subroutine (p. 224)
- Conditional Commands (p. 18)
- jumpout Jump to Another cT Program (p. 241)
Connections to Other Programs

Overview of Program Connections

Several commands are needed to allow one to

1) Stop a cT program in the current window and start a new one (-jumpout-)
2) Initiate a new (independent) program in a new window (-execute-)
3) Initiate a new program and send information back and forth (-execute-, sockets)
4) Call a subroutine in some other language (e.g., C or Pascal) (not yet available)

The -execute- command for initiating another program is currently available only on Unix machines. cT also offers the ability for different programs to communicate with each other through sockets (see the -socket- command, which can be used on Unix and Macintosh).

See Also:
Overview of Sockets (p. 311)

jumpout: Jump to Another cT Program

The -jumpout- command provides a way to quit running a program and (optionally) start running a different program:

jumpout       $$ no tag, ends the program as though user had chosen "Quit"
jumpout q    $$ q is equivalent to no tag
jumpout "test.t"  $$ marker expression naming a program to jumpout to
jumpout file name, arg1, arg2... $$ can pass arguments to the new program
jumpout \expr \x \ q  $$ x = do nothing; q or blank = quit the program

The first unit of a program may have value arguments of type byte, integer, float, or marker. A -jumpout- command may or may not pass arguments to a program that either does or doesn't have arguments. It is legal to pass no arguments to a program whose first unit has arguments, in which case these arguments will be zero or zempty. It is also legal to pass arguments to a program whose first unit has no arguments. However, if any arguments are passed to a program that accepts arguments, the number of arguments and the numeric/marker types must match.

If the -jumpout- command cannot execute because the target program doesn't exist, or because there isn't enough memory, the command will fail and fall through to the next command with zreturn set as for file operations.

When running in the "create" environment, the target of the -jumpout- is the source file, and the cT program will be automatically compiled if necessary. When running in the "execute" environment, the target is the binary file.

The system marker variable zfromprog gives the name of the program that did a -jumpout- to the current program (zfromprog = zempty if this is the first program to execute). You can jump back to the original program simply by -jumpout zfromprog-.

execute: Initiating Another Program

The -execute- command initiates another program without halting the current program. The new program is not connected in any way to the current program, unless the two programs use "sockets" to connect to each other.
NOTE: The -execute- command is currently available only on computers running the Unix operating system.

execute "other.t"  $$ equivalent to "ct -x other.t"
execute "graph"  $$ start program "graph"
execute "edittext proghelp"  $$ edit file "proghelp"
execute "prdoc ~zz09/zip.d"  $$ print a file
execute m1 + ".data"  $$ string expression

The tag of the -execute- command may be anything that would make sense as a "typescript" command. As a special case, any file named with a ".t" file extension will cause that file to be executed as a cT file. These two commands are equivalent:

execute "myfile.t"
execute "ct -x myfile.t"

The tag may include several "elements" separated by spaces. For each element, the directory in which the current cT program resides is examined. If the element is found in that directory, the -execute- command extends the name to a full path name (e.g., "proghelp" might be changed into "/cmu/csw/zz09/myprogs/proghelp"). If the element is not found in the current directory, it is passed on unchanged.

This procedure ensures that someone using your program will get the appropriate associated file or program. If the source file and all auxiliary files are moved to a different directory, the -execute- command will still work properly.

The file name can be built up of literal strings and marker variables:

execute "new"+myfile+version+".t"

Be Careful! If you say -execute "ls"- (ls is a Unix instruction) and your directory happens to contain a file named "ls", then the command issued will be the full path name for your "ls" and not the Unix command ls.

If the window size changes (causing the current main unit to be reexecuted), you probably don't want the -execute- command to be done again, since that would start up a second copy of the new program. The system variable "zreshape" is useful for avoiding an -execute- after the window changes size.

Example:

unit xexecute
if not(zreshape) $$ reshape not TRUE
execute "ls " + zhomedir
endif
at 50,50
write Look over in your typescript window.
You will see a list of the files in your home directory.
*

See Also:
File Name Specification (p. 286)
Overview of Sockets (p. 311)
**use: Using Library Files**

The -use- command specifies the name of a file whose contents will be included as part of the current program. This makes it possible to share routines among multiple cT programs. For example, a series of programs about astronomy could all refer to a single file containing a map of the heavens. Then, if the map is modified, all programs would see the new map.

```
use "filename"
```

The -use- command must be in the IEU (the initial entry unit). When the program is executed, the IEU of the originating program is executed, followed by the IEU of each of the -use-d files. A program may have multiple -use- commands. Also, a -use-d file may contain a -use- command, but -use-s may not be nested more than 10 deep.

The global variables declared in the IEU of a -use-d file are local to that file, except for named groups that can be referred to by "merge,name" in another file. Variable values may also be exchanged between files by using argumented -do- commands.

The -use- command resembles the linked-library facilities of C and other languages; each file has its own internal variables and its own IEU. Named groups of variables can be shared among all the files.

The $syntaxlevel statements at the beginning of all -use-d files must be the same.

It is very useful to be able to look at or change -use- files while working on the main program. The "Auxiliary file" option on the File menu lets you edit other files while maintaining control of the main source file.

**Example:**

This example uses five separate files to illustrate the hierarchy of -use-d files. The basic program is in Xuse.t. There are four subsidiary files: used1.t, used2.t, deeper.t, and stilldeeper.t. (This is a purposely complicated example. A more normal situation would be a "base" file with one or two -use-d files.)

The variable name "qqq" was intentionally used in several files to illustrate that these variables are indeed independent. Such duplication is certainly not recommended programming practice! Also note the use of group definitions and merge operations.

Notice how an embedded carriage return, <|cr|>, is used to move down one line after each comment. This makes a nice display without requiring a lot of -at- commands.

If you executed the program below, you would get this output:

```
Hello, this is the IEU of "Xuse.t"
More in the IEU of "Xuse.t"
This is the IEU of "used1.t"
IEU of "deeper.t"
IEU of "stilldeeper.t"
IEU of "used2.t"
county=79, add1 shows (a+b) = 3
Hello from "stilldeeper".
In "used1.t", qqq = 1
In "used2.t", qqq = 2.371
This is unit zip in deeper.t.
Back in "Xuse.t", qqq = 55
```
CONNECTING UNITS & PROGRAMS

**Stored in file Xuse.t**

```plaintext
$syntaxlevel 2
at 20,20
write Hello, this is the IEU of "Xuse.t" <cr>
use "used1.t" $$ make available routines in file used1.t
use "used2.t" $$ make available routines in file used2.t
define i: qqq $$ make qqq available in this file
write More in the IEU of "Xuse.t" <cr>
*
unit usexample
merge,global:
merge,mapvars: $$ defined in used1.t
calc qqq := 55 $$ assign a value to qqq for this file
write county=<s,county>, $$ from mapvars group
do add1(1,2) $$ from used1.t
do something $$ from deeperstill.t
do show1 $$ from used1.t
do show2 $$ from used2.t
do zip $$ from deeper.t
write Back in "Xuse.t", qqq = <|s,qqq|> <cr>
************
```

**Stored in file used1.t**

```plaintext
$syntaxlevel 2
define b: qqq $$ this qqq valid only in used1.t
group,mapvars: $$ a named group
  i: state, county, city
* Cannot reference group variables directly in the IEU, so do this:
do setup $$ initialize some group variables
* here is a -use- inside a -use-d file:
use "deeper.t" $$ make available routines in file deeper.t
*
unit setup
merge,b:
calc qqq := 1
county := 79
write This is the IEU of "used1.t" <cr>
*
unit add1(a,b) $$ uses values passed from Xuse.t
  i: a,b
write add1 shows (a+b) = <|s,a+b|> <cr>
*
unit show1
write In "used1.t", qqq = <|s,qqq|> <cr>
************
```

**Stored in file used2.t**

```plaintext
$syntaxlevel 2
define f: qqq $$ this valid only in used2.t
calc qqq := 2.371
write IEU of "used2.t" <cr>
```

244
*  
unit show2  
write In "used2.t", qqq = <|s,qqq|> <|cr|>  
*************  

Stored in file deeper.t

$syntaxlevel 2
* here is a -use- in a -use-d file inside a -use-d file :
use "stilldeeper.t" $$ get routines from stilldeeper.t
write IEU of "deeper.t" <|cr|>  
*  
unit zip  
write This is unit zip in deeper.t. <|cr|>  
*************  

Stored in file stilldeeper.t

$syntaxlevel 2
write IEU of "stilldeeper.t" <|cr|>  
*  
unit something  
write Hello from "stilldeeper". <|cr|>  
*******

See Also:
File Name Specification (p. 286)
define Global Variables and Groups (p. 193)
do Calling a Subroutine (p. 224)
IEU The Initial Entry Unit (p. 227)
7. Character Strings

Introduction to Strings

A "string" is a series of characters, such as "Hello there!" or "x^2 + 3x + 7." In cT such a string may involve styles such as bold or italics, or may even contain images or Japanese characters. We need to be able to manage such complex text—to store, display, and manipulate it. For this task, cT uses variables called "string markers," or more simply "markers."

A block of new text can be created in computer memory with -calc- or -string- (or by using -datain- to read text from a file):

```
define marker: newtext, m1, m2 $ define "marker" variables
...
calc newtext := "The handsome horses" $ create a new block of text
  or
string newtext $ alternative form, similar to -text- command

The handsome horses
```

After this new text has been created, the marker variable "newtext" brackets this new text. That is, the marker points to the location in computer memory where the text starts and also to the location in memory where it ends. You can think of a marker variable as containing two pointers, to the start and to the end of a character string.

Other markers can mark subsections of the same new text. Using marker functions such as zsearch to position markers m1 and m2, we could have marker variable m1 bracket the string "handsome" and have marker variable m2 bracket the string "horses."

```
The    handsome   horses
    [.....m1....]   [....m2..]
```

The marker m1 points to the beginning and end of "handsome" and marker m2 points to the beginning and end of "horses." There is a -replace- command for changing the contents of a region of text bracketed by a marker variable. Suppose we replace the text in memory that is bracketed by m1 with different text, "galloping black."

```
replace m1, "galloping black"
```

The markers automatically adjust themselves as follows:

```
The    galloping black   horses
    [........m1.........]   [....m2..]
```

Note how m1 expanded to cover the modified text, and how m2 moved to the right to continue to bracket "horses." Also, the original marker variable newtext expands to cover the entire modified text ("The galloping black horses"). This tracking behavior is the reason for calling these variables "marker" variables. They mark regions of text, and they continue to mark those regions even if changes are made in the underlying text.

Now consider what happens if you use -calc- (or -string-) with m1:

```
calc m1 := "cats and dogs"
```
This -calc- creates a new block of text in computer memory, bracketed by m1, and completely distinct from the still-existing block of text bracketed by newtext. It also breaks the connection that m1 used to have with the first block of text. The important point to keep in mind is that -calc- and -string- create new blocks of text, whereas the -replace- command modifies a section of text that is bracketed by a marker variable.

If, in addition, we use newtext and m2 in -calc- or -string- commands, creating third and fourth distinct blocks of new text, there are no longer any marker variables bracketing any portion of the original text dealing with horses. When the last connection is broken, cT recognizes that the original text is no longer accessible and frees up that portion of computer memory to be used for other purposes.

Marker variables are designed to serve two types of needs. To do simple tasks such as recording and re-displaying a person's name, the commands used are straightforward and simple. There are more sophisticated operations for complex string handling, which many users never need to learn. The simple operations are discussed under the general heading of "Basic Marker Operations." More complex operations are discussed under "Marker Commands" and "Marker Functions", and "Some Examples with Markers" provides extended examples such as alphabetizing a list or plotting expressions contained in marker variables.

See Also:
- Basic Marker Operations (p. 248)
- Marker Commands (p. 259)
- Marker Functions (p. 264)
- Some Examples with Markers (p. 277)
- arrow Soliciting a Response (p. 159)
- calc Assigning a Value to a Variable (p. 200)
- compute Storing and Evaluating Inputs (p. 165)
- Defining Variables (p. 190)
Basic Marker Operations

Defining Marker Variables

A marker variable brackets a section of complex text, which can include simple ASCII characters, styles such as bold or italics, characters from other Latin alphabets, Japanese characters, and images. It is defined with the keyword "marker" or "m":

```
define marker: m1, m2, m3
m: names(50)   $$ array of 50 names
```

Markers do not have a fixed length. When a marker is first defined, it has a length of zero. When text is added to the marker, it grows as needed.

In an array of markers, each element (each marker) in the array has its own length, which can increase or decrease as text is added or removed from the marker.

Markers may be defined locally:

```
unit something
m: temp1,temp2
```

See Also:
Defining Variables (p. 190)

string: Text in a Marker Variable

The -string- command stores text in a marker variable. This resulting text can be displayed, manipulated, or stored in a file.

```
string m1
```

The -string- command allows complex text to be stored in a string. The text can include carriage returns and tabs. Embeds such as `<s,3+x>` or `<s,m2>` are permitted. Styles such as bold and italic are allowed. There can even be images in the "text."

```
\`
```

The first line of the tag (on the same line as the command) is the marker variable that will bracket the new text created by this -string- command. If the marker is already associated with an existing block of text, that connection is broken. The text that will be stored in computer memory begins on the line immediately below the command itself. The text is terminated by a backslash at the beginning of a line. The text and backslash are not indented, even if the (indented) -string- command is inside an -if- or -loop- or -case- structure.

The text is stored in computer memory and the string marker `m1` is made to point to the beginning and the end of this text. The marker `m1` brackets the text. Later in the program, marker functions can be used to move `m1` so that it only brackets a portion of the text. However, the base text remains intact and can be recovered so long as some marker points at some portion of the text. If there are no markers that point to the base text, it is no longer accessible, and cT will reuse the space for other purposes.
The -string- command and the closely related -text- command are the only commands in cT that do not follow the pattern of command-at-the-left and tag-field-after-a-tab.

The -string- command allows styles such as centered, bold, and italic:

```
string m1

This is an example of a complex string, including centered and italic text.
```

A string may also be displayed with an embedded show command that is enclosed in a style. The marker contents appear on the screen in the enclosing style:

```
write <[s,m1]> $ note the italics
```

If a double-dollar ($$) is included in the body of a -string- command it is treated as part of the string, not as the beginning of a comment.

**Example:**

```
unit xstring
m: marker1
string marker1
Four-score and seven years ago,
our forefathers brought forth
on this continent a new nation . . .
\at 40,50; 240,150
show marker1
at 40,120; 240, 250
write The Gettysburg Address starts: <[cr]>
<[s,marker1]> $ note the italics
```

**See Also:**

- text Putting Text on the Screen (p. 39)

### calc: Simple Marker Calculations

The -string- command creates a whole paragraph of text and wraps a marker around it. Short text strings, including styles such as bold or italic, can be created with a -calc- statement. The -calc- command is also used to concatenate the contents of two marker variables and to equivalence one marker with another marker.

```
calc s1 := "Hello" $ assign text

calc s2 := m1+m2+<[s,PI]> $ concatenate; embeds permitted

calc s3 := s2 $ s3 equivalent to s2
```

When assigning new text to a marker, the text must be enclosed in quote marks:

```
calc m1 := "Welcome to Carnegie Mellon"
calc m2 := "University"
```
Spaces inside quote marks, such as the space before "University" are significant. Other spaces are not significant. Both markers and constant ("literal") strings can be concatenated using a plus (+):

\[
\begin{align*}
\text{calc} & \quad m3 := m1 + m2 \\
\text{calc} & \quad m4 := "Purdue" + m2
\end{align*}
\]

The concatenation operation creates new text. The marker m3 now contains the text "Welcome to Carnegie Mellon University." The marker m4 contains "Purdue University." The four markers, \( m1 \), \( m2 \), \( m3 \), and \( m4 \), bracket four different strings in memory.

You can use embedded display commands and the embedded forms <quote> (double-quotes), <cr> (carriage-return), and <tab> (TAB): "Purdue"+<tab>+"Indiana, " + <s,year> + <cr>

When the -calc- command is used with literal strings or to concatenate two markers, new text is created. When the -calc- command is used to assign one marker to another marker, the markers become equivalent; that is, the two markers point to the same string.

\[
\begin{align*}
\text{calc} & \quad \text{marker1} := "The tree is green." \\
\text{marker2} & := \text{marker1}
\end{align*}
\]

Now marker2 brackets the same string that marker1 brackets. No new text has been created.

See Also:
- calc Assigning a Value to a Variable (p. 200)
- Defining Variables (p. 190)

**show: Contents of a Marker Variable**

The text bracketed by a marker variable is displayed with a -show- command. The -show- command is sensitive to the type of variable in its tag, so it treats all variable types correctly. Marker variables may also be used with the embedded form of -show-.

\[
\begin{align*}
\text{calc} & \quad m1 := "It is very cold" \\
\text{show} & \quad m1 \\
\text{write} & \quad <s,m1>
\end{align*}
\]

Just as -show- can handle numerical expressions (e.g., 3x+5), it can also handle "expressions" of marker variables:

\[
\begin{align*}
\text{show} & \quad m1 + " in the winter."
\end{align*}
\]

The -show- above displays: It is very cold during the winter.

The -write- statement below uses an italic style around an embedded -show- to display in italics. The entire embedded form, including the <s>, is enclosed in the style.

\[
\begin{align*}
\text{write} & \quad <s,m1> \text{ in the winter. } $S$ It is very cold in the winter.\end{align*}
\]

This way of imposing a style on an embedded -show- only works with -write-, -text-, and -string- commands. In all other embedded -show- s such imposed styles are ignored.
However, when a style is imposed on a marker \( m_1 \) as in \(<|s,m_1|>\), some styles in \( m_1 \) may be overridden inappropriately. In complex situations (including the presence of superscripts or subscripts), the way to handle this is to use a -style- command to impose the additional style, as seen in the example below.

**Example:**

```plaintext
unit xshowm
m: m1, m2, m3, m4
m: complex, copycomplex
calc m1 := "It is"
m2 := " very"
m3 := " cold"
at 20,20
write <|s, m1|><|s, m2|><|s, m3|>
at 20,50
write <|s, m1|><|s, m2|>,<|s, m2|>,<|s, m2|>,<|s, m3|>.
calc m4 := m1+m2+m3
at 20,80
write <|s, m4|> during the winter.
calc complex := "H_2 O and 3 x^3"
copycomplex := zcopy(complex)
style copycomplex; bold; red
at 20,110
write The string <|s, copycomplex|> has
subscripts and superscripts.
```

*See Also:*

- show Displaying Variables (p. 47)
- Embedding Variables in Text (p. 50)
- text Putting Text on the Screen (p. 39)

**Using Embedded Marker Variables**

Marker variables can be embedded in -write- or -text- statements. Suppose we have the following definitions and calculations:

```plaintext
define m: m1, m2, A(6)
i: n=22
calc m1 := "yellow"
m2 := "purple"
n := 22
```

Then the -write- and -text- below both give "I saw a purple cow."

```plaintext
write I saw a <|s, m2|> cow.
text 200,200; 500,500
I saw a <|s, m2|> cow.
```

*Any* command that expects a string of characters will accept embeds. Here are some examples:
There are $n+3$ ships

[$s,m1$] $s,m2$ synonyms

See $s,m2$ run

$s,m1; s,m2$ someunit

$m1, n$ no embed needed

$m1+s,n>+.cars$

Note that commands that display or accept typed text, such as -write-, -text-, -menu-, and -answer-, don't take quote marks, and variables must be embedded. Other commands, such as -font- and -setfile-, take string expressions, so quote marks are needed for literal text and marker variables need not be embedded.

There are special embeds for carriage return (new line), tab, and quote. The following -show- command will execute a carriage return and a tab, then display a double quote mark followed by the contents of $m1$ (dog) and another double quote mark:

```
calc  m1 := "dog"
      m2 := $cr$ + $tab$ + $quote$ + $m1$ + $quote$
show  m2              $cr$, $tab$, "dog"
```

**See Also:**
- Embedding Variables in Text (p. 50)
- Putting Text on the Screen (p. 39)

### zempty: Logical Comparisons with Markers

Strings bracketed by marker variables can be compared using the standard logical operators (<, >, =, <=, >=, and ~= or !=). The system variable "zempty" is a marker of length zero.

The greater-than and lesser-than comparisons are made according to "dictionary order". That is, "$m1 > m2" is true if the string bracketed by $m1$ would come later in a dictionary than the string bracketed by $m2$.

**Examples:**

When a marker variable is defined, it has a length of zero; it is empty. When something is assigned to the marker, it is no longer empty. In the second example, try assigning various words into the variables word1, word2, and word3.

```
unit  xzempty
m: mystring
if    mystring = zempty
      at 30,10
      write Now "mystring" is empty.
      (Press any key.)
endif
pause
calc  mystring := "Hello"
if    not(mystring = zempty) $note the "not"
      at 30,60
      write Now it's not empty:
```
mystring = <|s,mystring|>.

endif
*
unit xzempty2
m: word1, word2, word3
calc word1 := "cat"
word2 := "antelope"
word3 := "ante" + "lope"
at 50,50
write \word1<word2
\<|s,word1|>
is alphabetized before \<|s,word2|>
\<|s,word1|>
is alphabetized after \<|s,word2|>
at 50, 75
write The variables word2 and word3
write \word2 = word3\ are equal.\ are different.
*

See Also:
Logical Operators (p. 201)

do: Passing Markers

Markers may be passed as arguments with a -do- command.

When a marker or string expression is passed by value to a subroutine, it is just as though -calc- had been used to assign the value. Consider this sequence:

```
d o print(5, m1, m2 + m3 + "runs")
.....
unit print(n, s, t)
    integer: n
    marker: s, t
```

These pass-by-value operations are equivalent to the following -calc- statements:

```
calc n := 5
    s := m1
    t := m2 + m3 + "runs"
```

In this case, the marker s becomes a copy of the marker m1, pointing at the same base text that m1 points at. The string expression (m2 + m3 + "runs") creates new text, and the marker t is made to bracket this new text. Executing "calc s := znext(s)" moves the local marker s to point to the next character in the base text without moving the original marker m1.

We could instead pass m1 to s by address:

```
d o print(5, m2+m3+"runs"; m1)
.....
unit print(n, t; s)
    integer: n
    marker: s, t
```

In this case, the statement "calc s := znext(s)" actually moves the marker m1.
File I/O with Marker Variables

Markers can be used with the -readln-, -datain-, and -dataout- commands:

- readln file; m1 $$ read until end of line
- readln file; m1,"!!" $$ read until !! encountered
- datain file; m1 $$ read entire file into m1
- dataout file; m1 $$ append m1 to end of file
- datain file; m1, count $$ optional character count
- dataout file; m1, count $$ optional character count

The -readln- command reads up to and including the end of a line (carriage return, or carriage return plus line feed). You can specify some other character string, in which case -readln- reads up to and including the specified string.

If you don't specify a character count, the -datain- command reads the entire remainder of an unstyled file into memory and initializes the marker to bracket all of that text. With a styled file, -readln- cannot be used (it gives an execution error), and -datain- reads the next section of text, corresponding to the section written to the file using -dataout-.

If you use -dataout- of a marker to write to an unstyled file, -dataout- strips out any styles such as italic, bold, superscript, etc.

See Also:
- Files, Serial Port, & Sockets (p. 285)
- readln Read a Line from a File (p. 297)
- datain Read Data from a File (p. 294)
- dataout Write Data to a File (p. 298)
- A File Editor Application (p. 155)

compute: Computing with Marker Variables

Marker variables can be used with a -compute- command to numerically evaluate a function that has been stored from the user's input or generated by some other process.

compute y
compute y, myfunction
compute y, "sqrt(x)"

The first argument in the tag must be a floating-point, integer, or byte variable. The second (optional) argument is a marker variable or a literal string.

When -compute- has only one argument, the string in the input buffer (zarrowm) is evaluated and the result is assigned to the named variable. The value of zarrowm can be changed, to allow -compute- to operate on arbitrary strings.
When `-compute-` has two arguments, the string in the second argument is evaluated and the numeric result is assigned to the variable named by the first argument.

If the string contains variables, these must have been defined as "user" variables (define user:) in the IEU of the same file (initial entry unit preceding the first `-unit-` command).

The first time a `-compute-` command is executed, the string is compiled and the compiled code is stored for later use (as long as the string isn't changed). Thus, execution of a `-compute-` in a loop is done very efficiently.

The operation of `-compute-` is affected by the `-specs-` options noops, novars, and okassign. These specifications are cleared by an `-arrow-` command but remain in effect across a change in main unit (e.g., with `-jump-`).

**Examples:**

**NOTE:** For both of the examples, the variable "x" (which is the independent variable in the examples), must be in the global `-define-` in the IEU:

```
define user:
  f: x
```

In the first example, the function that will be used for the `-compute-` ("myfunction") is set with a `-calc-` statement. The example is an illustration of the useful fact that, for small angles, the *sine* of an angle is approximately equal to the *size* of the angle, expressed in radians.

```
unit mxcompute1
merge,global:
m: myfunction
f: value
i: angle
calc myfunction := "sin(x*DEG)"
at 45,20
write degrees radians sine
at 50,50
loop x := 0, 7, 0.5 $$ x = 0, 0.5, 1, 1.5, etc.
compute value, myfunction
write<|t,x,1,1|><|t,x*DEG,5,4|><|t,value,5,4|><|cr|>
endloop
```

The example below allows the user to enter two functions of x. Both functions are then evaluated for some random value of "x". Note that the user-entered functions are *copied* into "fn1" and "fn2" using *zcoplastarzoom*.

```
unit mxcompute2
merge,global:
m: fn1, fn2
f: y1, y2
at 50,20
write Enter a function of x:
y1 =
arrow zwhere x+10, zwhere y
compute y1 $$ trial evaluation of response
ok zreturn $$ TRUE if response is well formed
```
endarrow
calc  fn1 := zcopy(zarrowm)  $$ save copy of response
at  50,60
write Enter another function:
y2 =
arrow zwhere x+10, zwhere y
calculate y2
ok zreturn
endarrow
calc  fn2 := zcopy(zarrowm)  $$ save second response
random x $$ select a random value for x
compute y1, fn1 $$ evaluate fn1 with selected value of x
compute y2, fn2 $$ evaluate fn2 with selected value of x
at 40,120
write For x = <|s,x|> $$ show selected value of x

<|s,fn1|> is <|s,y1|> $$ function values
<|s,fn2|> is <|s,y2|>

See Also:
compute Storing and Evaluating Inputs (p. 165)
znumeric Extract Number from Marker (p. 272)
zcode String to Integer Conversion (p. 265)
arrow Soliciting a Response (p. 159)
calc Assigning a Value to a Variable (p. 200)
IEU The Initial Entry Unit (p. 227)
Defining Variables (p. 190)
User Variables (p. 198)
specs Specifying Special Options (p. 169)

zarrown: Markers at an -arrow-

The user's input at an -arrow- is automatically bracketed by a system marker variable called "zarrown".

Most system variables can only be examined, but zarrown can be modified. This allows two useful operations at an -arrow-: the "response buffer" can be filled so that it appears that the user has already typed an input and/or the user's response can be modified before it is seen by an -answer- or some other response-handling command.

To prefille the response buffer, a string is assigned to zarrown with indented statements immediately following the -arrow-. These statements are executed only upon entry into the arrow region (the area between -arrow- and -endarrow-). If the response is judged "no", execution starts again at the first non-indented command.

  arrow  150,150
calc zarrown := "sunshine"
answer rain
answer sunshine

To modify the user's response, use -replace- or -append-.

  replace zlast(zarrown), "s!"
  append zarrown, "ing"
The -arrow- command initializes \texttt{zarrowm} as though you had said "calc \texttt{zarrowm} := zempty". This breaks the connection with any marker variables previously assigned to \texttt{zarrowm}. This initialization is also performed when -judge rejudge- or -judge ignore- is executed, and for each new response at the same -arrow-.

\textit{Examples:}

In this example the user may type anything at all, so any response at all is -ok-. The response (which is stored in \texttt{zarrowm}) is concatenated with "hello" and an explanation mark and displayed. Notice the comma and a space after "hello".

\begin{verbatim}
unit xzarrowm1
at 50,50
write What is your name?
arrow 75,75
ok
endarrow
show "Hello, " + zarrowm + 
"!
*
\end{verbatim}

In the second example, the user's response is repeated three times into "letters" and then displayed.

\begin{verbatim}
unit xzarrowm2
at 50,50
write Type something (short):
arrow 75,75
ok
endarrow
calc letters := zarrowm + zarrowm + zarrowm
at 50,100
show letters
*
\end{verbatim}

The next example pretends to be an -arrow-, but the user doesn't need to do anything. The -press zk(next)- initiates judging; the indented -calc- fills the response buffer as if the user had typed something; and the -replace- fixes the misspelling so that the response is "ok". Notice that the internal version of the response is changed to "cow", but that the response on the display is still "coy".

\begin{verbatim}
unit xzarrowm3
at 50,50
write What animal provides most of our milk?
arrow 100,100
calc zarrowm := "coy"
press zk(next)
replace zlast(zarrowm), "w"
answer cow
write Moo!
endarrow
*
\end{verbatim}

\textit{See Also:}
arrow Soliciting a Response (p. 159)
zarrowsel Selected Text at an -arrow- (p. 258)
**zarrowsel: Selected Text at an -arrow-**

The system variable `zarrowsel` is a marker on mouse-selected text within `zarrowm`, the input at the current -arrow- command.

*Example:*

```
unit    xarrowsel
at      10,10
write   Type something, then select part of your text
        with the mouse, then press Return:
arrow   10,50
show    zarrowsel
endarrow
```

*See Also:*

- `arrow` Soliciting a Response (p. 159)
- `zarrowm` Markers at an -arrow- (p. 256)
Marker Commands

append: Adding Characters to a String

The -append- command is used to append characters to the end of text bracketed by a marker.

```
append string, " and two cups of coffee"
append m1, m2
```

The first argument of the tag must be a marker variable. The second argument may be a marker variable, a literal string, or an "expression" (concatenation) of markers and literal strings.

The string in the second argument is appended to the end of the text bracketed by the first marker. The append operation is like a "Paste" in that the appended string (m2) has its own styles, not the styles of the string being appended to (m1).

All equivalent markers are extended to include the new text. Other markers are extended if they start before the append position and end at or after it. If they begin at or beyond the append position, they are moved so that they continue to bracket the same characters they used to bracket.

If you say -append zend(m1), m2-, the marker m1 is normally not extended to include the new text, because zend(m1) is considered to be after m1. You can use zbase(m1) to include the text that has been appended after the end. Similarly, if you say -append zstart(m1), m2-, the marker m1 is normally not extended to include the new text, because zstart(m1) is considered to be before m1. Again, zbase(m1) includes the new text.

You can use a -sticky- command to make the ends of a marker "sticky," in which case an -append- command involving either zstart(marker) or zend(marker) does extend the marker to include the new text.

Example:

```
unit xappend
m: m1
calc m1 := "one"
append m1, " two"     $$ note the initial space
append m1, " three"  at 20, 30
show m1
*
```

See Also:
- calc Simple Marker Calculations (p. 249)
- replace Replacing Characters in a String (p. 259)
- sticky Make Marker Ends Sticky (p. 261)

replace: Replacing Characters in a String

The -replace- command replaces the characters bracketed by a marker variable.

```
replace m1, m2
replace string, "something else"
```
CHARACTER STRINGS

The first argument of the tag must be a marker variable. The second argument may be a marker variable, a literal string, or an "expression" (concatenation) of markers and literal strings.

The -replace- command first appends the new text at the end of the marker, adjusting the markers in the same way that the -append- command does. The replace operation is like a "Paste" in that the resulting string (m2) has its own styles, not the styles of the original string (m1).

After appending the new text, the -replace- command deletes the old characters at the beginning of the marker. Any other markers in this deletion region shrink accordingly.

If you say -replace zend(m1), m2-, the marker m1 is normally not extended to include the new text, because zend(m1) is considered to be after m1. You can use zbase(m1) to include the text that has been placed after the end. Similarly, if you say -replace zstart(m1), m2-, the marker m1 is normally not extended to include the new text, because zstart(m1) is considered to be before m1. Again, zbase(m1) includes the new text.

You can use a -sticky- command to make the ends of a marker "sticky," in which case a -replace- command involving either zstart(marker) or zend(marker) does extend the marker to include the new text.

The functions zeditsel(zedit), zeditvis(zedit), zhot.sel(zedit), and zedit.text(zedit) cannot be used directly as the first argument in a -replace- command, to change text in an edit panel. Instead, you must first assign a marker to the region, then use that marker in the -replace- command, as follows:

```
calc   m1 := zeditsel(zedit)
replace m1, "cows"
```

Example:

```
unit xreplace
m: m1
calc m1 := "one"
at 20, 30
show m1
replace m1, "two"
at 20, 50
show m1
*
```

The next example builds a phrase by setting up a skeleton phrase, assigning markers to each appropriate position in the phrase, and then replacing each marker with a word. The markers belong to the same base string, but they mark different regions of the base string. To display the entire base string, zbase can be used. Note that zbase(noun) = zbase(adjective) = zbase(adverb) = phrase.

```
unit xreplace2
m: phrase, adverb, adjective, noun
calc phrase := "1 2 3" $$ three items separated by spaces
calc adverb := zfirst(phrase) $$ brackets "1"
adjective := znext(znext(adverb)) $$ brackets "2"
noun := zlast(phrase) $$ brackets "3"
replace adverb, "very"
replace adjective, "small"
replace noun, "house"
at 20, 30
show zbase(noun) $$ equivalent to phrase
at 20, 50
```
show zbase(adverb) $$ also equivalent to phrase
replace adjective, "big"
at 20, 90
show phrase $$ modified phrase
*

See Also:
calc Simple Marker Calculations (p. 249)
append Adding Characters to a String (p. 259)
sticky Make Marker Ends Sticky (p. 261)

**sticky: Make Marker Ends Sticky**

The -sticky- command lets you control whether the ends of markers are "sticky" or not. The default is that markers are not sticky. An -append- or -replace- command involving either zstart(m1) or zend(m1) does not normally extend the marker m1 to include the new text. Here are the possible forms:

- sticky marker $$ the ends of this marker are "sticky"
- sticky marker, FALSE $$ ends of this marker are not sticky
- sticky marker, expression $$ sticky if expression is TRUE

The system marker function associated with edit panels, `zedittext(editvar)`, is sticky by default. This facilitates adding additional visible text at the start and end of the displayed text.

Creation of new text by "string m1" or by "calc m1 := "new text" initializes the marker to be nonsticky. Only after placing a marker on some text will a -sticky- command make the ends sticky.

The easiest way to understand the issues is to execute and study the example given below.

**Example:**

```
unit xsticky
m: mark
calc mark := "little"
replace zstart(mark),"some ",
replace Zend(mark)," toys"
at 10,10
show mark $$ not sticky; shows "little"
* The entire base text includes everything:
at 10,30
show zbase(mark) $$ shows "some little toys"
* Next do similar operations with a sticky marker.
calc mark := "big" $$ new initial text
sticky mark $$ make "mark" sticky
replace zstart(mark),"fancy ",
replace Zend(mark)," cars"
at 10,50
show mark $$ shows "fancy big cars"
```

See Also:
calc Simple Marker Calculations (p. 249)
append Adding Characters to a String (p. 259)
replace Replacing Characters in a String (p. 259)
**style: Assigning Styles to Markers**

The `-style-` command lets you assign styles to text bracketed by markers:

```
style m1; bold; italic    $$ make marker m1 bold and italic

* Make marker m2 bold if expr1 is TRUE, italic if expr2 is TRUE:
style m2; bold, expr1; italic, expr2
```

Here are the styles that can be applied, *just as though you had used a menu option to apply such styles* (this means, for example, that applying an italic style to italicized text removes the italics):

- plain, plainest, bold, italic
- superscript, subscript, bigger, smaller
- full justify, left justify, right justify, center
- black, white, red, green, blue, yellow, cyan, magenta
- serif, sans serif, fixed, symbol
- hot, icon

You can also replace a section of a marker with a pixel image contained in a screen variable. One of the uses of this capability is to prepare a marker containing text and pixel graphics for direct printing.

```
style m1; screen, screen variable
```

The function `zhasstyle(marker, bold)` is TRUE if any text in the marker is bold (otherwise it is FALSE). The styles that can be checked for are the same as those given above.

If the style is "hot" you must also provide the text of the associated "hot" information:

```
style m1; hot, "This is the hot information"
style m2; hot, m3+<s,2x+y>
```

The "hot" style cannot be assigned to an empty (zempty) marker, since there would be no way to double-click such a region.

Conflicting styles override each other in order, left to right:

```
style m1; green; red    $$ text will be red
```

The "plain" and "plainest" styles are assigned first, no matter where they are listed in the statement:

```
style m1; green; italic; plain $$ text will be just green and italic
```

If the style is "icon" you must also provide the name of the icon file, and a list of icon numbers:

```
style marker; icon,icon file name,icon number(s)
```

The function `znicons(marker)` returns the number of icons contained in the specified marker. The function `ziconcode(marker, N)` returns the icon number for the Nth icon in the marker (equivalent to the argument of a `-plot-` command). The function `ziconfile(marker, N)` returns a marker containing the name of the icons file from which the Nth icon comes (equivalent to the argument of an -icons- command). If there is no Nth icon, either `ziconcode(marker, N)` or `ziconfile(marker, N)` gives an execution error, so you should first check how many icons there are by using `znicons(marker)`. 
Examples:

```
unit
  xstyles  $$ run through many -style- manipulations
  edit: ed
  marker: text, c
  i: nn
calc
text := "Setting styles"
ed: 10, 250, 65; xhot; text; editable; frame
calc
  c := zfirst(text)
loop
  nn := 1, 3
  style
    c; superscript
  style
    \nn-2 \ c; red \ c; green \ c; blue \n  pause
  0.5
  style
    c; plain  $$ remove the superscript and color
  calc
    c := znext(c)
endloop
* make "Sett" bold and red:
  style
    zextent(zstart(text), c); bold; red
  pause
  0.5
append
  text, <|cr|> + "Double-click the italicized words!"
pause
  0.5
calc
  c := znextline(zstart(text))  $$ first line of text
  style
    c; center
  calc
    c := zsearch(text,"italicized words")
  style
    c; hot, "You double-clicked on me!"
at
    25, 70
write
  zhasstyle(c, italic) = <|s, zhasstyle(c, italic)|>
* 
unit
  xhot  $$ handle hot text
erase
  20, 100; 300, 150
at
  30, 100
write
  Selection is [<|s, zeditsel(zedit)|>].
  Hot selection is [<|s, zhotsel(zedit)|>].
  Hot info is [<|s, zhotinfo(zedit)|>].
* 
unit
  xembedicons  $$ embed icons in a marker
  marker: circle
style
  circle; icon, zicons, 70, 69, 68  $$ three filled circles
at
  10, 20
show
  "Filled +circle+"circles"
* 
```

See Also:

Scrolling Text Panels (p. 149)
A File Editor Application (p. 155)
zhasstyle Check What Style Is on a Marker (p. 267)
Printing (p. 14)
Marker Functions

Comment

The marker functions in this section are covered alphabetically, but if you are just starting to use markers, it will be easier if you study them more or less in this order:

- zstart, znext, znexline, znextword, zcopy
- zfirst, zlast, zend, zprevious, zbase
- zchar, zcode, znumeric
- zextent, zsearch
- zaltered, zprecede, zsame
- zlength, zlocate, zsetmark
- zhasstyle, zhotinfo
- znicons, ziconcode, ziconfile

zaltered: Changed Marker Flag

The function `zaltered` is used to check whether a marked region has been affected by a change to an intersecting marked region. It returns either TRUE or FALSE. When a marker is altered directly, `zaltered(marker)` is set to FALSE. When a marker is altered because an action on another marker has affected the region bracketed by this marker, `zaltered(marker)` is set to TRUE. The act of examining `zaltered` causes it to be set to FALSE. That is, reading the marker clears it.

Example:

The first `-calc-` statement modifies m1 directly, therefore the first `zaltered(m1)` is FALSE; no indirect change has been made to m1. The second `-calc-` statement makes m2 point to the first character of the string m1. The `-replace-` modifies m2 and as a result modifies m1. Therefore, `zaltered(m1)` is TRUE.

```
unit xaltered
m: m1, m2
calc m1 := "abcde" $$ m1 directly modified
at 50,30
write m1 is |<s,m1>| 
zaltered = |<s,zaltered(m1)|>
calc m2 := zfirst(m1)
replace m2,"x" $$ m1 indirectly modified
at 50,100
write m1 is now |<s,m1>| 
zaltered = |<s,zaltered(m1)|>
```

The act of reading it modifies zaltered:

```
zaltered = |<s,zaltered(m1)|>
```
**zbase: The Entire Base String**

Text created with -calc- or -string- is called the *base string*. It is stored in computer memory and can always be retrieved so long as *some* marker points to *some* part of the string. The "zbase" function retrieves the entire string to which a marker belongs. Consider this sequence:

```
calc    m1 := "The sun is hot."
```

Now `m1` encloses "The sun is hot.", which is called the base string.

```
m1 := zfirst(m1)
```

At this point, `m1` encloses only "T", and there is no marker at all that encloses the entire string. However, the base string is still available and can be retrieved:

```
m2 := zbase(m1)
```

Now `m2` encloses "The sun is hot."

**zchar: Integer to String Conversion**

The function "zchar" operates on an integer. It creates a one-character string containing the character whose ASCII value is the integer. The 32 character codes in the range of 128 to 159 inclusive are special; zchar of these integers produces a character whose zcode value is -1 (like zempty).

```
zchar(97) $$ creates "a"
```

*Example:*

This little unit displays the ASCII characters numbered 32 through 64.

```
unit xzchar
  i: index
at 20,50
loop index := 32,64
  write <s,zchar(index)|>
endloop
*  
```

*See Also:*
- Embedding Variables in Text (p. 50)
- zcode: String to Integer Conversion (p. 265)
- Types of Variables (p. 191)

**zcode: String to Integer Conversion**

The function "zcode" operates on a marker variable. It produces an integer in the range 0 to 255, which is the ASCII value of the first character in the string bracketed by the marker variable.

```
zcode("abcd") $$ produces 97
```

The expression "zcode(zempty)" gives the special value -1.
CHARACTER STRINGS

Example:

```
unit       xzcode
m: S
calc      S := "abcdgoldfish"
at	50,75
write   ASCII value of the first character ($|s,zfirst(S)|$)
of the string "$|s,S|" is $|s,zcode(S)|$.
```

See Also:

zchar Integer to String Conversion (p. 265)
znumeric Extract Number from Marker (p. 272)
compute Computing with Marker Variables (p. 254)
Counting Vowels (p. 279)
Types of Variables (p. 191)

**zcopy: New Copy of a String**

The function **zcopy** is used to create a new copy of the text bracketed by a marker.

```
calc      marker1 := "The tree is green."
          marker2 := zcopy(marker1)
```

Now **marker2** brackets a separate copy of the text "The tree is green." that has no connection to **marker1**.

The `-calc-` command is used to cause two markers to bracket the same string.

```
calc      marker3 := marker1
```

Now **marker3** brackets the same string that **marker1** brackets. No new text has been created. If the text bracketed by **marker1** is changed, marker **marker3** is adjusted to bracket the same text, but **marker2** is unaffected.

**zend: After the Last Character**

The function **zend**(marker) returns a marker that points to a zero-length string just after the last character bracketed by the named marker.

Example:

```
unit       xzend
m: m1, m2
calc      m1 := "This is a string"
calc      m2 := zend(m1) $\$$ just after "g"
replace   m2, " with stuff in it." $\$$ put phrase into m2
at	50,30
write     m1 = $|s,m1|$
          m2 = $|s,m2|$
at	50,100
show      zbase(m1) $\$$ entire string
```
**zextent: Combine Marker Regions**

The function \texttt{zextent(marker1,marker2)} returns a marker that brackets everything between and including the regions bracketed by \texttt{marker1} and \texttt{marker2}.

The two markers (\texttt{marker1} and \texttt{marker2}) must bracket areas of the same base string. If the two markers are members of different base strings, \texttt{zextent} gives an execution error.

*Example:*

```plaintext
unit xzextent
m: m1, m2, m3, m4
calc m1 := "bubble gum is icky"
calc m2 := znext(zfirst(m1))
   m3 := zprevious(zlast(m1))
at 50,30
write m1 is <|s,m1|>  $$ contains first "u"
m2 is <|s,m2|>  $$ contains "k"
m3 is <|s,m3|>  $$ contains first "p"
calc m4 := zextent(m2, m3)
at 50,100
write m4 is <|s,m4|>  $$ all but first & last letters
```

**zfirst: First Character of a Marker**

The \texttt{zfirst} function operates on a string. It returns a marker bracketing the first character of the given string.

\texttt{zfirst(mystring)}

The given string is usually a marker variable, but it can be a literal string. If \texttt{"mystring"} is a zero-length string, then \texttt{zfirst(mystring)} returns \texttt{zempty}.

*Examples:*

```plaintext
unit xzfirst1
at 50,50
show zfirst("Hello")  $$ shows "H"
*
unit xzfirst2
m: S
calc S := "pneumonia"
at 50,50
write The first letter of <|s,S|> is <|s, zfirst(S)|>!
*```

**zhasstyle: Check What Style Is on a Marker**

The function \texttt{zhasstyle(marker, bold)} is TRUE if any text bracketed by the marker is bold (otherwise it is FALSE). The styles that can be checked for are the same as those used by the -style- command.
CHARACTER STRINGS

Example:

```
unit xhasstyle
m: m1
calc m1 := "tiger"
at 10,10
write Italic? <|s,zhasstyle(m1,italic)|>
Bold? <|s,zhasstyle(m1,bold)|>
```

See Also:

See Also: style Assigning Styles to Markers (p. 262)

zhotinfo: Information Associated with Hot Text

The system function `zhotinfo(myedit)` returns the information associated with the last click of hot text in edit panel "myedit". You can also use `zhotinfo(anymarker)`, which will return the string associated with that marker, which need not be in an edit panel. If you use `zhotinfo(anymarker)` on a marker that has no hot text, you get zempty. You can also use `zhasstyle(anymarker, hot)` to check whether there is any hot text associated with a marker. If you use `zhotinfo(anymarker)` on a marker that has more than one piece of hot text, you get all the information, concatenated into one string.

Example:

```
unit xhotinfo
m: m1, m2
calc m1 := "the quick brown fox"
m2 := zsearch(m1, "quick")
style m2; hot, "A synonym for fast"
at 10,10
show zhotinfo(m2)
```

See Also:

See Also: Scrolling Text Panels (p. 149)

ziconcode: Icon Code in a Marker

The function `ziconcode(marker, N)` returns the icon number for the Nth icon in the text bracketed by the marker (equivalent to the argument of a -plot- command).

Example:

```
unit xiconcode
marker: circle
style circle; icon,zicons,70,69,68 $$ three filled circles
at 10,20
show "Filled "+circle+"circles"
at 10,60
show ziconcode(circle,2)
```
**ziconfile: Icon File in a Marker**

The function `ziconfile(m, N)` returns a marker containing the name of the icons file from which the Nth icon comes (equivalent to the argument of an -icons- command).

**Example:**

```plaintext
unit xiconfile
marker: circle
style circle; icon,zicons,70,69,68 $$ three filled circles
at 10,20
show "Filled +circle+"circles"
at 10,60
show ziconfile(circle,2)
```

**See Also:**
- `style` Assigning Styles to Markers (p. 262)
- `znicons` Number of Icons in a Marker (p. 272)

**zlast: Last Character of a Marker**

The "zlast" marker function operates on a string. It returns a one-character string containing the last character of the given string.

```plaintext
zlast(mystring)
```

The given string is usually a marker variable, but it can be a literal string. If "mystring" is a zero-length string, then `zlast(mystring)` returns `zempty`.

**Examples:**

```plaintext
unit xzlast1
at 50,50
show zlast("Hello") $$ shows "o"
*
unit xzlast2
m: S
calc S := "bouquet"
at 50,50
write The last letter of <|s,S|> is <|s, zlast(S)|> !
*
```

**zlength: Number of Characters in a String**

The `zlength` function gives the number of elements in an array, or the number of characters in a marker variable:
More technically, \( z\text{length}(M(2)) \) gives the number of \( z\text{next} \) operations required to proceed through the marker.

**zlocate: Numerical Position of a Character**

The \( z\text{locate} \) function tells where the start of a marker \( m \) is with respect to the start of its base text \( z\text{base}(m) \). More technically, it gives the number of \( z\text{next} \) operations required to proceed from the start of the base text to the start of the region bracketed by the marker of interest. For example, if \( m \) is located at the start of its base text, \( z\text{locate}(m) \) is zero, since no \( z\text{next} \) operations are required to move to the start of \( m \).

The main use of \( z\text{locate} \) is associated with writing text to a file using -dataout-. If you write to an auxiliary file the \( z\text{locate} \) and \( z\text{length} \) information for markers associated with this text, the information can be read back later and used in the \( z\text{setmark} \) function to restore the markers to their original positions on the base text.

**znext: The "next" Character**

The "\( z\text{next} \)" function operates on a string. It returns a marker bracketing the next character (of the base string) after the end of the text bracketed by the current marker.

\[ z\text{next}(\text{current}) \]

If the named marker includes the last character of the base string, then \( z\text{next}(\text{current}) \) returns \( z\text{empty} \).

*Examples:*

In this example, the phrase "The sun is hot." is assigned to the variable \( S \). The function \( z\text{first}(S) \) is used to set the variable \( C \) to the first character of \( S \). Then the function \( z\text{next}(C) \) is used to step through the string, one character at a time.

Remember that a marker brackets something (or is a zero-length marker). It points to the beginning and the end of a string. When the marker \( C \) encloses the character "T", the next character after that is "h", so \( z\text{next}(C) \) is a marker around "h". The -loop- displays each character between vertical bars.
znextline: The "next" Line

The "znextline" function operates on a string. It returns a marker bracketing the next "line" (of the base string) after the end of the text bracketed by the current marker, including the end-of-line character(s).

`znextline(current)`

A "line" is defined as a sequence of characters ending with and including the end-of-line character or characters, or ending with the end of the text being searched.

If the named marker includes the last character of the base string, then `znextline(current)` returns `zempty`.

Examples:

In this example, a three-line paragraph is assigned to the variable `S`. The function `zstart(S)` is used to set the variable `L` to just before the first character of `S`. Then the function `znextline(L)` is used to step through the string, one line at a time. The -loop- displays each line preceded by an asterisk.

```
unit xznextline
m: S, L
string S
  The sun is hot.
  The moon is blue.
  The grass is green.
\calc L := zstart(S)
at 10,20
loop $line-by-line$
calc L := znextline(L)
outloop L = zempty
write * <|s,L|> <|cr|>
endloop
```

See Also:
- `readln` Read a Line from a File (p. 297)

znextword: The "next" Word

The "znextword" function operates on a string. It returns a marker bracketing the next "word" (of the base string) after the end of the text bracketed by the current marker.

`znextword(current)`

A "word" is defined as a sequence of characters other than "white space": space, tab, or end of line. Any initial white space is skipped, and the result brackets up to, but not including, the next white space.

If the named marker includes the last character of the base string, then `znextword(current)` returns `zempty`.

Examples:
In this example, the phrase "The sun is hot." is assigned to the variable \texttt{S}. The function \texttt{zstart(S)} is used to set the variable \texttt{W} to just before the first character of \texttt{S}. Then the function \texttt{znextword(W)} is used to step through the string, one word at a time. The -loop- displays each word between vertical bars.

\begin{verbatim}
unit xznicons
marker: circle
style circle; icon,zicons,70,69,68  $$ three filled circles
at 10,20
show "Filled "+circle+"circles"
at 10,60
show zicons(circle)
*
\end{verbatim}

\textbf{znicons: Number of Icons in a Marker}

The function \texttt{znicons(marker)} returns the number of icons contained in the text bracketed by the specified marker.

\textit{Example:}

\begin{verbatim}
unit xznicons
marker: circle
style circle; icon,zicons,70,69,68  $$ three filled circles
at 10,20
show "Filled "+circle+"circles"
at 10,60
show zicons(circle)
*
\end{verbatim}

\textit{See Also:}

\begin{verbatim}
style Assigning Styles to Markers (p. 262)
ziconfile Icon File in a Marker (p. 269)
ziconcode Icon Code in a Marker (p. 268)
\end{verbatim}

\textbf{znumeric: Extract Number from Marker}

There is a function that picks a number out of a string of characters (a marker expression):

\begin{verbatim}
znumeric("abc 10.1 def") returns floating value 10.1
\end{verbatim}

\textit{Example:}

\begin{verbatim}
unit xnumeric
m: m1
calc m1 := "Pi is 3.14, approximately."
at 10,10
show znumeric(m1)
*
\end{verbatim}
zprecede: Marker Order

The "zprecede" marker function operates on two markers. It returns TRUE if the beginning of the first marker precedes the beginning of the second marker. The function returns FALSE otherwise.

\[ \text{zprecede}(m_1, m_2) \]

If the two markers refer to different base strings, the function causes an execution error.

**Examples:**

```plaintext
unit xmzprecede
m: m1, m2, m3
calc m1 := "Where is the rain in Spain?"
at 50,50
write <|s,m1|>
calc m2 := zfirst(m1) \( \$\) contains "W"
m3 := zlast(m1) \( \$\) contains ":?"
at 50,100
write m2 = <|s,m2|>; m3 = <|s,m3|>
at 50,150
write zprecede(m2,m3) = <|s,zprecede(m2, m3)|>
zprecede(m3,m2) = <|s,zprecede(m3,m2)|>
*
unit xmzprecede2
m: m1, m2, m3, m4
calc m1 := "Mainly on the plain."
at 50,50
write <|s,m1|>
calc m2 := zsearch(m1, "e") \( \$\) find an "e"
calc m3 := zextent(zstart(m1), m2)
m4 := zextent(m2, zlast(m1))
at 50,75
write m2 = <|s,m2|>
m3 = <|s,m3|>
m4 = <|s,m4|>
at 50,125
write zprecede(m2,m1) = <|s,zprecede(m2, m1)|>
zprecede(m1,m2) = <|s,zprecede(m1, m2)|>
zprecede(m2,m3) = <|s,zprecede(m2,m3)|>
zprecede(m3,m2) = <|s,zprecede(m3,m2)|>
zprecede(m2,m4) = <|s,zprecede(m2,m4)|>
zprecede(m4,m2) = <|s,zprecede(m4,m2)|>
* 
```
**zprevious: The "previous" Character**

The "zprevious" marker function operates on a string. It returns a marker bracketing the character just before the beginning of the text bracketed by the current marker.

\[ \text{zprevious}(\text{current}) \]

If the named marker includes the first character of the base string, then \( \text{zprevious}(\text{current}) \) returns \( \text{zempty} \).

**Example:**

This example is exactly analogous to the example for \( \text{znext} \). It displays the characters one-by-one in reverse order. It starts with the last character in the string and works toward the beginning one character at a time.

```
unit xzprevious
m: S, C
calc S := "The sun is hot." \$ make a marker
calc C := zlast(S) \$ bracket last character
at 50,50
loop \$ get one character at a time, back to front
outloop C = zempty \$ no characters left
write \$[s,C]
calc C := zprevious(C)
endloop
*
```

**zsamemark: Marker Equivalence**

The \( \text{zsamemark} \) marker function operates on two markers. It returns TRUE if both bracket the same string and FALSE if they bracket different strings.

\[ \text{zsamemark}(m1,m2) \]

If the two markers point at different base strings, the function causes an execution error.

**Example:**

```
unit xmsavemark
m: m1, m2, m3, m4
calc m1 := "Where is the rain in Spain?"
at 50,50
write \$[s,m1]
calc m2 := zfirst(m1)
m3 := zlast(m1)
m4 := zfirst(zbase(m3))
at 50,100
write m2 = \$[s,m2]; m3 = \$[s,m3]; m4 = \$[s,m4]
at 50,150
write zsamemark(m2,m3) = \$[s,zsamemark(m2, m3)];
zsamemark(m2,m4) = \$[s,zsamemark(m2, m4)]
*
```

274
**zsearch: Search a String**

The `zsearch` function searches a marker for a given search object. It returns a marker bracketing the first occurrence of the search object. If the search object is not found, `zsearch` returns a zero-length marker pointing to the beginning of the searched region. The search object may be a literal string or it may be another marker. Styles such as bold or italic do not affect the search.

\[ \text{zsearch(region to search, string to search for)} \]

**Examples:**

The first two examples illustrate a successful search and an unsuccessful search. In the search that fails, the marker is set to the beginning of the searched string, so that the "remaining string" is the entire string.

```
unit xzsearch1 $$ successful search
m: region, result, rest
calc region := "This is a test of zsearch."
result := zsearch(region, "test")
rest := zextent(znext(result), zend(region))
at 50,50
write \result = zempty\Object not found.
\Object found: <\s,result>\text{Remaining string: <\s,rest>}
*

unit xzsearch2 $$ unsuccessful search
m: region, result, rest
calc region := "This is a test of zsearch."
result := zsearch(region, "zip")
rest := zextent(znext(result), zend(region))
at 50,50
write \result = zempty\Object not found.
\Object found: <\s,result>\text{Remaining string: <\s,rest>}
*
```

The third example searches for double-ells. Notice that "region" is reset each time through the loop, so that "region" is always the unsearched part of the original string. The fourth example replaces all occurrences of "ll" with "LL".

```
unit xzsearch3 $$ find all occurrences
m: region, result, object
i: count
calc region := "Molly's dolly has yellow braids."
object := "ll"
at 50,50
loop
calc result := zsearch(region, object)
outloop result = zempty
write <\s,zextent(zstart(region),result)> <\cr>
calc region := zextent(znext(result), zend(region))
endloop
*

unit xzsearch4 $$ find and replace
```
m: region, result
i: count
calc region := "Molly's dolly has yellow braids."
at 50,50
write Original string: <|s,region|> <|cr|>
loop
calc result := zsearch(region, "ll")
outloop result = zempty
replace result, "LL"
calc region := zextent(znext(result), zend(region))
endloop
write <|cr|>Modified string: <|s,zbase(region)|>
*

zsetmark: Bracket a Substring

The zsetmark function places a marker at a particular position within another marker, with a particular length. For example,

calc somem := zsetmark(sometext, 7, 3)

makes the marker variable somem start at location 7 from the start of the marker sometext, for a length of 3. Typically the starting location and length would have been derived from an earlier use of zlocate and zlength, and the situation usually involves reading text from a file with -datain-. Note that the start of a marker is location 0.

zstart: Before the First Character

The function zstart(marker) returns a marker that points to a zero-length string just before the first character of the text bracketed by the named marker.

Example:

unit xzstart
m: m1, m2
calc m1 := "abcdefg" $$ make a marker
calc m2 := zstart(m1) $$ marker just before "a"
replace m2, "xyz" $$ put "xyz" into m2
at 50,30
write <|s,m1|> $$ display m1
<|s,m2|> $$ display m2
at 50,100
show zbase(m1) $$ entire string containing m1
*

zsetmark: Bracket a Substring

The zsetmark function places a marker at a particular position within another marker, with a particular length. For example,

calc somem := zsetmark(sometext, 7, 3)

makes the marker variable somem start at location 7 from the start of the marker sometext, for a length of 3. Typically the starting location and length would have been derived from an earlier use of zlocate and zlength, and the situation usually involves reading text from a file with -datain-. Note that the start of a marker is location 0.

zstart: Before the First Character

The function zstart(marker) returns a marker that points to a zero-length string just before the first character of the text bracketed by the named marker.

Example:

unit xzstart
m: m1, m2
calc m1 := "abcdefg" $$ make a marker
calc m2 := zstart(m1) $$ marker just before "a"
replace m2, "xyz" $$ put "xyz" into m2
at 50,30
write <|s,m1|> $$ display m1
<|s,m2|> $$ display m2
at 50,100
show zbase(m1) $$ entire string containing m1
*

zsetmark: Bracket a Substring

The zsetmark function places a marker at a particular position within another marker, with a particular length. For example,

calc somem := zsetmark(sometext, 7, 3)

makes the marker variable somem start at location 7 from the start of the marker sometext, for a length of 3. Typically the starting location and length would have been derived from an earlier use of zlocate and zlength, and the situation usually involves reading text from a file with -datain-. Note that the start of a marker is location 0.

zstart: Before the First Character

The function zstart(marker) returns a marker that points to a zero-length string just before the first character of the text bracketed by the named marker.

Example:

unit xzstart
m: m1, m2
calc m1 := "abcdefg" $$ make a marker
calc m2 := zstart(m1) $$ marker just before "a"
replace m2, "xyz" $$ put "xyz" into m2
at 50,30
write <|s,m1|> $$ display m1
<|s,m2|> $$ display m2
at 50,100
show zbase(m1) $$ entire string containing m1
*

zsetmark: Bracket a Substring

The zsetmark function places a marker at a particular position within another marker, with a particular length. For example,

calc somem := zsetmark(sometext, 7, 3)

makes the marker variable somem start at location 7 from the start of the marker sometext, for a length of 3. Typically the starting location and length would have been derived from an earlier use of zlocate and zlength, and the situation usually involves reading text from a file with -datain-. Note that the start of a marker is location 0.

zstart: Before the First Character

The function zstart(marker) returns a marker that points to a zero-length string just before the first character of the text bracketed by the named marker.

Example:

unit xzstart
m: m1, m2
calc m1 := "abcdefg" $$ make a marker
calc m2 := zstart(m1) $$ marker just before "a"
replace m2, "xyz" $$ put "xyz" into m2
at 50,30
write <|s,m1|> $$ display m1
<|s,m2|> $$ display m2
at 50,100
show zbase(m1) $$ entire string containing m1
*
Some Examples with Markers

Reverse a List

This example collects a list of words or phrases and then displays the list in reverse order.

Unit "mx1getwords" loops until the user presses ENTER without typing a response or until the maximum number of entries allowed is reached. When a blank response is found, "mx1getwords" exits with an -outloop-.

Unit "mx1reverse" starts at the end of the list and works toward the beginning. (The index on the -loop- changes by -1.) The -reloop- skips any empty entries at the end of the list. Try taking out the -reloop-; notice how the positioning changes.

For this example, it is not necessary to pass the display position (xpos,ypos) into the -do-ne units. However, in a real program, where these routines may be used over and over, the display position will probably be different for each situation.

```
unit mx1Reverse_List
i: MAX=15 $$ maximum length of list
m: words(MAX) $$ array of words
do mx1instructions
do mx1getwords(50,100, MAX ; words)
do mx1reverse(200,100, MAX; words)
*
unit mx1instructions
at 10,10
write List some words. When you are finished, press ENTER again, and I will list them in reverse order.
*
unit mx1getwords(xpos,ypos, maxlength ; mtemp)
m: mtemp(*)$$ array of words
i: xpos,ypos $$ position of arrow
i: maxlength $$ list length
i: index $$ entry number
$$ collect a group of words or phrases
loop index := 1, maxlength
  allow blanks $$ blank response okay
  arrow xpos, ypos
  specs nookno
  ok
  endarrow
  outloop zarrowm = zempty $$ blank response found
  $$ save current response:
  calc mtemp(index) := zcopy(zarrowm)
  $$ update arrow position:
  calc ypos := ypos +15
endloop
*
unit mx1reverse(xpos,ypos, maxlength; mtemp)
m: mtemp(*)$$ array of words
i: xpos,ypos $$ display positions
```
**Alphabetize a List**

This example gathers a list of words and alphabetizes the words using a very simple sorting routine. Briefly, the inner loop \((i := 2, j)\) in the alphabetize routine compares two markers: \(mtemp(i-1)\) and \(mtemp(i)\). If the content of the first marker is "greater than" the content of the second marker (comes later in dictionary order), then the two markers are exchanged. In this way the "greatest" word works its way, one step at a time, to the end of the list. On each subsequent pass, one fewer word can be examined, since the "greatest" word is known to be at the end of the list.

The "getwords" routine is similar to the routine in the "reverse a list" example. However, this routine explicitly counts the number of entries. (You must not depend on the index of a loop for the "count." Refer to the example with -loop-.)

```
unit mx2OrderList $$ alphabetize a list
  i: MAX = 15 $$ maximum list length
  m: word(MAX) $$ array of markers
  i: count $$ # words entered
do mx2OrderInstructions
  do mx2getwords(50,100, MAX; word, count)
do mx2alphabetize(count; word)
do mx2list(200,100, count; word)
*
unit mx2getwords(xpos,ypos, maxlength; mtemp, count)
  i: xpos,ypos $$ position of arrow
  i: maxlength $$ maximum # words
  m: mtemp(*) $$ array of markers
  i: count $$ number of entries
calc count := 0
loop count < maxlength
  allow blanks $$ blank response okay
  arrow xpos, ypos
  specs nookno $$ no "ok" or "no"
  ok $$ any response okay
endarrow
outloop zarrowm = zempty $$ blank response found
$$ increment counter:
calc count := count+1
$$ copy response from arrow buffer into marker
calc mtemp(count) := zcopy(zarrowm)
```
**Example with Markers**

- **Update Arrow Position:**
  
  ```plaintext
  calc ypos := ypos + 15
  ```

- **Write:**

  - `count = <|s,count|>`

- **Unit:** `mx2OrderInstructions`

  - **At:** 10,10

  - **Write:** List some words.

    When you are finished, press ENTER again, and I will alphabetize them.

- **Unit:** `mx2alphabetize(count; mtemp)`

  - **I:** count  \(\text{number of words}\)
  
  - **M:** mtemp(*) \(\text{array of markers}\)
  
  - **M:** word1 \(\text{temporary work variable}\)

  - **I:** i, j \(\text{loop indexes}\)

  - **Loop:**

    ```plaintext
    j := count, 2, -1
    loop i := 2, j
        if mtemp(i-1) > mtemp(i)
            $$\text{exchange markers:}$$
            calc word1 := mtemp(i)
            mtemp(i) := mtemp(i-1)
            mtemp(i-1) := word1
        endif
    endloop
    endloop
    ```

- **Unit:** `mx2list(xpos,ypos, count; mtemp)`

  - **I:** xpos, ypos \(\text{display positions}\)
  
  - **I:** count \(\text{number of words}\)
  
  - **M:** mtemp(*) \(\text{array of markers}\)

  - **I:** index

  - **Loop:**

    ```plaintext
    index := 1, count
    at xpos, ypos + 15(index - 1)
    show mtemp(index)
    ```

**See Also:**

- Sorting an Array (p. 213)

**Counting Vowels**

In this example, the number of vowels (a, e, i, o, u) and semivowels (w, y) in a user’s sentence are counted.

- **Unit:** `mx3vowels` \(\text{count vowels and semivowels}\)

  - **M:** C \(\text{marker}\)
  
  - **I:** v, sv, count \(\text{counters}\)
  
  - **F:** time \(\text{time spent}\)

- **Arrow:** 10,20; 350,100 \(\text{user enters a string}\)

- **Ok**

  - **End Arrow**
calc time := zclock $\$$ record starting time
calc $v := sv := count := 0$ $\$$ initialize counters
C := zfirst(zarrowm) $\$$ get first character
loop C $\Rightarrow$ zempty $\$$ exit if no more chars
  calc count := count + 1
  if $C = \text{"a"} \mid C = \text{"e"} \mid C = \text{"i"} \mid C = \text{"o"} \mid C = \text{"u"}$
    calc $v := v + 1$
  elseif $C = \text{"w"} \mid C = \text{"y"}$
    calc sv := sv + 1
  endif
  calc C := znext(C) $\$$ get next character
endloop
calc time := zclock - time $\$$ elapsed time
do mx3show(v, sv, count, time)
  unit mx3show(v, sv, count, time)
i: v, sv, count
f: time
at 50, 75
write Your sentence contained:
\langle s, count \rangle characters
\langle s, v \rangle vowels
\langle s, sv \rangle semivowels
Processing time = \langle s, time \rangle seconds.
*

For many applications, the execution speed of markers is just fine. For sizable amounts of text, marker calculations can be rather slow if you have to examine every character one at a time. We hope to improve the speed, but markers are very complicated (internally) and they certainly will never be as fast as integers. When processing a tight character-by-character loop, you can speed up your program by using integers as discussed below. Whenever possible, use the \texttt{zsearch} or \texttt{znextline} or \texttt{znextword} functions to avoid having to examine every single character.

This second version of the vowels program uses the \texttt{zcode} and \texttt{zk} functions. These functions produce integers, so the comparisons are done with integers (fast) instead of strings (slow) as in the -if- statement above. A -case- is used instead of an -if- to illustrate another way the comparison might be structured.

unit mx3vowels2 $\$$ count vowels and semivowels
m: C $\$$ marker
i: v, sv, count $\$$ counters
f: time $\$$ time spent
arrow 10,20; 350,100 $\$$ user enters a string
ok endarrow
calc time := zclock $\$$ record starting time
calc $v := sv := count := 0$ $\$$ initialize counters
C := zfirst(zarrowm) $\$$ get first character
loop C $\Rightarrow$ zempty $\$$ exit if no more chars
  calc count := count + 1
  case zcode(C)
    zk(a), zk(e), zk(i), zk(o), zk(u)
Plot Two User Functions Simultaneously

The user is asked to enter two functions. The functions are then plotted "in parallel." That is, for each "x" the values of both functions are displayed. This makes a nice visual effect. (The other choice is to plot one function completely and then to plot the second function.) The first function is plotted with lines; the second function is plotted with dots.

The first unit, "mx4graph," is essentially nothing but calls to other units. Such a unit is often called a "driver" unit. Separating each little task into its own unit makes a program much easier to debug and to maintain.

The continued form, \texttt{-gdraw :x,y-}, is often used when plotting a function. However, this example plots two functions simultaneously. If the continued form were used, the result would be, not two distinct lines, but an envelope of the two functions. The values "oldx" and "oldy" save the point to draw from. If the second function were also plotted with a line, it would be necessary to save an "oldx2" and "oldy2". You might change the line labeled "\texttt{-plot 1st function}" to the form \texttt{-gdraw :x,y-} and see what happens.

The variable "x" must be a globally defined "user" variable:

```plaintext
define user:
  f: x
*
unit mx4graph $$ graph two functions together
  m: Fx(2) $$ array of two marker variables
next mx4graph
do
  do mx4graphsetup $$ prepare graph
  do mx4instructions $$ display instructions
  do mx4getfunct(1; Fx) $$ get first function
  do mx4getfunct(2; Fx) $$ get second function
mode erase
do
  do mx4instructions $$ erase instructions
mode write
do
  do mx4plot( ; Fx ) $$ plot functions
*
unit mx4getfunct(num; M)
merge.global:
  i: num $$ function number
  m: M(*) $$ array of markers
  f: yvalue $$ value of evaluated function
at
  10, 10+15*(num-1)
write Enter a function of x:
arrow zwherex,zwherey; 300, 10+15*num
compute yvalue $$ evaluate user's expression
```
ok \textbackslash{return}=7\Your expression should depend on x.\endarrow

c\textarrow=\zcopy\zarrowm\text\textarrow=\text{save function}

calc \text{M(num)}:=\zcopy\zarrowm\text\textarrow=\text{save function}

* unit \text{mx4plot( } : \text{ fn )}
merge,\text{global: }\text{\textarrow=\text{includes "x"}}
m: \text{fn(*)}\text\textarrow=\text{array of markers}
f: \text{ypos}\text\textarrow=\text{current function value}
\text{\textarrow=\text{oldx,oldy }}\text{\textarrow=\text{saved position for plotting}}
inhibit startdraw \text\textarrow=\text{don't show first line}

loop \text{x := 0,10,0.1}
compute \text{ypos, fn(1) } \text\textarrow=\text{evaluate 1st function}
color \text{zblue}
gdraw \text{oldx,oldy; x,ypos } \text\textarrow=\text{plot 1st function}
calc \text{oldx := x} \text\textarrow=\text{save point for -draw-}
\text{oldy := ypos}
compute \text{ypos, fn(2) } \text\textarrow=\text{evaluate 2nd function}
color \text{zred}
gdot \text{x,ypos } \text\textarrow=\text{plot 2nd function}

endloop

color \text{zdefaultf}

* unit \text{mx4graphsetup}
gorigin \text{50, 200}
axes \text{0,-100; 360,100}
scalex \text{10}
scaley \text{5}
labelx \text{1, .5}
labely \text{5,1}

* unit \text{mx4instructions}
at \text{120,90}
write \text{Try equations such as:}
\text{exp(x/10)sin(2x)}
or \text{5sin(2x)}

* See Also:
User Variables (p. 198)
compute Storing and Evaluating Inputs (p. 165)
compute Computing with Marker Variables (p. 254)
Graphing Commands (p. 102)

Plotting Parametric Equations

This example accepts three parametric equations and plots the resulting graph.

Unit "mx5GetEq" gets the equations. The example is written so that sample equations are provided; all you need
to do is press ENTER. The -do mx5provide- executes a unit that puts some suitable example equations into the
array of equations. The first -calc- (indented after the -arrow-) inserts the preprepared equations into the response
buffer. If you want the user to enter equations, omit the -do- and the -calc-. (Note that even with the prepared
responses, you can modify the input before pressing ENTER.)
We allow users of the program to use a simple equal sign for an assignment symbol (:=). Unit "mx5equal" checks the input and converts a bare "=" into ":=".

The -compute- in "mx5GetEq" does not affect the graph. It is used to get a value of zreturn so the equation can be checked for validity. In a real program, the -write Illegal form- would probably be a -do- of a diagnostic unit. (Refer to the zreturn discussion.)

The inner loop of unit "mx5plot" runs through all three equations to find one point for plotting. The outer loop continues until "Stop Plotting" is selected from the menu.

The "user:" variables must be defined in the IEU. The program assumes that "run" is also a global variable. Although the sample only uses t, x, and y, any single letter can be used by the equations. The IEU might look like this:

```
define i: run $$ TRUE while plotting
user:
  f: a, b, c, d, e, f, g, h, i, j, k, l, m
  f: n, o, p, q, r, s, t, u, v, w, x, y, z
font zsans,15
fine 500,350
rescale TRUE, TRUE, FALSE, TRUE

unit mx5graph $$ parametric plotter
  m: eqn(3) $$ array of markers
box $$ box active display area
next mx5graph $$ use "Run from Selected Unit"
do mx5axes $$ prepare axes & labels
do mx5GetEq(eqn) $$ get the equations
do mx5plot(; eqn) $$ plot equations
at 20,110
write Press ENTER to start over.
*
unit mx5GetEq( ;E) $$ get the equations
  m: E(*) $$ local marker array
  i: index
  f: result
do mx5provide( ;E) $$ get preprepared equations
loop index := 1, 3 $$ let user modify 3 equations
  arrow 10,15+20(index-1)
  specs okassign, nookno
  do compute $$ convert = into :=
    do ok $$ check for valid expression
      calc E(index) := zcopy(zarrowm)
      write Illegal form.
    endloop
  endloop
*
unit mx5provide( ;E) $$ provide some equations
  m: E(*)
calc E(1) := "t = t+.1"
```
E(2) := "x = 5+4\sin(2t)"
E(3) := "y = 4\cos(3t)\exp(-.01t)"

* unit mx5equal(; m) $$ This routine replaces = with :=
m: m, c
i: colon
calc c := zfirst(m) $$ first character
loop c ~= zempty $$ exit if no more characters
if c = ";: & znext(c) = ";="
    calc c := znext(c)
elseif c = ";="
    replace c, ";:="
endif
calc c := znext(c)
endloop

* unit mx5plot(; eqn) $$ does actual plotting
merge,global:
m: eqn(*)
i: index
f: result
inhibit startdraw
calc t := 0 $$ initialize
run := TRUE
menu Stop Plotting: mx5stopit
loop run $$ use menu to STOP
    loop index := 1, 3
        compute result, eqn(index)
    endloop
gdraw ; x,y $$ next line segment
endloop
menu Stop Plotting $$ remove menu item

* unit mx5axes $$ prepare for graphing
gorigin 220,166
axes 0,-115;250,115
scalex 10
scaley 5
labelx 2,1
labely 1

* unit mx5stopit $$ menu item
calc run := FALSE

See Also:
User Variables (p. 198)
compute Storing and Evaluating Inputs (p. 165)
compute Computing with Marker Variables (p. 254)
Graphing Commands (p. 102)
8. Files, Serial Port, & Sockets

File I/O Introduction

File input/output allows a program to store or retrieve data. Files are referred to indirectly using "file descriptors." The file descriptor is a variable that automatically keeps track of essential information about the file, such as its length, reading position in the file, whether the file is read-only or read/write, etc. Before using a file command you must define a file descriptor (see "Defining Variables"):

```
define file: filedescriptor
```

File descriptors may be defined as global or local variables. However, local variables should be used with caution. When a unit is completed, the local variables associated with the unit are gone, and any files associated with local file descriptors are closed.

The commands for file operations are

- **addfile** create a new file
- **setfile** select an existing file
- **delfile** destroy a file
- **setdir** select a directory or folder
- **dataout** append data to a file
- **numout** append one number to a file
- **datain** read data from a file
- **readln** read one line of text from a file
- **xout** write 8-bit (binary) bytes
- **xin** read 8-bit (binary) bytes
- **reset** set position within a file

The system variable **zreturn** gives status information about file operations.

The variable **zretinf** contains the number of elements read by -datain- or -xin-.

The length in bytes of a file is given by **zlength(file descriptor)**.

You can use **zfilename(file descriptor)** to get the alphanumeric name of the file associated with that file descriptor, and **zfilepath(file descriptor)** is the sequence of directories or folders enclosing the file, so that the marker expression **zfilepath(file descriptor)+zfilename(file descriptor)** provides the full name.

The system marker variables **zuser** and **zhomedir** contain the user's ID and the pathname of the user's home directory.

See Also:
- Defining Variables (p. 190)
- File Name Specification (p. 286)
File Name Specification

The file naming conventions discussed in this section apply to all commands that reference files: -addfile-, -setfile-, -execute-, -font-, -fontp-, -icons-, -pattern-, -cursor-, -get-, -put-, and -video-. In many cases, especially when using -addfile- or -setfile-, you reference a file by invoking a file dialog box to choose the file name. But if you have auxiliary data files that are essentially part of your program, you will explicitly refer to those files by name in your program. This section deals with how to provide the name of the file.

All modern computer systems provide "hierarchical" file storage. Each "file directory" or "file folder" contains a set of related files and may also contain additional directories or folders. To find a file in an arbitrary directory, one gives the "full path name," specifying in detail the entire hierarchy of directories or folders leading down to the file of interest. Depending on the system, the separators between directories may be slashes (/), colons (:), or backslashes (\).

/cmu/cil/jones/public/draw.data
\smith\tutor\physics\wells
volume:correspondence:april:jones

When you refer to a file in your program (e.g., with a -setfile- command), you should always use slashes (/) as separators. cT will make the necessary translation from slash to colon or backslash if necessary. Note that this means that a cT program cannot reference a file whose name includes a slash, because such a file has a name that looks as though it contains a directory name. In addition to always interpreting a slash on all systems as a separator, cT also recognizes colon or backslash on systems that use those separators. However, these separators will not work with a -setfile- command on a different system, whereas the slash will work on all systems.

The simplest and most common situation is that a cT program specifies a file that is in the same directory as the program itself, in which case the full path name is not necessary; only the file name is required. Suppose your program "analyze.t" is in the "public" directory, with this full path name:

/cmucil/jones/public/analyze.t

If your program needs to refer to a file named "grafdata" located in the "public" directory, all you need to give is just the file name itself, "grafdata", and cT will look for it in the "public" directory where "analyze.t" is located.

If the file is in a subdirectory "below" your cT program, the path may start at that level. A file such as "july.data"

/cmucil/jones/public/myinfo/july.data

could be referenced by "analyze.t" as "myinfo/july.data".

You can use the -setdir- command to change the "current" directory (the one cT looks in), and the system variable zcurrentdir gives the name of the current directory.

If the file is in some miscellaneous directory unrelated to the current directory (the one where your program resides, or the directory set by a -setdir- command), the full path-name must be specified:

/cmucil/jones/public/neat.stuff

On a multiuser system, there are four places where a file may be found: in the directory containing the cT program, in the user's current directory, in a default directory specified as part of the user's login, or in some miscellaneous directory.
If the file name starts with a dot-slash (./) or double-dot-slash (../), the path is expanded relative to the directory that was the user's "current directory" when the cT program was initiated.

The tilde (~) is an abbreviation for the user's home directory. For the user "zz09":

```
~/zip  -->  /cmu/cil/zz09/zip
~/zonk/B  -->  /cmu/cil/zz09/zonk/B
```

The tilde abbreviation with another user's ID specifies that user's home directory:

```
~aa0a/cat  -->  /cmu/depta/aa0a/cat
~aa0a/subone/dog  -->  /cmu/depta/aa0a/subone/dog
```

The file name can be built of a combination of text, marker variables, and embeds. The -setfile- below looks for a file named "newfoo3.seq".

```
calc myfile := "foo"
version := 3
setfile fd; "new"+myfile+<|s,version|>+.seq"; ro
```

The marker functions zfilepath and zfilename and the system marker variable zcurrentdir provide detailed information about file names and file locations. The system marker variables zuser and zhomedir contain the user's ID and the pathname of the user's home directory.

See Also:
- setdir Select a Directory (p. 294)
- User ID and Home Directory (p. 329)
- Defining Variables (p. 190)
- Character Strings (p. 246)

### File I/O Errors

It is very important to check every file operation for successful completion. All commands relating to file operations set the system variable zreturn. After a successful file operation, zreturn is TRUE. If some kind of error occurred, zreturn has a positive value:

```
-1   all okay
 2   illegal file variable (should not happen)
 3   file not open (no preceding setfile or addfile)
 4   file not found
 5   file improper type
 6   file code-word error
 7   duplicate file name (file already exists)
 8   file quota exceeded
 9   catchall error
10   file directory full
11   permission denied (wrong directory or need rw)
12   file in use (cannot be deleted)
13   directory not empty (directory cannot be deleted)
14   -dataout- out of range (not at end of file)
15   file is currently reserved
16   illegal character read with -datain-
17   user canceled making a selection in a file dialog box
```
window not wide enough for a file dialog box
not enough memory to create a file dialog box
file operation not supported (e.g. QuickTime not installed)
trying to bring up file dialog box in background window

The above list of `zreturn` values applies to all file operations, but not all possible values apply to each command. For example, for an `-addfile` command only the values `-1, 7, and 11` would be seen. As more file operations features become available, the missing values (2, 5, etc.) will be filled in.

Since each file operation depends on successful completion of previous operations, it does not make sense to continue after a bad `zreturn`. The program should give a message and exit from file operations after any `zreturn` that is not TRUE.

**Example:**

There are three ways this example may fail: the file `MyTestFile` may not exist in the current directory; `-dataout-` won't work unless the `-setfile-` specifies "read/write"; and if `MyTestFile` is not empty, a `-reset-` is required before the `-dataout-`.

```
unit xIOzreturn
file: fd
setfile fd; "MyTestFile"; ro $$ needs "rw"
do ShortZreturnText
reset fd; end $$ try with & without the -reset-
do ShortZreturnText
dataout fd; zclock
do ShortZreturnText
*
unit ShortZreturnText
write \zreturn\\file not open \file not found \\duplicate file name \\permission denied \\dataout- out of range
*
```

**See Also:**

File I/O Examples (p. 308)
A File Editor Application (p. 155)
datain Read Data from a File (p. 294)
zreturn The Status Variable (p. 332)
Logical Operators (p. 201)
Conditional Commands (p. 18)
File Handling Commands

addfile: Create a File

The -addfile- command creates a new file and opens it with read/write access so that data can be added to the initially empty file. The system variable zreturn should be checked to see if the file was successfully created.

   addfile file descriptor; file name (or zempty); styled (optional)

The first argument of -addfile- is a file descriptor variable, followed by a semicolon. This file descriptor must have been defined as type "file:"

The second argument is a string expression for the name of the new file. If the file resides in the "current" directory (the same directory as the source file, or the directory selected by a -setdir- command), only the file name needs to be given. If the file resides in the user's directory, the form "./filename" is used. If the file resides in some other directory, a full path name must be given.

If the file name is an empty marker, a dialog box is opened for the user to choose a file name and a directory or folder to put the file in. You can use the "name" option to initialize the file name in the file dialog box, which the user can then edit.

An optional third argument specifies a "styled" file, either as just the keyword "styled" or the form "styled, expr", where the file will be styled if expr is TRUE. Here are specific examples, assuming -define file: fd-:

   addfile fd; "filename" $$ create ordinary file
   addfile fd; "filename"; styled $$ create styled file
   addfile fd; "filename"; styled,expr $$ expr is TRUE or FALSE
   addfile fd; zempty $$ invoke dialog box to prompt for file name
   addfile fd; zempty; styled $$ dialog box, create styled file
   addfile fd; zempty; styled, expr $$ expr is TRUE or FALSE
   addfile fd; zempty; name,"abcd.ef" $$ initialize file name to "abcd.ef"

If you specify that the file is styled, you can use -dataout- to write out markers with styles such as bold or italic. Every -dataout- of a marker is bracketed in the file by special delimiters, and each -datain- into a marker from a styled file reads one bracketed region, not the whole file. If you execute 5 -dataout-s to a file, you would read the file with 5 matching -datain-s. It is illegal to use -xin-, -xout-, or -readln- with a styled file, but note that znextline(m) can be used to read a line from a marker that has been read from a styled file.

You can use zfpathname(fd) and zfilename(fd) to get the full file name:

   addfile fd; zempty $$ suppose use chooses file /abc/def/data1
   show zfpathname(fd)+zfilename(fd) $$ "/abc/def"+"data1"

There are special zreturn values that apply when a file selection dialog box is invoked (by specifying an empty file name). If the user chooses in the file selection dialog box to overwrite an existing file, the contents of that file are deleted and zreturn is set to -1, ready to write into the file as though it were a completely new file. If the user cancels making a selection, zreturn is set to 17. The window must be wide enough to display the file dialog box; otherwise zreturn is set to 18. If there is not enough memory to create a file dialog box, zreturn is set to 19. If the execution window is behind other windows, the file dialog box is not displayed, and zreturn is set to 21. Concerning the latter situation, the system variable zforeground is TRUE if the execution window is fully visible, in front of all other windows.
At any time while working with the file, the current number of bytes in the file is given by `zlength(fd)`.

After executing an `addfile` command the possible values of `zreturn` are:

- 1  file created
- 7  cannot create file (a file by that name already exists)
- 11 permission denied (cannot create file in specified directory)
- 17 user canceled making a selection in a file dialog box
- 18 window not wide enough for a file dialog box
- 19 not enough memory to create a file dialog box
- 21 execution window is not the forward-most window

Example:

Each example creates a file. Don't forget to delete the test files after you've tried the examples.

This unit creates a file named "TestA" in your current directory and writes the contents of the array "AnArray" into the file. After running this example, use "Auxiliary file" on the File menu to examine the file contents.

```plaintext
unit xaddfile $$ add file to current directory
f: AnArray(5)
file: td
set AnArray := 1,2,3,4,5 $$ put values in the array
addfile td; "TestA"
dataout td; AnArray
```

The example above shows a skeleton unit. In a real application, you should check `zreturn` after every file operation. Here is the unit rewritten with checks for unsuccessful operations.

```plaintext
unit xaddfile2
f: AnArray(5)
file: td
set AnArray := 1,2,3,4,5 $$ put values in the array
addfile td; "TestB" at 100,100
if zreturn
   write File created.
else $$addfile failed
   write Cannot create file.
zreturn=|s,zreturn|>
outunit $$ can't continue file operations
endif
dataout td; AnArray at 100,125
if zreturn
   write Dataout completed.
else
   write Dataout failed.
zreturn = |s,zreturn|>
endif
```

This example puts the new file in your home directory.
**setfile: Select a File**

The `-setfile-` command sets a file descriptor to an already existing file and opens the file so that data can be read from or written into the file.

`setfile fd; file name (or zempty); ro or rw; styled (optional)`

The first argument of `-setfile-` is a file descriptor variable, followed by a semicolon. This file descriptor must have been defined as type "file:"

The second argument is a string expression for the name of the new file. If the file resides in the "current" directory (the same directory as the source file, or the directory selected by a `-setdir-` command), only the file name needs to be given. If the file resides in the user's directory, the form "./filename" is used. If the file resides in some other directory, a full path name must be given. If the file name is an empty marker, a dialog box is opened for the user to choose a file.

The third argument is a keyword that specifies the read status of the file. The keyword "ro" means read-only and indicates that the file will be read but not written. The keyword "rw" opens the file for read/write. Without the "rw", the program can read information from the file, but cannot write data into the file. If you specify "ro, expr" the file will be read-only if "expr" is TRUE, otherwise it will be read-write. Similarly, if you specify "rw, expr" the file will be read-write if "expr" is TRUE, otherwise it will be read-only.

An optional fourth argument specifies a "styled" file, either as just the keyword "styled" or the form "styled, expr", where the file will be styled if expr is TRUE. Here are specific examples, assuming `-define file: fd-`:

```
setfile fd; "TestA"; ro $ open read-only
setfile fd; zempty; ro $ invoke dialog box to choose file
setfile fd $ blank tag to close file
setfile fd; "TestA"; rw $ read/write
setfile fd; "TestA"; rw, expr $ expr is TRUE or FALSE
setfile fd; "TestA"; ro; styled $ open styled file
setfile fd; "TestA"; ro; styled,expr $ expr is TRUE or FALSE
```

---

See Also:

- File Name Specification (p. 286)
- opendir Select a Directory (p. 294)
- File I/O Errors (p. 287)
- File I/O Examples (p. 308)
- A File Editor Application (p. 155)
- Defining Variables (p. 190)
- Character Strings (p. 246)
You can use \texttt{zfilepath(fd)} and \texttt{zfilename(fd)} to get the full file name:

\begin{verbatim}
setfile fd; zempty; ro $\$ suppose file is /abc/def/data1
show zfilepath(fd)+zfilename(fd) $\$/abc/def/\"+\"data1\"
\end{verbatim}

The number of bytes currently in the file is given by \texttt{zlength(fd)}.

There are special \texttt{zreturn} values that apply when a file selection dialog box is invoked (by specifying an empty file name). If the user cancels making a selection, \texttt{zreturn} is set to 17. The window must be wide enough to display the file dialog box; otherwise \texttt{zreturn} is set to 18. If there is not enough memory to create a file dialog box, \texttt{zreturn} is set to 19.

If you specify that the file is styled, you can use \texttt{-datain-} and \texttt{-dataout-} with markers that contain styles such as bold or italic. Every \texttt{-dataout-} of a marker is bracketed in the file by special delimiters, and each \texttt{-datain-} into a marker from a styled file reads one bracketed region, not the whole file. If you execute five \texttt{-dataout-s} to a file, you would read the file with five matching \texttt{-datain-s}. It is illegal to use \texttt{-xin-}, \texttt{-xout-}, or \texttt{-readln-} with a styled file, but note that \texttt{znexline(m)} can be used to read a line from a marker that has been read from a styled file.

If the tag is simply the file descriptor (no semicolon), then the file descriptor is "canceled," which closes the file. This makes sure that all the information is sent to permanent (disk) storage and breaks the connection to the file. It is not actually necessary to close a file in this way, because all files are closed automatically at the end of execution of the program. Executing \texttt{-reset td; end-} will make sure that all data have been sent to the file from the computer's internal buffers, without actually closing the file.

A file descriptor can be reused. Suppose we have these commands (assuming appropriate \texttt{-define-s}):

\begin{verbatim}
setfile td; "TestC"; ro
datain td; "AnArray", 7
setfile td; "TestD"; rw
dataout td; AnArray, 7
\end{verbatim}

The first two commands read information from TestC into AnArray. The third command first closes TestC and then opens TestD. The last command writes the array into TestD.

The possible values of \texttt{zreturn} for \texttt{-setfile-} are

-1 file opened
4 file not found
11 permission denied
17 user canceled making a selection in a file dialog box
18 window not wide enough for a file dialog box
19 not enough memory to create a file dialog box
21 execution window is not the forward-most window

Concerning the situation where \texttt{zreturn = 21}, the system variable \texttt{zforeground} is TRUE if the execution window is fully visible, in front of all other windows.

\textit{Example:}

\begin{verbatim}
unit xsetfile
file: fd $\$ define file descriptor
at 50,50
setfile fd; "zonkity"; rw $\$ nonexistent file
write zreturn = <|s,zreturn|> (probably 4).
\end{verbatim}
FILE HANDLING COMMANDS

at 50,90
loop
setfile fd; zempty; ro $$ invoke dialog box
if zreturn = 18 $$ window too small
   write Make window bigger.
   pause
else
   outloop
endif
endloop
write zreturn = <|s,zreturn|> (probably -1).
*

See Also:
  File Name Specification (p. 286)
  setdir Select a Directory (p. 294)
  File I/O Errors (p. 287)
  File I/O Examples (p. 308)
  A File Editor Application (p. 155)
  Defining Variables (p. 190)
  Character Strings (p. 246)
  serial Serial Port (p. 309)

delfile: Delete a File

The -delfile- command removes the file pointed to by the file descriptor in its tag. Before using -delfile-, you must execute either -addfile- or -setfile- to specify the file descriptor.

delfile td

It is possible to have read/write permission to a file without having permission to delete the file. Thus, for -delfile-, a zreturn of 11 means "does not have delete access for this file."

The relevant zreturn values for -delfile- are

-1 everything ok
  3 file not open
  11 permission denied

Example:

unit xdelfile
file: td
setfile td; "TestA”; rw $$ set file descriptor
at 100,100
if not(zreturn)
   write \zreturn=4\ File not found.
   \Permission denied.
   outunit
endif
delfile td
if zreturn
   write File deleted.
else
write \zreturn=3;File not open.
\Permission denied -- cannot delete.
endif
*

See Also:
File Name Specification (p. 286)
File I/O Errors (p. 287)
File I/O Examples (p. 308)
Defining Variables (p. 190)

setdir: Select a Directory

The -setdir- command changes the "current" directory or folder:

    setdir "~/rjk/stuff" $$ later -setfile-s relative to this directory

When a -setfile- command specifies just a file name, that file is looked for in the current directory or folder. At the beginning of a program, by default the current directory or folder is the one where the program itself is located. The -setdir- command changes this default, so that later -setfile- commands look in the new current directory. The name of the current directory is available in the system marker variable zcurrentdir.

A major use of -setdir- is in association with -setfile- commands that invoke a file dialog box for the user to choose a file. If the user chooses a file from a different directory, it is usually appropriate to change the current directory to match that new directory, so that in later user choices the file dialog box will start in the new directory, not the program directory. To achieve this effect, do the following:

    setfile fd; zempty; ro $$ bring up file dialog box
    setdir zfilepath(fd) $$ make the new directory the "current" directory

After executing a -setdir- command the possible values of zreturn are

    -1 directory or folder created
    4 no directory or folder of this name

See Also:
File Name Specification (p. 286)
setfile Select a File (p. 291)

datain: Read Data from a File

The -datain- command reads information from a human-readable file (or from the serial port or from a socket). Just as with the -show- command, the information may be numbers or a text string.

    define file: td $$ file descriptor
    f: A(10,5) $$ a numeric array
    i: value $$ an integer variable
    m: m1 $$ a marker variable
    datain td; value $$ read 1 number
    datain td; A $$ fill numeric array
The first argument is a file descriptor followed by a semicolon.

The second argument is a variable or an array into which the input is placed. If the argument is a numeric array name with no modifiers, the entire array is read from the file. If the array name is followed by a comma and a number (N), the next N elements are read from the file. If the argument is an array element or a simple numeric variable, a single number is read. If the argument is an array element followed by a comma and a number (N), then N consecutive array elements are read.

If the second argument is a marker variable, and no character count is specified, the entire remainder of the file is read into memory and the marker variable is wrapped around that text. If a character count is specified, that many characters (or however many remain in the file) are read into memory and the marker is wrapped around the text. See the -readln- command for how to read one line of text from a file.

If you read from a styled file, each -datain- into a marker can contain styles such as bold or italic. Every -datain- into a marker from a styled file reads one bracketed region, not the whole file. If you execute five -dataout-s to a file, you would read the file with five matching -datain-s.

The system variable \texttt{zretinf} tells how many items were actually transferred. Attempting to -datain- at the end of the file, where there is no more data, does not cause a failure (\texttt{zreturn} stays TRUE), but \texttt{zretinf} = 0 (no information read).

In the case of a numeric -datain-, numbers may be terminated by space, tab, new line (carriage return), or end of file. The number of elements read by -datain- is given by the system variable \texttt{zretinf}. After reading a number, the next character in the file is the one that terminated the number. For example, if a number is terminated by a carriage return, and next you read text from the file, the first character of that text will be the carriage return.

If there is something wrong with the actual data in the file (such as a number containing illegal characters), \texttt{zreturn} has the value 16, and the position within the file is set to the point where the error was detected. One could then use -xin- to examine the file byte by byte.

For -datain- the relevant values of \texttt{zreturn} are

\begin{itemize}
  \item -1 operation successful
  \item 3 file not open
  \item 16 illegal character (in the case of numeric -datain-)
\end{itemize}

A file that is being read numerically can have more than one number on a line, even though the simple numeric form of the -dataout- command puts only one number on a line. You could, for example, use -datain- to read multiple numbers per line that had been output using embeds, or which were typed into a file using a simple text editor, or which were generated by a Fortran program. Legal separators between numbers are space, tab, and carriage return. Numbers may be in "scientific notation" as "2.103E+07" (which means $2.103 \times 10^7$). Both lowercase "e" and uppercase "E" are recognized.

See the -serial- command and the discussion of sockets for special considerations that apply to using -datain- when communicating with apparatus connected to the serial port or with other processes.
**Example:**

In this example, the `-datain- attempts to read in enough numbers from TestB to fill the entire array "Vector". If there are not 100 numbers in TestB, it will read as many numbers as possible. The value of `zretinf` tells how many numbers were actually found.

```plaintext
unit xdatain1
f: Vector(100)
file: td
setfile td; "TestB"; ro $$ in the current directory
datain td; Vector
at 100,100
write <|s,zretinf|> numbers found
```

In the next example, we read the same file as text rather than numerically, then display the number of characters and the text that was read:

```plaintext
unit xdatain2
m: filetext
file: td
setfile td; "TestB"; ro $$ in the current directory
datain td; filetext, 80 $$ read 80 characters
at 100,100
write <|s,zretinf|> characters found:
<|s,filetext|>
```

Units "xdatain1" and "xdatain2" are skeleton units just for illustration. Unit "xdatain3" does the same thing as "xdatain1", but includes the checks for successful completion.

```plaintext
unit xdatain3
f: Vector(100)
file: td
setfile td; "TestB"; ro
at 100,100
if not(zreturn)
    write Setfile failed.
    zreturn = <|s,zreturn|>
    outunit $$ no point in continuing
endif
datain td; Vector
if zreturn
    write datain got <|s,zretinf|> numbers
else
    write Datain failed.
    zreturn = <|s,zreturn|>
endif
```

**See Also:**
- File Name Specification (p. 286)
- File I/O Errors (p. 287)
- File I/O Examples (p. 308)
**readln: Read a Line from a File**

The `-readln-` command reads a line of text from a file (or serial port or socket) into a marker. Whereas the `-datain-` command reads an entire file or a specified number of characters into a marker, the `-readln-` operation stops after encountering a newline or a specified character string.

```
readln fd; m1 $$ read up to and including a newline
readln fd; m1, m2   $$ read up to and including a match to m2
```

The optional final argument is a string whose contents specify when to stop reading. If the final argument is omitted, the terminator is a newline. For example,

```
readln fd; m1, "st"
```

will read into m1 up to and including the sequence of "s" followed by "t". If no terminator is found, or if the final argument is `zempty`, the entire remainder of the file will be read.

It is illegal to use `-readln-` with a styled file, and an execution error will result. Note that `znextline(m)` can be used to read a line from a marker that has been read from a styled file.

See the `-serial-` command and the discussion of sockets for special considerations that apply to using `-readln-` when communicating with apparatus connected to the serial port or with other processes.

**Example:**

```
unit xreadln
file: testing
marker: line
addfile testing; "xreadln" $$ create a test file
string line
This is the first line
and this is the second.
\
dataout testing: line $$ write out two-line text
reset testing: start
readln testing: line $$ read first line
readln testing: line $$ read second line
at 10,20
show line $$ show second line
delfile testing $$ delete the test file
*  
```

**See Also:**

File Name Specification (p. 286)
File I/O Errors (p. 287)
dataout: Write Data to a File

The `-dataout-` command writes information into a file in human-readable format. Just as with the `-show-` command, the information may be numbers or a text string. One cannot write into the middle of a file but can only append to the end of a file: use `-reset ff; end-` to position to the end of the file or `-reset ff; empty-` to delete the current contents of the file.

```
define file: td $$ file descriptor
define f: A(10,5) $$ a numeric array
define m: m1 $$ a marker variable

dataout td; A $$ entire numeric array
ndataout td; A,10 $$ first 10 elements
ndataout td; A(3,2) $$ one array element
ndataout td; A(2,1),7 $$ A(2,1) through A(3,2)
dataout td; m1 $$ append m1 to file
ndataout td; m1, 65 $$ append 65 characters to file
ndataout td; m1+<\s,x+5|>+" end" $$ embedded -show-
```

The first argument of `-dataout-` is a file descriptor, followed by a semicolon. The second argument is a variable or an array. If the argument is a numeric array name with no modifiers, the entire array is appended to the file. If the array name is followed by a comma and a number (N), the first N elements are appended to the file. If the argument is an array element or a simple numeric variable, that single number is appended. If the argument is an array element followed by a comma and a number (N), then N consecutive array elements are appended.

If the second argument is a marker variable, and no character count is specified, the entire text bracketed by the marker variable is appended to the file. If a character count is specified, that many characters from the start of the marker are appended to the file. If the file is not a styled file, styles (such as bold, italic, superscript, etc.) are discarded before appending the text to the file. Marker expressions with embedded `-show-` commands can be used to produce arbitrary formats for the output.

If you look at the file by choosing "Auxiliary file" on the File menu after writing into it with the numeric form of `-dataout-`, you will see one number per line, with a space at the beginning of each line. The text form of `-dataout-` permits constructing your own output format, using marker expressions and embedded `-show-`. The following statement could be used to append to the file the string "The 1979 population was 25000." (Note the explicit carriage return at the end of the line.)

```
dataout td; "The +<\s,year>|>+," population was +<\s,pop>|>+."+<\cr>
```

You can also use the `-numout-` command to output a number without spaces or carriage returns.

If the numbers sent by a numeric `-dataout-` are very large or very small, they are written in the so-called "scientific" or "E" format, as in "2.103E+07" (which means 2.103 times 10^7).
If you write into a styled file, you can use -dataout- with markers that contain styles such as bold or italic. Every -dataout- of a marker is bracketed in the file by special delimiters, and each -datain- into a marker from a styled file reads one bracketed region, not the whole file. If you execute five -dataout-s to a file, you would read the file with five matching -datain-s.

Because -dataout- buffers the output, it may not be sent immediately. The output buffer is flushed by any of these conditions:

- -setfile- with no filename is executed;
- -reset  fd; end- is executed;
- the output buffer is full;
- the program finishes execution.

The values of zreturn for -dataout- are

-1 data written
3 file not open
11 permission denied
14 out of range

The "permission denied" error is caused by trying to write to a read-only file. The "out of range" error is caused by trying to write at the start or in the middle of a nonempty file. The position within the file must be modified with -reset-: one can write data only by appending to the end of a file.

Examples:

The nested -loop-s fill the array with the values 101,102,103,104, 201,202,203, etc. The -dataout- writes the entire 40 numbers contained in AnArray into the newly created file.

```
unit xdataout1
f: AnArray(10,4), i, j
file: td
addfile td; zempty $$ create a file (need wide window)
loop i := 1,10        $$ put numbers into array
loop j := 1,4 
  calc AnArray(i,j) := 100i+j
endloop
endloop
dataout td; AnArray $$ output entire array
```

The next example assumes that you already have FileOne and FileTwo. It reads from one file (FileOne) and appends the modified data onto a second file (FileTwo). Note that FileTwo must be opened as a read/write file.

```
unit xdataout2
f: A(100), B(100), i
file: file1, file2
setfile file1; "FileOne"; ro $$ open FileOne
setfile file2; "FileTwo"; rw $$ open FileTwo
datain file1; A
loop i := 1,zretinf $$ modify values of A
  calc B(i) := 2A(i)
endloop
reset file2; end $$ prepare for output
```
FILES, SERIAL PORT, & SOCKETS

dataout file2; B, zretinf $ append to FileTwo

* The examples above omitted checks of \texttt{zreturn} so that the structure would be easier to see. Here is the example with checks for successful operations.

unit xdataout2b $ with error info
  f: A(100), B(100), i
  file: file1, file2
next StartOver
at 100,50 $ position for \texttt{zreturn} comments
setfile file1; "FileOne"; ro $ open FileOne
if not(zreturn) $ if -setfile- failed
  write FileOne not found
outunit $ exit if error
endif
setfile file2; "FileTwo"; rw $ open FileTwo
if not(zreturn) $ if -setfile- failed
  write \texttt{zreturn} = 4 FileTwo not found
  \texttt{Cannot write in FileTwo}
outunit $ exit if error
endif
datain file1; A $ read data into A
if not(zreturn) $ if -datain- failed
  write datain failed; zreturn = <|s,zreturn|>
outunit $ exit if error
endif
loop i := 1,zretinf $ prepare values for B
  calc B(i) := 2A(i)
endloop
reset file2; end $ prepare for dataout
if not(zreturn) $ if -reset- failed
  write reset failed; zreturn = <|s,zreturn|>
outunit $ exit if error
endif
dataout file2; B,zretinf $ read B out into FileTwo
if not(zreturn) $ if -dataout- failed
  write dataout failed; zreturn = <|s,zreturn|>
outunit $ exit if error
endif
write All Done
next NextUnit $ successful transfer; continue
*
unit StartOver
write .... instructions or information for the user
*
unit NextUnit
write Data transferred; now continue . . .
*

\textit{See Also:}
numout Output a Number to a File (p. 301)
File Name Specification (p. 286)
File I/O Errors (p. 287)
**numout: Output a Number to a File**

The `-numout-` command outputs a single number to a file, with no leading space and no trailing carriage return (unlike the `-dataout-` command):

```
numout file descriptor; number
```

The `-dataout-` command is useful for writing numbers to a file because it automatically separates successive numbers by putting them on sequential lines (with a leading space). The `-numout-` command is useful when you want to control the exact format of the file, although this can also be done using "embeds."

**Example:**
```
unit xnumout
file: datafile
i: nn
addfile datafile; "numout" $s create file
if ~zreturn
  write zreturn = <|s, zreturn|>
outunit
endif
loop nn := 1, 10
  if frac(nn/5) = 0
    * -dataout- outputs a leading space
    * and a trailing carriage return:
    dataout datafile; sqrt(nn)
  else
    * -numout- just outputs the number:
    numout datafile; sqrt(nn)
    * add two spaces after the number:
    dataout datafile; "  "
  endif
endloop
write Done
```

**See Also:**
- `dataout` Write Data to a File (p. 298)
- `xout` Write Bytes to a File (p. 303)

**xin: Read Bytes from a File**

The `-xin-` command reads information from a file, just like `-datain-`, except `-xin-` expects 8-bit bytes while `-datain-` expects text for a marker variable or a series of human-readable numbers separated by spaces, tabs, or carriage returns. No special formatting is required for `-xin-`; any file can be read with `-xin-` including, in
particular, binary files such as are used for images. The number of elements read by -xin- is given by the system variable \texttt{zretinf}.

\begin{verbatim}
define b: MyBytes(10,5) $$ sample array
xin td; value        $$ read 1 number
xin td; MyBytes $$ fill array
xin td; MyBytes,12 $$ first 12 elements
xin td; MyBytes(3,6) $$ single element
xin td; MyBytes(3,6),5 $$ five elements
\end{verbatim}

The first argument is a file descriptor followed by a semicolon. The second argument is the variable into which information is placed. If the variable is an array name, the -xin- attempts to fill the array. If the variable is an array element, the -xin- starts with that element and continues filling the array. When there are more bytes than will fit in the array, the remaining bytes can be read later. That is, the file is positioned so that the next unread byte is ready.

The optional third argument is the number of bytes (\texttt{N}) to read. If a third argument (\texttt{N}) is given, the first \texttt{N} elements of the array are filled. If an element of an array is specified, then \texttt{N} elements, starting with the specified element, are filled.

The system variable \texttt{zretinf} tells how many items were actually transferred. Attempting to -xin- at the end of the file, where there are no more data, does not cause a failure (\texttt{zreturn} stays \texttt{TRUE}), but \texttt{zretinf} = 0 (no information read).

The \texttt{zreturn} values for -xin- are the same as for -datain-:

\begin{verbatim}
-1  operation successful
 3  file not open
\end{verbatim}

It is illegal to use -xin- with a styled file, and an execution error will result.

See the -serial- command and the discussion of sockets for special considerations that apply to using -xin- when communicating with apparatus connected to the serial port or with other processes.

\textit{Example:}

This example lets you select a file and then reads the file in 1000-byte chunks. The lines between -outloop- and -endloop- count and display the number of uppercase letters in each chunk.

\begin{verbatim}
unit xxin1 $$ read file & count capitals
b: A(1000)
i: count, totalbytes, capitals, temp
file: fd
loop
setfile fd; zempty; ro $$ invoke dialog box
if \texttt{zreturn} = 18 $$ window too small
write Make window bigger.
pause
else
endif
outloop
endloop
at 30,90;350,350 $$ for display of values
\end{verbatim}
FILE HANDLING COMMANDS

```pascal
do zreturns
calc totalbytes := capitals := temp := 0
loop $$ outloop used to exit from loop
xin fd; A $$ read 1000 bytes
if not(zreturn) $$ -xin- failed
do zreturns
outunit $$ must exit
endif
outloop zretinf = 0 $$ no more data
loop count := 1, zretinf $$ count capitals
if zk(A) <= A(count) <= zk(Z)
calc temp := temp+1
endif
endloop
$$ show totals for this chunk:
write <|cr|><|s,zretinf|>; <|s,temp|>
calc totalbytes := totalbytes + zretinf
capitals := capitals + temp
temp := 0
if zwherey > (zymax-20) $$ near bottom
pause
erase 30,96;200,zymax
at 30,90 $$ reposition -write-
endif
endloop
write <|cr|>Bytes in file = <|s,totalbytes,6|>
Upper case letters = <|s,capitals|>
*
unit zreturns
write \zreturn\ok\\file not open
\file not found
\\duplicate file name (file already exists)
\\permission denied (can't write in directory)
\\-dataout- out of range (not at end of file)
\canceled file request
write <|cr|>
*

See Also:
File Name Specification (p. 286)
File I/O Errors (p. 287)
File I/O Examples (p. 308)
Defining Variables (p. 190)
readln Read a Line from a File (p. 297)
serial Serial Port (p. 309)
Overview of Sockets (p. 311)

xout: Write Bytes to a File

The -xout- command writes information into a file, just like -dataout-, except -xout- writes 8-bit bytes to produce a "binary" file such as is used for images. If you try to -xout- a floating-point number, it will be rounded to an integer. If you try to -xout- a number that is negative, or greater than 255, you will get an error
message. One cannot write into the middle of a file but can only append to the end of a file: use -reset ff; end-
to position to the end of the file or -reset ff; empty- to delete the current contents of the file.

```define b: q(10,5)  $$ sample byte array
xout td; 4x-9  $$ write value of (4x-9)
xout td; q  $$ write array
xout td; q,10  $$ first 10 bytes of q
xout td; q(3,2)  $$ 1 byte: q(3,2)
xout td; q(2,1),7  $$ q(2,1) thru q(3,2)
```

The first argument of -xout- is a file descriptor, followed by a semicolon. The second argument is a variable or
an array. If the argument is an array name, the entire array is written into the file. If the argument is an array
element, writing starts with that element of the array.

The optional third argument is the number of bytes (N) to write. If a third argument (N) is given, the first N
elements of the array are written. When an element of an array is specified by the second argument, then N
elements, starting with the specified element, are written.

Because -xout- buffers the output, it may not be sent immediately. The output buffer is flushed when

- a -setfile- with no filename is executed;
- reset fd; end- is executed;
- the output buffer is full;
- the program finishes execution.

The **zreturn** values for -xout- are the same as for -dataout-:

-1  data written
  3  file not open
  11  permission denied
  14  out of range

It is illegal to use -xout- with a styled file, and an execution error will result.

**Examples:**

In this example, the ASCII values of a, b, c, d, and e are added to the end of a file.

```unit xxout1  $$ append to file
b: A(5)  $$ array of 5 bytes
file: fd
set loop
  A := zk(a), zk(b), zk(c), zk(d), zk(e)
  setfile fd; zempty; rw  $$ invoke dialog box
  if zreturn = 18  $$ window too small
    write Make window bigger.
  else
    outloop
  endif
endloop
if not(zreturn)  $$ setfile failed
  write setfile failed: <|s,zreturn|>
```
outunit $$ exit on failure

reset fd; end $$ move to end of file
if not(zreturn) $$ reset failed
write reset failed: <|s,zreturn|>
outunit $$ exit on failure
endif

xout fd; A $$ append array A to file
write \zreturn \Done. \xout failed: <|s,zreturn|>

The next example uses -xout- and -dataout- to store the same information in two different ways. Use "Auxiliary file" to examine the resultant files to see how the contents differ.

unit xcontrast
file: file1, file2
do make_file( ; file1) $$ make first file
outunit ~zreturn
do make_file( ; file2) $$ make second file
outunit ~zreturn
at 50,150
write Type something and press ENTER.
at 50,170
loop $$ exit from loop with outloop
 . pause keys=all $$ collect typed keys
outloop zkey = zk(next) $$ exit on ENTER
outloop ~(-31 < zkey < 127) $$ misc. illegal keys
 . plot zkey $$ show keypress
 . xout file1; zkey $$ write key to file1
 . write \zreturn\bad xout: <|s,zreturn|>
 . dataout file2; zkey $$ write key to file2
 . write \zreturn\bad dataout: <|s,zreturn|>
endloop
write <|cr>|<cr|> DONE
setfile file1
write \zreturn\file1 not released
setfile file2
write \zreturn\file2 not released
*
utiunit make_file( ; fd) $$ create file
file: fd
loop
addfile fd; zempty $$ invoke dialog box
if zreturn = 18 $$ window too small
write Make window bigger.
pause
else
outloop
endif
endloop
*

See Also:
numout Output a Number to a File(p. 301)
reset: Changing Position within a File

The -reset- command is used to reposition a file for reading or writing or to empty a file.

```
reset  td;start  $$ position to start
reset  td;end   $$ position to end
reset  td;empty $$ delete file contents
```

The first argument of the tag is a file descriptor followed by a semicolon. The second argument is a keyword: start, end, or empty.

The statement -reset td; end- not only positions to the end of the file but also flushes any internal buffers of data out to the file, without closing the file.

The "empty" keyword empties the file and leaves the file descriptor ready for writing to the file with -dataout- or -xout-.

When a file is opened with -setfile-, it is positioned to the beginning of the file for reading the file with -datain- or -xin-. Before writing to the file with -dataout- or -xout-, you must either reposition to the end of the file or delete the contents of the file.

The zreturn values for -reset- are

```
-1     operation successful
 3     file not open
11    permission denied
```

Any open file can be repositioned to the start or the end, but only a file with read/write access can be emptied.

See Also:

File Name Specification (p. 286)
File I/O Errors (p. 287)
File I/O Examples (p. 308)
Defining Variables (p. 190)

zretinf: Number of Elements Read

The system variable zretinf (return information) contains the number of elements read by -datain- or -xin-.

Attempting to do a -datain- at the end of the file, where there are no more data or not "enough" data does not cause a zreturn failure. The end of the file is "noticed" by using zretinf. If you ask to read 50 variables and there are only 30 in the file, zretinf returns 30. If you try to read at the very end of the file, zretinf will be 0.

Note that zlength(file descriptor) gives the current length in bytes of the associated file.
Examples:

The first example illustrates reading from an empty file, something you ordinarily wouldn't want to do! Even so, the \texttt{zreturn} for \texttt{-datain-} is \texttt{-1}.

\begin{verbatim}
unit xzretinf
file: td
addfile td; "TestZ"
write not(zreturn)\ cannot create file\/
outunit not(zreturn)
datain td; AnArray \$S read from empty file
write not(zreturn)\ datain failed\/
outunit not(zreturn)
at 100,100
write zreturn = <\{s,zreturn\}>
zretinf = <\{s,zretinf\}>
delfile td \$S delete the file
\end{verbatim}

In the next example, a file is created and an array of five numbers is stored in the file. Then the \texttt{-datain-} tries to read 100 values (from the same file) in order to fill ArrayB. However, the data file only has five values in it, so \texttt{zretinf} returns five. Note that after the \texttt{-dataout-}, the file must be \texttt{-reset-} to the beginning for \texttt{-datain-}.

\begin{verbatim}
unit xzretinf2
file: td
f: AnArray(100)
set AnArray := 3, 6, 9, 12, 15
addfile td; "TestZ"
write not(zreturn)\ cannot create file\/
outunit not(zreturn)
dataout td; AnArray, 5 \$S store the 5 values
write not(zreturn)\ dataout failed\/
outunit not(zreturn)
reset td; start \$S prepare for datain
write not(zreturn)\ reset failed\/
outunit not(zreturn)
datain td; ArrayB \$S read 100 values
write not(zreturn)\ datain failed\/
outunit not(zreturn)
at 50,100
write Number of values read = <\{s,zretinf\}>
delfile td
\end{verbatim}

See Also:

- File Name Specification (p. 286)
- File I/O Errors (p. 287)
- File I/O Examples (p. 308)
- Defining Variables (p. 190)
- \texttt{zreturn} The Status Variable (p. 332)
**File I/O Examples**

Every command that does a file operation should test `zreturn`. The program should give a message and make a "graceful" exit from file operations after any `zreturn` that is not TRUE. The example below uses `-outunit-`; you might use `-jump-` instead.

For another major example of using files, see "A File Editor Application" mentioned in the "See Also" below.

The following unit "zreturnText", which reports on the file operations, is a very "verbose" version: it makes a comment after every operation. A finished program might return messages only after unsuccessful operations. The "system errors" comment is included because an undefined `zreturn` means something is very wrong and it should be reported.

```plaintext
unit xfilezreturn $$ file operation zreturns
  f: A(10), B(25)
  i: nn
  file: file1
  do xgetfile( ;file1) $$ create file
     do zreturnText(30,40, "addfile")
     loop nn := 1, 10
        calc   A(nn) := 100+nn
     endloop
     dataout file1; A $$ get some data
     do zreturnText(70,zwherey+20, "dataout")
     outunit not(zreturn)
     reset file1; start $$ reset to start of file
     do zreturnText(70,zwherey+20, "reset")
     outunit not(zreturn)
     datain file1; B $$ read data into B
     do zreturnText(30,100, "datain")
     outunit not(zreturn)
     loop nn:= 1, zretinf $$ a miscellaneous calculation
        calc   B(nn) := 2A(nn)
     endloop
     dataout file1; B, zretinf $$ append data
     do zreturnText(70,zwherey+20, "dataout")
* unit xgetfile( ; fd)
  file: fd
  loop
     addfile fd; zempty $$ invoke dialog box
     if zreturn = 18 $$ window too small
        write Make window bigger.
        pause
     else
        outloop
     endif
* unit zreturnText(tx,ty, command) $$ conservative form
  f: tx,ty
```
FILE HANDLING COMMANDS

m: command
at tx,ty
write <$s,command|>

```plaintext
write \zreturn \ OK
\file not open (need setfile or addfile)
\file not found
\duplicate file name (file already exists)
\permission denied (wrong directory?)
\-dataout- out of range (not at end of file)
\illegal character read with -datain-
\user canceled selection in file dialog box
\window not wide enough for file dialog box
\not enough memory to create file dialog box
```

else
write system error <$s,zreturn|>
Please report this error to
the distributors of cT.
endcase

See Also:
A File Editor Application (p. 155)

**serial: Serial Port**

The -serial- command allows you to specify the details of how to send data to and receive data from external devices attached to the "serial port" of the computer. The -serial- command is similar to the -setfile- command in that it establishes a connection with a file descriptor, after which you can perform -xin-, -xout-, -datain-, and -dataout- operations. These are the arguments of the -serial- command:

```
serial file descr.; port no., baud rate, data bits, parity, stop bits
```

For example,

```
serial fd; 1, 9600, 8, even, 1
```

specifies port number 1, 9600 baud, 8 bits per data byte, and 1 stop bit. The number of data bits is typically 6, 7, or 8. The parity can be even, odd, or none. The stop bits are typically 1, 1.5, or 2. While the port number, baud rate, data bits, and stop bits can be arbitrary expressions, the only way to specify the parity is with the specific words even, odd, or none.

On a Macintosh, the modem port is port number 1, and the printer port is port number 2. On a PC, serial port COM1 is port number 1, and serial port COM2 is port number 2.

An -xin- command reads whatever data are already available at that moment, so it is important to check zretinf to find out how many elements were read:

```
xin fd; 5 $$ read up to 5 bytes if available now
show zretinf $$ show the number of bytes read
```
Important note: When reading from the serial port or a socket, the -datain-, -readln-, and -xin- commands will finish immediately if there are no bytes (characters) available, setting zretinf to zero. If one or more characters are initially available the -xin- command will continue reading until the specified number of bytes has been read, or until there are no bytes available. The -datain- command when reading into a marker with no specified character count will also continue reading until no bytes are available. The other commands and options (numeric -datain-, -datain- into a marker with a character count, and -readln-) will try very hard to complete their work if there are bytes initially available (numeric -datain- requires that the first character be a legitimate start of a number). If they don't find what they are looking for (nonnumeric character signaling the end of a number, end of line, etc.), they will wait a long time before giving up. For this reason it is usually appropriate when reading the serial port or a socket to use -xin-, or -datain- into a marker with no character count specified.

As with the -setfile- command, "serial fd" closes the connection between the file descriptor fd and the serial port.

See Also:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>setfile</td>
<td>Select a File</td>
<td>291</td>
</tr>
<tr>
<td>xin</td>
<td>Read Bytes from a File</td>
<td>301</td>
</tr>
<tr>
<td>xout</td>
<td>Write Bytes to a File</td>
<td>303</td>
</tr>
</tbody>
</table>
Socket Commands

Overview of Sockets

Sockets in cT are a general facility for communicating between different programs running on the same or different computers. These are two-way, read/write, reliable channels. Some computer systems support separate processes running simultaneously on the same machine, and a socket can carry messages between these processes. In Unix-based and AppleTalk-based computer networks, cT programs running on different computers can be connected by a socket. This provides the capability for two or more users of cT programs to communicate with each other interactively at high speed.

While interprogram communication between programs running on the same machine can be achieved by reading and writing ordinary files, socket communication is much faster. Sockets can be used for communication between two cT programs or between a cT program and a program written in some other language, provided that the other program can cooperate with cT (by using the same protocol for sending and receiving messages).

An important use of sockets is to connect between a cT program that handles the graphical user interface and a program written in some other language. At present this is supported on Macintosh System 7 and on Unix. Sample C-language programs are provided in the "socket" programs distributed with cT. Interprocess communication on Windows has a very different structure, and cT supports Dynamic Data Exchange rather than sockets on Windows. Eventually it is intended to support the somewhat similar AppleEvents on the Macintosh.

A socket is referenced with a file variable, just as with a file or a serial port. Once the socket is opened and connected it acts very much like a file: input and output operations are done with -datain-, -readln-, and -dataout-, or -xin- and -xout-.

Important note: When reading from a socket or the serial port, the -datain-, -readln-, and -xin- commands will finish immediately if there are no bytes (characters) available, setting zretinf to zero. If one or more characters are initially available the -xin- command will continue reading until the specified number of bytes has been read, or until there are no bytes available. The -datain- command when reading into a marker with no specified character count will also continue reading until no bytes are available. The other commands and options (numeric -datain-, -datain- into a marker with a character count, and -readln-) will try very hard to complete their work if there are bytes initially available (numeric -datain- requires that the first character be a legitimate start of a number). If they don't find what they are looking for (nonnumeric character signaling the end of a number, end of line, etc.), they will wait a long time before giving up. For this reason it is usually appropriate when reading a socket or the serial port to use -xin-, or -datain- into a marker with no character count specified.

The greatest difficulty in using a socket is in making the initial connection between the two independent processes. For the two to communicate they each need to know where the other is. Conceptually, the way this works in cT is that one process (called the server) advertises its presence with a -server- command and waits for connection attempts. The other process (called the client) executes a -socket- command to attempt to connect. The server also uses a -socket- command to find out that the final connection has been made. Once the server and client are connected to each other there is no remaining asymmetry between the two. Both of them can simply read and write the socket as they would a file.

See Also:

server Advertising a Server (p. 312)
socket Connecting to Another Process (p. 311)
getserv Asking about Servers (p. 316)
server: Advertising a Server

The -server- command allows a program to advertise itself. It has these possible forms:

```plaintext
server fd; logical, "serverA", "Joe's" $$ kind and instance of server
server fd; logical, "serverA" $$ instance of server is zempty
server fd; local,"serverB" $$ only for connections on same machine
server fd; ptp,"serverC" $$ program-to-program connection on Macintosh
server fd; absolute, "N5000" $$ mostly for Unix
```

The first argument of the tag is a file variable. Since this only advertises availability, after executing a -server- command the socket is NOT yet open for reading and writing. A subsequent -socket- command is needed for that (see -socket- documentation).

The second argument of the tag specifies the address type and is the word "logical", "local", "ptp" (program-to-program on Macintosh), or "absolute". Programmers are strongly encouraged to avoid absolute addresses, because absolute addresses are machine, system software, and network software dependent.

The third argument of the tag is the name of the server. Other programs can use the -server- command to connect by name to this server.

Logical addresses can contain two parts. The first part is the kind of server, the second part is the particular server of this kind. The first part describes the program that is running (server kind); the second part describes who is running that program (server instance). On a given network there might be many instances of a given kind of server advertising itself. A client can decide which one to connect to based on the second part of the server address, which should contain information allowing a client to do just that. (See the documentation of the -getserv- command on how to obtain instance information from a server.) Note that while the system variable zuser can be useful for the second part on some multiuser systems, this isn't portable, since on a single-user system there may not be a log-in name.

The keyword "local" means that you intend to connect with another program running on the same machine. To talk to another program on the same computer you don't have to use "local", because cT will find a server on your computer just as it will on some other networked host. However, "local" is much faster when that is really what is intended, because it is not necessary to search for a remote host.

An absolute address is contained in a marker. The specific content of the marker will depend on the network software, but in general the first character specifies the kind of address. An absolute Internet domain socket address consists of two parts: the workstation name and the socket number. An absolute address for the -server- command only needs to specify the socket number: the absolute address starts with "N" and is followed by the socket number (e.g., "N5000"). The workstation name does not have to be specified because it is the name of the machine the server is running on. Generally speaking, the only time an absolute address is needed is when a non-cT program is involved.

Macintosh: On Macintosh System 7 or on any Macintosh with the program-to-program (ptp) toolbox installed, you can use the ptp layer instead of the raw AppleTalk protocol. This means that a cT program running on a Macintosh without this toolbox (System 6, for example) can't talk to a cT program that is running on another machine that has the toolbox. The "local" argument is interpreted to mean to use a "ptp" connection on the same machine, whereas with the "ptp" argument cT will also try to make connections with other Macintoshes in the same AppleTalk zone. The "logical" or "absolute" arguments are interpreted to mean that the
program should use the raw AppleTalk protocol to connect between two different Macintoshes in the same AppleTalk zone (you cannot use this to connect two programs on the same machine).

At present, connections between cT programs on one or more Macintoshes are not validated. You don't have to prove who you are before the connection is established (connections are made as "Guest"). This means that "Program Linking" must be turned on (even for making connections between two programs on the same machine), and "Guest" must have access to program linking if a connection is made between two hosts.

You can stop being a server by executing -setfile FileDescriptor-, since a socket connection is similar to a file connection.

See the -socket- command for sample programs using the -server- command.

See Also:
- Overview of Sockets (p. 311)
- socketConnecting to Another Process (p. 313)
- getserv Asking about Servers (p. 316)

socket: Connecting to Another Process

The -socket- command is the command used to actually make the initial connection between two processes (one called a "server" and the other a "client"). It is essentially equivalent to -setfile-. The -socket- command has several forms, the first used by servers, the others by clients of those servers.

\[
\begin{align*}
\text{socket} & \quad \text{fd} \quad $$ used by a server to connect to a client \\
\text{socket} & \quad \text{fd; logical, "serverA", nn} \quad $$ connect to nnth server of type "serverA" \\
\text{socket} & \quad \text{fd; logical, "serverA"} \quad $$ choose first server of type "serverA" \\
\text{socket} & \quad \text{fd; local,"serverB"} \quad $$ connect to server on this machine \\
\text{socket} & \quad \text{fd; ptp, "serverC", nn} \quad $$ Macintosh; connect to nnth server \\
\text{socket} & \quad \text{fd; ptp, "serverC"} \quad $$ Macintosh; choose first server \\
\text{socket} & \quad \text{fd; absolute, "N5000"} \quad $$ mostly for Unix
\end{align*}
\]

In all cases zreturn is set to TRUE if the connection was successful and set to an error return if the connection was not successful. Currently the only error value is 4, similar to "file not found."

The first argument of the tag is a file variable that will be used for reading and writing data, using commands such as -datain-.

The server form (-socket fd- with no other arguments) will succeed (zreturn = TRUE) if at least one other process has attempted to connect to this server, in which case it will connect to the first process that attempted to connect. The command will fail (zreturn = 4) if a previous -server- command has not been executed by this program. When the connection is established the -server- advertising is removed so that no other processes will waste time attempting to connect.

The second argument of the tag specifies the address type and is the word "logical", "local", "ptp" (program-to-program on Macintosh), or "absolute". Programmers are strongly encouraged to avoid absolute addresses, because absolute addresses are machine, system software, and network software dependent.
The keyword "local" means that you intend to connect with another program running on the same machine. To talk to another program on the same computer you don't have to use "local", because cT will find a server on your computer just as it will on some other networked host. However, "local" is much faster when that is really what is intended, because it is not necessary to search for a remote host.

A logical address contains two parts. The first part is the kind of server, the second is a number that indicates which of the servers of that kind we wish to connect to. (Note that this is the same server-kind string or marker expression that is used in the -server- or -getserv- command.) If the number is omitted it is taken to be one. While you would typically execute a -getserv- command before executing a -socket- command, this is not necessary.

An absolute address is contained in a marker. The specific content of the marker will depend on the network software, but in general the first character specifies the kind of address. An absolute Internet domain socket address for the client -socket- command needs to specify the workstation name and a socket number. The first character for the absolute address is "I". The rest of the address is the workstation name, followed by a comma, followed by the socket number (e.g., "Iripley.andrew.cmu.edu,5000").

A very simple connection would be

```
socket fd;logical,"file_server" $$ try to connect to first
    $$ available server of type "file_server"
```

This connects to the first found server of kind "file_server".

Once a server and client are connected, either process can use another file variable and open another socket to some other process.

**Macintosh:** On Macintosh System 7 or on any Macintosh with the program-to-program (ptp) toolbox installed, you can use the ptp layer instead of the raw AppleTalk protocol. This means that a cT program running on a Macintosh without this toolbox (System 6, for example) can't talk to a cT program that is running on another machine that has the toolbox. The "local" argument is interpreted to mean to use a "ptp" connection on the same machine, whereas with the "ptp" argument cT will also try to make connections with other Macintoshes in the same AppleTalk zone. The "logical" or "absolute" arguments are interpreted to mean that the program should use the raw AppleTalk protocol to connect between two different Macintoshes in the same AppleTalk zone (you cannot use this to connect two programs on the same machine).

At present, connections between cT programs on one or more Macintoshes are not validated. You don't have to prove who you are before the connection is established (connections are made as "Guest"). This means that "Program Linking" must be turned on (even for making connections between two programs on the same machine), and "Guest" must have access to program linking if a connection is made between two hosts.

**Examples:** The example given below shows how two cT programs can connect to each other on a Macintosh or Unix. An important use of sockets is to connect between a cT program that handles the graphical user interface and a program written in some other language running on the same machine. At present this is supported on Macintosh System 7 and on Unix. Sample C-language programs are provided in the "socket" programs distributed with cT. Interprocess communication on Windows has a very different structure, and cT supports Dynamic Data Exchange rather than sockets on Windows.

In addition to the "chat" example shown below, see the socket programs in the set of sample programs distributed with cT.
**Example:**

Here is a "chat" program that permits two people using networked computers to type messages back and forth to each other. This program is interesting because it will try to connect to another chat program as a client and if it fails it will become a server. So two copies of this program will be able to connect no matter in what order they are started. (See the -getserv- command for a more general way to find out what servers are available.)

If you would like to try this on a single Macintosh, use the Sharing Setup control panel to enable file sharing and program linking. In the -socket- command change "logical" to "local". Change the tag of the -server- command to fd; local, "chatter" (that is, change "logical" to "local" and delete the "myname" argument). Make a binary and double-click that binary to run it with the cT Executor. In cT Create, run the program. Now there are two cT programs running that can send messages to each other.

```cT
define group,connect:
   file: fd

unit SetupChat
   merge,connect:
   * try to connect to server
   at 5,5
   write connecting
   socket fd; logical, "chatter", 1 $$ connect to first available
   $$ "chatter" server
   if ~zreturn
      * no server, become server & wait
      * for someone to connect to us
      write No server available, we will be server
      server fd; logical, "chatter", "myname"
      loop
         socket fd $$ try to accept connection
         outloop zreturn $$ got connection
         write . $$ indicate waiting......
         pause 1 $$ wait a while (don't want
         $$ to use up the computer,
         $$ we're just waiting)
      endloop
   endif
   * At this point fd is a valid connection, either because
   * original socket connected, or because we became
   * a server and someone connected to us.
   jump Chat
   *
   unit Chat
   merge,connect:
   m: m1
   at 10,80
   write We are connected! Choose menu item to send a message.
   menu Send: DoMessage $$ set up menu to allow sending
   *
   * We just keep looking for input on socket,
   * and display it if it comes.
   loop
      datain fd;m1 $$ try to read socket
   * Note that zreturn = TRUE and zretinf = 0 if connection still good
```
* (socket is still open) but there is nothing to read.

if zreturn & zretinf > 0
* got something
erase 10,80;zxmax,zymax
at 10,80;zxmax-10,zymax
show m1
endif
pause 1 $$ wait a bit not to consume the computer
endloop
*

unit DoMessage $$ menu unit for sending a message
merge,connect:
* Get a string to send (terminate with Return):
erase 5,5;zxmax-10,75
arrow 5,5;zxmax-10,75
ok
donarrow
dataout fd;zarrom $$ send the message
*

See Also:
Overview of Sockets (p. 311)
server Advertising a Server (p. 312)
getserv Asking about Servers (p. 316)
Sample Programs (p. 28)

getserv: Asking about Servers

The -getserv- command allows a program to find out what programs have executed -server- commands to advertise themselves:

```
getserv server_kind $$ single argument; set zretinf to the
$$ number of servers of this kind

getserv server_kind, nn; m1 $$ full form; get information on
$$ the nnth server of this kind
```

Here server_kind is a string or marker expression (e.g. "test-server"). The number nn indicates that we want information on the nnth server of this kind, and marker m1 will be filled with the string the particular server gave as the second part of the address in its -server- command (server instance). zreturn is TRUE if there is an nnth host of this kind or 4 if not.

When there are several servers of the same kind on the network how do you decide which one to try to connect to? If you execute the single-argument form of the command,

```
getserv server_kind
```
you are told (1) via zreturn, whether there are any servers of this kind, and (2) via zretinf how many servers there are. You can then execute several full -getserv- commands to get the particular server information for each one, in order to decide which one to connect to.

The single-argument -getserv- command asks the outside world for information about available servers, which may be quite slow because of the query to the network or other parts of the computer. This query is also made
on other -getserv- commands if you have not executed the single-argument form. Once the query has been made, subsequent full-form -getserv- commands will be fast. However, programmers should beware of letting their server information get stale. Executing another single-argument -getserv- command will update the information. There is no guarantee that the server information will be in the same order as before.

Example:

This subroutine uses the single-argument form of the -getserv- command to find out what servers are available, makes the connection to a server, and returns the connected file descriptor, as well as a success flag indicating whether the connection was made successfully.

```
unit ConnectClient(serverKind; fd,success)
m: serverKind $$$ has the server kind
i: success $$ TRUE if successfully connected
file: fd $$ will have connected client
m: m1 $$ will contain server information
i: jj, nservers

calc success := FALSE
geterv serverKind $$ find out what servers are
          $$ available on the network
outunit ~zreturn $$ no such servers are available

calc nservers := zretinf
if nservers = 1
  * connect to the only server there is:
socket fd; logical, serverKind, 1
calc success := TRUE
outunit
endif
write There are <|s,nservers|> servers of that kind.
      Please choose one by typing X, or type anything else to see more.
loop jj := 1,nservers
          geterv serverKind,jj;m1
at 10,jj*20
show m1
pause
if zkey = zk(X)
      socket fd; logical, serverKind, jj
      calc success := TRUE
      outunit
endif
endloop
write You didn't choose any!
```
9. System Variables

Overview of System Variables

While a program is executing, the computer keeps track of a lot of information about the user and about the general environment. The "system variables" make this information available to the author.

Associated with screen graphics:
- zxmin, zxmax, zymin, zymax
- zwherex, zwherey, zwidth, zheight, zwidth(), zheight()
- zcolorf, zcolorb, zdefaultf, zdefaultb, zwcolor, zncolors
- zxpixels, zypixels
- zmode
- zreshape
- zvt ime, zvwidth, zvheight, zvlength, zvplaying (for video)
- zforeground

Associated with mouse interactions:
- ztouchx, ztouchy, zleftdown, zrightdown, zdblclick
- zgtouchx, zgtouchy, zrtouchx, zrtouchy

Error information:
- zreturn, zretinf

Associated with the -edit- command:
- zedit, zeditkey, zeditx, zedity, ztextat
- zedit sel, zeditvis, zedittext, zhot sel, zhotinfo

Associated with the -button- and -slider- commands:
- zvalue, zslider, zbutton

Length of arrays, strings, and files:
- zlength

Associated with the -arrow- command:
- zarrown, zarrowsel, zcaps, zentire, zextra, zjcount, zjudged
- znentries, zopcnt, zorder, zspell, zvarcnt, zwcount, zanscnt

Miscellaneous:
- zkey, zdevice
- zclock, ztime, zdate, zday
- zfilepath, zfilename, zcurrentdir, zhomedir, zuser
- zcurrentu, zmainu, zfromprog
- zclipboard
- zmachine

One of the most important system variables is **zreturn**. It reports whether or not a command was successful. Not all commands provide **zreturn**; if a -draw- command fails, the whole system has failed and there is no point in issuing a **zreturn** value. However, if -compute- fails, there is some fault either with the program or with the user, and the **zreturn** value can be used to provide feedback about the type of failure. A successful **zreturn** is always TRUE (-1). The specific error values for **zreturn** are discussed with the individual commands and are summarized in the **zreturn** documentation.
In addition, the -sysinfo- command gives you a way to find out about some specific features of the particular computer the program is running on, such as default font size, and size and contents of the system-defined color palette.

See Also:

sysinfo Get System Information (p. 319)

**sysinfo: Get System Information**

The -sysinfo- command provides detailed information about the particular computer on which your program is running:

```plaintext
sysinfo default newline, variable
Sets the indicated (integer or floating-point) variable to the newline height of the default application font of this computer. This gives an indication of what the user considers a readable font on this computer/display.

sysinfo palette size, variable
Returns the (hardware) palette size. This value is 0 on a true-color display.

sysinfo default colors, variable
Returns the number of colors the system or hardware has already set up, out of the total number of colors available.

sysinfo foreground, red, green, blue
Returns the RGB definition of the current foreground color.

sysinfo background, red, green, blue
Returns the RGB definition of the current background color.
```

More than one keyword/variable set may appear in one sysinfo command, separated by semicolons:

```plaintext
define integer: palsize, dcolor
...
sysinfo palette size, palsize; default colors, dcolor
```

**Example:**

```plaintext
unit xsysinfo
i: lineheight, palsize, dcolor
i: rfore, gfore, bfore
i: rback, gback, bback
i: ii, jj, imax
sysinfo default newline, lineheight
sysinfo palette size, palsize
sysinfo default colors, dcolor
sysinfo foreground, rfore, gfore, bfore
sysinfo background, rback, gback, bback
at 10,10
write The default newline height is $<s, lineheight>$.
There are $<s, palsize>$ palette slots.
The system uses $<s, dcolor>$ slots for controls, title bars, etc.
Foreground RGB: $<s, rfore>$, $<s, gfore>$, $<s, bfore>$.
```
SYSTEM VARIABLES

Background RGB: <|s.rback|>, <|s.gback|>, <|s.bback|>.

outunit palsize <= 0

calc imax := sqrt(palsize)-1

loop ii := 0, imax
    loop jj := 0, imax
        outloop ii+(imax+1)*jj >= palsize
        rorigin 10+10ii,100+10jj
        color ii+(imax+1)*jj
        rfill 0,0; 8,8
    endloop
endloop

See Also:
zmachine Current Machine (p. 337)

System Literals and Constants

TRUE A "true" logical expression evaluates to -1.
FALSE A "false" logical expression evaluates to 0.

PI = 3.14159.....
DEG = 2PI/360 (the number of radians in one degree)

zred, zyellow, zgreen, zcyan, zblue, zmagenta: standard palette slots
System Variables for Graphics and Mouse

Current Screen Size

The system variables `zxmin`, `zymin`, `zxmax`, and `zymax` give the maximum and minimum values of x and y that are visible on the display (if it is large enough). They give the current active display size.

When there is no `-fine-` command, `zxmin = zymin = 0`. The maximum values can be found from `zwidth` and `zheight`: `zxmax = zwidth - 1` and `zymax = zheight - 1`.

When `-fine-` is active, `zxmin` and `zymin` are equal to the minimum values given by the `-fine-`. The maximum values, `zxmax` and `zymax`, are one less than the x and y in the `-fine-`.

Examples:

Try this example with different window sizes. Notice that when the `-fine-` is active, the right end of the long box is always lost, because its right end is at 450, but the active screen defined by the `-fine-` only extends to 300.

The blank-tag `-box-` draws a box around the active screen area.

```plaintext
unit zxmax
do zxmaxdraw(1) $$ no -fine-
pause $$ wait for keypress
erase $$ erase entire display
fine 400,300 $$ -fine-
do zxmaxdraw(2) $$ version 2
pause $$ wait for keypress
erase $$ erase entire display
fine 50,0; 400,300 $$ -fine-
resize -1,-1,0,0 $$ and -rescale-
do zxmaxdraw(3) $$ version 3
*
unit zxmaxdraw(temp)
i: temp
box $$ box active display area
box 5,150; 300,300; -20 $$ thick box
box 50,200;450,250; -5 $$ long box
at 50,10
write \temp\First \Second \Third \\nwrite version

zxmax,zymax = <s,zxmax>, <s,zymax>
zxmin,zymin = <s,zxmin>, <s,zymin>
zwidth = <s,zwidth>
zheight = <s,zheight>
```

Actual Screen Size

The system variables `zwidth` and `zheight` give the actual display width and height in pixels.
The variables *zwidth* and *zheight* may be used to send an explicit message if the user's window is an inappropriate size. These variables are also used when the display depends on the actual size and shape of the space available. Such displays are much harder to manage than ones that use -fine- and -rescale-!

The system variables *zxpixels* and *zypixels* indicate the actual number of screen dots ("pixels") inside the region defined by the -fine- command (and framed by a blank-tag -box- command). If no -rescale- command has been executed, *zxpixels* and *zypixels* are the same as the bounds specified by the -fine- command, unless the -fine- region is larger than the window, in which case they are the number of available pixels inside the window.

Note the contrast with *zwidth* and *zheight*. They indicate the number of pixels in the entire window, which may be a larger display area than determined by the -fine- and -rescale- commands.

The uses of *zxpixels* and *zypixels* are rather technical. They can be used to calculate a correspondence between the (rescaled) coordinates of an -at- or -draw- and the actual screen pixels. For example, icons are not rescaled, and it may be necessary to calculate their size in terms of rescaled coordinates to know whether a mouse click is within the border of the icon. Suppose you have designed an icon that is 20 by 20 pixels. The corresponding icon size in rescaled coordinates can be calculated in this way:

```
fine 500,300
rescale TRUE,TRUE,FALSE,TRUE
.....
calc iconx := 20(500/zxpixels)
cony := 20(300/zypixels)
```

*Example:*
This example shows how to check for a window that is "too small." Each time the window is reshaped, the current main unit (xzwidth) is reexecuted, and the size is checked by -do checker-. If the size is too small, a message is displayed. In a typical program, unit checker would be specified as the -imain- unit.

```
unit xzwidth
do checker
at 50,50
write This is unit "xzwidth."
*
unit checker
f: needwide = 500 $S requires 500 wide
f: needtall = 200 $S requires 200 tall
loop zwidth < needwide | zheight < needtall
       erase .1 $S optional, these two lines
       pause .1 $S make a blink onkeypress
       at zwidth/4, zheight/4
       if zwidth<needwide & zheight<needtall
          write Window is too small!
       endif
       elseif zwidth < needwide
          write Window is too narrow!
       else
          write Window is too short!
       endif
       pause $S wait for window change
endloop
```

See Also:
- *imain* Modifying Every Unit (p. 232)
- *rescale* Adjusting the Display (p. 37)
Current Screen Position

The system variables for the current screen position give the point where the next display will start:

\[ \text{zwherex, zwherey} \]

Often the "current screen position" is not explicitly considered. Usually we say, "The -at- command sets the position for the beginning of a -write- statement." A more precise statement would be, "The -at- command sets the current screen position; the -write- statement begins its display at the current screen position." For example, the position for the center of a circle is usually set by an -at- command, but the circle can just as easily be centered around the end of a line. The lines below put the center of the circle at 300,100.

```
draw 100,100; 300,100
circle 50
```

Every command that modifies the screen display leaves the current screen position at a predictable location. The positions selected are described in the individual command descriptions.

The variables \text{zwherex} and \text{zwherey} are not updated until a command is finished. In the example below, the -circle- leaves the current screen position at the center of the circle. The -vector- command does not modify the screen position until after it is all finished, so \text{inside} the vector command, \text{zwherex}, \text{zwherey} still points to the center of the circle.

Example:

```
unit x
i: x, y
at 50,50; zxmax-50,140
write The screen position at the end of a -write- statement
is the position where the next character would have been printed.
circle 7 $$ center circle at end of -write-
vector 65,145; zwherex,zwherey; 0.2
at 60,150; zwidth-40, 300
write The little circle is centered on
the (zwherex, zwherey) at the
end of the first -write- statement.
*
unit x
i: x,y
at 100,100 $$ center for -circle-
circle 75, 180, 330 $$ arc from 180 to 330 degrees
calc x := zwherex
y := zwherey
vector 100,100; x-4,y+4
at 100,100 $$ uses embedded -show-s:
write The end of the arc
is at \langle s,x \rangle, \langle s,y \rangle.
*```

```
**SYSTEM VARIABLES**

### zmode: Current Mode

The system variable `zmode` tells what the current graphics mode is. An important use is in a general subroutine that must alter the mode and then restore it to its previous value.

```
zmode =
1   mode write  
2   mode rewrite  
3   mode erase  
4   mode inverse  
5   mode xor  
```

**Example:**

```plaintext
unit xzmode1
mode rewrite
fill 50,10; 150,100
at 10,15
write This is in mode rewrite,
do xzmode2(75,55)
at 110,80
write and so is this.
*
unit xzmode2(x,y)
i: x, y, savemode
calc savemode := zmode $$ save current mode
mode erase
at x,y
disk 10
mode \savemode-2 \write \rewrite $$ restore original mode \erase \inverse \xor
```

*See Also:*  
mode Changing Modes (p. 60)

### zreshape: Screen Reshape Status

The system variable `zreshape` is TRUE if the current main unit was started due to the user reshaping the window. It is FALSE if the current main unit was entered normally (-next-, -back-, -jump-).

The -jump- command allows passing of arguments to main units, though this is not advised. If the display is reshaped, execution starts again, but the arguments are not passed again. Numeric local variables but not other kinds of local variables retain the values they had just before the reshape.

**Example:**

In this example, the house is drawn only if the screen is reshaped.

```plaintext
unit xzreshape $$ use Run from Selected Unit
next xzreshape
do \zreshape\house\x
at 91,110
write The House
That Jack Built
```
Mouse Status

The x,y position of the mouse when a click is entered is given by three sets of system variables, one for each of the coordinate systems:

\[ \text{ztouchx, ztouchy} \quad \$\$ \text{absolute} \]
\[ \text{zgtouchx, zgtouchy} \quad \$\$ \text{graphing} \]
\[ \text{zrtouchx, zrtouchy} \quad \$\$ \text{relative} \]

These variables are all zero unless the most recent user input was a mouse click.

It is also possible to determine the current mouse coordinates, independent of whether the most recent input was from the mouse:

\[ \text{zmousex, zmousey} \quad \$\$ \text{current mouse position} \]

Special note: the current mouse position given by zmousex/zmousey may be outside the cT window due to dragging, and the reported position is affected by -fine- and -rescale-.

The system variables \text{zleftdown} and \text{zrightdown} are TRUE if the left or right button is currently held down, independent of the current value of \text{zkey} (which refers to the most recent -pause- or -getkey- event). For example, in a unit driven by a -slider-, if zleftdown is TRUE you might choose to do nothing, waiting for the mouse button to be released before action is taken.

If the time between two down events is less than the time given by system variable \text{zdblclick}, and the two clicks are near each other on the screen (within two pixels, say), you may want to consider the second click to be a "double click." Some computer systems allow the user to specify the double-click time, in which case \text{zdblclick} is set to the user-specified time. You can identify double-click events by keeping track of the time between down events (using \text{zclock}).

Example:

\begin{verbatim}
unit   xmousestatus
at     10,10
write  I will make an O wherever you click
       with the left mouse button.
loop
   .    pause   keys=touch
   .    at      ztouchx,ztouchy
   .    circle    5
endloop
\end{verbatim}
SYSTEM VARIABLES

See Also:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>pause</td>
<td>Mouse Inputs</td>
<td>126</td>
</tr>
<tr>
<td>enable</td>
<td>Allowing Mouse Input</td>
<td>132</td>
</tr>
<tr>
<td>zkey</td>
<td>Last User Input</td>
<td>333</td>
</tr>
</tbody>
</table>

Color Status

Here is a summary of system variables associated with color:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zcolorf</td>
<td>current foreground color</td>
</tr>
<tr>
<td>zcolorb</td>
<td>current background color</td>
</tr>
<tr>
<td>zdefaultf</td>
<td>default foreground color on this machine</td>
</tr>
<tr>
<td>zdefaultb</td>
<td>default background color on this machine</td>
</tr>
<tr>
<td>zwcolor</td>
<td>current window color (set by -wcolor-)</td>
</tr>
<tr>
<td>zncolors</td>
<td>number of available colors</td>
</tr>
</tbody>
</table>

For additional color status information, see the -sysinfo- command.

Example:

```
unit xcolorstatus
color zred,zyellow
mode rewrite
at 10,10
write Red on yellow
do xdefaultcolors
at 10,40
write More red on yellow, rewrite
*
unit xdefaultcolors
i: savemode, savef, saveb
calc savemode := zmode
savef := zcolorf
saveb := zcolorb
color zdefaultf,zdefaultb
mode inverse
at 150,10
write Default colors, inverse
*
 restore colors and mode:
color savef,saveb
mode \savemode-2\write \rewrite \erase \inverse \xor
*
```

See Also:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>color Introduction</td>
<td>Color Graphics &amp; Text</td>
<td>77</td>
</tr>
<tr>
<td>sysinfo</td>
<td>Get System Information</td>
<td>319</td>
</tr>
<tr>
<td>color</td>
<td>Color Graphics &amp; Text</td>
<td>77</td>
</tr>
<tr>
<td>wcolor</td>
<td>The Window Color</td>
<td>79</td>
</tr>
<tr>
<td>palette</td>
<td>Creating a Color Palette</td>
<td>81</td>
</tr>
<tr>
<td>zncolors</td>
<td>The Number of Available Colors</td>
<td>86</td>
</tr>
</tbody>
</table>
Video Status

After executing a -video- command, the system variable \texttt{zvlength} give the total duration in seconds, \texttt{zvwidth} gives the original width of the video image in pixels, and \texttt{zvheight} gives the original height of the video image in pixels. While playing a video sequence, the system variable \texttt{zvtime} gives the current time measured from the start of the movie. The system variable \texttt{zvplaying} is TRUE while a video clip (even if sound-only) is playing.

Example:

```c
unit xvideoinfo
video movie; "Some File"; 10,50; 100,200  $$ change file name!
at 10,10
if ~zreturn
  write Can't play this movie.
  zreturn = <|s,zreturn|>.
pause
jumpout
endif
vset rectangle, 10,50; 10+zvwidth-1,50+zvheight-1
write Video clip is <|s,zvlength|> seconds long.
vplay zvlength/3, 2zvlength/3  $$ play middle third of clip
loop zvtime <= zvlength/2  $$ watch for half-way point
  pause 0.1
endloop
write Movie time = <|s,zvlength/2|> seconds.
pause
```

See Also:

- \texttt{video Initialize Video} (p. 117)
- \texttt{vplay Play a Video Sequence} (p. 120)

zforeground: Window Forward

The system variable \texttt{zforeground} is TRUE if the execution window is the active window, fully visible, in front of all other windows. File selection dialog boxes, and -dialog- boxes, are not created if the window is not the active, forward-most window.

Generic Font Names

Because different systems assign different names to their standard fonts, cT has several generic font family names. These names are recognized on all machines:

- \texttt{zserif} - a standard font with serifs (Times, except for New York on Macintosh)
- \texttt{zsans} - a standard sans-serif font (Helvetica, except for Geneva on Macintosh)
- \texttt{zfixed} - a standard fixed-width font (Courier, except for Monaco on Macintosh)
- \texttt{zsymbol} - a math symbol font (Symbol on all platforms)

In addition, there are generic patterns, icons, and cursors:

- \texttt{zpatterns} - gray scales & some figures
**SYSTEM VARIABLES**

- `z cursors` - some cursors
- `z icons` - some icons

The generic names can be used anywhere that a font name can be used:

```
font zsans,16
icons zicons
pattern zpatterns, zk(a)
icons zserif+"12"
pattern zsans+"16"
```

The `-font-` command specifies a family name, so the format of the tag is "familyname", followed by a comma, then the size. The scaling of the font is determined appropriately for whatever machine the program is running on. The `-icons-` and `-pattern-` command refer to explicit font sets and *not* to families. If a font set is used with `-icons-` or `-pattern-`, the format is "familynamesize" with no commas or spaces.

If no font is specified by the program, the default font used is "zserif,15". The scaling of the font will be determined appropriately for whatever machine the program is running on.

In the future additional patterns, cursors, and icons may be added to these sets. If there is no `-icons-` command, the icon set is the same as the default font.

*See Also:*

- `font` Selecting a Typeface (p. 43)
- `icons` Selecting an Icon (p. 93)
- `pattern` Making Textured Areas (p. 61)
- `rescale` Adjusting the Display (p. 37)
Other System Variables

zfilename: Getting File Name

You can use `zfilename(file descriptor)` to get the main part of a file name associated with that file descriptor by `setfile` or `setfile-`, and `zfilepath(file descriptor)` gives the rest. For example:

```
setfile fd; zempty; ro $$ suppose file is /abc/def/data1
show zfilepath(fd)+zfilename(fd) $$ `/abc/def/` + `"data1"
```

See Also:
addfile Create a File (p. 289)
setfile Select a File (p. 291)
setdir Select a Directory (p. 294)

User ID and Home Directory

There are two system marker variables that give the user's home directory and the user's ID. These are useful for keeping records and for use when a file-name must be constructed.

```
zuser the user's ID
zhomedir the user's home directory
```

If the user's home directory is `/cmu/cdec/zz9z`, then `zhomedir` contains the full path name for that directory `/cmu/cdec/zz9z` and `zuser` contains `"zz9z"`.

Examples:

```
unit xzuser $$ show ID & directory
at 50,50
write User's ID is `<s,zuser>`.  
at 50,75
write User's home directory is `<s,zhomedir>`.  
```

The two examples below (.1a and .1b) both create a file in the user's subdirectory "mysub". The first generates the file name with a `-calc-` statement and then uses that file name in the `-addfile-` command. The second composes the file name directly, inside the `-addfile-` command.

It is very important to check `zreturn` after every file operation. "Permission denied" can mean that the user does not have write access to the specified (sub)directory or folder, or that the (sub)directory or folder does not exist. You probably don't have a subdirectory named "mysub".

Note that these examples actually create files in your directory or folder. If you run the example successfully and then try it again, you will get "duplicate file name."

```
unit xzhomedir1a $$ create a file
file: fd
m: newfile $$ store new file name in a marker
at 50,50
write Generated newfile is:
```

*
SYSTEM VARIABLES

```plaintext
<|s,newfile|>.
addfile fd; newfile
do xzhomedir2 $$ zreturn feedback
*
unit xzhomedir1b $$ create a file
  file: fd
at 50,50 $$ use composed file name directly:
write Adding file:
  <|s,zhomedir+/mysub/helptestfile2|>
addfile fd; zhomedir+/mysub/helptestfile2
* 
unit xzhomedir2 $$ zreturn results
at 50,120 
if zreturn
  write -addfile- was successful
else 
case zreturn
  7 
    write duplicate file name
  11 
    write permission denied
endcase
endif
```

See Also:
File Name Specification (p. 286)

Judging System Variables

The system variables listed below are set when a user's response is judged at an -arrow-. The user's input is available as the system marker variable `zarrowm`, and any portion of the input that has been selected with the mouse is available as the system marker variable `zarrowsel`.

These variables are TRUE (-1) if no errors were found, FALSE (0) if the condition was not satisfied:

- `zcaps`: no capitalization errors
- `zentire`: every required word is present
- `zextra`: no extra words in the response
- `zorder`: word order is correct
- `zspell`: no misspelled words

This variable, `zjudged`, gives the current status of the judgment of the user's response:

- `-1`: response judged "ok"
- `0`: response judged "wrong"
- `1`: response judged "no"
- `2`: no judgment yet

These variables give relevant counts:

- `zanscnt`: # of response-command that was matched
OTHER SYSTEM VARIABLES

zntries  # attempts made at the current -arrow-
zjcount  # characters in the response
zwcount  # words in the response
zopcnt   # of operations (+-*/) in the response
zvarcnt  # number of defined variables in response

Examples:

The system variable zntries gives the number of attempts the user has made at the current arrow. In this example, if the second try is not correct, the answer is given and the program continues.

unit xzntries
at 50,50
write How many years are in a century?
arrow 50,100
ansv 100.
write Very Good!
if zntries=2
write There are 100 years in a century.
judge quit $$ force past the endarrow
endif
endarrow
at 50,200
write after the <endarrow-

The system variable zopcnt gives the number of arithmetic operations in the user's response at an arrow. In this example, if the user responds "3+7", the -ansv- will be matched, but zopcnt is 1, so the message will be given and the response marked "no".

unit xzopcnt
at 50,50
write What is 3 x 7 ?
arrow 100,100
ansv 21
if zopcnt != 0
write You must not use any arithmetic operators.
See Also:
judge Changing the Judgment (p. 182)
Basic Judging Commands (p. 159)
zarrowm Markers at an -arrow- (p. 256)
zarrowsel Selected Text at an -arrow- (p. 258)

zreturn: The Status Variable

For certain commands, the system variable zreturn reports whether or not a command was successful. If the command was successful, zreturn is TRUE (-1). If a command failed, zreturn gives a diagnostic number telling how the command failed.

A zreturn value is provided for commands that handle numerical input from the user and for file operations. The interpretation of zreturn values for numerical input from the user -compute-, -ansv-, -wrongv-) is covered in the example below.

A detailed discussion of zreturn for file operations -addfile-, -setfile-, -delfile-, -datain-, -dataout-, -reset-, -xin- and -xout-) is with the file operations discussion. Here is a summary of the values:

-1 all okay
2 illegal file variable (should not happen)
3 file not open (no preceding setfile or addfile)
4 file not found
5 file improper type
6 file code-word error
7 duplicate file name (file already exists)
8 file quota exceeded
9 catchall error
10 file directory full
11 permission denied
12 file in use (cannot be deleted)
13 directory not empty (directory cannot be deleted)
14 -dataout- out of range (not at end of file)
15 file is currently reserved
16 illegal character read with -datain-
17 user canceled making a selection in a file dialog box
18 window not wide enough for a file dialog box
19 not enough memory to create a file dialog box
20 file operation not supported (e.g. QuickTime not installed)
21 trying to bring up file dialog box in background window

The file operation zreturn is complemented by the system variable zretinf (return information).

Example:

unit xComputeZreturn
f: value
arrow 50,50
compute  value
ok     zreturn  $$ -1 is TRUE
no

door

endarrow
*

unit  errors  $$ tell what is wrong
write \zreturn

\-1  valid expression, everything okay
\0  zopcnt > 0 with -specs noops- in effect
\1  illegal character
\2  decimal point error (such as 3...5)
\3  currently unused
\4  currently unused
\5  error in form (such as 34+)
\6  missing left parenthesis
\7  unrecognized variable name
\8  illegal function argument (such as sqrt -1))
\9  missing right parenthesis
\10  zvarcnt > 0 with -specs novars- in effect
\11  invalid assignment (=)

*

More detailed comments about some of the \texttt{zreturn} values for numerical input:

\texttt{zreturn = 0}: The user entered an expression such as "2+5*8", but the author had specified that no arithmetic operations -specs noops-) were allowed. The only legal response is a single number or variable, but one can say -3 or sin(0.3).

\texttt{zreturn = 1}: The expression includes a character that is illegal in an arithmetic statement, such as \ or \%.

\texttt{zreturn = 10}: The user entered an expression that includes a variable, but the author had specified that no variables -specs novars-) were allowed. The only legal response includes only explicit numbers or functions of numbers, such as "5 + sin(45)".

\texttt{zreturn = 11}: Unless the author explicitly specifies that assignments may be included -specs okassign-), the user cannot use the symbol "\==" in his or her response.

See Also:
  File I/O Errors (p. 287)
  \texttt{zretinf} Number of Elements Read (p. 306)
  \texttt{specs} Specifying Special Options (p. 169)

\textbf{zkey: Last User Input}

The system variable \texttt{zkey} gives the last user input, from the keyset or from the mouse.

Although you can use or display the value of \texttt{zkey} (as shown in the Example), \texttt{zkey} is usually compared with a \texttt{zk} function, as in

\begin{verbatim}
write \zkey=zk(Q)
\end{verbatim}

You pressed a Q!
SYSTEM VARIABLES

Selecting a menu item that was created with a `-menu-` command does not change the value in \texttt{zkey}. Selecting a menu item that was automatically generated by cT does cause \texttt{zkey} to change. These menu items generate a "next" keypress, so that \texttt{zkey} = \texttt{zk(next)}:

(Next Page)
(Proceed)
(Enter Response)

The menu item "(Back)" generates a "back" keypress, so that \texttt{zkey} = \texttt{zk(back)}.

When a mouse input is received, the actual position is stored in \texttt{ztouchx} and \texttt{ztouchy}, while \texttt{zkey} = \texttt{zk(some-touch-input)}.

Keep in mind that \texttt{zkey} only holds one input. If the user touches (mouse clicks) a position and then presses ENTER to continue, the coordinates of the touch position are lost.

\textit{Example:}

\begin{verbatim}
unit xzkey at 50,50
write Press a key.
pause at 80,80
write The numeric value of the key you pressed is <|s, zkey|>.
*
\end{verbatim}

\textit{See Also:}
The Keyname Function: \texttt{zk()} (p. 204)
pause Mouse Inputs (p. 126)
Mouse Status (p. 325)
Moving between Main Units (p. 237)

\textbf{zdevice: Last Input Device}

The system variable \texttt{zdevice} identifies the source of the last user input. Currently, the only input devices are the keyset and the mouse.

\texttt{zdevice} = 0 \ $\$\$\ keyset input
\texttt{zdevice} = 1 \ $\$\$\ mouse input

\textit{Example:}

\begin{verbatim}
unit xzdevice at 50,50
write Press a key or click the left mouse button.
pause keys = touch,all
at 80,80
write xzkey = <|s, zkey|>
zdevice = <|s,zdevice|>
ztouchx,ztouchy = <|s,ztouchx|>, <|s,ztouchy|>
if zdevice = 1 \ $\$\ show mouse position
at ztouchx,ztouchy
\end{verbatim}
OTHER SYSTEM VARIABLES

See Also:
  zkey Last User Input (p. 333)
  The Keyname Function: zk() (p. 204)

zclock: Finding Time Spent

The system variable zclock is used to calculate elapsed time. It is measured in seconds and is accurate to about .01 seconds.

Example:

```
unit xzclock
f: start, finish
at 60,40
write What is 15 times 15 ?
calc start := zclock $$ save starting value
arrow 100,100
ansv 15*15
wrongv 15*15,5%
write You're within 5%.
endarrow
calc finish := zclock
at 60,150
write You took <| s, finish-start |> seconds.
```

Time and Date

The system marker variables ztime and zdate give the current time (hour:minute:second) and date (month/day/year).

ztime is "hour:minute:second"

zdate is "month/day/year" (January 1, 2000 will be 01/01/00)

The numeric system variable zday gives the current Julian date in total seconds measured from 00:00:00 Coordinated Universal Time, Jan. 1, 1970. However, since PCs and Macintoshes usually don't know what time zone they are in (unlike most Unix systems), the Julian date may be skewed by time-zone differences. Since one typically uses the difference between two Julian dates on the same computer, this skewing typically doesn't really matter.

zday gives the numeric Julian date

Example:

```
unit xtime
m: time, date
i: hour, minute, second
```
i: month, day, year

calc time := ztime \$\$ save current values
date := zdate
at 10,10
write The time is \$s, time\$. The date is \$s, date\$.
<\$s, zday/(365*24*60*60)\$> years have
gone by since 1/1/70.
do xClock(time; hour, minute, second)
at 10,100
write The time is \$s, hour+12*(hour > 12)\$:
write \$s, minute\$:
write \$s, second\$:
show second
write \$s, hour > 12 \ PM\ AM\$.
do xDate(date; month, day, year)
at 10,130
write The date is
write \$s, January\February\March\April\May\June\July\August\September\October\November\December\$:
write \$s, day\$, 19\$s, year\$.
*
unit xClock(hrminsec; hr, min, sec)
* From hrminsec in ztime format, extract numeric hr, min, sec.
m: hrminsec, colon
i: hr, min, sec
calc hr := znumeric(hrminsec)
colon := zsearch(hrminsec,"\:")
min := znumeric(zextent(colon,zend(hrminsec)))
colon := zsearch(zextent(znext(colon),zend(hrminsec)),"\:")
sec := znumeric(zextent(colon,zend(hrminsec)))
*
unit xDate(monthdayyear; month, day, year)
* From monthdayyear in zdate format, extract numeric month, day, year.
m: monthdayyear, slash
i: month, day, year
calc month := znumeric(monthdayyear)
slash := zsearch(monthdayyear,"/"")
day := znumeric(zextent(slash,zend(monthdayyear)))
slash := zsearch(zextent(znext(slash),zend(monthdayyear)),"/"")
year := znumeric(zextent(slash,zend(monthdayyear)))
*

**Unit Markers zcurrentu & zmainu**

The system variable \texttt{zcurrentu} is a marker containing the name of the unit that is being executed at this moment.

The system variable \texttt{zmainu} is a marker containing the name of the current main unit; that is, the first unit in the program, or a unit reached by -jump-, -next-, or -back-.
**zclipboard: Clipboard Contents**

The system variable `zclipboard` is a marker bracketing the text that is currently on the system clipboard.

```
show        zclipboard $$ show current contents of system clipboard
calc        zclipboard := "Hello there!" $$ put this text on clipboard
```

Assigning some text to `zclipboard` is essentially equivalent to doing a cut or copy operation.

**zmachine: Current Machine**

The system variable `zmachine` is a marker containing the name of the type of computer the program is currently running on, according to this list:

"macintosh"
PC running Windows: "windows"
Unix: "pmax" (DECStation 3100), "sparc", "sun3", "vax", "rt"

*See Also:*
- `sysinfo` Get System Information (p. 319)
Index

answer, 162
algebraic, 167
capitalization, 169
-contingent commands, 159, 162
-contingent erasing, 159, 177
markup, 169
modify defaults, 169
numerical, 163
optional words, 169
out-of-order words, 169
reserved words for status, 330
spelling, 169
unexpected, 164
word or phrase, 162
append, 259
arc
relative, 114
syntax for, 53
arccos(x), 202
arccot(x), 202
arccsc(x), 202
arcsec(x), 202
arcsin(x), 202
arctan(x), 202
area
erase, 58
fill, 56
move, 98
pattern, 61
arguments
of zk(x), 204
passing, 223
arithmetic operations, 200
array
assign values to, 210
length, 213
pass-by-address, 208
sort, 213
special indices, 208
using, 208
zero, 211
arrays, 208
arrow, 159
blank responses, 179
display of, 178
initializations, 173
modifying defaults, 169
number of tries, 330
selected text, 258
with marker variable, 256
aspect ratio, 37
assign, 200
INDEX

assignment
at an arrow, 169
symbol, 200
to a variable, 200
assignments, 200
asterisk
comments, 11
at, 34
current screen position, 34, 323
implicit after response, 159
with no margin, 35
with -text-, 39
atnm, 35
author variables, 198
axes, 104
labels, 106
lengths, 104
marks along, 106
not shown, 105
back, 239
background color, 77
backslash
in conditional commands, 18
with -text-, 39
bar graphs, 110
bar width, 110
bases
number, 191
beep, 122
binaries, 16
binary
constants, 191
display, 49
bit
count, 206
manipulations, 206
bitcnt(x), 206
blank lines, 11
blanks
allow/inhibit, 179
block, 212
bmp, 14
bold
after -answer-, 169
text, 39
boolean, 201
boolean operators, 201
bounds, 105
box, 55
braces
used with -answer-, 162
used with -calc-, 200
branching commands, 237
bugs in cT, 27
button, 142
buttonfont
inhibit, 69
byte, 191
bytes
file I/O, 301, 303
calc, 200
into a marker variable, 249
calculation
introduction, 187
calendar date, 335
call-by-address, 224
call-by-value, 224
calling arrays, 208
cancel, 154
arrow defaults, 169
back, 239
file descriptor, 291, 293
iarrow, 173
ijudge, 175
imain, 232
next, 237
capitalization at an arrow, 169
carriage return, 42
embedded in a string, 249
case, 216
character
add to string, 259
assign to a marker, 249
coordinates, 38
count at an -arrow-, 179, 330
creation of, 43, 46
sets, 43, 46
strings, 246
characters
add to string, 259
circle, 53
arcs of, 53
dashed, 53
relative, 114
circleb, 53
clear
file descriptor, 291
input buffer, 135
specs options, 169
click, 126
mouse, 126, 132, 325
clip, 63
closing a file, 291
clrkey, 135
coarse, 38
coarse grid, 38
code
from another file, 243
**SYSTEM VARIABLES**

- **color**, 77
- closest from existing palettes, 88
- fallback slots, 81
- get values, 86
- **introduction**, 71
- newpal, 81
- palette, 81
- **status**, 326
- window, 79
- **color images**, 91
- color menu, 13
- **colors**
  - number of, 86
- **columns of numbers**, 48
- **combin(x,y)**, 203
- combining logical expressions, 201
- **command**
  - conditional, 18
  - graphing, 102
  - **indent**, 39, 162, 181, 195, 215, 216
  - relative coordinates, 112
  - response handling, 124
  - syntax, 200, 201
- **commands**
  - animation, 98
  - **calculating**, 187
  - **drawing**, 52
  - **graphics**, 32
  - **looping**, 215
  - modify defaults, 169
  - mouse input, 125
  - random, 221
  - screen description, 34
  - sequencing, 237
  - single key input, 125
  - **text**, 39
- comment, 11
- **comp(x)**, 206
- compare
  - markers, 252
- complement, 206
- compute, 165
  - computing with marker variables, 254
  - on a string, 254
  - with one tag, 165
  - with two tags, 254
  - zreturn values, 332
- conditional commands, 18
- constant, 191
- **constants**
  - floating-point, 191
  - integer, 191
- continued lines, 52
- convert
  - integer to string, 265
  - string to integer, 265
- coordinates
  - absolute, 34
  - character, 38
  - coarse, 38
  - **fine**, 35
  - graphing, 102
  - log scales, 108
  - mouse, 325
  - polar, 109
  - relative, 112
  - scaled, 37
  - screen, 34
  - touch, 325
- copy
  - string to a new string, 266
- **cos(x)**, 202
- **cosh(x)**, 202
- **cot(x)**, 202
- CR, 50
- **csc(x)**, 202
- current screen position, 323
- current screen size, 321
- cursor, 62
  - editing, 159
  - mouse, 62
- data
  - about mouse position, 325
  - about user input, 330
  - input/output, 285, 289
  - numerical input errors, 332
  - read from file, 294, 297
  - write to file, 298, 301
- datain, 294
  - marker variables, 254
  - **text**, 254
- dataout, 298
- datastream, 23
- date, 335
- debugging, 20
- defaults
  - arrow, 169
  - erase, 66
  - **graphics**, 32
  - graphing, 103
  - judging, 169
  - main unit, 232
  - margins, 34
  - modify, 65, 169, 177
  - modifying unit, 232
  - screen erase, 232
- define, 193
global variables, 193
groups, 193
local variables, 195
markers, 248
student:, 198
summary, 190
user:, 198
DEG, 191, 202
dergee, 202
inhibit, 70
delete file, 293
delfile, 293
delimiter
condenotional, 18
text, 39
delta, 110
device sending input, 334
dialog, 144
dialog box, 144
diff (of bits), 206
differences from other languages, 17
dimensions
screen, 35, 321
directory
select, 294
disable, 132
disk, 57
display
answer markup, 169
arrow, 178
binary, 49
characters, 94
clipping, 63
columns, 48
hexadecimal, 49
inhibit, 67
inhibit first display, 66
marker contents, 250
octal, 49
reshaping, 37
size, 113, 321, 324
text, 39, 41
variables, 47, 50
divide
floating, 200
integer, 200
divr, 200
divt, 200
do, 224
-eraseu- unit, 175
-iarrow- unit, 173
-ijudge- unit, 175
-imain- unit, 232
-menu- unit, 136
passing markers, 253
dollar signs, 11
dot, 53
drag, 126
drag (mouse), 126
draw, 52
continued lines, 52
-inhibit startdraw-, 66
drawing commands, 52
dynamic arrays, 212edit, 149edit files, 155edit menu, 11edit panel, 149editdraw
inhibit, 68
eclipse, 114
eelse, 215
eelseif, 215
embed, 50
embedded variables, 50
empty a file, 306
enable, 132
endarrow, 159
endcase, 216
endif, 215
endloop, 218
end-of-line comments, 11
ENTER, 237
equal to, 201
equality
almost equal, 201
logical, 201
equation judging, 167
erase, 58
default, 232
inhibit, 66
mode, 60
eraseu, 175
error returns
file operations, 287, 306
numerical input, 332
evaluate
algebraic responses, 167
numerical responses, 163, 165
word responses, 162
exact, 181
exactw, 181
example program, 5
exclusive or (of bits), 206
execute, 241
executing a new program, 241
execution
of a program, 241

341
quitting, 237
exit
  from a program, 237
  from a unit, 234
exp(x), 203
exponentiation, 200
ext, 204
external device, 126, 204, 334
factorial(x), 203
fallback color slots, 81
FALSE, 191, 201, 203
file
  changing position within, 306
  close, 291
  create, 289
  delete, 293
descriptor definition, 191
extension, 241, 286, 289
I/O examples, 308
introduction, 285
library, 243
names, 286
read a line, 297
read bytes from, 301
read data from, 294, 297
select directory, 294
set to, 291
write bytes to, 303
write data to, 298, 301
zreturn value, 332
file editor, 155
file I/O, 285
  with string variables, 254
file menu, 11
file name, 329
file names, 286
file path, 329
fill, 56
  area, 56
  circle, 57
  pattern, 61
fine, 35
fixed format variable display, 48
flag
  changed marker, 264, 266
flags
  modify defaults, 65, 177
float, 191
floating point
  constants, 191
  range of, 191
focus, 153
folder
  select, 294
font, 43
  generic names, 327
  typewriter, 327
font menu, 13
fontp, 46
for loops, 218
forcing a keypress, 133
foreground color, 77
frac(x), 203
frames, 55
frequency of sound, 122
frequency of tone, 122
function keys, 186
functions, 200
  arrays, 208
  combin, 203
  factorial, 203
  gamma, 203
  hyperbolic, 202
  keynames, 204
  markers, 264
  math, 203
  modulo, 203
  trig, 202
  zk(x), 204
fuzzy zero, 201
  inhibit, 71
fuzzyeq
  inhibit, 71
gamma(x), 203
garrow, 102
gat, 102
gatnm, 102
gbox, 102
gbutton, 102, 142
gcircle, 102
gcircleb, 102
gclip, 63
gdisk, 102
gdot, 102
gdraw, 102
gedit, 102, 149
g erase, 102
generic font names, 327
gset, 89, 91
getserv, 316
gfill, 102
gget, 102
global variables, 193
  with local variables, 196
gmove, 102
INDEX

gorigin, 103
gput, 102
graph
  mouse coordinates, 325
graphics
  defaults, 32
  dot pictures, 53, 93
  drawings, 52
  fast updates, 68
  graphs, 102
  icons, 94
  introduction, 32
  moving of, 98
  position, 34, 35, 323
  system variables, 321
  window size, 34
graphics editing, 10
graphing
  axes, 104, 105
  bar width, 110
  bars
    vertical & horizontal, 110
  commands, 102
  defaults, 103
  histograms, 110
  labels, 106
  labels for log scales, 108
  log scales, 108
  origin, 103
  polar coordinates, 109
  scales, 105
  semi-log scales, 108
  tick marks, 106
graphing commands, 102
gray areas, 61
greater than, 201
grid
  coarse, 38
  fine, 35
group, 193
gslider, 102, 146
gtext, 102
gvector, 102
halt program, 241
hbar, 110
height
  of video, 327
hexadecimal
  constants, 191
  display, 49
hidden units, 11
histograms, 110
home directory, 329
horizontal bars, 110
hot text, 153
hsv, 84
hue, 84
hyperbolic functions, 202
hypertext, 153
iarrow, 173
Icon Maker, 95
icon.t, 96
icons, 93
  Macintosh, 95
  PC & Unix, 96
ID of user, 329
IEU, 227
if, 215
ifmatch, 181
ignorable words, 162
ijudge, 175
image height, 91
image width, 91
images, 88
images & files, 91
imain, 232
indenting, 11
index
  calculations, 187
inhibit, 65, 177
  answer erasing, 177
  answer markup, 169
  arrow display, 178
  blank responses, 179
  buttonfont, 69
  degree, 70
  display, 67
  draw, 66
  erase, 66
  first display, 66
  fuzzyeq, 71
  objdelete, 69
  objects, 68, 69
  optionmenu, 69
  screen updating, 68
  startdraw, 66
  summary, 65, 177
  supsubadjust, 70
initial entry unit, 227
initializations
  arrow, 173
  judging, 175
  unit, 232
initialize
  arrays to zero, 211
input
  algebraic, 167
  alphanumeric, 159
buffer
get one key, 134
clear buffer, 135
from a file, 294, 297
keyset, 159
last device, 334
last key, 333
mouse, 126
non-keyset, 132
numerical, 165
single key, 125
input handling, 124
overview, 124
input/output, 285
error returns, 287, 306
insert file, 11
int(x), 203
integer, 191
constants, 191
convert to string, 265
integer divide, 200
integers
range of, 191
intersection, 201
intersection (logical), 201
inverse mode, 60
italic
after -answer-, 169
text, 39
iteration, 218
iterative loops, 218
judge, 182
changing the judgment, 182
initializations, 175
Judging
overview, 158
jump, 239
jumpout, 241
keynames, 204
keypress
current, 333
forcing, 133
single keys, 125
value of, 204
keys=, 125
keyset, 184
keywords, 318
labels on graphs, 106
labeix, 106
labeled, 106
languages, 184
left shift, 206
length, 213
arrow input, 330
of axes, 104
of video, 327
pass-by-address arrays, 208
screen size, 321
length of a string, 269
less than, 201
library files, 243
line
with arrowhead, 54
line thickness, 62
lines
blank, 11
drawing, 52
ln(x), 203
local variables, 195
with global variables, 196
log
base 10, 203
natural, 203
log graphs, 108
log(x), 203
logical comparisons with markers, 252
logical expressions, 201
loop, 218
lscalex, 108
lscaley, 108
lsh, 206
main unit
and -enable touch-, 132
initializations, 232
introduction, 228
when window is reshaped, 228
with -jump, 239
margin
-atnm- no margin, 35
default, 34, 232
set with -at-, 34
with -arrow-, 159
with -text-, 39
with -write-, 41
markers, 246
appending characters, 259
assigning value to, 249
at an arrow, 256
base string, 265
basic operations, 248
before the first character, 276
changed marker flag, 264, 266
combine marker regions, 267
commands, 259
convert integer to string, 265
convert string to integer, 265
copy to a new string, 266
INDEX

creating text, 248

defining, 248
display contents, 250
embedded, 251
embedded CR, 249
embedded in text, 250
embedded quote, 249
embedded TAB, 249
equivalent markers, 273, 274
example
  alphabetize a list, 278
counting vowels, 279
parametric equations, 282
plot two functions, 281
reverse a list, 277
examples, 277
extract number, 272
file I/O, 254
first character, 267
functions, 264
has style, 267
hot info, 268
icon code, 268
icon file, 269
introduction, 246
last character, 269
length, 269
location, 270
logical comparisons, 252
next character, 270
next line, 271
next word, 271
number of icons, 272
pass-by-address, 253
pass-by-value, 253
position, 270
previous character, 274
replacing characters, 259
search for a string, 275
selection at an arrow, 258
set, 276
sticky, 261
style, 262
styles in text, 250
with compute, 254
marks along axes, 106
markup
  answer, 169
markx, 106
marky, 106
mask, 206
math functions, 203
mathematical operators
  in responses, 167, 198, 330, 332

menu, 136
  like -do-, 140
ordering of items, 138
passing a parameter, 139, 141
simple formats, 136
summary of formats, 136
titles with variables, 141
transfer of control, 140
merge, 193
merge,global:, 196
merge,group:, 193
minus, 200
mod(x,y), 203
mode, 60
erase, 60
inverse, 60
rewrite, 60
write, 60
xor, 60
zmode, 324
modify
defaults, 65, 177
judging defaults, 169
mouse, 126
click, 126
coordinates, 325
cursor, 62
drag, 126
enable, 132
input, 126
rubberband, 126
system variables, 321
mouse inputs, 125
move, 98
move a block of variables, 212
moving ahead, 237
name
  path, 329
  user, 329
natural log, 203
negative indices on arrays, 208
nested inputs, 159
new line, 42
newline, 42
newpal, 81
next, 237
no, 164
nomark, 169
non-English text, 184
nookno, 169
noops, 169
noorder, 169
nospell, 169
not (logical), 201

345
SYSTEM VARIABLES

not equal, 201
not(x), 201
novars, 169
number bases, 191
numerical responses, 163, 165
numout, 301
obj delete
    inhibit, 69
objects
    inhibit, 69
octal
    constants, 191
display, 49
ok, 164
okassign, 169
okcaps, 169
okextra, 169
okorder, 169
okspell, 169
operators, 200
    arrays, 208
operators in responses, 169
optional words in -answer-, 162, 169
optionmenu
    inhibit, 69
or (logical), 201
or (of bits), 206
origin
    for rotations, 113
graphing, 103
    screen coordinates, 32
other languages
    differences, 17
outcase, 216
outif, 215
outloop, 218
out-of-order words in -answer-, 169
output
    to a file, 298, 301
outunit, 234
Overview
    input handling, 124
judging, 158
page
    next, 237
    previous, 239
palette, 81
paper coordinates, 37
parametric equations example, 282
pass-by-address
    arrays, 208
to a subroutine, 224
to a unit, 223
pass-by-value
to a unit, 223
path names, 329
pattern, 61
    area, 56
circle, 57
creating, 93
pause, 125
    enable input, 132
    for keyset input, 125
    for mouse input, 126
timed, 125
pause inputs, 125
permutations, 221
phrase inputs, 159
PI, 191, 202
pict, 14
picture icons, 93
pixel data display, 91
plot, 94
plot two functions example, 281
plotting a graph, 102
plus, 200
polar, 109
polar coordinates, 109
polygon
    erase, 58
    fill, 56
polyline, 52
portability, 21
porting programs, 21
position
    in a file, 306
    on the screen, 323
ppm, 14
precise labels on graph, 106
precision
    floating-point, 191
    integers, 191
    numbers near zero, 201
preferences, 13
press, 133
previous page, 239
print, 14
printing, 14
program
    exit from, 237
    starting a new, 241
program example, 5
pull-down menus, 136
punc, 169
punctuation
    of responses, 162, 169
purpose of cT, 4
put, 89, 91
INDEX

q tag
  back, 239
iarrow, 173
ijudge, 175
imain, 232
next, 237
quit execution, 237
quit running, 241
quote, 50
quote marks
  embedded in a string, 249
radians, 202
random
  values, 221
random access files, 285
random values, 221
randu, 221
rarrow, 112
rat, 112
ratnm, 112
rbox, 112
rbutton, 112, 142
rcircle, 112, 114
rcircleb, 112, 114
rclip, 63
rdisk, 112
rdot, 112
rdraw, 112
readln, 297
real numbers, 191
rectangles, 55
recursion, 195
redit, 112, 149
relative
  origin, 113
relative graphics commands, 112
relocatable commands, 112
reloop, 218
repeat, 218
replace, 259
rerase, 112
rescale, 37
  display, 37
  part of a display, 113
reset, 306
reshape
  display, 37
  status, 324
response
  -contingent commands, 159
  -contingent erasing, 159, 177
response handling commands
  judging commands, 124
responses
  actions after judging, 181
  algebraic, 167
  blank, 179
  changing the judgment, 182
  erasing of, 175
  exact, 181
  forcing a keypress, 133
  numerical, 163, 165
  operators in, 169
  punctuation in, 169
  single key, 125, 126, 134
  unexpected, 164
  word or phrase, 162
restoring a marker, 276
restoring the screen, 89
RETURN, 237
reverse-list example, 277
review, 239
rewrite mode, 60
rfill, 112
rgb, 84
rget, 112
right shift, 206
rmove, 112
rorigin, 113
rotate, 113
round(x), 203
roundoff errors, 201
rput, 112
rsh, 206
rslider, 112, 146
rtext, 112, 115
rubberband, 126
rubberband (mouse), 126
rvector, 112
sample programs, 28
sanserif, 327
saturation, 84
saving the screen, 89
scaled coordinates, 102, 112
scalex, 105
scaley, 105
scientific notation, 191
screen
  defaults, 32
  erase, 58
  erasure, 66, 239
  position, 34, 35, 323
  reshape status, 324
  reshaping, 37
  size, 35
    actual, 321
    current, 321
screen description, 34
scrollbar, 146
search menu, 13
sec(x), 202
semi-log graphs, 108
sentence inputs, 159
sequencing commands, 237
sequential files, 285, 289
serial, 309
serial port, 117, 309
serif, 327
server, 312
set, 210
set file descriptor, 291
setdir, 294
setfile, 291
setting a marker, 276
show, 47
marker contents, 250
string, 250
showb, 49
showh, 49
showo, 49
showt, 48
side-effect changed marker flag, 264, 266
sign(x), 203
sin(x), 202
single key inputs, 125
sinh(x), 202
size, 113
of display, 321
of text, 43, 46, 115
skip, 52
slider, 146
socket, 313
sockets, 311
solid circle, 57
solid-color areas, 56
sort, 213
array, 213
strings, 278
sound, 122
source format, 23
specs, 169
sqrt(x), 203
square root, 203
squares, 55
startdraw, 66
step, 20
sticky, 261
stop program, 241
stopping a program, 241
string, 248
strings-see "markers", 246
student variables, 198
style, 262
with string variables, 262
style menu, 13
subroutine
calling, 224
calling an array, 208
subscripts, 43
superscripts, 43
supsubadjust
inhibit, 70
switch file, 11
synonyms, 162
syntax
calculations, 200
syntax level, 27
sysinfo _ TC "'sysinfo
Get System Information'.i.sysinfo " \l 4
_, 319
system literals and constants, 320
system variables, 318
calculations, 332
color status, 326
current mode, 324
file i/o, 287, 306
judging, 330
mouse status, 325
reshape status, 324
screen position, 323
screen size, 321
timing, 335
window forward, 327
zclipboard, 337
zdate, 335
zday, 335
zmachine, 337
ztime, 335
TAB, 50
embedded in a string, 249
tan(x), 202
tanh(x), 202
text, 39
changing size, 115
display of, 39, 41
text display, 39
text panel, 149
textured areas, 61
thick, 62
thickness, 62
tick marks, 106
timed pause, 125
times, 200
timeup, 125, 204
timing, 335
tolerance
  in -ansv-, 163
  in logical expressions, 201
tone, 122
touch, 126
  enable, 132
  keyword, 204
touch command, 129
touch regions, 129
transportability, 21
tries at an -arrow-, 330
trigonometric functions, 202
TRUE, 191, 201, 203
typeface, 43, 46
typing function keys, 186
typing non-English text, 184
union, 201
union (logical), 201
union (of bits), 206
unit, 223
  before first, 227
  doing a, 224
IEU, 227
introduction, 223
  modifying main units, 232
  next, 237
units
  hidden, 11
until, 218
update
  screen, 68
use, 243
user input
  most recent device, 334
  most recent key, 333
user name, 329
user variables, 198
using code from another file, 243
value
  colors, 84
variables, 190
author, 198
  combining groups of variables, 193
  combining local and global, 196
display of, 47, 49
embedded in text, 50
formatted display of, 48
global, 193
graphics, 321
local, 195
merge, global:, 196
merge, group:, 193
mouse, 321
random, 221
summary, 190, 191
system, 287, 306, 318, 321, 323, 324, 325, 326, 327, 330, 332, 335, 336, 337
user, 198
vbar, 110
vector, 54
vertical bars, 110
video, 117
video height, 327
video length, 327
video time indicator, 327
video width, 327
voiding the stack, 239
volume of sound, 122
volume of tone, 122
vowel-counting example, 279
vplay, 120
vset, 119
vshow, 121
vstep, 121
wcolor, 79
while loops, 218
width
  of video, 327
window
  color, 79
    main unit on reshape, 228
  zforeground, 327
window menu, 13
window size, 34
window title, 34
word
  count at an -arrow-, 330
word inputs, 159
write, 41
  mode, 60
wrong, 162
wrongv, 163
wtitle, 34
xin, 301
xor mode, 60
xout, 303
zaltered(m), 264
zansctn, 330
zarrows, 256
zarrowsel, 258
zbase(m), 265
zbutton, 142
zcaps, 330
zchar(m), 265
zclipboard, 337
zclock, 335
zcode(m), 265
zcolorb, 77, 326
SYSTEM VARIABLES

zcolorf, 77, 326
zcopy(m), 266
zcurrentdir, 294
zcurrentu, 336
zcursors, 327
zdate, 335
zday, 335
zdblclick, 325
zdefaultb, 77, 326
zdefaultf, 77, 326
zdevice, 334
zeditkey, 154
zeditsel, 149
zedittext, 149
zeditvis, 149
zemptyp, 252
zend(m), 266
zentire, 330
zero, 211
  an array, 211
  values near, 201
zextent(m), 267
zextra, 330
zfilename, 329
zflepel, 329
zfirst(m), 267
zfixed, 327
zforeground, 327
zgtouchx, 325
zgtouchy, 325
zhasstyle(m,style), 267
zheight, 321
zhomedir, 329
zhotsel, 153
zhotinfo(m), 268
zhotsel, 153
zhsvn, 88
ziconcode(m,N), 268
ziconfile(m,N), 269
zicons, 327
zjcount, 330
zjudged, 330
zk(x), 204
zkkey, 333
zksw(x), 204
zlast(m), 269
zleftdown, 325
zlength(array), 213
zlength(m), 269
zlocate(m), 270
zmachine, 337
zmode, 324
zncolors, 86, 326
znext(m), 270
znextline(m), 271
znextword(m), 271
znicons(m), 272
zntries, 330
znpossible(m), 272
zopcnt, 167, 330
zorder, 330
zpatterns, 327
zpreceed(m), 273
zprevious(m), 274
zreshape, 324
zretinfo, 306
zreturn, 332
  file operations, 287, 306
  summary, 332
  the status variable, 332
zrgbn, 88
zrightdown, 325
zrtouch, 325
zrtouchy, 325
zsamemark(m), 274
zsans, 327
zsearch(m), 275
zserif, 327
zsetmark(m), 276
zheight(), 89, 91
zsheight, 327
zslider, 146
zspell, 330
zstare(m), 276
zswidth(), 89, 91
ztexat, 154
zttime, 335
ztouchx, 325
ztouchy, 325
zuser, 329
zvalue, 142, 146
zvarcnt, 330
zvheight, 327
zvlength, 327
zvplaying, 327
ztvtime, 327
zvwidth, 327
zwcolor, 79, 326
zwcount, 330
zwcount, 330
zwherex, 323
zwherex, 323
zwidth, 321
zxmax, 321
zmax, 321
zxmin, 321
zmin, 321
zwcolor, 79, 326
zyixels. 321