Alphabetical List of All Turing Elements

- <u>abs absolute value function</u>
- <u>addr address of a variable</u>
- <u>addressint type</u>
- <u>all all members of a set</u>
- and operator
- anyclass the ancestor of all classes
- arctan arctangent function (radians)
- arctand arctangent function (degrees)
- <u>array type</u>
- <u>assert statement</u>
- assignability of expression to variable
- <u>assignment statement</u>
- <u>begin statement</u>
- <u>bind declaration</u>
- <u>bits extraction</u>
- <u>body declaration</u>
- <u>boolean true-false type</u>
- break debugger pause statement
- buttonchoose switch mouse modes
- buttonmoved has a mouse event occurred
- buttonwait get a mouse event procedure
- <u>case selection statement</u>
- <u>catenation (+) joining together strings</u>
- <u>ceil real-to-integer function</u>
- char(n) type
- <u>char type</u>
- <u>cheat type cheating</u>
- <u>checked compiler directive</u>
- chr integer-to-character function
- <u>class declaration</u>
- clock millisecs used procedure
- <u>close file statement</u>

- <u>cls clear screen graphics procedure</u>
- <u>collection declaration</u>
- <u>color text color graphics procedure</u>
- <u>colorback background color procedure</u>
- colour text color graphics procedure
- <u>colourback background color procedure</u>
- <u>comment remark statement</u>
- <u>comparisonOperator</u>
- <u>Concurrency</u>
- <u>Concurrency.empty</u>
- <u>Concurrency.getpriority</u>
- <u>Concurrency.setpriority</u>
- <u>Concurrency.simutime</u>
- <u>condition declaration</u>
- Config
- Config.Display
- Config.Lang
- <u>Config.Machine</u>
- const constant declaration
- constantReference use of a constant
- <u>cos cosine function (radians)</u>
- <u>cosd cosine function (degrees)</u>
- <u>date procedure</u>
- <u>declaration create a variable</u>
- deferred subprogram declaration
- <u>delay procedure</u>
- <u>Dir</u>
- <u>Dir.Change</u>
- <u>Dir.Close</u>
- <u>Dir.Create</u>
- <u>Dir.Current</u>
- <u>Dir.Delete</u>
- <u>Dir.Get</u>
- <u>Dir.GetLong</u>
- <u>Dir.Open</u>
- div integer truncating division operator
- <u>Draw</u>
- Draw.Arc
- Draw.Box

- <u>Draw.Cls</u>
- <u>Draw.Dot</u>
- Draw.Fill
- Draw.FillArc
- Draw.FillBox
- <u>Draw.FillMapleLeaf</u>
- Draw.FillOval
- Draw.FillPolygon
- Draw.FillStar
- <u>Draw.Line</u>
- Draw.MapleLeaf
- <u>Draw.Oval</u>
- <u>Draw.Polygon</u>
- <u>Draw.Star</u>
- <u>Draw.Text</u>
- drawarc graphics procedure
- drawbox graphics procedure
- drawdot graphics procedure
- drawfill graphics procedure
- drawfillarc graphics procedure
- drawfillbox graphics procedure
- <u>drawfillmapleleaf</u> <u>graphics</u> procedure
- drawfilloval graphics procedure
- <u>drawfillpolygon graphics procedure</u>
- drawfillstar graphics procedure
- drawline graphics procedure
- drawmapleleaf graphics procedure
- drawoval graphics procedure
- drawpic graphics procedure
- drawpolygon graphics procedure
- drawstar graphics procedure
- empty condition function
- enum enumerated type
- <u>enumeratedValue enumerated value</u>
- <u>eof end-of-file function</u>
- <u>equivalence of types</u>
- erealstr real-to-string function
- <u>Error</u>
- Error.Last

- Error.LastMsg
- <u>Error.LastStr</u>
- Error.Msg
- <u>Error.Str</u>
- <u>Error.Trip</u>
- <u>ErrorNum</u>
- Exceptions
- <u>exit statement</u>
- <u>exp exponentiation function</u>
- explicitCharConstant character literal
- explicitConstant literal
- explicitIntegerConstant integer literal
- explicitRealConstant real literal
- explicitStringConstant string literal
- explicitTrueFalseConstant boolean literal
- <u>expn expression</u>
- export list
- external declaration
- false boolean value (not true)
- fetcharg fetch argument function
- <u>File</u>
- <u>File.Copy</u>
- <u>File.Delete</u>
- <u>File.DiskFree</u>
- <u>File.Exists</u>
- <u>File.Rename</u>
- <u>File.Status</u>
- <u>flexible array initialization</u>
- <u>floor real-to-integer function</u>
- <u>Font</u>
- Font.Draw
- <u>Font.Free</u>
- Font.GetName
- Font.GetSize
- <u>Font.GetStyle</u>
- <u>Font.Name</u>
- Font.New
- Font.Sizes
- <u>Font.StartName</u>

- <u>Font.StartSize</u>
- Font.Width
- <u>for statement</u>
- <u>fork statement</u>
- forward subprogram declaration
- <u>frealstr real-to-string function</u>
- <u>free statement</u>
- <u>function declaration</u>
- <u>functionCall</u>
- get file statement
- getch get character procedure
- getchar get character function
- getenv get environment function
- getpid get process id function
- <u>getpriority function</u>
- <u>GUI</u>
- <u>GUI.AddLine</u>
- <u>GUI.AddText</u>
- <u>GUI.Alert</u>
- <u>GUI.Alert2</u>
- <u>GUI.Alert3</u>
- <u>GUI.AlertFull</u>
- <u>GUI.Choose</u>
- <u>GUI.ChooseFull</u>
- <u>GUI.ClearText</u>
- <u>GUI.CloseWindow</u>
- <u>GUI.CreateButton</u>
- <u>GUI.CreateButtonFull</u>
- <u>GUI.CreateCanvas</u>
- <u>GUI.CreateCanvasFull</u>
- <u>GUI.CreateCheckBox</u>
- <u>GUI.CreateCheckBoxFull</u>
- <u>GUI.CreateFrame</u>
- <u>GUI.CreateHorizontalScrollBar</u>
- <u>GUI.CreateHorizontalScrollBarFull</u>
- <u>GUI.CreateHorizontalSlider</u>
- <u>GUI.CreateLabel</u>
- <u>GUI.CreateLabelFull</u>
- <u>GUI.CreateLabelledFrame</u>

- <u>GUI.CreateLine</u>
- <u>GUI.CreateMenu</u>
- <u>GUI.CreateMenuItem</u>
- <u>GUI.CreateMenuItemFull</u>
- <u>GUI.CreatePicture</u>
- <u>GUI.CreatePictureButton</u>
- <u>GUI.CreatePictureButtonFull</u>
- <u>GUI.CreatePictureRadioButton</u>
- <u>GUI.CreatePictureRadioButtonFull</u>
- <u>GUI.CreateRadioButton</u>
- <u>GUI.CreateRadioButtonFull</u>
- <u>GUI.CreateTextBox</u>
- <u>GUI.CreateTextBoxFull</u>
- <u>GUI.CreateTextField</u>
- <u>GUI.CreateTextFieldFull</u>
- <u>GUI.CreateVerticalScrollBar</u>
- <u>GUI.CreateVerticalScrollBarFull</u>
- <u>GUI.CreateVerticalSlider</u>
- <u>GUI.Disable</u>
- <u>GUI.Dispose</u>
- <u>GUI.Draw...</u>
- <u>GUI.Enable</u>
- <u>GUI.FontDraw</u>
- <u>GUI.GetCheckBox</u>
- <u>GUI.GetEventTime</u>
- <u>GUI.GetEventWidgetID</u>
- <u>GUI.GetEventWindow</u>
- <u>GUI.GetHeight</u>
- <u>GUI.GetMenuBarHeight</u>
- <u>GUI.GetScrollBarWidth</u>
- <u>GUI.GetSliderValue</u>
- <u>GUI.GetText</u>
- <u>GUI.GetVersion</u>
- <u>GUI.GetWidth</u>
- <u>GUI.GetX</u>
- <u>GUI.GetY</u>
- <u>GUI.Hide</u>
- <u>GUI.HideMenuBar</u>
- <u>GUI.OpenFile</u>

- <u>GUI.OpenFileFull</u>
- <u>GUI.Pic...</u>
- <u>GUI.ProcessEvent</u>
- <u>GUI.Quit</u>
- <u>GUI.Refresh</u>
- <u>GUI.SaveFile</u>
- <u>GUI.SaveFileFull</u>
- <u>GUI.SelectRadio</u>
- <u>GUI.SetActive</u>
- <u>GUI.SetBackgroundColor</u>
- <u>GUI.SetBackgroundColour</u>
- <u>GUI.SetCheckBox</u>
- <u>GUI.SetDefault</u>
- <u>GUI.SetDisplayWhenCreated</u>
- <u>GUI.SetKeyEventHandler</u>
- <u>GUI.SetLabel</u>
- <u>GUI.SetMouseEventHandler</u>
- <u>GUI.SetNullEventHandler</u>
- <u>GUI.SetPosition</u>
- <u>GUI.SetPositionAndSize</u>
- <u>GUI.SetScrollAmount</u>
- <u>GUI.SetSelection</u>
- <u>GUI.SetSize</u>
- <u>GUI.SetSliderMinMax</u>
- <u>GUI.SetSliderReverse</u>
- <u>GUI.SetSliderSize</u>
- <u>GUI.SetSliderValue</u>
- <u>GUI.SetText</u>
- <u>GUI.SetXOR</u>
- <u>GUI.Show</u>
- <u>GUI.ShowMenuBar</u>
- handler exception handler
- hasch has character function
- id (identifier) name of an item in a program
- <u>#if used for conditional compilation</u>
- <u>if statement</u>
- implement by clause
- <u>implement clause</u>
- import list

- <u>in member of a set</u>
- <u>include source files</u>
- index find pattern in string function
- <u>indexType</u>
- <u>indirection operator (@)</u>
- <u>infix operator</u>
- inherit inheritance clause
- init array initialization
- Input
- <u>Input.getch</u>
- <u>Input.getchar</u>
- Input.hasch
- Input.KeyDown get keyboard state
- Input.Pause pause for keystroke
- <u>int integer type</u>
- <u>intn n-byte integer type</u>
- intreal integer-to-real function
- intstr integer-to-string function
- <u>invariant assertion</u>
- Joystick
- Joystick.GetInfo
- <u>Keyboard</u>
- length of a string function
- Limits
- <u>ln natural logarithm function</u>
- <u>locate procedure</u>
- locatexy graphics procedure
- <u>loop statement</u>
- <u>lower bound</u>
- <u>Math</u>
- max maximum function
- maxcol maximum column function
- maxcolor graphics function
- maxcolour graphics function
- maxint maximum integer function
- maxnat maximum natural number function
- <u>maxrow maximum row function</u>
- <u>maxx graphics function</u>
- maxy graphics function

- <u>min minimum function</u>
- minint minimum integer function
- minnat minimum natural number function
- <u>mod modulo operator</u>
- <u>module declaration</u>
- monitor declaration
- <u>Mouse</u>
- <u>Mouse.ButtonChoose</u>
- <u>Mouse.ButtonMoved</u>
- <u>Mouse.ButtonWait</u>
- <u>Mouse.Where</u>
- <u>mousewhere</u>
- <u>Music</u>
- Music.Play
- Music.PlayFile
- <u>Music.PlayFileStop</u>
- <u>Music.Sound</u>
- <u>Music.SoundOff</u>
- <u>named type</u>
- <u>nargs number of arguments</u>
- <u>nat natural number type</u>
- <u>natn n-byte natural number type</u>
- natreal natural number to real function
- <u>natstr natural-number-to-string function</u>
- <u>Net</u>
- <u>Net.BytesAvailable</u>
- <u>Net.CharAvailable</u>
- <u>Net.CloseConnection</u>
- Net.HostAddressFromName
- <u>Net.HostNameFromAddress</u>
- <u>Net.LineAvailable</u>
- <u>Net.LocalAddress</u>
- <u>Net.LocalName</u>
- <u>Net.OpenConnection</u>
- <u>Net.OpenURLConnection</u>
- <u>Net.TokenAvailable</u>
- <u>Net.WaitForConnection</u>
- <u>new statement</u>
- <u>nil pointer to a collection</u>

- <u>not true/false (boolean) operator</u>
- <u>objectclass of a pointer</u>
- <u>opaque type</u>
- <u>open file statement</u>
- <u>or operator</u>
- ord character-to-integer function
- parallelget parallel port function
- parallelput parallel port procedure
- paramDeclaration parameter declaration
- pause statement
- <u>PC</u>
- <u>PC.ParallelGet</u>
- <u>PC.ParallelPut</u>
- pervasive declaration modifier
- <u>Pic</u>
- <u>Pic.Blend</u>
- <u>Pic.Blur</u>
- <u>Pic.Draw</u>
- <u>Pic.DrawFrames</u>
- <u>Pic.DrawFramesBack</u>
- <u>Pic.DrawSpecial</u>
- <u>Pic.DrawSpecialBack</u>
- <u>Pic.FileNew</u>
- <u>Pic.FileNewFrames</u>
- <u>Pic.Flip</u>
- <u>Pic.Frames</u>
- <u>Pic.Free</u>
- <u>Pic.Height</u>
- <u>Pic.Mirror</u>
- <u>Pic.New</u>
- <u>Pic.Rotate</u>
- <u>Pic.Save</u>
- <u>Pic.Scale</u>
- <u>Pic.ScreenLoad</u>
- <u>Pic.ScreenSave</u>
- <u>Pic.SetTransparentColor</u>
- <u>Pic.SetTransparentColour</u>
- <u>Pic.Width</u>
- play procedure

- playdone function
- pointer type
- post assertion
- pre assertion
- precedence of operators
- pred predecessor function
- prefix operator
- procedure declaration
- procedureCall statement
- process declaration
- program a (main) program
- <u>put statement</u>
- <u>quit fail statement</u>
- <u>Rand</u>
- <u>Rand.Int</u>
- <u>Rand.Next</u>
- <u>Rand.Real</u>
- <u>Rand.Reset</u>
- <u>Rand.Seed</u>
- <u>Rand.Set</u>
- rand random real number procedure
- randint random integer procedure
- <u>randnext procedure</u>
- randomize procedure
- <u>randseed procedure</u>
- <u>read file statement</u>
- real the real number type
- <u>realn n-byte real number type</u>
- realstr real-to-string function
- <u>record type</u>
- register use machine register
- <u>rem remainder operator</u>
- repeat make copies of string:function
- <u>result statement</u>
- <u>return statement</u>
- <u>RGB</u>
- <u>RGB.AddColor</u>
- <u>RGB.AddColour</u>
- <u>RGB.GetColor</u>

- <u>RGB.GetColour</u>
- <u>RGB.maxcolor</u>
- <u>RGB.maxcolour</u>
- <u>RGB.SetColor</u>
- <u>RGB.SetColour</u>
- round real-to-integer function
- <u>scalar type</u>
- seek (file) statement
- <u>self pointer to current object</u>
- separator between tokens in a program
- <u>set type</u>
- <u>setConstructor</u>
- <u>setpriority procedure</u>
- setscreen graphics procedure
- <u>shl shift left operator</u>
- <u>shr shift right operator</u>
- <u>sign function</u>
- <u>signal wake up a process statement</u>
- simutime simulated time function
- <u>sin sine function (radians)</u>
- <u>sind sine function (degrees)</u>
- <u>sizeof size of a type</u>
- sizepic graphics function
- <u>skip used in get statement</u>
- <u>skip used in put statement</u>
- <u>sound statement</u>
- <u>Sprite</u>
- Sprite.Animate
- Sprite.ChangePic
- <u>Sprite.Free</u>
- <u>Sprite.Hide</u>
- <u>Sprite.New</u>
- <u>Sprite.SetFrameRate</u>
- Sprite.SetHeight
- <u>Sprite.SetPosition</u>
- <u>Sprite.Show</u>
- <u>sqrt square root function</u>
- <u>standardType</u>
- <u>statement</u>

- <u>statementsAndDeclarations</u>
- <u>Str</u>
- <u>Stream</u>
- <u>Stream.eof</u>
- <u>Stream.Flush</u>
- <u>Stream.FlushAll</u>
- <u>string type</u>
- <u>string comparison</u>
- strint string-to-integer:function
- <u>strintok string-to-integer:function</u>
- <u>strnat string to natural number function</u>
- strnatok string to natural number function
- <u>strreal string-to-real function</u>
- strrealok string-to-real function
- <u>subprogramHeader</u>
- <u>subprogramType</u>
- <u>subrangeType</u>
- <u>substring of another string</u>
- <u>succ successor function</u>
- <u>Sys</u>
- <u>Sys.Exec</u>
- <u>Sys.FetchArg</u>
- Sys.GetComputerName
- <u>Sys.GetEnv</u>
- Sys.GetPid
- <u>Sys.GetUserName</u>
- <u>Sys.Nargs</u>
- sysclock millisecs used procedure
- <u>system statement</u>
- <u>tag statement</u>
- takepic graphics procedure
- <u>tell file statement</u>
- <u>Text</u>
- <u>Text.Cls</u>
- <u>Text.Color</u>
- <u>Text.ColorBack</u>
- <u>Text.Colour</u>
- <u>Text.ColourBack</u>
- <u>Text.Locate</u>

- <u>Text.LocateXY</u>
- <u>Text.maxcol</u>
- <u>Text.maxrow</u>
- <u>Text.WhatCol</u>
- <u>Text.WhatColor</u>
- <u>Text.WhatColorBack</u>
- <u>Text.WhatColour</u>
- <u>Text.WhatColourBack</u>
- <u>Text.WhatRow</u>
- <u>Time</u>
- <u>Time.Date</u>
- <u>Time.DateSec</u>
- <u>Time.Delay</u>
- <u>Time.Elapsed</u>
- <u>Time.ElapsedCPU</u>
- <u>Time.PartsSec</u>
- <u>Time.Sec</u>
- <u>Time.SecDate</u>
- <u>Time.SecParts</u>
- time time of day as a string procedure
- token in input
- true boolean value (not false)
- <u>type declaration</u>
- <u>TypeConv</u>
- typeSpec type specification
- <u>unchecked compiler directive</u>
- <u>union type</u>
- unit file containing module, monitor, or class
- <u>unqualified export</u>
- <u>upper bound</u>
- <u>var declaration</u>
- variableReference use of a variable
- <u>View</u>
- <u>View.ClipAdd</u>
- <u>View.ClipOff</u>
- <u>View.ClipSet</u>
- <u>View.maxcolor</u>
- <u>View.maxcolour</u>
- <u>View.maxx</u>

- <u>View.maxy</u>
- <u>View.Set</u>
- <u>View.Update flicker-free animation</u>
- <u>View.WhatDotColor</u>
- <u>View.WhatDotColour</u>
- wait block a process statement
- wallclock seconds since 1/1/1970 procedure
- whatcol cursor position function
- whatcolor text color graphics function
- whatcolorback color of background function
- whatcolour text color graphics function
- whatcolourback color of background function
- whatdotcolor graphics function
- whatdotcolour graphics function
- whatrow cursor position function
- <u>Window</u>
- <u>Window.Close</u>
- <u>Window.GetActive</u>
- <u>Window.GetPosition</u>
- <u>Window.GetSelect</u>
- <u>Window.Hide</u>
- <u>Window.Open</u>
- Window.Select
- <u>Window.Set</u>
- <u>Window.SetActive</u>
- <u>Window.SetPosition</u>
- <u>Window.Show</u>
- <u>Window.Update</u>
- <u>write file statement</u>
- xor exclusive "or" operator

abs	absolute value function
Syntax	abs (expn)
Description	The abs function is used to find the absolute value of a number (the <i>expn</i>). For example, abs (-23) is 23.
	This program outputs 9.83.
Example	<pre>var x : real := -9.83 put abs (x) % Outputs 9.83</pre>

Execute

	The abs function accepts numbers that are either int 's or real 's. The type of the result is the same type as the accepted number.
Details	The abs function is often used to see if one number <i>x</i> is within a given distance <i>d</i> of another number <i>y</i> ; for example:
	if abs $(x - y) \le d$ then

See also predefined unit <u>Math</u>.

address of a variable

Dirty	
Syntax	addr (reference)
Description	The addr attribute is used to find the integer address of a variable or non scalar constant. This is implementation-dependent. This address may be used in an indirection operation <i>@</i> .
	Set <i>a</i> to be the address of <i>x</i> .
Example	<pre>var x : real var a : addressint := addr (x)</pre>
	The value of the address produced by addr is of type addressint , an integer type whose range is that of the underlying memory addresses.
Details	The concept of an address is implementation-dependent. For example, an optimizing compiler could determine that a variable does not require space because the program could be computed without the variable with no change in output. However, in most implementations, types have a predictable size and variables of that type occupy that number of bytes in memory.
See also	the <u>indirection operator @</u> , cheat , <i>explicitIntegerConstant</i> (how to write hexadecimal constants), and pointer type (in particular unchecked pointer type). See also <u>sizeof</u> , which returns the size of a variable.

addr

addressint

Dirty

Syntax addressint

The addressint (address integer) type is an integer type whose ran; value is the same as that of the underlying computer. This range is, its nature, implementation-dependent. On 32-bit architectures, it is commonly the same range as nat4 (4-byte natural number).

Record r contains three fields, one of which has type **char**(28). Var *a* is an integer whose range of values is the same as the addresses c underlying computer. This assigns *B* to the seventh character of a record of type r which is assumed to be located at absolute address

Example

type	er:	
	reco	ord
		i : int
		<i>c28</i> : char (28)
		<i>c11</i> : char (11)
	end	record
var	a :	addressint % An integer
		% a is assigned an integer v
r @	(a)	. c28 (7) := 'B' % Use indirectic

Although **addressint** is called an integer type, it is commonly equivalent to a natural type such as **nat4** (for 32-bit machines).

Be careful not to confuse addressint with pointer types. In low lev languages such as assembler and C, addresses and pointers are the same. In Turing, however, a pointer is a high level concept that is n abstract than a machine address. A Turing pointer is a reference to object, and the representation of this reference depends upon the implementation. In current Turing implementations, pointers (whic by default checked) are represented as a time stamp (a unique num together with an address. The time stamp is used to make sure that pointer actually locates an object. There are also unchecked pointer An unchecked pointer's internal representation is a machine addres. You can use type cheats (a dangerous feature) to translate between addressint and unchecked pointers. This is meaningful in current

implementations.

the <u>indirection operator @</u>, <u>cheat</u>, <u>explicitIntegerConstant</u> (how to write hexadecimal constants), and <u>pointer</u> type (in particular unchecked pointer type). See also <u>addr</u>, which returns the address variable.

all	all members of a set
Syntax	setTypeName (all)
Description	Given a set type named <i>S</i> , the set of all of the possible elements of <i>S</i> is written <i>S</i> (all).
Example	type smallSet : set of 0 2 var x : smallSet := smallSet (all) % Set x contains elements 0, 1 a
See also	set type for details about sets.

and

Syntax	A and B
Description	The and (boolean) operator yields a result of true if, and only if, both operands are true. The and operator is a short circuit operator. For example, if <i>A</i> is false in <i>A</i> and <i>B</i> then <i>B</i> is not evaluated.
Example	<pre>var success : boolean := false var continuing := true % The type is boolea continuing := continuing and success</pre>
Details	The <i>continuing</i> variable is set to true if, and only if, both <i>continuing</i> and <i>success</i> are true . Since Turing uses short circuit operators, once <i>continuing</i> is false , <i>success</i> will not be looked at.
	The and operator can also be applied to natural numbers. The result is the natural number that is the bit-wise and of the operands. See nat (natural number).
Example	This masks out the everything but the lower two bytes of <i>number</i> .
	var number : nat := 16#ABCD var mask : nat := 16#FF put number and mask % Outputs 205 (CD16)
See also	boolean (which discusses true/false values), <u>explicitTrueFalseConstant</u> (which discusses the values true and <u>false</u>), <u>precedence</u> and <u>expn</u> (expression).

anyclass

Syntax anyclass

DescriptionThere is a predefined class called **anyclass**, which is the root of the
All classes that do not have **inherit** lists are considered to be expan
The main purpose of **anyclass** is to allow pointers that can locate o

Here is the declaration of a pointer p that can locate an object of an

var p : pointe r	to anyclass	% Short form: va
var q : pointe r	to stack	% Short form: va
new q	% Create	a stack object
p := q	% Legal b	ecause p's class
	% is an a	ncestor of q's clas

Assuming *stack* is a class, this creates a *stack* object and places its The compiler will not allow a call to *stack*'s exported subprograms in:

p -> push (14) % ILLEGAL! anyclass has no

Example An assignment from *p* to *q* is legal, as in:

q := p % Checks that p locates a stack object (

This implies a run time check to make sure that *p* locates an object descendant of a *stack*).

Here is a way to call a subprogram exported from *stack* using *p*:

stack (p) . push (14) % Checks that p locates

This checks to see that *p* locates a *stack* object (or a descendant) be *stack* operation *push*.

It is legal to create objects of the class called **anyclass**, but this is n because there is nothing you can do with these objects (they have r legal to assign these objects to other objects of the same class (**any** this accomplishes nothing.

objectclass, which takes a class pointer and produces the class of t by the pointer. This is used for testing to determine the class of the pointer.

See also

See also <u>class</u>. See also <u>export</u> list, <u>import</u> list, <u>implei</u> list, <u>implei</u> list, <u>implei</u> list, <u>implei</u> list.

arccosine function (radians)

arccos

arccos (*r* : **real**) : **real Syntax**

The **arccos** function is used to find the arc cosine of a value. The **Description** result is given in radians. For example, **arccos** (0.5) is p / 3.

This program prints out the arccosine of -1 through 1 in radians.

Example

```
for i : -4 .. 4
     const arg := i / 4
     put "Arc cosine of ", arg, " is ",
     arccos (arg), " radians"
end for
```

Execute

	the <u>arcsin</u> and <u>arctan</u> functions for calculating arcsine and arctangent.
See also	the arccosd function which finds the arc cosine of a value with the result given in degrees. (2p radians are the same as 360 degrees.)
	See also predefined unit Math

See also predefined unit Math.

arccosd

arccosine function (degrees)

Syntax arccosd (*r* : real) : real

Description The **arccosd** function is used to find the arc cosine of an angle given in degrees. For example, **arccosd** (0.5) is 60.

This program prints out the arccosine of values from -1 to 1 in degrees.

Example for i : -4 .. 4
 const arg := i / 4
 put "Arc cosine of ", arg, " is ",
 arccosd (arg), " degrees"
end for

	the arcsind and arctand functions for calculating arcsine and arctangent
See also	the arccos function which finds the arc cosine of a value with the result given in radians. (2p radians are the same as 360 degrees.)
	See also predefined unit <u>Math</u> .

arcsin

arcsine function (radians)

Syntax arcsin (*r* : real) : real

Description The **arcsin** function is used to find the arc sine of a value. The result is given in radians. For example, **arcsin** (0.5) is p / 6.

This program prints out the arcsine of -1 through 1 in radians.

Example

```
for i : -4 .. 4
    const arg := i / 4
    put "Arc sine of ", arg, " is ",
        arcsin (arg), " radians"
end for
```

Execute

the **arccos** and **arctan** functions for calculating arccosine and arctangent.

See also the <u>arcsind</u> function which finds the arc sine of a value with the result given in degrees. (2p radians are the same as 360 degrees.)

See also predefined unit **Math**.

arcsind

arcsine function (degrees)

Syntax arcsind (*r* : real) : real

Description The **arcsind** function is used to find the arc sine of an angle given in degrees. For example, **arcsind** (0.5) is 30.

This program prints out the arcsine of values from -1 to 1 in degrees.

Execute

See alsothe arccosd and arctand functions for calculating arccosine and
arctangentSee alsothe arcsin function which finds the arc sine of a value with the
result given in radians. (2p radians are the same as 360 degrees.)
See also predefined unit Math.

arctan

arctangent function (radians)

Syntax arctan (*r* : real) : real

Description The **arctan** function is used to find the arc tangent of a value. The result is given in radians. For example, **arctan** (1) is p / 4.

This program prints out the arctangent of 0 through 3 in radians.

Example

```
for i : 0 .. 12
    const arg := i / 4
    put "Arc tangent of ", arg, " is ",
        arctan (arg), " radians"
end for
```

	the <u>arcsin</u> and <u>arccos</u> functions for calculating arcsine and arccosine
See also	the arctand function which finds the arc tangent of a value with the result given in degrees. (2p radians are the same as 360 degrees.)
	See also predefined unit Math.

arctand

arctangent function (degrees)

Syntax arctand (r : real) : real

Description The **arctand** function is used to find the arc tangent of an angle given in degrees. For example, **arctand** (0) is 0.

This program prints out the arctangent of values from 0 to 3 in degrees.

Example for i : 0 .. 12
 const arg := i / 4
 put "Arc tangent of ", arg, " is ",
 arctand (arg), " degrees"
end for

	the arcsind and arccosd functions for calculating arcsine and arccosine
See also	the arctan function which finds the arc tangent of a value with the result given in radians. (2p radians are the same as 360 degrees.)
	See also predefined unit Math.

Syntax	<pre>array indexType { , indexType } of typeSpec</pre>
Description	An array consists of a number of elements. The <i>typeSpec</i> gives the elements. There is one element for each item in the (combinations <i>indexType(s)</i> . In the following example, the array called <i>marks</i> con each of which is an integer.
	var marks : array 1 100 of int
Example	<pre>var sum : int := 0 for i : 1 100 % Add up the elements of sum := sum + marks (i)</pre>

end for

Details	In the above example, <i>marks(i)</i> is the <i>i</i> -th element of the <i>marks</i> arra or <i>subscript</i> of <i>marks</i> . In Turing, a subscript is surrounded by parer brackets as is the case in the Pascal or C-like languages.
	The <i>prices</i> array shows how an array can have more than one dime one dimension for the year (1988, 1989 or 1990) and another for th There are 36 elements of the array, one for each month of each yea
	var price : array 1988 1990, 1 12 of int
Example	 var sum : int := 0 for year : 1988 1990 % For each year for month : 1 12 % For each month sum := sum + price (year, month) end for end for

Execute

Each *indexType* must contain at least one item. The range 1 .. 0, for be allowed. Each index type must be a subrange of the integers, ch type), or of an enumerated type, an (entire) enumerated type, the **cl** type, or a named type which is one of these.

Arrays can also be declared in the form

var a : array 1 ... * of typeSpec := init (...)

The upper bound of *a* will be computed from the count of the initia Details **var** and **const** arrays can be declared this way. An array variable/cc "*" as an upper bound must have an initializing list. Only one dime be declared in this form.

> Arrays can be assigned as a whole (to arrays of an equivalent type) compared.

> An array can be initialized in its declaration using **init**. For details, declarations.

In this example, the size of the array is not known until run time.

```
var howMany : int
get howMany
var height : array 1 .. howMany of real
    ...read in all the elements of this array...
function total (a : array 1 .. * of real) : real
    var sum : real := 0
    for i : 1 \dots upper (a)
        sum := sum + a (i)
    end for
    result sum
end total
put "Sum of the heights is ", total (height)
```

Example

Execute

The ends of the range of a subscript are called the *bounds* of the an are not known until run time, the array is said to be *dynamic*. In the *height* is a dynamic array. Dynamic arrays can be declared as varia for *height*. However, dynamic arrays cannot appear inside other typ and cannot be named types. Dynamic arrays cannot be assigned an initialized using **init**.

Details In the above example, **upper**(*a*) returns the size of *a*. See also **upp**

In the declaration of an array parameter, the upper bound can be gir as is done in the above example. This means that the upper bound i the corresponding actual parameter (from *height* in this example).

You can have arrays of other types, for example arrays of record. If records, then *R*(*i*).*f* is the way to access the *f* field of the *i*-th element

Arrays can also be made resizeable. This is done using the **flexible** declaration syntax is:

var name : flexible array indexType { , indexTyp

The indices may have compile-time or run-time upper bounds (the compile-time). The upper bounds can be changed by using:

new name , newUpper1 {, newUpper2}

The existing array entries will retain their values, except that any ir will have the corresponding array entries lost. Any index made larg array entries uninitialized (if applicable).

Details Additionally, the upper bound (both in the declaration and the **new** made one less than the lower bound. This effectively makes an arra

elements. It can later be increased in size with another **new**.

In the current implementation (2002), with a multi-dimensional arr number of total elements, it is a run-time error to change any but th (unless one of the new upper bounds is one less than the correspon giving 0 elements in the array) as the algorithm to rearrange the ele locations has not yet been implemented.

Currently, only variables can be declared in this form. There is no 1 parameter type, although a flexible array can be passed to an array as the upper bound.

In this example, the array is resized to fit the number of elements in

```
function getLines (fileName : string) : int
                        var f, numLines : int
                        var line : string
                        open : f, fileName, get
                        numLines := 0
                        100p
                            exit when eof (f)
                            get : f, line : *
                            numLines += 1
                        end loop
                        close : f
                        result numLines
                    end getLines
Example
                    procedure readFile (var lines : array 1 .. * of
                        var f : int
                        var line : string
                        open : f, fileName, get
                        for i : 1 .. upper (lines)
                            get : f, lines (i) : *
                        end for
                                    close : f
                    end readFile
                    var lines : flexible array 1 .. 0 of string
                    new lines, getLines ("text.dat")
                    readFile (lines, "text.dat")
                    for i : 1 .. upper (lines)
                        put lines (i)
                    end for
```

Execute

init to initialize arrays, **flexible** to declare resizable arrays and <u>inde</u> index of an array.

assert

Syntax	assert trueFalseExpn
Description	An assert statement is used to make sure that a certain requirement is met. This requirement is given by the <i>trueFalseExpn</i> . The <i>trueFalseExpn</i> is evaluated. If it is true, all is well and execution continues. If it is false, execution is terminated with an appropriate message.
Example	Make sure that <i>n</i> is positive.
	assert $n > 0$
Example	This program assumes that the <i>textFile</i> exists and can be opened, in other words, that the open will set the <i>fileNumber</i> to a positive stream number. If this is not true, the programmer wants the program halted immediately. var <i>fileNumber</i> : int open : <i>fileNumber</i> , "textFile", read assert <i>fileNumber</i> > 0
Details	In some Turing systems, checking can be turned off. If checking is turned off, assert statements may be ignored and as a result never cause termination.

assignability

of expression to variable

A value, such as 24, is assignable to a variable, such as *i*, if certain rules are followed. These rules, given in detail below, are called the *assignability* rules. They must be followed in assignment statements as well as when passing values to non-var parameters.

var i : int
i := 24 % 24 is assignable to i **var** width : 0 .. 319 width := 3 * i % 3 * i is assignable to wic var a : array 1 .. 25 of string a (i) := "Ralph" % "Ralph" is assignable var name : string (20) name := a (i) % a(i) is assignable to name var b : array 1 .. 25 of string b := a % Array a is assignable to b **type** personType : record age : int name : string (20) end record **var** r, s : personType s := r % Record r is assignable to s

Example

The expression on the right of := must be *assignable* to the variable on the left. An expression passed to a non-**var** parameter must be assignable to the corresponding parameter.

An expression is defined to be *assignable* to a variable if the two *root* types are *equivalent* or if an integer value is being assigned to a **real** variable (in which case the integer value is automatically converted to **real**). Two types are considered to be equivalent if they are essentially the same type (see *equivalence* for the detailed definition of this term).

In most cases a *root* type is simply the type itself. The exceptions are subranges and strings. The *root* type of a subrange, such as 0 .. 319, is the type of its bounds (**int** type in this example). The *root* type of a string, such as the type **string**(9), is the most general string type, namely **string**.

When a subrange variable, such as *width*, is used as an expression, for example on the right side of an assignment statement, its type is considered to be the *root* type (integer in this case) rather than the subrange. When an expression is assigned to a subrange variable such as *width*, the value (3**i* in this example) must lie in the subrange. Analogously, any string variable used in an expression is considered to be of the most general type of string. When a string value is assigned to a string variable, its length must not exceed the variable's maximum length.

Turing's assignability rule applies to characters and strings in this way. A **char** value can be assigned (or passed to an non **var** parameter) with automatic conversion to a **char**(1) variable and vice versa. String values of length 1 can be assigned to **char** variables. Character, that is **char**, values can be assigned to string variables, yielding a string of length 1. String values of length *n* are assignable with automatic conversion to **char**(*n*) variables. Values of type **char**(*n*) can be assigned with automatic conversion to **string** variables.

Turing's assignability rule applies to pointers to classes in this way. A pointer that locates an object created as class E, can be assigned to a pointer to class B only if B is an ancestor of (or the same as) E. For example, a pointer to an object that is a *stackWithDepth* can be assigned to a pointer to *stack*, where *stackWithDepth* is a child of *stack*, but not vice versa. The pointer **nil** can be assigned to any pointer variable, but the value **nil**(C) can only be assigned to a pointer to an ancestor of C.

Objects of classes can be assigned to each other only if both were created as the same class.

Details

assignment

statement

An assignmentStatement is:

Syntax

variableReference := expn

Description An assignment statement calculates the value of the expression (*ex* assigns that value to the variable (*variableReference*).

var i : inti := 24% Variable i becomes 24var a : array 1 .. 25 of stringa (i) := "Ralph"% The i-th element of a......var b : array 1 .. 25 of stringb := a% Array b becomes (is assigned)

The expression on the right of := must be *assignable* to the variable left. For example, in the above, any integer value, such as 24, is ase to *i*, but a **real** value such as 3.14 would not be not assignable to *i*. arrays, records and unions can be assigned. For example, in the abc array *a* is assigned to array *b*. See *assignability* for the exact rules c allowed assignments.

You cannot assign a new value to a constant (const).

There are short forms that allow you to write assignment statement compactly. For example,

Details

i := i + 1

can be shortened to

i += 1

In Turing, there are short forms for combining +, = and * with assign For example, i *= 2 doubles i.

There are also short forms to allow any binary operator to be comb

with assignment. For example, *i* **shl**= 2 shifts *i* by 2 to the left.

begin

statement

A beginStatement is:

Syntax	begin statementsAndDeclarations end
Description	A begin statement limits the scope of declarations made within it to the confines of the begin/end block. In Turing, begin is rarely used, because declarations can appear wherever statements can appear, and because every structured statement such as if ends with an explicit end .
Example	<pre>begin var bigArray : array 1 2000 of real bigArray will exist only inside this begin end</pre>
Details	In Pascal programs, begin statements are quite common because they are required for grouping two or more statements, for example, to group the statements that follow then . This is not necessary in Turing as where ever you can write a single statement, you can also write several statements.

declaration

A bindDeclaration is:

Syntax	<pre>bind [var] id to variableReference { , [var] id to variableReference }</pre>
Description	The bind declaration creates a new name (or names) for a variable reference (or references). You are allowed to change the named item only if you specify var . You can also bind to named non scalar constants.
L.	While <i>variableReference</i> is bound it does not disappear in the scope.
Example	Rename the <i>n</i> -th element of array <i>A</i> so it is called <i>item</i> and then change this element to 15.
	bind var item to A (n) item := 15
	The scope of the identifier (<i>item</i> above) begins with the bind declaration and lasts to the end of the surrounding program or statement (or to the end of the surrounding part of a case or if statement). During this scope, a change to a subscript (<i>n</i> above) that occurs in the variable reference does not change the element to which the identifier refers.
	You are not allowed to use bind at the outermost level of the main program (except nested inside statements such as if) or at the outermost level in a module.
Details	You can also optionally use the register keyword to request that the bind be done using a machine register. The syntax for <i>bindDeclaration</i> is actually:
	bind [var] [register] id to variableReference
	{ , [var] [register] id to variableReference }

In the current (2002) implementation, programs are run interpretively using pseudo-code and the **register** keyword is ignored.

Syntax	bits (expn, subrange)
Description	The bits operator is used to extract a sequence of bits from a natura negative) number expression. The bits are numbered from right to 2
	Set bits 2 and 1 (third and second from the right) in the variable d t We first set b to be the bit string 1100.
Example	<pre>type T12 : 1 2 % Use to specify bit var d : nat2 := 2#1100 % Two byte natural r % At this point bits(d, T12) = 2#10 bits (d, T12) := 2#01 % At this point d = 2#1010</pre>
	Set bit 7 in variable <i>n</i> to be 1. As a result, <i>n</i> will equal 2#10000000
Example	var n : nat1 := 0% A one byte variable sebits (n, 7) := 1% n now contains the pat
	The form of <i>subrange</i> must be one of:
	(a) typeSpec % Subrange type(b) compileTimeIntegerExpression
Details	In form (a) the subrange type specifies a range from L to M (for <i>lec most</i> significant). This is a little confusing because the subrange is M with L on the left and M on the right, but in a number, the least s bit is on the right and the most significant is on the left. The subran can be either the name of a type, for example $T12$, or an explicit su example 3 7. The values in the explicit subrange must be compile values.
	Form (b) represents the range $n \dots n$ where n is the non-negative val expression. In other words, both L and M equal n . The expression c non-negative integer value or natural number value.

bits

If the expression *expn* is a variable reference, the **bits** operation cal assigned to, but cannot be passed to, a **var** parameter. For example above, **bits** (d, T12) has the value 2#01 assigned to it. For this assig be allowed, the expression *expn* must be a natural number type (**na nat2** or **nat4**).

explicitIntegerConstant (for description of constants such as 16#FF the following functions that convert one type to another in a machi independent manner: **ord**, **chr**, **intstr**, **strint**, **natstr**, and **strnat**. S and **shl** (shift right and left).

See also

body

A *bodyDeclaration* is one of:

	(a)	body procedure procedureId
		statementsAndDeclarations
		end procedureId
	(b)	body function functionId
		statementsAndDeclarations
		end functionId
Syntax	(C)	body procedure id [(paramDeclaration
		<pre>{, paramDeclaration })]</pre>
		statementsAndDeclarations
		end id
	(d)	body function <i>id</i> [([paramDeclaration {,
		<pre>paramDeclaration }])] : typeSpec</pre>
		statementsAndDeclarations
		end id

A body declaration is used to resolve either a forward subprogram or a deferred subprogram.

You declare a procedure or function forward when you want to define its header but not its body. This is the case when one procedure or function calls another, which in turn calls the first. This situation is called *mutual recursion*. The use of forward is necessary in this case because every item must be declared before it can be used. The forward declaration must be followed by a body declaration for the same procedure or function. For details, see forward declarations.

When a procedure or function in a class is declared to be **deferred** (or simply exported from the class), it can be resolved

or overridden afterward by giving its body further down in that class or in descendant classes. The overriding procedure must use the keyword **body**. See **class** or "**implement by**" for examples. You can specify the parameter and return values of the subprogram in the **body** declaration. However, the names and types of the parameters and return values must match the initial Details declaration exactly, or a warning results and the parameter list and return values from the **body** declaration are ignored. The example given here is part of a complete Turing program that includes an explanation of **forward** declarations. var token : string forward procedure expn (var eValue : real) import forward term, var token ... other declarations appear here ... body procedure expn var *nextValue* : **real** Example % Evaluate t term (*eValue*) **100**p % Evaluate { + t exit when token not= "+" **aet** token term (nextValue) eValue := eValue + nextValue end loop end expn

Execute

The syntax of a *bodyDeclaration* presented above has been simplified by omitting the optional result identifier, **import** list, **pre** and **post** condition and **init** clause. See **procedure** and **function** declarations for descriptions of these omissions.

The "function" or "procedure" token in the declaration is now

Details optional. In other words the following code fragment is legal

forward procedure p... body p... end p

See also **class**, **forward** and **implement by**

boolean

Syntax	boolean	
Description	The boolean type is used for values that are either true or false . These true-false values can be combined by various operators such as or and and .	
Example	<pre>var success : boolean := false var continuing := true % The type is boolea success := mark >= 60 continuing := success and continuing if continuing then</pre>	
	This type is named after the British mathematician, George Boole, who formulated laws of logic.	
	The operators for true and false are and , or , xor , =>, and not . For two true/false values <i>A</i> and <i>B</i> , these operators are defined as follows:	
Details	<i>A</i> and <i>B</i> is true when both are true <i>A</i> or <i>B</i> is true when either or both are true <i>A</i> xor <i>B</i> is true when either but not both are true $A \Rightarrow B$ (A implies B) is true when both are true or when A is false	
	not <i>A</i> is true when A is false	
	The and operator has higher precedence than or , so <i>A</i> or <i>B</i> and <i>C</i> means <i>A</i> or (<i>B</i> and <i>C</i>).	
	The operators or , and and => are short circuit operators. For example, if <i>A</i> is true in <i>A</i> or <i>B</i> , <i>B</i> is not evaluated.	
Details	The boolean type can be used as an index to an array.	
	Declaration of an array with boolean index.	

Example	<pre>var a : array boolean of int a (false) := 10 a (true) := 20</pre>
Details	The put and get semantics allow put's and get's of boolean values. true values will be output as "true" and false values will be output as "false". The only legal input values are "true" and "false", which are case sensitive.
See also	<i>explicitTrueFalseConstant</i> (which discusses the values true and false), <i>precedence</i> and <i>expn</i> (expression).

break

debugger pause statement

Syntax break

On systems with a debugger, the environment "pauses" when execution reaches the break statement. While "pausing" is environment specific, in general, the program stops execution until the user presses the "Resume" or "Continue" button. While paused, the program variables can be inspected, stack traces done, etc.

for i : 1 .. 100 put i break end for

Example

buttonchoose

switch mouse modes

Syntax **buttonchoose** (*choice* : string)

The **buttonchoose** procedure is used to change the mode of the mouse. In Turing, the mouse can either be in "*single-button mode*" or in "*multi-button mode*". In "*single-button mode*" the mouse is treated as a one button mouse. A button is considered pressed when any button is pressed and released only when all buttons have been released.

Description

In Turing, the mouse starts in "single-button mode".

The parameter *choice* can be one of "singlebutton", "onebutton" (which switch the mouse into "*single-button mode*") or "multibutton" (which switches the mouse into "*multi-button mode*").

A program that displays the status of the mouse at the top left corner of the screen.

```
buttonchoose ("multibutton")
                   var x, y, button, left, middle, right : int
                    mousewhere (x, y, button)
                    left := button mod 10
                                                    % left = 0 or 1
                    middle := (button - left) mod 100 % middle = 6
                    right := button - middle - left
                                                       % right = 0
Example
                    if left = 1 then
                        put "left button down"
                    end if
                    if middle = 10 then
                        put "middle button down"
                    end if
                    if right = 100 then
                        put "right button down"
                    end if
```

Execute

buttonmoved and buttonwait to get mouse events saved in a
queue. See also mousewhere to get the current status of mouse
button(s).See also

See also predefined unit **Mouse**.

Syntax **buttonmoved** (motion : string) : boolean

The **buttonmoved** function indicates whether there is a mouse event of the appropriate type on the mouse queue. Events are either "up", "down", "updown" or "downup" events (although the "downup" and "updown" are the same event).

The parameter *motion* must be one of "up", "down", "updown" or "downup". If an event of the type requested is in the queue, **buttonmoved** returns **true**. If the event is not in the queue, then **buttonmoved** returns **false**.

Description

In "*single-button mode*" (where the mouse is treated like a onebutton mouse), a "down" event occurs whenever all the buttons are up and a button is pressed. An "up" event takes place when the last button is released so that no buttons remain pressed.

In "*multi-button mode*", a "down" event occurs whenever any button is pressed, and an "up" event occurs whenever any button is released.

This program draws random circles on the screen until the user clicks the mouse button, whereupon is starts drawing random boxes. Clicking the mouse button switches between the two.

```
var circles: boolean := true
loop
var x, y, radius, clr: int
if buttonmoved ("down") then
var buttonnumber, buttonupdown : int
buttonwait ("down", x, y, buttonnumber,
circles := not circles
end if
randint (x, 0, maxx)
randint (y, 0, maxy)
randint (radius, 0, 100)
randint (clr, 0, maxcolor)
if circles then
drawfilloval (x, y, radius, radius, clr)
```

Example

```
else
     drawfillbox (x, y, x + radius, y + radiu
end if
end loop
```

Execute

This is an example demonstrating how to check for both character and mouse input at the same time.

var ch : string (1)
var x, y, btnnum, btnupdown : int
loop
 if hasch then
 getch (ch)
 locate (1, 1)
 put "The character entered is a: ", ch
 end if
 if buttonmoved ("down") then
 buttonwait ("down", x, y, btnnum, btnupc
 locate (1, 1)
 put "The button was clicked at position:
 end if
end loop

Execute

Detailsbuttonmoved can be thought of as the mouse equivalent of hasch
in that they both check for something in a queue and both return
immediately.

buttonwaitto get mouse events saved in the queue. See alsobuttonchooseto switch between "single-button mode" andSee also"multi-button mode".

See also predefined unit **Mouse**.

buttonwait

get a mouse event procedure

Syntax	<pre>buttonwait (motion : string, var x, y, buttonnumber, buttonupdown : int)</pre>
	The buttonwait procedure gets information about a mouse event a removes it from the queue.
Description	The parameter <i>motion</i> must be one of "up", "down", "updown" or "downup". If an event of the type requested is in the queue, buttonwait returns instantly. If there isn't such an event, buttonwa waits until there is one and then returns (much like getch handles keystrokes).
	In " <i>single-button mode</i> " (where the mouse is treated like a one-butt mouse), a "down" event occurs whenever all the buttons are up and button is pressed. An "up" event takes place when the last button is released so that no buttons remain pressed.
	In " <i>multi-button mode</i> ", a "down" event occurs whenever any buttc pressed, and an "up" event occurs whenever any button is released.
	The parameters <i>x</i> and <i>y</i> are set to the position of the mouse cursor when the button was pressed. The parameter <i>buttonnumber</i> is set to when in " <i>single-button mode</i> ". In " <i>multi-button mode</i> ", it is set to 1 the left button was pressed, 2 if the middle button was pressed, and if the right button was pressed and 0 if a button was released.
	This program draws lines. It starts a line where the user presses do and continues to update the line while the mouse button is held dow When the button is released, the line is permanently draw and the u can draw another line.
Example	<pre>var x, y, buttonnumber, buttonupdown, buttons : var nx, ny : int loop buttonwait ("down", x, y, buttonnumber, butt nx := x</pre>

Example

```
ny := y
loop
    drawline (x, y, nx, ny, 0) % Erase prev
    exit when buttonmoved ("up")
    mousewhere (nx, ny, buttons)
    drawline (x, y, nx, ny, 1) % Draw line
end loop
buttonwait ("up", nx, ny, buttonnumber, butt
drawline (x, y, nx, ny, 2) % Draw line to f
end loop
```

Execute

In the previous example, when the mouse moves, the line is erased drawing it in white. This erases anything that the line was drawn or top of, including previous lines. A more complete example involvin the clicking and dragging of filled rectangles using the mouse is available. In this example, the background is fully restored when the rectangle is moved by using "xor".

Example

This is an example demonstrating how to check for both character mouse input at the same time.

var ch : string (1)
var x, y, btnnum, btnupdown : int
loop
 if hasch then
 getch (ch)
 locate (1, 1)

Example

```
put "The character entered is a: ", ch
end if
if buttonmoved ("down") then
buttonwait ("down", x, y, btnnum, btnupc
locate (1, 1)
put "The button was clicked at position:
end if
end loop
```

Execute

Details	buttonwait can be thought of as the mouse equivalent of getch in they both read something in a queue and both wait until they get the thing they're looking for.
See also	buttonwait to see if an appropriate event is in the queue. See also buttonchoose to switch between " <i>single-button mode</i> " and " <i>multi-button mode</i> ".
	See also predefined unit Mouse .

A caseStatement is:

case expn of

}:

{ **label** *compileTimeExpn* {, *compileTimeExpn*

Syntax

statementsAndDeclarations }

[label:

statementsAndDeclarations]

end case

A **case** statement is used to choose among a set of statements (and declarations). One set is chosen and executed and then execution continues just beyond **end case**.

Description The expression (*expn*) following the keyword **case** is evaluated and used to select one of the alternatives (sets of declarations and statements) for execution. The selected alternative is the one having a label value equaling the case expression. If none are equal and there is a final **label** with no expression, that alternative is selected.

Output a message based on value of mark.

case mark oflabel 9, 10 :put "Excellent"label 7, 8 :put "Good"label 6 :put "Fair"label 1 :put "Poor"end case

Execute

Output a message based on value of name.

case name of
label "horse", "cow" : put "Farm animal"Label "tiger", "lion" : put "Jungle animal"
label "cat", "dog" : put "Pet"
label : put "Unknown animal"end case

Execute

Details

The case expression is required to match one of the labels. If it does not, there must be a final **label** with no expression. Label expressions must have values known at compile time. All label values must be distinct. The case expression and the label values must have the same equivalent type, which must be an integer, **char**, **boolean**, an **enum** type or strings.

Note that there is no way to express a range of values (for example from 5 to 10) in a label. Each individual value must be expressed in the label.

catenation (+)

joining together strings

A catenation is:

Syntax

stringExpn + stringExpn

Description Two strings (*stringExpns*), **char** or **char**(*n*) values can be joined together (catenated) using the + operator.

var lastName, wholeName : string
lastName := "Austere"
wholeName := "Nancy" + " " + lastName
% The three strings Nancy, a blank and A
% catenated together to make the string
% "Nancy Austere". This string becomes
% value of wholeName

Example

The length of a string catenation is limited to 255 characters.

Catenation is sometimes called *concatenation*.

Catenation can also be applied to **char** and **char**(n) values. See **char** and **char**(n). If either operand, s or t in s + t, is a **string** or a dynamic **char**(n) (length not known at compile time), the result type is **string**. Otherwise (when both s and t are **char** or non-dynamic **char**(n)) the result type is **char**(n).

Details The result of catenation is considered to be a compile time value if both operands are compile time values.

If both operands have the type **char** or **char**(n) neither of which is a dynamic **char**(n), the result is of type **char**(n), which is also of a non dynamic type. This allows the creation of very long **char**(n) values that can effectively span line boundaries using catenation to join lines. If either operand is a dynamic type or a string type, the catenation produces a string, whose length is limited to 255 characters.

substrings (for separating a strings into parts), repeat (for making

See also	repeated catenations), string type, length , and index (to determine where one string is located inside another).
	See also string , <u>char</u> , <u>char</u> (<i>n</i>), <u><i>explicitStringConstant</i></u> , <u><i>explicitCharConstant</i></u> , <u>substring</u> and <u>length</u> .

ceil	real-to-integer function
Syntax	ceil (<i>r</i> : real) : int
Description	Returns the smallest integer greater than or equal to <i>r</i> .
Details	The ceil (ceiling) function is used to convert a real number to an integer. The result is the smallest integer that is greater than or equal to <i>r</i> . In other words, the ceil function rounds up to the nearest integer. For example, ceil (3) is 3, ceil (2.25) is 3 and ceil (-8.43) is -8.
See also	See also the floor and round functions.

char(*n*)

Syntax	char (numberOfCharacters)
Description	Each variable whose type is a char (<i>n</i>) contains exactly <i>n</i> characters.
Example	Canadian postal codes contain six characters, for example, M4V 1Y9. This is represented in a char (6) variable:
•	<pre>var postalCode : char (6) := 'M4V1Y9'</pre>
	Explicit constants for the char (n) type use single quotes as in 'M4V1Y9', as opposed to explicit string constants which use double quotes, as in "Nancy". A single character single quoted character, such as 'A', is considered to have the type char instead of char (n), but since these two types can be assigned to each other and compared to each other, this fact has little consequence.
	The type char (<i>n</i>) is generally more difficult to use than the string type, which is favored for most simple programs. The type char (<i>n</i>) has the advantage that it is efficient in terms of both space and time. In particular, it is represented as <i>n</i> bytes in the computer's memory. By contrast, the string type must use extra space (a trailing zero byte in current implementations) to represent the current length and allocates space for the maximum value it can hold.
	The form of <i>numberOfCharacters</i> is one of:
	 (a) expn % Integer value (b) * % Only in subprogram parameters
	The first form determines n . If the expression is a run time value, the type is considered to be <i>dynamic</i> char (n). The value of n must be at least 1. The second form is used only for subprogram parameters and uses the length of the actual parameter. This too, is considered to be a <i>dynamic</i> char (n) type. Dynamic char (n)

types can only be passed to char(*) parameters. Dynamic char(n) types have the same restrictions as dynamic arrays. This implies they cannot be assigned as a whole and cannot appear in record and union types.

An implementation may impose a limit, recommended to be at least 32767, on the length *n*.

Details Values of the **char**(*n*) type can be assigned and they can be compared for both equality and for ordering, but only if they have the same length and they are not dynamic (i.e. the length must be known at compile time).

Values of the **char**(*n*) type can be read and written by **get** and **put** statements.

The **char**(n) type is a nonscalar, which implies that its parameters are always passed by reference (by means of an implicit pointer).

As is true for the **char** type, all 256 possible values of an 8-bit byte are allowed for each character in **char**(n) type. There is no pattern left to be used for the "initialized value", so there is no uninitialized checking for **char**(n).

In general, you can freely intermix the values of the types **char**, **char**(n) and **string**. This means that catenation (+), comparisons, **length** and substrings can be applied to any of these types. See **catenation** and **substring**. If two non dynamic **char**(n) values (or **char** values) are catenated, the result is a **char**(n)value. If either are dynamic, it is a **string** value. This implies that very long **char**(n) values can be created by catenating them together, for example to initialize a **char**(n) variable.

A **char** value can be assigned (or passed to an non **var** parameter) with automatic conversion to a **char**(1) variable and vice versa. String values of length 1 can be assigned to **char** variables. Character (**char**) values can be assigned to string variables, yielding a string of length 1. String values of length *n* are assignable with automatic conversion to **char**(*n*) variables. Values of type **char**(*n*) can be assigned with automatic conversion to

string variables.

When comparing two **char**(n) values, as in s > t, if both are nondynamic and of the same length, they are compared without converting to strings. If either are dynamic, they are converted to strings and then compared.

the char type which is much like char(1). See also the string type.

char

Syntax	char
Description	Each variable whose type is a char contains a single character, such the letter <i>A</i> , the digit <i>3</i> or the special character &.
	Count characters until a period is found. Character c is read using ϵ statement and is compared to the explicit character constant '.'.
	var c : char
	var counter := 0
Example	loop
	exit when eof <pre>get c</pre> % Read a single character
	exit when $c = '.'$ % Single quotes for char
	counter := counter + 1
	end loop
	put counter, " characters before the period"

Execute

Count capital letters. This example illustrates the use of the **char** ty as the subscript type for the *frequency* array, the use of character variable *c* as a subscript, and the use of *d* as a **for** counter that rang across the letters A to Z.

```
var frequency : array 'A' .. 'Z' of nat
for d : 'A' .. 'Z'
    frequency (d) := 0
end for
loop % Tabulate use of capital le
    exit when eof
    var c : char
    get c % Read one character
    if c >= 'A' and c <= 'Z' then</pre>
```

```
frequency (c) := frequency (c) + 1
end if
end loop
for d : 'A' .. 'Z' % Print frequency of cap
put d, " ", frequency (d)
end for
```

Execute

The type **string** (or **char**(n)) is used instead of **char** when more that one character needs to be stored, such as the string of characters H_{ℓ} . Unless the program needs to be quite efficient, it is usually easier to the **string** type. See also the **char**(n) type, which always stores exa n characters.

The **char** type differs from the **string**(1) type in the following way **char** always represents exactly one character, while **string**(1) can represent either the null string or a string containing one character. **char** type is similar to the **char**(1) type in that both contain at most character.

The **char** type is an index type and can be used, for example, as subscripts, **for** ranges and **case** labels. For example, this declaration

var charCounts : array char of int

creates an array whose subscripts are characters.

The **char** type is a scalar type, which implies that its parameters are passed by value, instead of by reference (which is the case for **cha**) and **string**).

Details

Values of the **char** type can be assigned and they can be compared both equality and ordering. Explicit **char** constants are written as a character surrounded by single quotes, for example, 'A'. For details including how to write control characters, see *explicitCharConstan*

Characters can be read and written by **get** and **put** statements.

There are 256 **char** values, corresponding to the distinct patterns ir 8-bit byte. This allows the patterns *eos* (internal value 0) and *unini* (internal value 128) to be **char** values (these patterns are not allow the **string** type; see the **string** type). All 256 patterns are used, so t is no pattern left to be the "uninitialized value". Uninitialized check is not done for the **char** type.

The **ord** and **chr** functions convert between the **char** values and the corresponding numeric representation in a byte. See **ord** and **chr**.

In general, you can freely intermix the values of the types **char**, **char**(n) and **string**. This means that catenation (+), comparisons, **length** and substrings can be applied to any of these types. See **cha** for details about conversions between **char**, **char**(n) and **string**.

string and char(n) for related types. See ord and chr functions for
conversion from and to ASCII values. See explicitCharConstant fo
information on special characters like caret (^) and backslash (\).

cheat

Syntax

type cheating

Dangerous

A *typeCheat* is one of:

- (a) **cheat** (*targetType*, *expn* [: *sizeSpec*])
- (b) # *expn*
- (c) *id* : **cheat** *typeSpec*

A type cheat interprets the representation (bits) of one type as anoth type. Type cheats are dirty (machine-dependent) and sometimes dangerous (arbitrary corruption) and should be used only by programmers who know the underlying computer representation of values.

Description

Form (b) is a short form type cheat in which the target type is a nat number.

Form (c) is used as a parameter in a subprogram declaration. It cau whatever is passed in to the parameter to be interpreted as *typeSpec*

The character 'B' is assigned to variable *i*, whose type is considered be **char** (although it is really **int1**).

var i : int1 % One byte integer
cheat (char, i) := 'B'

Example This assignment is equivalent (on byte oriented computers) to eithe the following:

```
i := cheat (int1, 'B')
i := ord ('B')
```

The form of *targetType* must be one of:

(a) [*id* .] *typeId*

(b) int, int1, int2 or int4

- (c) **nat**, **nat1**, **nat2** or **nat4**
- (d) **boolean**
- (e) **char** [(*numberOfCharacters*)]
- (f) **string** [(maximumLength)]
- (g) addressint

In form (a) the beginning identifier *id* must be the name of a modul monitor or class that exports the *typeId*. Each of *numberOfCharact* and *maximumLength* must be compile time integer expressions.

If the *expn* in a type cheat is a variable reference and the *sizeSpec* i omitted, the type cheat is considered to be a variable whose type is *targetType*. This allows, for example, the type cheat to be assigned as in:

Details

cheat (**char**, *i*) := 'B'

If the *expn* is a value that is not a variable reference, or if *sizeSpec* present, the type cheat is an expression value whose type is *target*?

The *sizeSpec* is a compile time integer expression giving the size o *expn*'s value. It can be specified only for integer or natural number values (where it must be 1, 2 or 4) or real values (where it must be 8).

A type cheat is carried out in two steps. The first step converts the if necessary to the size given by *sizeSpec*. The second step, which involves no generated code, interprets the value as the target type.

The prefix operator # is a short form for a class of type cheats. It interprets its argument as a natural number. In general, # *expn* is the same as **cheat** (**nat***n*, *expn*) where *n* is determined as follows. If the *expn* is a variable or expression of size 1, 2 or 4, *n* is the size of the item, otherwise *n* is 4.

Set the second character of *d* so it has the numeric representation 2 general, if *c* is a character, then #c = ord(c). Note that #c can have number value assigned to it, but ord(c) cannot.

Example

	var d : char (3) #d (2) := 24 % Same as d(2) := chr(24)
Example	The notation 16#FFFF means FFFF in base 16, which is 32767 in l 10 and is 16 1's in a row in base 2. This same pattern is the two's complement representation of the value -1 in a 2-byte integer.
	var i : int2 #i := 16#FFFF % Equivalent to i := -1
	The following example prints out a string located at addressint <i>my</i> .
Example	<pre>procedure PrintString (str : cheat string) put str end PrintString</pre>
	var myAddr : addressint % Assigned a value to myAddr PrintString (myAddr) % myAddr will be treateo
Details	An implementation may prohibit certain type cheats. Memory alignment requirements may render some type cheats unfeasible. It dangerous to consider a value to have a <i>targetType</i> larger than the value's type. An implementation may prohibit certain type cheats o register scalar items.
See also	<i>explicitIntegerConstant</i> (for description of constants such as 16#FF and the following functions that convert one type to another in a machine-independent manner: ord , chr , intstr , strint , natstr , and strnat .

checked

compiler directive

Unchecked means that certain run time tests, which take place by default, can be eliminated, usually to make the program more efficient at the risk of unreliability. The keyword **checked**, used as a statement, requests that the disabling of checking, previously

Description as a statement, requests that the disabling of checking, previous requested by the keyword **unchecked**, be re-enabled. See **unchecked** for details and an example.

Syntax **chr** (*i* : **int**) : **char**

The **chr** function is used to convert an integer to a character. The character is the *i*-th character of the ASCII sequence of characters (except on the IBM mainframe, which uses the EBCDIC sequence.) For example, **chr** (65) is "A".

Description

The **ord** function is the inverse of **chr**, so for any character *c*,:

chr(ord(c)) = c.

Execute

DetailsThere are two sets of characters representing values 128-255: the
ANSI values (the default values for Windows), and the MS-DOS
OEM character set, which contains various line drawing
characters. To use the MS-DOS character set, you must use the
setscreen ("msdos") command.

Another example is available that displays the entire characters set, both the regular ANSI character set and the MS_DOS OEM character set.

Example

chr

See also ord, **intstr** and **strint** functions.

class

Pointer *p* is used to locate individual objects of the class. The **new** statement creates one of these objects. The statement

p -> push ("Harvey")

is a short form for:

stackClass (p) . push ("Harvey")

This inserts the string *Harvey* into the stack object located by *p*.

A classDeclaration is:

[monitor]
class id
 [inherit inheritItem]
 [implement implementItem]
 [implement by implementByItem]
 [import [var] importItem {, [var] importIten
 [export [howExport] id {, [howExport] id }]
 statementsAndDeclarations
end id

Description A class declaration defines a template for a package of variables, consubprograms, etc. The name of the class (*id*) is given in two places just after **end**. Items declared inside the class can be accessed outsi if they are exported. Items from outside the class that are to be used to be imported (unless they are predefined or pervasive). Instances are created using the **new** statement. Each object is essentially a monometer.

This class is a template for creating objects, each of which is a stac **module** description for the corresponding module that implements strings.)

Syntax

class stackClass % Template for creating indi export push, pop **var** *top* : **int** := 0 var contents : array 1 .. 100 of string procedure push (s : string) top := top + 1contents (top) := s Example end push procedure pop (var s : string) s := contents (top) top := top - 1end pop end stackClass var p: pointer to stackClass % Short form: va % Short form: **new** p **new** stackClass, p p -> push ("Harvey") **var** name : **string** $p \rightarrow pop (name)$ % This sets name to be H

Execute

The **new** statement is used to create objects of a class. Many instan exist at a given time, each located by a pointer. The **free** statement objects that are no longer of use. Turing does not support *garbage* (automatic recovery of space belonging to inaccessible objects).

See **modules** for a discussion of importing, exporting and related c object is created by **new**, its initialization code is executed. In this *top* variable is set to 0. As is true in modules, an exported subprogr class cannot be called until the object is completely initialized.

You are not allowed to create variables of a class, as in:

var s : stack % Not legal!

If the **monitor** keyword is present (just before **class**), the objects at means that only one process at a time can be active in the object. S **process**.

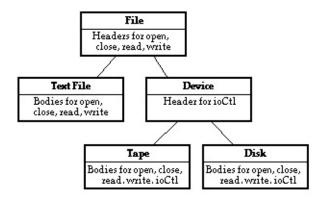
Details Inherit lists are used to specify inheritance. See **inherit** list. Implen by lists provide a special kind of expansion which supports the sep interface from its implementation. See **implement** list and **implem** cannot contain both an inherit and an implement list.

Class declarations can be nested inside modules and monitors but c inside other classes or inside procedures or functions. A class must as one of its (outermost) declarations. A **return** statement cannot b (outermost) statements in a class.

A class cannot export variables (or run time constants) as **unqualif** object has a distinct set of variables).

The syntax of a *classDeclaration* presented above has been simplif **pre**, **invariant** and **post** clauses. The full syntax which supports **pr post** is the same as that for modules. The initialization of classes is modules. See **module**.

We will give an example in which a subprogram in one class overricorresponding subprogram in a class that is being inherited. The exprogram that implements a file system inside an operating system. *close, read* and *write* operations. Some files, called *Device* files, all called *ioCtl* (input/output control). The kind of file determines the imethod. Here is the expansion (inheritance) hierarchy among the class of the expansion (inheritance) hierarchy among the expansion (inheritance) hierarchy among the expansion (inheritance) hierarchy among the expansio



The class called *File* gives the interface to all possible kinds of file implements files that are text (ASCII characters). The *Device* class

to all files that have the *ioCtl* operation in addition to *open*, *close*, *r Tape* and *Disk* classes implement files that are actually physical tap the declaration of the *File* class:

```
class File
    export open, close, read, write
    deferred procedure open (... parameters for op
    deferred procedure close (... parameters for c
    deferred procedure read (... parameters for re
    deferred procedure write (... parameters for w
    end File
```

The *TextFile* class implements the *File* interface by giving variable procedure bodies for ASCII files:

textFilePtr -> read (... actual parameters ...) % F

The *Device* class adds the *ioCtl* procedure to the *File* interface.

class Device inherit File export ioCtl deferred procedure ioCtl (... parameters for i end Device

The *Disk* class provides data and procedures to implement a file the (the *Tape* class is analogous):

class Disk

Example

A pointer that can locate any kind of *File* object is declared this wa

var filePtr : ^ File

This may locate, for example, a *TextFile*:

filePtr := textFilePtr

This assignment is allowed because *filePtr*'s corresponding class (*I* of *textFilePtr*'s corresponding class (*TextFile*). It is guaranteed that located by *filePtr* supports a version of all the operations of a *File* (and *write*).

When we call a procedure in the object located by *filePtr*, the actuation will depend upon the object:

```
filePtr -> read (... actual parameters ...)
```

For example, if *filePtr* currently locates a *Disk* file, this will call the from the *Disk* class. This is an example of *dynamic binding* in whic *read* to be used is selected at run time and this choice is based on the *filePtr*. This is called *polymorphism*, because *File* objects can have form.

As another example, consider class C, which contains headers and f and g. C exports functions f and g. There is also a class D, which f class D contains a body that overrides the body for g. D also contained body for function h. D exports function h.

Pointer *p* has been declared to locate an object of class *C*, but at run object of class *D*. When p is used to call *f*, by means of p->*f*, the bc appears in *C*, is invoked. When p is used to call *g*, by means of p-> body in *D* is invoked. Any attempt to use *p* to call *h* is illegal becau

```
class C
                             export f, g
                             procedure f
                                  put "C's f"
                             end f
Example
                             procedure g
                                  put "C's q"
                             end g
                        end C
                        class D
                             inherit C % Inherit f and g
                             body procedure g % Overrides g in C
                                 put "*** D's q ***"
                             end g
                             procedure h
                                 put "*** D's h ***"
                             end h
                        end D
                        var p : pointer to C % p can point to any des
                        new D, p% p locates an object of class Lp -> f% Outputs "C's f"p -> g% Outputs "*** D's g ***"p -> h% Causes error "'h' is not in ex
```

used to call functions that are exported from *C*.

Execute

See also

module, monitor and **unit**. See also **import** list, **export** list, **imple implement by** list, and **inherit** list. See also **deferred** subprogram, and **objectclass**.

clock	millisecs used procedure
Syntax	clock (var c : int)
Description	The clock statement is used to determine the amount of time since program (process) started running. Variable <i>c</i> is assigned the number of milliseconds since the program started running.
Example	This program tells you how much time it has used. <pre>var timeRunning : int clock (timeRunning) put "This program has run ", timeRunning, " mill</pre>

Details	On Apple Macintoshes, the hardware resolution of duration is in units of 17 milliseconds (1/60-th of a second).
	delay, time, sysclock, wallclock and date statements.
See also	See also predefined unit <u>Time</u> .

file statement

A closeStatement is:

close : *fileNumber*

Description In Turing, files are read and written using a *fileNumber*. In most cases, this number is given a value using the **open** statement, whicl translates a file name, such as "Master", to a file number, such as 5 When the program is finished using the file, it disconnects from the file using the **close** statement.

This program illustrates how to open, read and then close a file.

Example	<pre>var fileName : string := "Master" % Name of fi var fileNo : int % Number of file var inputVariable : string (100) open : fileNo, fileName, read</pre>
-	 read : fileNo, inputVariable
	 close : fileNo

In a Turing implementation, there will generally be a limit on the number of currently open files. This limit will typically be around . To avoid exceeding this limit, a program that uses many files one after another should close files that are no longer in use.

Details If a program does not close a file, the file will be automatically closed when the program finishes.

There is an older and still acceptable version of **close** that has this syntax:

close (fileNumber : int)

See also the open, get, put, read, write, seek and tell statements.

close

Syntax

clear screen	graphics	procedure
--------------	----------	-----------

Syntax	cls
Description	The cls (clear screen) procedure is used to blank the output window. The cursor is set to the top left (to row 1, column 1).
	The entire output window is set to the current text background color (as set by colorback or Text.ColorBack).
Details	The output window must be in " <i>graphics</i> " mode. See setscreen for details.
See also	See also predefined unit Text .

collection

declaration

A collectionDeclaration is one of:

Syntax(a) var id { , id } : collection of typeSpec(b) var id { , id } : collection of forward typeId

A collection declaration creates a new collection (or collections).
 A collection can be thought of as an array whose elements are dynamically created (by new) and deleted (by free). Elements of a collection are referred to by the collection's name subscripted by a pointer. See also new, free and pointer.

Create a collection that will represent a binary tree.

```
var tree : collection of
    record
        name : string (10)
        left, right : pointer to tree
    end record

var root : pointer to tree
new tree, root
tree (root) . name := "Adam"
```

The statement "**new** C,p" creates a new element in collection C and sets p to point at i. If there is no more memory space for the element, though, p is set to nil(C), which is the null pointer for collection C. The statement "**free** C,p" deletes the element of C pointed to by p and sets p to nil(C). In each case, p is passed as a **var** parameter and must be a variable of the pointer type of C.

The keyword **forward** (form b above) is used to specify that the *typeId* of the collection elements will be given later in the collection's scope. The later declaration must appear at the same level (in the same list of declarations and statements) as the original declaration. This allows cyclic collections, for example, when a collection contains pointers to another collection, which in turn contains pointers to the first collection. In this case, the

Example

typeId is the name of the type that has not yet been declared; *typeId* cannot be used until its declaration appears. A collection whose element type is **forward** can be used only to declare pointers to it until the type's declaration is given.

Suppose pointer q is equal to pointer p and the element they point to is deleted by "**free** C,p". We say q is a *dangling pointer* because it seems to locate an element, but the element no longer exists. A dangling pointer is considered to be an uninitialized value. It cannot be assigned, compared, used as a collection subscript, or passed to **free**.

Details

Collections cannot be assigned, compared, passed as parameters, bound to, or named by a **const** declaration. Collections must not be declared in procedures, functions, records or unions.

The same short forms for classes can be also used for collections. These include omission of the collection name in **new**, **free** and **nil** together with the ^ and -> notations. Pointers to types (see **pointer**) can also be used, which are often more convenient to use than collections.

The syntax of a *collectionDeclaration* presented above has been simplified by leaving out **unchecked** collections. With this feature, a *collectionDeclaration* is one of:

(a) var *id* { , *id* } : [unchecked] collection of *typeSpec*(b) var *id* { , *id* } : [unchecked] collection of forward *typeId*

When **unchecked** is specified, the checking to verify that pointers actually locate elements is removed. This checking is done using a "time stamp" attached to each element and pointer, and making sure that these match with each other. When **unchecked** is specified, the execution is dangerous, but faster and smaller, and the pointers become simply machine addresses (as in C).

color	text color graphics procedure
Syntax	color (Color : int)
Description	The color procedure is used to change the currently active color. This is the color of characters that are to be put on the screen. The alternate spelling is colour .
	This program prints out the message "Bravo" three times, each in a different color.
Example	<pre>setscreen ("graphics") for i : 1 3 color (i) put "Bravo" end for</pre>

Execute

This program prints out a message. The color of each letter is different from the preceding letter. For letter number i the color number is i mod maxcolor + 1. This cycles repeatedly through all the available colors.

Example	setscreen ("graphics")
-	<pre>const message := "Happy New Year!!"</pre>
	<pre>for i : 1 length (message)</pre>
	color (<i>i</i> mod maxcolor + 1)
	put message (i)
	end for

Details	See maxcolor for the number of colors available in the various " <i>graphics</i> " modes. The background color that text appears upon can be set using the colorback procedure.
Details	The screen must be in a " <i>graphics</i> " mode to use the color procedure. See setscreen for details.
	colorback, whatcolor and maxcolor.
See also	See also predefined unit <u>Text</u> .

colorback

background color procedure

Syntax	colorback (Color : int)
Description	The colorback procedure is used to change the color upon which to appears. The alternate spelling is colourback .
	When the message "Greetings" is output, the background surround letter will be inred.
Example	setscreen (" <i>graphics</i> ") colorback (red) put "Greetings"

Details	When a newline is output, such as when the put statement does not a at the end, the rest of the line is cleared from cursor to the right the output window in the text background color.
Example	This program will result in the word "Hello" being displayed in rec green background followed by the word "Again" displayed in red c yellow background. The yellow background stretches across the w: <pre>setscreen ("graphics") color (red) colorback (green) put "Hello " % The "" at the end of line st colorback (yellow) put "Again" % The rest of the line is cleared in</pre>

Execute

This program displays the letter 'X' in 16 different colors on 16 diff backgrounds. Note that the letter is not visible when the text color bacground color are the same

	setscreen ("graphics")
	for row : 0 15
Example	colorback (<i>row</i>)
prc	for column : 0 15
	color (column)
	locate (row, column)
	put "X"`´
	end for
	end for

Details	See maxcolor for the number of background colors available . The which text appears can be set using the color procedure.
	The screen must be in a " <i>graphics</i> " mode to use the color procedur setscreen for details.
	<u>color</u> and <u>whatcolorback</u> .
See also	See also predefined unit <u>Text</u> .

comment	remark statement
Description	A <i>comment</i> is a remark to the reader of the program, which the computer ignores. The most common form of comment in Turing starts with a percent sign (%) and continues to the end of the currer line; this is called an <i>end-of-line</i> comment. There is also the <i>bracketed</i> comment, which begins with the /* and ends with */ and which can continue across line boundaries.
Example	% This is an end-of-line comment var x : real % Here is another end-of-lin const s := "Hello" /* Here is a bracketed comment that lasts for two lines */ const pi := 3.14159
Details	In the BASIC language, comments are called <i>remarks</i> and start wit the keyword REM. In Pascal, comments are bracketed by (* and *)

comparisonOperator

A comparisonOperator is one of:

	(a) <	% Less than
	(b) >	% Greater than
Syntax	(c) =	% Equal
	(d) <=	% Less than or equal; subset
	(e) >=	% Greater than or equal; superset
	(f) not=	% Not equal

A comparison operator is placed between two values to determine their equality or ordering. For example, 7 > 2 is true and so is "Adam" < "Cathy". The comparison operators can be applied to numbers as well as to enumerated types. They can also be applied to strings to determine the *ordering* between strings (see the string type for details). Arrays, records, unions and collections cannot be compared. Boolean values (true and false) can be compared only for equality (= and not=); the same is true of pointer values. Set values can be compared using <= and >=, which are the subset and superset operators. The not= operator can be written as ~=.

Comparisons among classes is also supported (see **class**). If *C* and *D* are classes, $C \le D$ means *D* is a descendant of (inherits from) *C*. See **class**.

See alsoSee also infix operators and precedence of operators. See also the
int, real, string, set, boolean and enum types. See also string
comparison.

Concurrency

	This unit contains the predefined procedures that deal with concurrency. It contains one predefined function, although conceptually it contains three other subprograms.		
Description	All routines in the Concurrency module are exported unqualified. (This means you can call the entry points directly.)		

	<u>empty</u> *	Returns true if no processes are waiting on the condition queue.
	<mark>getpriority</mark> *	Returns the priority of the current process.
	<u>setpriority</u> *	Sets the priority of the current process.
Entry Points	<u>simutime</u>	Returns the number of simulated time units that have passed.

* Part of the language, conceptually part of the **Concurrency** unit.

Concurrency.empty

Syntax	empty (variableReference) : boolean		
Description	The empty function is used in a concurrent program. It returns true if the <i>variableReference</i> , which must be a condition variable, has no processes waiting for it. Processes join the queue of a condition variable by executing the wait statement, and are awakened by the signal statement.		
Status	Part of the language and only conceptually part of the Concurrency unit.		
	This means that you can only call the function by calling empty , not by calling Concurrency.empty .		
See also	condition, wait, signal, fork and monitor.		

Concurrency.getpriority

Part of <u>Concurrency</u> module

Syntax	getpriority : nat		
Description	The getpriority function returns the priority of an executing process in a concurrent program. A smaller value means a faster speed.		
Status	Part of the language and only conceptually part of the Concurrency unit.		
	This means that you can only call the function by calling getpriority , not by calling Concurrency.getpriority .		
See also	setpriority, fork and monitor.		

Concurrency.setpriority

Syntax setpriority (*p* : nat)

Description	The setpriority procedure is used to set the priority of a process in a concurrent program. This priority cannot be counted on to guarantee critical access to shared variables. A smaller value of p means increased speed. The argument to setpriority may be limited to the range 0 to $2^{**}15 - 1$.
	Part of the language and only concentually part of the

Part of the language and only conceptually part of the **Concurrency** unit.

Status This means that you can only call the function by calling setpriority, not by calling Concurrency.setpriority.

See also getpriority, fork and monitor.

Concurrency.simutime

Syntax	simutime : int	
Description	The simutime function returns the number of simulated time units that have passed since program execution began.	
Details	Simulated time only passes when all process are either paused or waiting. This simulates the fact that CPU time is effectively infinitely faster than "pause" time.	
Example	This prints out the simulated time passing between two processes. This will print out 3, 5, 6, 9, 10, 12, 15, 15, 18, 20, 21, process p (t : int) loop pause t put simutime end loop end p fork p (3) fork p (5)	

Execute

Exported unqualified.

StatusThis means that you can call the function by calling simutime or
by calling Concurrency.simutime.

condition

A conditionDeclaration is:

Syntax

var id { , id } : [array indexType {, indexType } of
]

[conditionOption] condition

A condition is essentially a queue of sleeping processes. It is used in a concurrent program to allow processes to block themselves (by the wait statement) and later to be awakened (by the signal statement). A condition variable, which can occur only inside a monitor (a special kind of module that handles concurrency) or monitor class, is used by the wait and signal statements for putting processes to sleep and later waking them up.

The processes use this monitor to gain exclusive access to a resource. A process wanting to use the resource calls the *request* entry point and is blocked until the resource is free. When the process is finished with the resource, it calls the *release* entry point. This monitor is essentially a binary *semaphore* in which the semaphore's *P* operation is the *request* and the *V* is the *release*.

	resource ort request, release
	available : boolean := true nowAvailable : condition
pro	cedure request if not available then
	wait nowAvailable % Go to sleep end if
	assert available available := false % Allocate resou
Example end	request
pro	cedure release
	assert not available% Resource is alavailable := true% Free the resousignal nowAvailable % Wake up one proces

```
% If any are sleeping
end release
end resource
process worker
loop
....
resource.request
.... use resource ...
resource.release
end loop
end worker
fork worker
fork worker
% Activate one worker
% Activate another worke
```

Execute

A conditionOption is one of:

- (a) **priority**
- (b) **deferred**
- (c) timeout

The **priority** option requires that the corresponding **wait** statements include priorities. Options (b) and (c) declare *deferred* conditions. A signal to a deferred condition causes the signaled process to become ready to enter the monitor when the monitor becomes inactive. The signaling process continues running in the monitor. A signal to an *immediate* (non deferred) condition causes the signaled process to begin running in the monitor immediately. The signaling process waits to re-enter the monitor when the monitor becomes inactive. All conditions in a device monitor must be deferred (or **timeout**).

A **timeout** option means the signaling is deferred and that an

Details	extra parameter to the wait statement must give a <i>timeout interval</i> . If a process waits longer than its interval, it is automatically signaled. Beware that the <i>empty</i> function can be non-repeatable when applied to timeout conditions. For example, empty (c) may not be equal to empty (c) in a single expression. In the current (1999) version of Turing, the time for time outs is measured in simulation time rather than real time. See the pause statement.
	Conditions cannot be named as types, cannot be contained in records, unions or collections and cannot be declared in statements (such as begin or loop) or in subprograms. They can only be declared in monitors and monitor classes.
	There is no guaranteed order of progress among awakened deferred processes, processes signaling immediate conditions, and processes attempting to enter an active monitor.
	Note that <i>conditionOption</i> must precede the keyword condition .
See also	wait and signal. See also monitor and fork. See also empty. See also pause.

Config

Description	This unit contains the predefined subprograms that deal with getting configuration information about the machine and environment on which the program is being run. It exists in order to allow users to obtain information about the system that may only be available at run time.		
	All routines in the Config module are exported qualified (and thus must be prefaced with " Config. ").		
	<u>Display</u>	Returns information about the display currently attached.	
Entry Points	Lang	Returns information about the language environment that the program is currently running within.	
	<u>Machine</u>	Returns information about the hardware on which the program is running.	

Config.Display

Syntax **Config.Display** (*displayCode* : int) : int

Config.Display returns information about the display (or displays) the computer. The parameter *displayCode* determines what sort of passed back. *displayCode* has a number of possible values, all sum set of predefined constants.

At the time of this writing, the following constants were defined:

Description	cdScreenHeight	return the height of the screen in pixels.
	cdScreenWidth	return the width of the screen in pixels.
	cdMaxNumColors	return the maximum number of colors suppo display.
	cdMaxNumColours	return the maximum number of colors suppo display.

Execute

This program prints the screen width and height.

Example	<pre>const width : int := Config.Display (cdScreenWic const height: int := Config.Display (cdScreenHei put "The screen width is ", width, " the screer</pre>	
Details	On the Macintosh, it's possible to have multiple displays attached t computer. To get information about the extra displays, you can call Config.Display with any of the first four constants above plus one etc. This will return the height, width or maximum number of color second, third and beyond displays.	

This program prints the screen width and height of the second disp Macintosh.

Example

const width : int := Config.Display (cdScreenWic const height: int := Config.Display (cdScreenHei put "The second display size is ", width, " x "

Exported qualified.

Status This means that you can only call the function by calling **Config.D** calling **Display**.

Part of **Config** module

Config.Lang

Syntax Config.Lang (langCode : int) : int

Config.Lang returns information about the language and the limita of the implementation that the program is currently running. The parameter *langCode* determines what sort of information is passed *langCode* has a number of possible values, all summarized by a set predefined constants.

At the time of this writing, the following constants were defined:

	clRelease	return the current release number of the environment (e.g. $4.02 = 400$).			
Description	clLanguageVersion	return the current version number of the language (e.g. $1.81 = 181$).			
	clMaxNumStreams	return the maximum number of I/O stre (used by the open and close statements) can be opened at once.			
	clMaxNumDirStreams	return the maximum number of director streams that can be opened at once.			
	clMaxNumRunTimeArgs	return the maximum number of run-time arguments.			
	This program prints the current environment version.				
Example	const <i>version</i> : int := Config.Lang (<i>clLanguageV</i> & put "The language version number is ", <i>version</i>				

Exported qualified.

Status This means that you can only call the function by calling **Config.L** not by calling **Lang**.

Config.Machine

Part of Config module

Syntax	Config.Machine (machineCode : int) : int			
	Config.Machine returns information about the machine that the program is currently running on. The parameter <i>machineCode</i> determines what sort of information is passed back. <i>machineCode</i> has a number of possible values, all summarized by a set of predefined constants.			
Description	At the time of this writing, the following constants were defined:			
	cmProcessor cmFPU cmOS	return an encoding of the processor number. return 1 if there is an FPU installed, 0 if not. return the current version number of the operating system (e.g. $6.07 = 607$).		
	This program prints whether the machine has an FPU or not.			
Example	<pre>if Config.Machine (cmFPU) = 1 then put "The machine has an FPU installed" else put "The machine does not have an FPU insta end if</pre>			
	Exported quali	ified.		
Status		at you can only call the function by calling i ne , not by calling Machine .		

constant declaration

const

A constantDeclaration is:

Syntax const *id* [: *typeSpec*] := *initializingValue*

Description A const declaration creates a name *id* for a value.

	const $c := 3$
	const s := "Hello" % The type of s is strin
	$const \times := sin (y) ** 2$
Example	<pre>const a : array 13 of int := init (1, 2, 3)</pre>
-	<pre>const b : array 13 of int := a</pre>
	const <i>c</i> : array 12, 12 of int := init (1, 2,
	% So c(1,1)=1, c(1,2)=2, c(2,1)=3, c(2,2

The initializing value can be an arbitrary value or else a list of item commas inside **init** (...). The syntax of *initializingValue* is:

- a. expn
- b. **init** (initializingValue, initializingValue)

Each **init** (...) corresponds to an array, record or union value that is These must be nested for initialization of nested types. In the Pasca must have values known at compile time; Turing has no such restri

When the typeSpec is omitted, the variable's type is taken to be the initializing expression, for example, **int** or **string**. The typeSpec ca dynamic arrays or when the initializing value is of the form **init** (... **init** (...) must be known at compile time.

Details

The keyword **pervasive** can be inserted just after **const**. When this is visible inside all subconstructs of the constant's scope. Without **p** is not visible inside modules, monitors or classes unless explicitly i constants need not be imported. You can abbreviate **pervasive** as a

You can also optionally use the **register** keyword to request that th a machine register. The syntax for constantDeclaration is actually:

```
const [pervasive] [register] id [ : typeSpec ] :
```

In the current (2002) implementation, programs are run interpretive which has no machine registers, and the **register** keyword is ignore for restrictions on the use of register constants.

constantReference

use of a constant

A constantReference is:

Syntax constantId { componentSelector }

In a Turing program, a constant is declared and given a name (*constantId*) and then used. Each use is called a *constant reference*.

If the constant is an array, record or union, its parts (*components*) can be selected using subscripts and field names (using *componentSelectors*). The form of a *componentSelector* is one of:

Description

(a) (*expn* {, *expn*})(b) . *fieldId*

Form (a) is used for subscripting (indexing) arrays. The number of array subscripts must be the same as in the array's declaration

of array subscripts must be the same as in the array's declaration. Form (b) is used for selecting a field of a **record** or **union**. Component selectors are used in the same manner as variable references. See *variableReference* for details. See also **const** declaration and *explicitConstant*.

- 1	<pre>var radius : real const pi := 3.14159</pre>	% Constant declaration
Example	put "Area is: ", <i>pi</i>	* radius **2 % pi is a constant reference

cosine function (radians)

Syntax **cos** (*r* : **real**) : **real**

Description The **cos** function is used to find the cosine of an angle given in radians. For example, **cos** (0) is 1.

This program prints out the cosine of p/6, 2p/6, 3p/6, up to 12p/6 radians.

T 1	const <i>pi</i> := 3.14159
Example	for <i>i</i> : 1 12
	const angle := i * pi / 6
	<pre>put "Cos of ", angle, " is ", cos (angle)</pre>
	end for

Execute

	the sin and tan functions for calculating sine and tangent.
See also	the <u>cosd</u> function which finds the cosine of an angle given in degrees. (2p radians are the same as 360 degrees.)
	See also predefined unit Math.

cosd	cosine function (degrees)
Syntax	cosd (r : real) : real
Description	The cosd function is used to find the cosine of an angle given in degrees. For example, cosd (0) is 1.
	This program prints out the cosine of 30, 60, 90, up to 360 degrees.
Example	<pre>for i : 1 12 const angle := i * 30 put "Cos of ", angle, " is ", cosd (angle) end for</pre>

Execute

the sind and tand functions for calculating sine and tangent.
 the cos function which finds the cosine of an angle given in radians. (2p radians are the same as 360 degrees.)
 See also predefined unit Math.

procedure

Syntax	date (var d : string)	
Description	The date statement is used to determine the current date. Variable <i>d</i> is assigned a string in the format " <i>dd mmm yy</i> ", where <i>mmm</i> is the first 3 characters of the month, e.g., " <i>Apr</i> ". For example, if the date is Christmas 2002, <i>d</i> will be set to " <i>25 Dec 02</i> ".	
	This program greets you and tells you the date.	
Example	var today : string date (<i>today</i>) put "Greetings!! The date today is ", <i>today</i>	

Execute

Details	Be warned that Turing gets the date from the operating system. If the date is is not correctly set in the operating system, then the date procedure will give incorrect results.		
	delay, clock, sysclock, wallclock and time statements.		
See also	See also predefined unit Time .		

date

declaration

create a variable

A *declaration* is one of:

- (a) variableDeclaration
- (b) constantDeclaration
- (c) *typeDeclaration*
- (d) bindDeclaration
- (e) procedureDeclaration
- (f) *functionDeclaration*
- (g) moduleDeclaration
- (h) classDeclaration
- (i) processDeclaration
- (j) monitorDeclaration
- (k) conditionDeclaration

A *declaration* creates a new name (or names) for a variable, constant, type, procedure, function, module, class, process, monitor, or condition. These names are called *identifiers*, where *id* is the abbreviation for *identifier*.

Example	var width : int const pi := 3.14159 type range : 0 150	% Variable declaration % Constant declaration % Type declaration
Lampic	<pre>procedure greet put "Hello world" end greet</pre>	% Procedure declaration

Ordinarily, each new name must be distinct from names that are already visible; that is, redeclaration is not allowed. There are certain exceptions to this rule, for example, names of parameters and fields of records can be the same as existing visible variables. Variables declared inside a subprogram (a procedure and function) are allowed to be the same as variables global to

Syntax

(outside of) the subprogram.

The effect of a declaration (its *scope*) lasts to the end of the construct in which the declaration occurs; this will be the end of the program, the **end** of the surrounding procedure, function or module, the **end** of a **loop**, **for**, **case** or **begin** statement, or the end of the **then**, **elsif**, or **else** clause of an **if** statement, or the end of the **case** statement alternative.

A name must be declared before it can be used; this is called the *DBU* (*Declaration Before Use*) rule. The exceptions to this rule use the keyword **forward**, as in **import** lists and in **collection** declarations.

A *declaration* can appear any place a *statement* can appear. This differs from the Pascal language, in which declarations are allowed only at the beginning of the program or at the beginning of a procedure or function. Each declaration can optionally be followed by a semicolon (;).

There are certain restrictions on the placement of declarations. Procedures and functions cannot be declared inside other procedures and functions nor inside statements (for example, not inside an **if** statement). A **bind** declaration cannot appear at the outer level of either the main program or a module. A **condition** declaration can appear only inside a monitor. Processes cannot be declared inside procedures, functions, monitors or classes. Classes cannot be declared inside classes. However, modules and monitors can be declared inside classes and vice versa. Monitors can be declared inside modules, not vice versa.

Details

deferred

subprogram declaration

A deferredDeclaration is:

Syntax deferred subprogramHeader

Description A procedure or function is declared to be **deferred** when you want to be able to override the subprogram in an expansion. The procedure or function must be in a module, monitor or class.

The *display* procedure is deferred in this class of stacks to allow various ways of graphically displaying the stack on the screen:

class stack
 export push, pop
 ... local declarations ...
 ... declarations of the push and pop procedure
 deferred procedure display (howbig : int)
end stack

An expansion to the *stack* class can give a body for *display*, as in:

Example

class stackWithSimpleDisplay
 body procedure display % (howbig : int)
 ... graphically display the stack on th
 end display
end stackWithSimpleDisplay

The following creates a stack that can be displayed and displays it:

var p : ^stackWithSimpleDisplay
new p
...
p -> display (25) % Display the stack on t

A deferred procedure is *resolved* by giving its body. This can be done in the scope (**module**, **monitor** or **class**) containing the **deferred** declaration (following the **deferred** declaration) or in any expansion of that scope. Only one resolution per scope is allowed. Unresolved subprograms can be called, but they immediately abort

All exported subprograms are implicitly deferred and can be

Details overridden in expansions.

During initialization of a **module**, **monitor** or **object** of a **class**, deferred subprograms (including exported subprograms) cannot be called. This restriction prevents accessing an object before it is full initialized.

A **deferred** declaration must not appear in the main program.

module, monitorand class. See also exportlist, importSee alsoinheritlist, implementlist and implement by

delay

Syntax	delay (<i>duration</i> : int)
Description	The delay statement is used to cause the program to pause for a given time. The time duration is in milliseconds.
	This program prints the integers 1 to 10 with a second delay between each.
Example	for i : 1 10 put i delay (1000) % Pause for 1 second end for

Execute

Details	On Apple Macintoshes, the hardware resolution of duration is in units of 17 milliseconds (1/60th of a second). For example, delay (500) will delay the program by about half a second, but may be off by as much as 17 milliseconds.	
	sound, clock, sysclock, wallclock, time and date statements.	
See also	See also predefined unit <u>Time</u> .	

Dir

Description	This unit contains the predefined subprograms that deal with directories. You can use these subprograms to list the contents of directories, create directories, change directories and return the current directory.		
		in the Dir module are exported qualified (and thus faced with " Dir. ").	
	<u>Open</u>	Opens a directory stream in order to get a listing of the directory contents.	
	<u>Get</u>	Gets the next file name in the directory listing.	
	<u>GetLong</u>	Gets the next file name and other information in the directory listing.	
Entry	<u>Close</u>	Closes the directory stream.	
Points	<u>Create</u>	Creates a new directory.	
	<u>Delete</u>	Deletes a directory (must be empty).	
	<u>Change</u>	Changes the current execution directory.	
	<u>Current</u>	Returns the current execution directory.	
	<u>Exists</u>	Returns whether a directory exists.	
	slashes or ba Dir.Change	in Turing can be expressed using either forward ackslashes. For example, ("d:\\turing files\\assignment 3") or	
Details	Dir.Change ("d:/turing files/assignment 3") are both legal. Note that backslashes must be doubled in string literals. The "." directory represent the current directory (as in Dir.Open (".")), and the "" directory represents the parent directory. For example, if the current execution directory is "d:/turing files/assignment 3", then Dir.Change ("") changes the current execution directory to "d:/turing files".		
See also	File unit for more explanation of the different ways of specifying a path name of a file or directory under the different operating systems.		

Dir.Change

Part of <u>Dir</u> module

Syntax	Dir.Change (directoryPathName : string)	
	Dir.Change changes the execution directory to that specified by the parameter <i>directoryPathName</i> . This is the equivalent of doing a cd in UNIX.	
Description	Under Microsoft Windows, specifying a drive in the <i>directoryPathName</i> parameter causes the drive to become the default drive (unlike the DOS cd command).	
Details	If the Dir.Change call fails, then Error.Last will return a non- zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	This program changes to the directory called <i>/usr/west</i> and then lists the current directory.	
Example	<pre>Dir.Change ("/usr/west") if Error.Last = eNoError then put "Directory changed" else put "Did not change the directory." put "Error: ", Error.LastMsg end if put "The current execution directory is ", Dir.C</pre>	
	An example demonstrating the use of several of the Dir subprograms is available. In this example, a directory is created, a file is written to the directory, an attempt is made to delete the directory, the file is deleted, and then the directory is deleted.	
	This example demonstrates the use of Dir.Change , Dir.Current , Dir.Create , and Dir.Delete .	

Example

Exported qualified.

StatusThis means that you can only call the function by calling
Dir.Change, not by calling Change.

Dir.Close	Part of <u>Dir</u> module	
Syntax	Dir.Close (streamNumber : int)	
Description	Dir.Close is part of a series of four subprograms that help users get directory listings. Dir.Close is used to close a directory stream number opened by Dir.Open . After the directory stream number is closed, it can not be used with Dir.Get or Dir.GetLong .	
Details	If the Dir.Close call fails, then Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	This program prints a listing of all the files in the directory <i>datafiles</i> .	
Example	<pre>var streamNumber : int var fileName : string streamNumber := Dir.Open ("datafiles") assert streamNumber > 0 loop fileName := Dir.Get (streamNumber) exit when fileName = "" put fileName end loop Dir.Close (streamNumber)</pre>	

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Dir.Close, not by calling Close.

Dir.Create

Part of <u>Dir</u> module

Syntax	Dir.Create (<i>directoryPathName</i> : string)	
Description	Dir.Create is used to create the directory specified by the parameter <i>directoryPathName</i> . This is the equivalent of doing a mkdir in DOS or UNIX. On the Macintosh, it creates a folder.	
Details	If the Dir.Create call fails, then Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	This program creates the directory called <i>information</i> .	
Example	Dir.Create ("information") if Error.Last = eNoError then put "Directory created" else put "Did not create the directory." put "Error: ", Error.LastMsg end if	
	An example demonstrating the use of several of the Dir subprograms is available. In this example, a directory is created, a file is written to the directory, an attempt is made to delete the directory, the file is deleted, and then the directory is deleted.	
	This example demonstrates the use of Dir.Change , Dir.Current , Dir.Create , and Dir.Delete .	

Example

Exported qualified.

StatusThis means that you can only call the function by calling
Dir.Create, not by calling Create.

Dir.Current

Syntax	Dir.Current : string	
Description	Dir.Current returns the full path name of the current execution directory. This is the equivalent of doing a pwd in UNIX.	
Details	If the Dir.Current call fails, then Error.Last will return a non- zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	This program changes to the directory called <i>/usr/west</i> and then lists the current directory.	
Example	<pre>Dir.Change ("/usr/west") if Error.Last = eNoError then put "Directory changed" else put "Did not change the directory." put "Error: ", Error.LastMsg end if put "The current execution directory is ", Dir.C</pre>	
	An example demonstrating the use of several of the Dir subprograms is available. In this example, a directory is created, a file is written to the directory, an attempt is made to delete the directory, the file is deleted, and then the directory is deleted.	
	This example demonstrates the use of Dir.Change , Dir.Current , Dir.Create , and Dir.Delete .	

Example

Exported qualified.

StatusThis means that you can only call the function by calling
Dir.Current, not by calling Current.

Dir.Delete

Syntax	Dir.Delete (directoryPathName : string)	
Description	Dir.Delete is used to delete the directory specified by the parameter <i>directoryPathName</i> . This is the equivalent of doing a rmdir in DOS or UNIX. On the Macintosh, it removes a folder.	
	Dir.Delete will fail if it attempts delete a directory that has files in it.	
Details	If the Dir.Delete call fails, then Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	This program deletes the directory called <i>information</i> .	
Example	<pre>Dir.Delete ("information") if Error.Last = eNoError then put "Directory delete" else put "Did not delete the directory." put "Error: ", Error.LastMsg end if</pre>	
	An example demonstrating the use of several of the Dir subprograms is available. In this example, a directory is created, a file is written to the directory, an attempt is made to delete the directory, the file is deleted, and then the directory is deleted.	

This example demonstrates the use of **Dir.Change**, **Dir.Current**, **Dir.Create**, and **Dir.Delete**.

Example

Exported qualified.

StatusThis means that you can only call the function by calling
Dir.Delete, not by calling Delete.

Dir.Exists

Syntax	Dir.Exists (directoryPathName : string) : boolean	
Description	Dir.Exists returns true if a directory by the name of <i>directoryPathName</i> exists. It will return false if <i>directoryPathName</i> is a file.	
Details	If the Dir.Exists returns false , you can examine Error.Last or Error.LastMsg for more information (i.e. whether the path failed or the directory was simply not found).	
	The following program determines if the directory "d:/usr/west" exists, and outputs an error message if it does not.	
Example	<pre>if Dir.Exists ("d:/usr/west") then put "Directory exists" else put "Directory does not exists: ", Error.Las end if</pre>	
	Exported qualified.	
Status	This means that you can only call the function by calling Dir.Exists , not by calling Exists .	

Dir.Get

Syntax	Dir.Get (streamNumber : int) : string	
Description	Dir.Get is part of a series of four subprograms that help users get directory listings. Dir.Get is used to get the file names in the directory. Each time the function is called, it returns the next file name in the directory. The names are not sorted. When there are no more file names in the directory, Dir.Get returns the empty string.	
Details	If the Dir.Get call fails, then Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	This program prints a listing of all the files in the directory <i>datafiles</i> .	
Example	<pre>var streamNumber : int var fileName : string streamNumber := Dir.Open ("datafiles") assert streamNumber > 0 loop fileName := Dir.Get (streamNumber) exit when fileName = "" put fileName end loop Dir.Close (streamNumber)</pre>	

Execute

Exported qualified.

Status This means that you can only call the function by calling **Dir.Get**,

not by calling **Get**.

Dir.GetLong

	Dir.GetLong (streamNumber : int, var entryName : string var size, attribute, fileTime : int)	
Syntax		
	Dir.GetLong is part of a series of four subprograms that help users directory listings. Dir.GetLong is used to get the names and assort information of the files in the directory. Each time the function is c returns the name and information of the next file in the directory. T names are not sorted. When there are no more file names in the director. Dir.GetLong returns the empty string in the <i>entryName</i> parameter.	
	The <i>size</i> parameter is the size of the file in bytes. The <i>attribute</i> para has its individual bits set as follows (the individual bits can be extrusing the bits operator):	
	Bit 0 <i>attrDir</i> set to 1 if entry is a directory.	
	Bit 1 <i>attrRead</i> set to 1 if the program can read the file.	
	Bit 2 <i>attrWrite</i> set to 1 if the program can write the file.	
Description	Bit 3 <i>attrExecute</i> set to 1 if the program can execute the file.	
	Bit 4 <i>attrHidden</i> set to 1 if the entry if a hidden file (PC, Mac).	
	Bit 5 <i>attrSystem</i> set to 1 if the entry is a system file (PC only).	
	Bit 6 <i>attrVolume</i> set to 1 if the entry is a volume name (PC only).	
	Bit 7 <i>attrArchive</i> set to 1 if the entry has archive bit set (PC only).	
	The <i>attr</i> constants are defined in the Dir unit. They correspond to values of <i>attribute</i> if a specified bit is set. For example, <i>attrSystem</i> value of the <i>attribute</i> parameter if bit 5 is set to 1. You can and or constants to get combinations of specific file attributes.	
	The <i>fileTime</i> is the time of last modification of the file. It is returne number of seconds since 00:00:00 GMT 1/1/1970. To convert this	

string, use **Time.SecDate**

DetailsIf the Dir.GetLong call fails, then Error.Last will return a non-zeiindicating the reason for the failure. Error.LastMsg will return a s
which contains the textual version of the error.

This program prints a listing of all the files in the directory datafile

var streamNumber : int
var fileName : string
var size, attribute, fileTime : int
streamNumber := Dir.Open ("datafiles")
assert streamNumber > 0
loop
Dir.GetLong (streamNumber, fileName, size, a
exit when fileName = ""
put fileName, " ", Time.SecDate (fileTime)
end loop
Dir.Close (streamNumber)

This program prints a listing of the attributes of all the files in the c directory.

```
var streamNumber : int
                    var fileName : string
                    var size, attribute, fileTime : int
                    streamNumber := Dir.Open (Dir.Current)
                    assert streamNumber > 0
                    1000
                        Dir.GetLong (streamNumber, fileName, size, a
                        exit when fileName = ""
                        put fileName, " "...
                        if (attribute and attrDir) not= 0 then
                            put "Directory "...
Example
                        end if
                        if (attribute and attrRead) not= 0 then
                            put "Readable "..
                        end if
                        if (attribute and attrWrite) not= 0 then
                            put "Writeable "...
                        end if
                        if (attribute and attrExecute) not= 0 then
                            put "Executable "...
                        end if
                        put ""
                    end loop
```

Dir.Close (streamNumber)

Execute

Exported qualified.

StatusThis means that you can only call the function by calling **Dir.GetL**
by calling **GetLong**.

Dir.Open

Syntax	Dir.Open (directoryPathName : string) : int	
Description	Dir.Open is part of a series of four subprograms that help users get directory listings. Dir.Open returns a directory stream number if the directory could be opened. This stream number can be used to get file names and information using the Dir.Get and Dir.GetLong subprograms. After getting the listing, the user should call Dir.Close .	
Details	If the Dir.Open call fails, then Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	This program prints a listing of all the files in the current directory.	
Example	<pre>var streamNumber : int var fileName : string streamNumber := Dir.Open (Dir.Current) assert streamNumber > 0 loop fileName := Dir.Get (streamNumber) exit when fileName = "" put fileName end loop Dir.Close (streamNumber)</pre>	

Execute

Exported qualified.

Status This means that you can only call the function by calling

Dir.Open, not by calling **Open**.

integer truncating division operator	
div	
The div operator divides one number by another and produces the integer result, truncated in the direction of zero. For example, 7 div 2 produces 3 and -7 div 2 produces -3.	
In this example, <i>eggCount</i> is the total number of eggs. The first put statement determines how many dozen eggs there are. The second put statement determines how many extra eggs there are beyond the last dozen. var eggCount : int get eggCount put "You have ", eggCount div 12, " dozen eggs" put "You have ", eggCount mod 12, " left over"	

Execute

See also *infix operators, <u>precedence</u> of operators and the <u>mod</u> operator.*

Draw

Entry Points This unit contains the predefined subprograms that deal with drawing pixel graphics to the screen.

Description

All routines in the **Draw** unit are exported qualified (and thus must be prefaced with "**Draw.**").

<u>Cls</u>	Clears the screen to color 0.
Dot	Draws a dot.
<u>Line</u>	Draws a line.
DashedLine	Draws a dashed or dotted line.
ThickLine	Draws a thick line.
Box	Draws a box.
FillBox	Draws a filled box.
<u>Oval</u>	Draws an oval.
FillOval	Draws a filled oval.
Arc	Draws an arc.
<u>FillArc</u>	Draws a filled arc or a wedge.
<u>Polygon</u>	Draws a polygon.
<u>FillPolygon</u>	Draws a filled polygon.
MapleLeaf	Draws a maple leaf.
<u>FillMapleLeaf</u>	Draws a filled maple leaf.
<u>Star</u>	Draws a star.
<u>FillStar</u>	Draws a filled star.
<u>Fill</u>	Does a flood fill.
<u>Text</u>	Draws text as graphics

Draw.Arc

Draw.Arc (*x*, *y*, *xRadius*, *yRadius* **: int**, *initialAngle*, *finalAngle*, *Color* **: int**)

Syntax

Description

The **Draw.Arc** procedure is used to draw an arc whose center is at (*x*, *y*). This is just like **Draw.Oval**, except that you must also give two angles, *initialAngle* and *finalAngle*, which determine where to start and stop drawing. Zero degrees is "three o'clock", 90 degrees is "twelve o'clock", etc. The horizontal and vertical distances from the center to the arc are given by *xRadius* and *yRadius*.



This program draws a quarter circle whose center is (*midx*, *midy*) the center of the screen, using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen.

Example

```
View.Set ("graphics")
const midx := maxx div 2
const midy := maxy div 2
Draw.Arc (midx, midy, midx, midy, 0, 90, 1)
```

Execute

DetailsThe screen must be in a "*graphics*" mode. See the **View.SetDetails**procedure for details. If the screen is not in a "*graphics*" mode, an
error will occur

Exported qualified.

StatusThis means that you can only call the function by calling
Draw.Arc, not by calling Arc.

View.Set, maxx, maxyand the various procedures in the DrawSee alsounit.

Part of **Draw** module

Syntax Draw.Box (*x*1, *y*1, *x*2, *y*2, *Color* : **int**)

The **Draw.Box** procedure is used to draw a box on the screen with bottom left and top right corners of (x1, y1) to (x2, y2) using the specified *Color*.

Description



This program draws a large box, reaching to each corner of the screen using color number 12. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The point (0,0) is the left bottom of the screen and (**maxx**, **maxy**) is the right top.

View.Set ("graphics")
Draw.Box (0, 0, maxx, maxy, 12)

Execute

Example

Details	The screen must be in a " <i>graphics</i> " mode. See the View.Set procedure for details. If the screen is not in a " <i>graphics</i> " mode, an error will occur.
	Exported qualified.
Status	This means that you can only call the function by calling Draw.Box , not by calling Box .
	View.Set, maxx, maxy and the various procedures in the Draw

Draw.Box

See also unit.

Draw.Cls

Syntax	Draw.Cls	
Description	The Draw.Cls (clear screen) procedure is used to blank the output window The cursor is set to the top left (to row 1, column 1).	
Details	The Draw.Cls procedure sets all pixels in the output window to color 0. In this way it differs from the cls and Text.Cls procedures which set the screen to the text background color.	
	The screen must be in " <i>graphics</i> " mode. See View.Set for details.	
	Exported qualified.	
Status	This means that you can only call the function by calling Draw.Cls , not by calling Cls .	
See also	<u>View.Set</u> , <u>maxx</u> , <u>maxy</u> and the various procedures in the <u>Draw</u> unit.	

Draw.DashedLine

Description

Syntax Draw.DashedLine (*x*1, *y*1, *x*2, *y*2, *lineStyle*, *Color* : **int**)

The **Draw.DashedLine** procedure is used to draw a dotted or dash on the screen from (x1, y1) to (x2, y2) using the specified *Color*.

There are five possible line styles: drawSolid Draws a solid line (sa Draw.Line) drawDash Draws a dashed line drawDot Draws a dotte drawDashDot Draws a line that alternates dashes and dots drawDashDotDot Draws a line that alternates dash and dot-dot



This program draws a large X, reaching to each corner of the scree two different colors. The **maxx** and **maxy** functions are used to det the maximum x and y values on the screen. The point (0,0) is the le bottom of the screen, (**maxx**, **maxy**) is the right top, etc.

Example	View.Set ("graphics") % Draw a line in each of the styles.
	Draw.DashedLine (0, 50, maxx, 50, drawSolid, bri
	Draw.DashedLine (0, 100, maxx, 100, drawDash, br
	Draw.DashedLine (0, 150, maxx, 150, drawDot, bri
	Draw.DashedLine (0, 200, maxx, 200, drawDashDot,
	Draw.DashedLine (0, 250, maxx , 250, drawDashDotE

Execute

DetailsThe screen must be in a "*graphics*" mode. See the **View.Set** proced
details. If the screen is not in a "*graphics*" mode, an error will occu

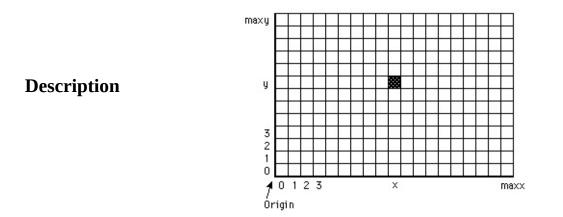
Exported qualified.

- StatusThis means that you can only call the function by calling
Draw.DashedLine, not by calling DashedLine.
- See also <u>View.Set</u>, <u>maxx</u>, <u>maxy</u> and the various procedures in the <u>Draw</u> un

Draw.Dot

Syntax Draw.Dot (*x*, *y*, *Color* : **int**)

The **Draw.Dot** procedure is used to color the dot (pixel) at location (*x*, *y*) using the specified *Color*.



This program randomly draws dots with random colors. The **maxx**, **maxy** and **maxcolor** functions give the maximum x, y and color values.

Example View.Set ("graphics") var x, y, c : int loop x := Rand.Int (0, maxx) % Random x y := Rand.Int (0, maxy) % Random y c := Rand.Int (0, maxcolor) % Random color Draw.Dot (x, y, c) end loop

Execute

The screen must be in a "graphics" mode. If the screen is not in a

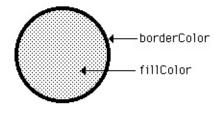
Details

"graphics" mode, an error will occur. See View.Set for details.
 Exported qualified.
 Status This means that you can only call the function by calling Draw.Dot, not by calling Dot.
 See also View.Set, maxx, maxy and the various procedures in the Draw unit.

Syntax Draw.Fill (*x*, *y* : int, *fillColor*, *borderColor* : int)

The **Draw.Fill** procedure is used to color in a figure that is on the screen. Starting at (x, y), the figure is filled with *fillColor* to a surrounding border whose color is *borderColor*.

Description



This program draws an oval with x and y radius of 10 in the center of the screen in bright green. Then the oval is filled with red. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen.

Example

View.Set ("graphics")
const midx := maxx div 2
const midy := maxy div 2
Draw.Oval (midx, midy, 10, 10, brightgreen)
Draw.Fill (midx, midy, red, brightgreen)

Execute

Details	The screen must be in a " <i>graphics</i> " mode. See the View.Set procedure for details. If the screen is not in a " <i>graphics</i> " mode, an error will occur.		
	Exported qualified.		
Status	This means that you can only call the function by calling		

Draw.Fill, not by calling Fill.

Draw.FillArc

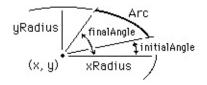
Syntax

Description

Draw.FillArc (*x*, *y*, *xRadius*, *yRadius* **: int**, *initialAngle*, *finalAngle*, *Color* : **int**)

The **Draw.FillArc** procedure is used to draw a filled arc whose center is at (*x*, *y*). It then fills in the pie-shaped wedge using the specified *Color*. To outline a filled arc, use **Draw.FillArc** with the *Color* parameter set to the fill color and then **Draw.Arc** with the *Color* parameter set to the border color. For *initialAngle* and *finalAngle*, which determine the edges of the wedge, zero degrees is "three o'clock" and 90 degrees is "twelve o'clock", etc. The horizontal and vertical distances from the center to the arc are

given by *xRadius* and *yRadius*.



This program draws a filled semicircle (actually, an approximation to a semicircle) whose center is (*midx*,0) the bottom center of the screen, in bright red. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen.

View.Set ("graphics")
const midx := maxx div 2
Draw.FillArc (midx, 0, maxy, maxy, 0, 180, brigh

Execute

Example

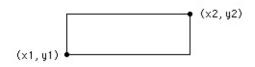
The screen must be in a "graphics" mode. See the View.Set

Details	procedure for details. If the screen is not in a " <i>graphics</i> " mode, a error will occur.			
	Exported qualified.			
Status	This means that you can only call the function by calling Draw.FillArc , not by calling FillArc .			
See also	View.Set, maxx, maxy and the various procedures in the Draw unit.			

Draw.FillBox

Syntax Draw.FillBox (*x*1, *y*1, *x*2, *y*2, *Color* : **int**)

The **Draw.FillBox** procedure is used to draw a filled box on the screen with bottom left and top right corners of (*x*1, *y*1) to (*x*2, *y*2) filled using the specified *Color*. To get a box outlined in a different color, use **Draw.FillBox** with the *Color* parameter set to the fill color and then call **Draw.Box** with the *Color* parameter set to the border color.



This program will fill the bottom half of the screen with color 1 and then outline it in color 2. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The point (0,0) is the left bottom of the screen and (**maxx**, **maxy**) is the right top.

View.Set ("graphics")
Draw.FillBox (0, 0, maxx, maxy div 2, 1)
Draw.Box (0, 0, maxx, maxy div 2, 2)

Execute

Example

DetailsThe screen must be in a "graphics" mode. See the View.Set
procedure for details. If the screen is not in a "graphics" mode, an
error will occur.

Exported qualified.

StatusThis means that you can only call the function by calling
Draw.FillBox, not by calling FillBox.

Draw.FillMapleLeaf

Syntax Draw.FillMapleLeaf (*x*1, *y*1, *x*2, *y*2, *Color* : **int**)

The **Draw.FillMapleLeaf** procedure is used to draw a filled maple leaf on the screen bounded by a rectangle with bottom left and top right corners of (*x*1, *y*1) to (*x*2, *y*2) and filled using the specified *Color*. To get a maple leaf outlined in a different color, use **Draw.FillMapleLeaf** with the *Color* parameter set to the fill color and then call **Draw.MapleLeaf** with the *Color* parameter set to the border color. If *y*1 is greater than *y*2, then the mapleleaf is drawn upside down.

(x1,y1) (x1,y1) (x1,y1) (x2,y2) (x2,y2)

This program will draw two maple leaves beside each other. The first will be outlined in black and filled in brightred. The second maple leaf will be upside down and both filled and outlined in green.

Example

Description

View.Set ("graphics")
Draw.FillMapleLeaf (0, 0, 100, 100, brightred)
Draw.MapleLeaf (0, 0, 100, 100, black)
Draw.FillMapleLeaf (150, 100, 250, 0, green)

Execute

The **Draw.FillMapleLeaf** procedure is useful for drawing the

Canadian flag.

Details The screen should be in a "graphics" mode. See the View.Set procedure for details. If the screen is not in a "graphics" mode, it will automatically be set to "graphics" mode.
Exported qualified.
Status This means that you can only call the function by calling Draw.FillMapleLeaf, not by calling FillMapleLeaf.
See also View.Set, maxx, maxy and the various procedures in the Draw unit.

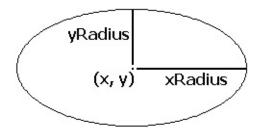
Draw.FillOval

Syntax

Draw.FillOval (*x*, *y*, *xRadius*, *yRadius*, *Color* : int)

The **Draw.FillOval** procedure is used to draw a filled oval whose center is at (x, y). The horizontal and vertical distances from the center to the oval are given by *xRadius* and *yRadius*. To get an oval outlined in a different color, use **Draw.FillOval** with the *Color* parameter set to the fill color and then call **Draw.Oval** with the *Color* parameter set to the border color.

Description



This program draws a large filled oval that just touches each edge of the screen using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The center of the oval is at (*midx*, *midy*), which is the middle of the screen.

Example

```
View.Set ("graphics")
const midx := maxx div 2
const midy := maxy div 2
Draw.FillOval (midx, midy, midx, midy, 1)
```

Execute

Ideally, a circle is drawn when *xRadius* = *yRadius*.

Details The screen must be in a "*graphics*" mode. See the **View.Set**

procedure for details. If the screen is not in a "*graphics*" mode, am error will occur.

Exported qualified.

StatusThis means that you can only call the function by calling
Draw.FillOval, not by calling FillOval.

Draw.FillPolygon

Syntax	Draw.FillPolygon (<i>x</i> , <i>y</i> : array 1 * of int , <i>n</i> : int , <i>Color</i> : int)		
Description	The Draw.FillPolygon procedure is used to draw a filled polygon with <i>n</i> points. The polygon is described by the points ($x(1)$, $y(1)$) to ($x(2)$, $y(2)$) to ($x(3)$, $y(3)$) and so on to ($x(n)$, $y(n)$). The polygon will be drawn and filled with <i>Color</i> .		
	To get an polygon outlined in a different color, use Draw.FillPolygon with the <i>Color</i> parameter set to the fill color and then call Draw.Polygon with the <i>Color</i> parameter set to the border color.		
Example	This program will create a filled octagon and display it in bright blue and then outline it in cyan.		
	<pre>View.Set ("graphics") var x : array 18 of int := init (100, 100, 135</pre>		

Execute

	The PC allows a maximum of 256 points. As well, Draw.FillPolygon can fail (due to lack of memory). If failure occurs, it will try to draw an outline of the polygon. If that also		
Details	fails, it will not draw anything. The screen must be in a " <i>graphics</i> " mode. See the View.Set		
	procedure for details. If the screen is not in a " <i>graphics</i> " mode, an		

error will occur.

Exported qualified.

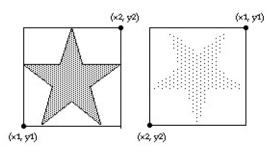
- StatusThis means that you can only call the function by calling
Draw.FillPolygon, not by calling FillPolygon.
- View.Set, maxx, maxyand the various procedures in the DrawSee alsounit.

Draw.FillStar

Syntax Draw.FillStar (x1, y1, x2, y2, Color : int)

The **Draw.FillStar** procedure is used to draw a filled five pointed star on the screen bounded by a rectangle with bottom left and top right corners of (x1, y1) to (x2, y2) and filled using the specified *Color*. To get a star outlined in a different color, use **Draw.FillStar** with the *Color* parameter set to the fill color and then call **Draw.Star** with the *Color* parameter set to the border color. If y1 is greater than y2, then the star is drawn upside down.

Description



This program will draw two stars beside each other. The first will be outlined in color 1 and filled in color 2. The second star will be upside down and both filled and outlined in color 3.

Example

View.Set ("graphics")
Draw.FillStar (0, 0, 100, 100, brightred)
Draw.Star (0, 0, 100, 100, green)
Draw.FillStar (150, 100, 250, 0, brightblue)

Execute

The **Draw.FillStar** procedure is useful for drawing the American flag.

Details	The screen must be in a " <i>graphics</i> " mode. See the View.Set procedure for details. If the screen is not in a " <i>graphics</i> " mode, error will occur.		
	Exported qualified.		
Status	This means that you can only call the function by calling Draw.FillStar , not by calling FillStar .		
See also	View.Set, maxx, maxy and the various procedures in the Draw unit.		

Draw.Line

Part of **Draw** module

Syntax Draw.Line (*x*1, *y*1, *x*2, *y*2, *Color* : **int**)

The **Draw.Line** procedure is used to draw a line on the screen from (x1, y1) to (x2, y2) using the specified *Color*.

Description



This program draws a large X, reaching to each corner of the screen using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The point (0,0) is the left bottom of the screen, (**maxx**, **maxy**) is the right top, etc.

Example

View.Set ("graphics")
% First draw a line from the left bottom to righ
Draw.Line (0, 0, maxx, maxy, 1)
% Now draw a line from the left top to right bot
Draw.Line (0, maxy, maxx, 0, 1)

Execute

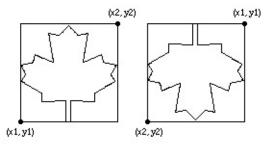
DetailsThe screen must be in a "graphics" mode. See the View.Set
procedure for details. If the screen is not in a "graphics" mode, an
error will occur.Exported qualified.StatusThis means that you can only call the function by calling
Draw.Line, not by calling Line.

Draw.MapleLeaf

Syntax Draw.MapleLeaf (*x*1, *y*1, *x*2, *y*2, *Color* : **int**)

The **Draw.MapleLeaf** procedure is used to draw a maple leaf on the screen bounded by a rectangle described by the bottom left and top right corners of (x1, y1) to (x2, y2) using the specified *Color*. If y1 is greater than y2, then the maple leaf is drawn upside down.

Description



This program will draw two maple leaves beside each other. The first will be in red and the second maple leaf will be upside down and in yellow.

Example

View.Set ("graphics")
Draw.MapleLeaf (0, 0, 100, 100, red)
Draw.MapleLeaf (150, 100, 250, 0, yellow)

Execute

The Draw.MapleLeaf procedure is useful for drawing the Canadian flag.
Details The screen must be in a "*graphics*" mode. See the View.Set procedure for details. If the screen is not in a "*graphics*" mode, an error will occur.

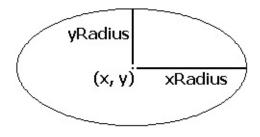
Exported qualified.

StatusThis means that you can only call the function by calling
Draw.MapleLeaf, not by calling MapleLeaf.

Syntax Draw.Oval (*x*, *y*, *xRadius*, *yRadius*, *Color* : **int**)

The **Draw.Oval** procedure is used to draw an oval whose center is at (*x*, *y*). The horizontal and vertical distances from the center to the oval are given by *xRadius* and *yRadius*.

Description



This program draws a large oval that just touches each edge of the screen in magenta. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The center of the oval is at (*midx*, *midy*), which is the middle of the screen.

Example

```
View.Set ("graphics")
const midx := maxx div 2
const midy := maxy div 2
Draw.Oval (midx, midy, midx, midy, magenta)
```

Execute

	Ideally, a circle is drawn when <i>xRadius</i> = <i>yRadius</i> . In fact, the aspect ratio (the ratio of height to width of pixels displayed on the
	screen) of the IBM PC compatibles is not 1.0, so this does not draw a true circle. In CGA graphics mode this ratio is 5 to 4.
Details	The screen must be in a " <i>graphics</i> " mode. See the View.Set procedure for details. If the screen is not in a " <i>graphics</i> " mode, an

error will occur.

Exported qualified.

- StatusThis means that you can only call the function by calling
Draw.Oval, not by calling Oval.
- View.Set, maxx, maxyand the various procedures in the DrawSee alsounit.

Draw.Polygon

Part of **Draw** module

Syntax	Draw.Polygon (x, y : array 1 * of int , n : int , <i>Color</i> : int)		
Description	The Draw.Polygon procedure is used to draw a polygon with <i>n</i> points. A line is drawn in <i>Color</i> from the point $(x(1), y(1))$ to $(x(2), y(2))$ to $(x(3), y(3))$ and so on. After drawing the line to $(x(n), y(n))$, a line will be drawn back to $(x(1), y(1))$, closing the polygon. The Draw.Polygon procedure is equivalent to:		
-	<pre>for i : 1 n - 1 Draw.Line (x (i), y(i), x (i + 1), y (i + 1) end for Draw.Line (x (n), y (n), x (1), y (1), Color)</pre>		
	This program will create an octagon and display it in color 1.		
Example	View.Set ("graphics") var x : array 18 of int := init (100, 100, 135 220, 220, 185, 135) var y : array 18 of int := init (100, 150, 185 150, 100, 65, 65) Draw.Polygon (x, y, 8, brightblue)		

Execute

The IBM PC limits **Draw.Polygon** to a maximum of 256 points.

Details The screen must be in a "*graphics*" mode. See the **View.Set** procedure for details. If the screen is not in a "*graphics*" mode, an error will occur.

Exported qualified.

Status This means that you can only call the function by calling

Draw.Polygon, not by calling Polygon.

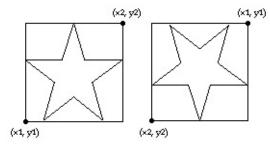
Draw.Star

Part of **Draw** module

Syntax Draw.Star (*x*1, *y*1, *x*2, *y*2, *Color* : **int**)

The **Draw.Star** procedure is used to draw a star on the screen bounded by a rectangle described by the bottom left and top right corners of (x1, y1) to (x2, y2) using the specified *Color*. If y1 is greater than y2 then the star is drawn upside down.

Description



This program will draw two stars beside each other. The first star will be in color 1 and the second star will be upside down and in color 2.

Example

View.Set ("graphics")
Draw.Star (0, 0, 100, 100, brightred)
Draw.Star (150, 100, 250, 0, brightblue)

Execute

The **Draw.Star** procedure is useful for drawing the American flag.

Details The screen must be in a "*graphics*" mode. See the **View.Set** procedure for details. If the screen is not in a "*graphics*" mode, an error will occur.

Exported qualified.

StatusThis means that you can only call the function by calling
Draw.Star, not by calling Star.

Draw.Text

Syntax	Draw.Text (<i>txtStr</i> : string , <i>x</i> , <i>y</i> , <i>fontID</i> , <i>Color</i> : int)			
Description	Draw.Text is used to actually draw text in a specified font. The <i>tex</i> parameter contains the string to be drawn. The <i>x</i> and <i>y</i> parameters as location of the lower left hand corner of the text to be displayed. The parameter is the number of the font in which the text is to be drawn parameter is used to specify the color in which the text is to appear			
	Note that the text that appears is completely unrelated to the text th using put . Draw.Text is a graphics command and thus does not use the cursor location.			
	The text drawn by the Draw.Text procedure does not erase the bac			
Details	If Draw.Text is passed an invalid font ID, a fatal error occurs. If th Draw.Text call fails for other (non-fatal) reasons then Error.Last non-zero value indicating the reason for the failure. Error.LastMs a string which contains the textual version of the error.			
Details	Draw.Text is identical to Font.Draw . It is placed here for consiste other pixel graphics drawing routines.			
	The program draws a phrase in red surrounded by a box in bright b			
Example	<pre>var font : int font := Font.New ("serif:12") assert font1 > 0 var width : int:= Font.Width ("This is in a seri var height, ascent, descent, internalLeading : i Font.Sizes (font, height, ascent, descent, inter Draw.Text ("This is in a serif font", 50, 30, fc Draw.Box (50, 30 + descent, 50 + width, 30 + hei Font.Free (font)</pre>			

Execute

Detaile	To use the same font as is used by the put statement, use defFontI font number. This font does not have to be greated or freed by the
Details	font number. This font does not have to be created or freed by the u allows a program to quickly place text in any location on the scree
	The program draws two strings in the default font (defFontID).

Example	Draw.Text	("Drawing He	ere", 100,	120, defFontID ,
-	Draw.Text	("and Here"	, 180, 90,	defFontID, brigh

Execute

Exported qualified.

- Status This means that you can only call the function by calling Draw.Te: calling Text.
- **See Also** Font module for more information about selecting the font to be di

Draw.ThickLine

Part of **Draw** module

Syntax Draw.ThickLine (*x*1, *y*1, *x*2, *y*2, *lineWidth*, *Color* : **int**)

The **Draw.ThickLine** procedure is used to draw a line on the scree to (*x*2, *y*2) using the specified *Color*.

Description



This program draws a large X, reaching to each corner of the scree different colors. The **maxx** and **maxy** functions are used to determ x and y values on the screen. The point (0,0) is the left bottom of th (**maxx**, **maxy**) is the right top, etc.

Example

View.Set ("graphics")
% First draw a line of 5 pixel width from the le
Draw.ThickLine (0, 0, maxx, maxy, 5, brightred)
% Now draw a line of 5 pixel width from the left
Draw.ThickLine (0, maxy, maxx, 0, 5, brightgreer

Execute

See also	View.Set, maxx, maxy and the various procedures in the Draw un		
Status	This means that you can only call the function by calling Draw.Th calling ThickLine .		
	Exported qualified.		
Details	The screen must be in a " <i>graphics</i> " mode. See the View.Set proced the screen is not in a " <i>graphics</i> " mode, an error will occur.		

drawarc

Description

Syntax

graphics procedure

The **drawarc** procedure is used to draw an arc whose center is at (*x*, *y*). This is just like **drawoval**, except that you must also give two angles, *initialAngle* and *finalAngle*, which determine where to start and stop drawing. Zero degrees is "three o'clock", 90 degrees is "twelve o'clock", etc. The horizontal and vertical distances from the center to the arc are given by *xRadius* and *yRadius*.



This program draws a semicircle (actually, an approximation to a semicircle) whose center is (*midx*,0) the bottom center of the screen, using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen.

Example

setscreen ("graphics")
const midx := maxx div 2
drawarc (midx, 0, maxy, maxy, 0, 180, 1)

Execute

The meaning of the *Color* number depends on the current palette. See the **palette** statement.

Details The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it

will automatically be set to "*graphics*" mode.

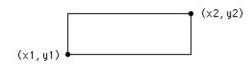
See also setscreen, **maxx**, **maxy** and the various **draw...** procedures.

graphics procedure

Syntax drawbox (x1, y1, x2, y2, Color : int)

The **drawbox** procedure is used to draw a box on the screen with bottom left and top right corners of (x1, y1) to (x2, y2) using the specified *Color*.

Description



This program draws a large box, reaching to each corner of the screen using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The point (0,0) is the left bottom of the screen and (**maxx**, **maxy**) is the right top.

```
setscreen ("graphics")
drawbox (0, 0, maxx, maxy, 1)
```

Execute

Example

The meaning of the *Color* number depends on the current palette. See the **palette** statement.

Details The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it will automatically be set to "*graphics*" mode.

See also setscreen, **maxx**, **maxy** and the various **draw...** procedures.

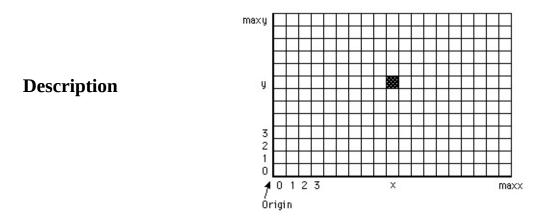
drawbox

graphics procedure

drawdot

Syntax drawdot (*x*, *y*, *Color* : **int**)

The **drawdot** procedure is used to color the dot (pixel) at location (*x*, *y*) using the specified *Color*.



This program randomly draws dots with random colors. The **maxx**, **maxy** and **maxcolor** functions give the maximum x, y and color values.

Example setscreen ("graphics") var x, y, c : int loop randint (x, 0, maxx) % Random x randint (y, 0, maxy) % Random y randint (c, 0, maxcolor) % Random color drawdot (x, y, c) end loop

Execute

The screen should be in a "*graphics*" mode. If the screen is not in

Details a "*graphics*" mode, it will automatically be set to "*graphics*" mode. See **setscreen** for details.

This program draws a line of dots that bounce off the "sides" of the screen. It also demonstrates that you can use real coordinates to store position (x, y), as long as you convert the coordinates to **int** values using **round** in the **drawdot** call.

```
var x, y : real
                     var dx, dy : real
                     var clr : int := 1
                     x := \text{Rand.Int} (1, \text{maxx} - 1)
                     y := Rand.Int (1, maxy - 1)
                     dx := Rand.Real - 0.5
                     dy := Rand.Real - 0.5
Example
                     100p
                         drawdot (round (x), round (y), clr)
                         clr := (clr + 1) \mod \max
                         x := x + dx
                         y := y + dy
                         if x \le 0 or x \ge maxx then
                             dx := -dx
                         end if
                         if y \le 0 or y \ge maxy then
                             dy := -dy
                         end if
                     end loop
```

Execute

See also setscreen, maxx, maxy and the various draw... procedures.

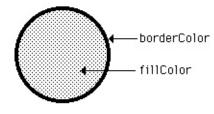
drawfill

graphics procedure

Syntax drawfill (*x*, *y* : **int**, *fillColor*, *borderColor* : **int**)

The **drawfill** procedure is used to color in a figure that is on the screen. Starting at (x, y), the figure is filled with *fillColor* to a surrounding border whose color is *borderColor*.

Description



This program draws an oval with x and y radius of 10 in the center of the screen using color 1. Then the oval is filled with color 2. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen.

Example

setscreen ("graphics")
const midx := maxx div 2
const midy := maxy div 2
drawoval (midx, midy, 10, 10, 1)
drawfill (midx, midy, 2, 1)

Execute

Details

The meaning of the *Color* number depends on the current palette; see the **palette** statement.

The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it will automatically be set to "*graphics*" mode.

Warning: In Turing for IBM PC compatibles, **drawfill** fails to completely fill in some complicated figures that contain "islands" within them surrounded by the *borderColor*.

See also setscreen, **maxx**, **maxy** and the various **draw...** procedures.

drawfillarc

Syntax

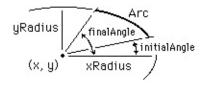
Description

graphics procedure

drawfillarc (x, y, xRadius, yRadius : **int**, initialAngle, finalAngle, Color : **int**)

The **drawfillarc** procedure is used to draw a filled arc whose center is at (x, y). It then fills in the pie-shaped wedge using the specified *Color*. To outline a filled arc, use **drawfillarc** with the *Color* parameter set to the fill color and then **drawarc** with the *Color* parameter set to the border color. For *initialAngle* and *finalAngle*, which determine the edges of the wedge, zero degrees is "three o'clock" and 90 degrees is "twelve o'clock", etc. The horizontal and vertical distances from the center to the arc are

given by *xRadius* and *yRadius*.



This program draws a filled semicircle (actually, an approximation to a semicircle) whose center is (*midx*,0), the bottom center of the screen, using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen.

Example

setscreen ("graphics")
const midx := maxx div 2
drawfillarc (midx, 0, maxy, maxy, 0, 180, 1)

Execute

On the PC, drawfillarc fills the pie-shaped wedge by using a

See also	setscreen, maxx, maxy and the various draw procedures.
	The screen should be in a " <i>graphics</i> " mode. See the setscreen procedure for details. If the screen is not in a " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode.
Details	The meaning of the <i>Color</i> number depends on the current palette. See the palette statement.
	"flood" fill and is thus subject to all the conditions of a flood fill.

drawfillbox

graphics procedure

Syntax drawfillbox (x1, y1, x2, y2, Color : int)

The drawfillbox procedure is used to draw a filled box on the
screen with bottom left and top right corners of (x1, y1) to (x2, y2)
filled using the specified *Color*. To get a box outlined in a
different color, use drawfillbox with the *Color* parameter set to
the fill color and then call drawbox with the *Color* parameter set
to the border color.Description



This program will fill the bottom half of the screen with color 1 and then outline it in color 2. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The point (0,0) is the left bottom of the screen and (**maxx**, **maxy**) is the right top.

setscreen ("graphics")
drawfillbox (0, 0, maxx, maxy div 2, 1)
drawbox (0, 0, maxx, maxy div 2, 2)

Execute

Example

The meaning of the *Color* number depends on the current palette. See the **palette** statement. **Details** The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it will automatically be set to "*graphics*" mode.

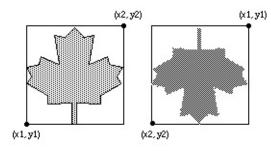
drawfillmapleleaf

graphics procedure

Syntax drawfillmapleleaf (*x*1, *y*1, *x*2, *y*2, *Color* : int)

The **drawfillmapleleaf** procedure is used to draw a filled maple leaf on the screen bounded by a rectangle with bottom left and top right corners of (x1, y1) to (x2, y2) and filled using the specified *Color*. To get a maple leaf outlined in a different color, use **drawfillmapleleaf** with the *Color* parameter set to the fill color and then call **drawmapleleaf** with the *Color* parameter set to the border color. If y1 is greater than y2, then the maple leaf is drawn upside down.

Description



This program will draw two maple leaves beside each other. The first will be outlined in color 1 and filled in color 2. The second maple leaf will be upside down and both filled and outlined in color 3.

Example

```
setscreen ("graphics")
drawfillmapleleaf (0, 0, 100, 100, 1)
drawmapleleaf (0, 0, 100, 100, 2)
drawfillmapleleaf (150, 100, 250, 0, 3)
```

Execute

The **drawfillmapleleaf** procedure is useful for drawing the

Canadian flag.

DetailsThe meaning of the *Color* number depends on the current palette.
See the **palette** statement.DetailsThe screen should be in a "graphics" mode. See the **setscreen**
procedure for details. If the screen is not in a "graphics" mode, it
will automatically be set to "graphics" mode.See alsosetscreen, maxx, maxy and the various draw... procedures.

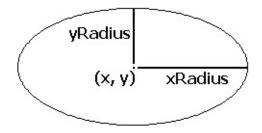
drawfilloval

graphics procedure

Syntax drawfilloval (*x*, *y*, *xRadius*, *yRadius*, *Color* : int)

The **drawfilloval** procedure is used to draw a filled oval whose center is at (x, y). The horizontal and vertical distances from the center to the oval are given by *xRadius* and *yRadius*. To get an oval outlined in a different color, use **drawfilloval** with the *Color* parameter set to the fill color and then call **drawoval** with the *Color* parameter set to the border color.

Description



This program draws a large filled oval that just touches each edge of the screen using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The center of the oval is at (*midx*, *midy*), which is the middle of the screen.

Example

```
setscreen ("graphics")
const midx := maxx div 2
const midy := maxy div 2
drawfilloval (midx, midy, midx, midy, 1)
```

Execute

Ideally, a circle is drawn when xRadius = yRadius. In fact, the aspect ratio (the ratio of height to width of pixels displayed on the screen) of the IBM PC compatibles is not 1.0, so this does not

See also	setscreen, maxx, maxy and the various draw procedures.
	The screen should be in a " <i>graphics</i> " mode. See the setscreen procedure for details. If the screen is not in a " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode.
Details	The meaning of the <i>Color</i> number depends on the current palette. See the palette statement.
	draw a true circle. In CGA graphics mode this ratio is 5 to 4.

drawfillpolygon

graphics procedure

Syntax	drawfillpolygon (x, y : array $1 *$ of int, $n : int$, Color : int)			
Description	The drawfillpolygon procedure is used to draw a filled polygon with <i>n</i> points. The polygon is described by the points ($x(1)$, $y(1)$) to ($x(2)$, $y(2)$) to ($x(3)$, $y(3)$) and so on to ($x(n)$, $y(n)$). The polygon will be drawn and filled with <i>Color</i> .			
	To get an polygon outlined in a different color, use drawfillpolygon with the <i>Color</i> parameter set to the fill color and then call drawpolygon with the <i>Color</i> parameter set to the border color.			
	This program will create a filled octagon and display it in color and then outline it in color 3.			
Example	<pre>setscreen ("graphics") var x : array 18 of int := init (100, 100, 135</pre>			
	drawpolygon (x, y, 8, 3)			

Execute

	The PC allows a maximum of 256 points. As well, drawfillpolygon can fail (due to lack of memory). If failure occurs, it will try to draw an outline of the polygon. If that also fails, it will not draw anything.	
Details	The meaning of the <i>Color</i> number depends on the current palette. See the palette statement.	

The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it will automatically be set to "*graphics*" mode.

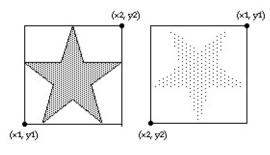
drawfillstar

graphics procedure

Syntax drawfillstar (*x*1, *y*1, *x*2, *y*2, *Color* : int)

The **drawfillstar** procedure is used to draw a filled five pointed star on the screen bounded by a rectangle with bottom left and top right corners of (x1, y1) to (x2, y2) and filled using the specified *Color*. To get a star outlined in a different color, use **drawfillstar** with the *Color* parameter set to the fill color and then call **drawstar** with the *Color* parameter set to the border color. If y1 is greater than y2, then the star is drawn upside down.

Description



This program will draw two stars beside each other. The first will be outlined in color 1 and filled in color 2. The second star will be upside down and both filled and outlined in color 3.

Example

setscreen ("graphics")
drawfillstar (0, 0, 100, 100, 1)
drawstar (0, 0, 100, 100, 2)
drawfillstar (150, 100, 250, 0, 3)

Execute

The **drawfillstar** procedure is useful for drawing the American flag.

DetailsThe meaning of the *Color* number depends on the current palette.
See the **palette** statement.

The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it will automatically be set to "*graphics*" mode.

drawline

graphics procedure

Syntax drawline (*x*1, *y*1, *x*2, *y*2, *Color* : int)

The **drawline** procedure is used to draw a line on the screen from (x1, y1) to (x2, y2) using the specified *Color*.

Description



This program draws a large X, reaching to each corner of the screen using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The point (0,0) is the left bottom of the screen, (**maxx**, **maxy**) is the right top, etc.

Example

setscreen ("graphics")
% First draw a line from the left bottom to righ
drawline (0, 0, maxx, maxy, 1)
% Now draw a line from the left top to right bot
drawline (0, maxy, maxx, 0, 1)

Execute

The meaning of the *Color* number depends on the current palette. See the **palette** statement.

Details The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it will automatically be set to "*graphics*" mode.

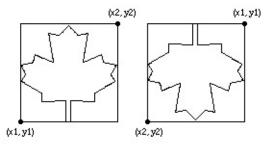
drawmapleleaf

graphics procedure

Syntax drawmapleleaf (*x*1, *y*1, *x*2, *y*2, *Color* : int)

The **drawmapleleaf** procedure is used to draw a maple leaf on the screen bounded by a rectangle described by the bottom left and top right corners of (x1, y1) to (x2, y2) using the specified *Color*. If *y1* is greater than *y2*, then the maple leaf is drawn upside down.

Description



This program will draw two maple leaves beside each other. The first will be in color 1 and the second maple leaf will be upside down and in color 2.

Example

setscreen ("graphics")
drawmapleleaf (0, 0, 100, 100, 1)
drawmapleleaf (150, 100, 250, 0, 2)

Execute

The **drawmapleleaf** procedure is useful for drawing the Canadian flag.

The meaning of the *Color* number depends on the current palette. See the **palette** statement.

Details

The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it will automatically be set to "*graphics*" mode.

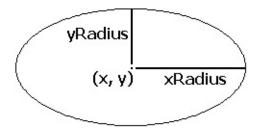
drawoval

graphics procedure

Syntax drawoval (*x*, *y*, *xRadius*, *yRadius*, *Color* : int)

The **drawoval** procedure is used to draw an oval whose center is at (x, y). The horizontal and vertical distances from the center to the oval are given by *xRadius* and *yRadius*.

Description



This program draws a large oval that just touches each edge of the screen using color number 1. The **maxx** and **maxy** functions are used to determine the maximum x and y values on the screen. The center of the oval is at (*midx*, *midy*), which is the middle of the screen.

Example

setscreen ("graphics")
const midx := maxx div 2
const midy := maxy div 2
drawoval (midx, midy, midx, midy, 1)

Execute

Ideally, a circle is drawn when xRadius = yRadius. In fact, the aspect ratio (the ratio of height to width of pixels displayed on the screen) of the IBM PC compatibles is not 1.0, so this does not draw a true circle. In CGA graphics mode this ratio is 5 to 4.

Details The meaning of the *Color* number depends on the current palette.

See the **palette** statement.

The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it will automatically be set to "*graphics*" mode.

drawpic

graphics procedure

Syntax	drawpic (<i>x</i> , <i>y</i> : int , <i>buffer</i> : array 1 * of int , <i>picmode</i> : int)		
Description	The drawpic procedure is used to copy of a rectangular picture onto the screen. The left bottom of the picture is placed at (x, y) . In the common case, the buffer was initialized by calling takepic . The values of <i>picmode</i> are:		
	0: Copy actual picture on screen.1: Copy picture by XORing it onto the screen.		
	XORing a picture onto the screen twice leaves the screen as it was (this is a convenient way to move images for animation). XORing a picture onto a background effectively superimposes the picture onto the background.		
Details	See takepic for an example of the use of drawpic and for further information about buffers for drawing pictures.		
	The screen must be in a " <i>graphics</i> " mode. See the setscreen procedure for details. If the screen is not in a " <i>graphics</i> " mode, an error will occur.		
Details	The drawpic , takepic , and sizepic subprograms have been made obsolete by the subprograms <u>Pic.Draw</u> and <u>Pic.New</u> of the <u>Pic</u> module. Users are strongly suggested to use those routines instead. The <u>Pic</u> module also contains subprograms to load images from files.		
See also	takepic and sizepic . See also setscreen , maxx , maxy and the various draw procedures.		
	See also predefined unit Draw and Pic .		

drawpolygon

graphics procedure

```
drawpolygon (x, y : array 1 .. * of int, n : int, Color : int)
Syntax
             The drawpolygon procedure is used to draw a polygon with n
             points. A line is drawn in Color from the point (x(1), y(1)) to
             (x(2), y(2)) to (x(3), y(3)) and so on. After drawing the line to
             (x(n), y(n)), a line will be drawn back to (x(1), y(1)), closing the
             polygon. The drawpolygon procedure is equivalent to:
Description
                      for i : 1 .. n - 1
                           drawline (x (i), y(i), x (i + 1), y (i + 1),
                      end for
                      drawline (x (n), y (n), x (1), y (1), Color)
             This program will create an octagon and display it in color 1.
                      setscreen ("graphics")
                      var x : array 1..8 of int := init (100, 100, 135)
Example
                                                     220, 220, 185, 135)
                      var y : array 1..8 of int := init (100, 150, 185)
                                                     150, 100, 65, 65)
                      drawpolygon (x, y, 8, 1)
```

Execute

	The IBM PC limits drawpolygon to a maximum of 256 points.	
Details	The meaning of the <i>Color</i> number depends on the current palette. See the palette statement.	
	The screen should be in a " <i>graphics</i> " mode. See the setscreen procedure for details. If the screen is not in a " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode.	

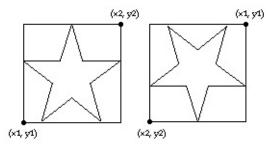
drawstar

graphics procedure

Syntax drawstar (x1, y1, x2, y2, Color : int)

The **drawstar** procedure is used to draw a star on the screen bounded by a rectangle described by the bottom left and top right corners of (x1, y1) to (x2, y2) using the specified *Color*. If y1 is greater than y2, then the star is drawn upside down.

Description



This program will draw two stars beside each other. The first will be in color 1 and the second star will be upside down and in color 2.

Example

setscreen ("graphics")
drawstar (0, 0, 100, 100, 1)
drawstar (150, 100, 250, 0, 2)

Execute

The **drawstar** procedure is useful for drawing the American flag.

The meaning of the *Color* number depends on the current palette. See the **palette** statement.

Details

The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in a "*graphics*" mode, it

will automatically be set to "*graphics*" mode.

empty	condition function		
Syntax	empty (variableReference) : boolean		
Description	The empty function is used in a concurrent program. It returns true if the <i>variableReference</i> , which must be a condition variable, has no processes waiting for it. Processes join the queue of a condition variable by executing the wait statement, and are awakened by the signal statement.		
	condition, wait, signal, fork and monitor.		
See also	See also predefined unit Concurrency .		

enumerated type

Syntax	An enumeratedType is: enum (id { , id })	
Description	The values of an enumerated type are distinct and increasing. They can be thought of as the values 0, 1, 2 and so on, but arithmetic is not allowed with these values.	
Example	type color : enum (red, green, blue) var c : color := color . red var d : color := succ (c) % d becomes gree	
Details	Each value of an enumerated type is the name of the type followed by a dot followed by the element's name, for example, <i>color.red</i> . Enumerated values can be compared for equality and for ordering. The succ and pred functions can be used to find the value following or preceding a given enumerated value. The ord function can be used to find the enumeration position of a value, for example, ord (<i>color.red</i>) is 0.	
	Enumerated types cannot be combined with integers or with other enumerated types.	
	It is illegal to declare an "anonymous" enum. The only legal declaration for an enum is in a type declaration. For example, the following is now illegal:	
Details	var a : array enum (<i>red</i> , green, blue) of int	
	Given that there is no (easy) way of generating an enum value without it being a named type, this should not impact any but the most bizarre code.	
	The "put" and "get" statement semantics have been expanded to allow put's and get's of enum values. The values printed and input are the element names themselves, case sensitive. For example,	

enum

Details for

type colors : enum (red, green, blue)
var c : colors := colors . red
put c % outputs "red" (without the quotes)

enumeratedValue

enumerated value

An enumeratedValue is:

Syntax	enumeratedTypeId . enumeratedId			
Description	The values of an enumerated type are written as the type name (<i>enumeratedTypeId</i>) followed by a dot followed by one of the enumerated values of the type (<i>enumeratedId</i>).			
	In this example, <i>color.red</i> is an <i>enumeratedValue</i> .			
Example	type color : enum (red, green, blue) var c : color := color . red var d : color := succ (c) % d becomes greε			
Details	The above description has been simplified by ignoring the possibility that the enum type can be exported from a module . If this is the case, each use of one of the enumerated values outside of module <i>M</i> must be preceded by the module name and a dot, as in <i>M.color.red</i> .			
See also	the <u>enum</u> type and <i><u>explicitConstant</u></i> .			

eof	end-of-file function		
Syntax	eof (streamNumber : int) : boolean		
Description	The eof (end of file) function is used to determine if there is any more input. It returns true when there are no more characters to be read. The parameter and its parentheses are omitted when referring to the standard input (usually this is the keyboard); otherwise the parameter specifies the number of a stream. The stream number has been determined (in most cases) by an open statement.		
Example	<pre>This program reads and outputs all the lines in the file called "info". var line : string var fileNumber : int open : fileNumber, "info", get loop exit when eof (fileNumber) get : fileNumber, line : * put line end loop</pre>		

Execute

See also the description of the get statement, which gives more
examples of the use of eof. See also the open and read
statements.When the input is from the keyboard, the user can signal end-of-
file by typing control-Z on a PC (or control-D on UNIX). If a
program tests for eof on the keyboard, and the user has not typed
control-Z (or control-D) and the user has typed no characters

beyond those that have been read, the program must wait until the next character is typed. Once this character is typed, the program knows whether it is at the end of the input, and returns the corresponding **true** or **false** value for **eof**.

Another example is available that illustrates detecting EOF from the keyboard.

Example

equivalence

Example

Two types are *equivalent* to each other if they are essentially the same types (the exact rules are given below). When a variable is passed to a **var** formal parameter, the types of the variable and the formal parameter must be equivalent because they are effectively tl same variable. When an expression is assigned to a variable, their types must be equivalent, except for special cases. For example, Turing allows you to assign an integer expression to a **real** variable (see *assignability* for details).

var j : **int** var b : array 1 .. 25 of string **type** *personType* : record age : int name : string (20) end record procedure p (var i : int, var a : array 1 .. 25 var r : personType) ... body of procedure p, which modifies each of i, end p var s : personType p (j, b, s) % Procedure call to p % i and j have the equivalent type i % Arrays a and b have equivalent typ % Records r and s have equivalent ty Two types are defined to be *equivalent* if they are:

- (a) the same standard type (**int**, **real**, **boolean** or **string**),
- (b) subranges with equal first and last values,
- (c) arrays with equivalent index types and equivalent component types,
- (d) strings with equal maximum lengths,
- (e) sets with equivalent base types, or

- (f) pointers to the same collection; in addition, a declared type identifier is also equivalent to the type it name
- (g) (and to the type named by that type, if that type is a named type, etc.)
- (h) both **char**,
- (i) both **char**(*n*) with the same length,
- (j) both procedure types, with corresponding equivalent paramet types and corresponding **var** or non-**var** of the parameters,
 - both function types, with corresponding equivalent parameter
- (k) types and corresponding var or non-var of the parameters an equivalent result types,
- (l) both pointer types to the same class or equivalent type and bc are checked or unchecked.

Each separate instance of a record, union or enumerated type (writt out using one of the keywords **record**, **union** or **enum**) creates a distinct type, equivalent to no other type. By contrast, separate instances of arrays, strings, subranges and sets are considered equivalent if their parts are equal and equivalent.

Opaque type T, exported from a module, monitor or class M as **opaque**, is a special case of equivalence. Outside of M this type is written M.T, and is considered to be distinct from all other types. B contrast, if type U is exported non-**opaque**, the usual rules of equivalence apply. The parameter or result type of an exported procedure or function or an exported constant is considered to have type M.T outside of M if the item is declared using the type identifi T. Outside of M, the **opaque** type can be assigned, but not compare

It is not required that subprogram types have the same names and parameter names to be equivalent. They also do not require the san factoring of parameters across their types, as in *i*, *j*: **int** instead of *i*. **int**, *j*: **int**.

Details

erealstr

	erealstr (<i>r</i> : real,		
Syntax	width, fractionWidth, exponentWidth : int) :string		
	The erealstr function is used to convert a real number to a string; for example, erealstr (2.5e1, 10, 3, 2)=" $b2.500e+01$ " where <i>b</i> represents a blank. The string (including exponent) is an approximation to <i>r</i> , padded on the left with blanks as necessary to a length of <i>width</i> .		
Description	The <i>width</i> must be a non-negative int value. If the <i>width</i> parameter is not large enough to represent the value of <i>r</i> , it is implicitly increased as needed.		
	The <i>fractionWidth</i> parameter is the non-negative number of fractional digits to be displayed. The displayed value is rounded to the nearest decimal equivalent with this accuracy. In the case of a tie, the value is rounded to the larger of the two values.		
	The <i>exponentWidth</i> parameter must be non-negative and give the number of exponent digits to be displayed. If <i>exponentWidth</i> is not large enough to represent the exponent, more space is used as needed. The string returned by <i>erealstr</i> is of the form:		
	{blank}[-]digit.{digit}e sign digit {digit}		
	where <i>sign</i> is a plus or minus sign. The leftmost digit is non-zero, unless all the digits are zeros.		
	The erealstr function approximates the inverse of strreal , although round-off errors keep these from being exact inverses.		
See also	frealstr, realstr, strreal, intstr and strint functions.		

Error

	This unit contains the predefined subprograms that deal with errors returned from other predefined subprograms.			
Description	All routines in the Error unit are exported qualified (and thus must be prefaced with " Error. ").			
	The constants representing the possible errors returned by this module can be found in the ErrorNum module.			
Entry Points	Last	Returns the (integer) error code produced by the last call to a predefined subprogram.		
	<u>LastMsg</u>	Returns the error string produced by the last call to a predefined subprogram.		
	<u>LastStr</u>	Returns the string version of the error constant produced by the last call to a predefined subprogram.		
	<u>Msg</u>	Returns the string that corresponds to a specified error code.		
	<u>Str</u>	Returns the string version of the error constant that corresponds to a specified error code.		
	<u>Trip</u>	This causes the specified error code to be set.		
	<u>TripMsg</u>	This causes the specified error code and error message to be set.		
	<u>Halt</u>	This causes execution to halt with the specified error message.		

Error.Last

Syntax Error.Last : int

Error.Last is a function that returns the error code set by the last called predefined subprogram. If there is no error, then it returns *eNoError* (which is 0). If there is an error, you can use **Error.LastMsg** to obtain a textual form of the error or **Error.LastStr** to obtain a string version of the error constant.

Description The fact that **Error.Last** is not *eNoError* does not necessarily mean that the previous predefined function failed or failed completely. **Error.Last** also returns a number of warning codes. For example, if a user specifies a number larger than **maxcolor** for the *color* parameter of the **Draw.Line** procedure, the line is still drawn, only in color **maxcolor**. However, **Error.Last** will return a code that warns the user of the fact.

This program creates the directory called *information*. If the creation fails, it prints out the error number and an error message.

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Error.Last, not by calling Last.

Error.LastMsg

Syntax	Error.LastMsg : string
	Error.LastMsg is a function that returns the error message set by the last called predefined subprogram. If there is no error, then it returns the empty string. If there is an error, you can use Error.Last to obtain the error code.
Description	The fact that Error.LastMsg is not "" does not necessarily mean that the previous predefined function failed or failed completely. Error.LastMsg also returns a number of warning messages. For example, if a user specifies a number larger than maxcolor for the <i>color</i> parameter of the Draw.Line procedure, the line is still drawn, only in color maxcolor . However, Error.LastMsg will return a message that indicates that the color was out of range
Example	<pre>This program creates the directory called information. If the creation fails, it prints out the error number and an error message. Dir.Create ("testdata/information") if Error.Last = eNoError then put "Directory created" else put "Did not create the directory." put "Error Number: ", Error.Last put "Error Message: ", Error.LastMsg end if</pre>

Execute

Exported qualified.

Status This means that you can only call the function by calling

Error.LastMsg, not by calling **LastMsg**.

Error.LastStr

Syntax Error.LastStr : string

Error.LastStr is a function that returns the string version of the error code set by the last called predefined subprogram (i.e. it would return the string "eDrawClrNumTooLarge" for using a color greater than **maxcolor** in a **Draw** command). If there is no error then it returns the empty string. If there is an error, you can use **Error.Last** to obtain the actual error code.

The fact that **Error.LastStr** is not "" does not necessarily mean that the previous predefined function failed or failed completely.

Description Error.LastStr also returns a number of error codes for warning messages. For example, if a user specifies a number larger than maxcolor for the *color* parameter of the Draw.Line procedure, the line is still drawn, only in color maxcolor. However, Error.LastStr will return a string version of the error code that indicates that the color was out of range.

You can take a look at the error constants defined by looking at the unit **ErrorNum** which contains all defined error codes.

This program creates the directory called *information*. If the creation fails, it prints out the error number and an error message.

	Dir.Create ("testdata/information") if Error.Last = <i>eNoError</i> then
Example	<pre>put "Directory created"</pre>
	else
	put "Did not create the directory."
	put "Error Number: ", Error.Last
	<pre>put "Error Constant: ", Error.LastStr</pre>
	end if

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Error.LastStr, not by calling LastStr.

Error.Msg

Part of <u>Error</u> module

Syntax	Error.Msg (errorCode : int): string		
Description	Error.Msg is a function that returns the error message related to a specified error code. If the error code is <i>eNoError</i> , or if there is no such error code, it returns the empty string. If there is such an error, it returns the textual message associated with that error.		
Example	This program prints out the error message associated with <i>eFsysFileNotFound</i> ("File not found"). <pre>put Error.Msg (eFsysFileNotFound)</pre>		
	Exported qualified.		
	Exported quanned.		
Status	This means that you can only call the function by calling Error.Msg , not by calling Msg .		

Error.Str

Syntax	Error.Str (errorCode : int): string			
Description	Error.Str is a function that returns the error message related to a specified error code. If the error code is <i>eNoError</i> or if there is no such error code, it returns the empty string. If there is such an error, it returns the textual message associated with that error.			
Example	This program prints out the string "eFsysFileNotFound". put Error.Str (<i>eFsysFileNotFound</i>)			
	Exported qualified.			
Status	This means that you can only call the function by calling Error.Str , not by calling Str .			

Error.Trip

Syntax	Error.Trip (errorCode : int)		
Description	Error.Trip is a procedure that sets the error number that is returned by Error.Last and Error.LastMsg . It does not halt the program.		
	Error codes that do not correspond to recognized errors will cause an abort with the error message "Unknown Error #n" where n is the error passed in.		
	You can find a list of constants for the legal error codes in the module ErrorNum . Any call to Error.Trip should use a constant found in the ErrorNum module.		
	This program sets an error code. The program outputs 201 for the error number and "File not found" for the message.		
Example	Error.Trip (<i>eFsysFileNotFound</i>) put "Error code = ", Error.Last put "Error message = ", Error.LastMsg		

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Error.Trip, not by calling Trip.

ErrorNum

This unit contains all the constants representing errors used by the Error module.

Description All constants in the ErrorNum module are exported unqualified. (This means you can use the constants directly without having to use the qualifier "**ErrorNum.**".)

Exceptions

This unit contains all the constants corresponding to exception numbers in Turing for use in building exception handlers.

Description All constants in the Exceptions module are exported unqualified. (This means you can use the constants directly without having to use the qualifier "**Exceptions.**".)

statement

exit	statement	
	An <i>exitStatement</i> is one of:	
Syntax	(a) exit when trueFalseExpn(b) exit	
Description	An exit statement is used to stop the execution of a loop or for statement. Form (a) is the most common. Here, the true/false expression is evaluated. If it is true, the loop is terminated and execution jumps down and continues just beyond end loop or end for . If it is false, the loop keeps on repeating. Form (b) always causes the loop to terminate. This form is almost always used inside another conditional statement such as if .	
Example	<pre>Input names until finding Jones. var name : string loop get name exit when name = "Jones" end loop</pre>	
Details	Exit statements must occur only inside loop or for statements. An exit takes you out of the closest surrounding loop or for . The only other ways to terminate a loop or for is by return (in a procedure or in the main program, in which case the entire procedure or main program is terminated) or by result (in a function, in which case the entire function is terminated and a result value must be	

supplied).

The form "**exit when** *trueFalseExpn*" is equivalent to "**if** trueFalseExpn then exit end if".

exp	exponentiation function		
Syntax	exp (<i>r</i> : real) : real		
Description	The exp function is used to find e to the power r, where e is the natural base and r is the parameter to exp . For example, exp (0) returns 1 and exp (1) returns the value of e.		
	This program prints out the exponential values of 1, 2, 3, up to 100.		
Example	<pre>for i : 1 100 put "Exponential of ", i, " is ", exp (i) end for</pre>		
	In (natural logarithm function).		
See also	See also predefined unit Math.		

explicitCharConstant

An *explicitCharConstant* is a sequence of characters surrounded bysingle quotation marks, for example, 'Renzo'.

In the following, the explicit character constants are 'H' and 'Hi'.

Example var c : char := 'H' var d : char (2) := 'Hi'

An explicit character constant must contain at least one character. If it contains exactly one character, as in 'A', its type is **char**. If it contains two or more characters (n characters), as 'Width', its type is **char**(n). The difference between the **char** and **char**(1) types is rarely of significance, but does make a difference in declarations without an explicit type, for example:

var c := 'H' % Type is char var d := 'Hi' % Type is char (2) var e := "H" % Type is string

The backslash \ is used in explicit string and char(n) constants to specify special values, for example, '\T' is the tab character. Similarly, the carat ^ is used to specify ASCII control characters, for example, '^H' is the ASCII backspace. See *explicitStringConstants* for details.

Details

Explicit character constants cannot cross line boundaries. To represent a constant that is longer than a line, break it into two or more strings on separate lines and use + (catenation) to join the individual strings. See **catenation**.

An explicit character constant may be limited in length by the implementation. We recommend that this limitation be at least 32767.

Explicit character constants, but not strings, are allowed to contain the character internal values 0 (called *eos* for end of string) and 128 (called *uninitchar*, used as the uninitialized string value).

explicitConstant

An *explicitConstant* is one of:

	(a)	explicitStringConstant	% e.g.: "Hello world"
	(b)	explicitIntegerConstant	% e.g.: 25
Syntax	(C)	explicitRealConstant	% e.g.: 51.8
	(d)	<i>explicitTrueFalseConstant</i> % e.g.: true	
		% e.g.: true	
	(e)	explicitCharConstant	% e.g.: 'Hi'

An *explicitConstant* gives its value directly. For example, the**Description**value of the explicit constant 25 is twenty-five.

In the following, the explicit constants are "Hello world", 3.14159 and 2. Note that *pi* is a *named* constant rather than an explicit constant.

Example	<pre>put "Hello world" var diameter : real const pi := 3.14159 diameter := pi * r ** 2 var x := diameter</pre>
Details	In some programming languages, <i>explicit constants</i> are called <i>literals</i> or <i>literal values</i> , because they literally (explicitly) give their values.
See also	<u>explicitStringConstant, explicitIntegerConstant, explicitRealConstant, explicitTrueFalseConstant</u> and <u>explicitCharConstant</u> . See also <u>enumeratedValue</u> .

literal

explicitIntegerConstant

SyntaxAn *explicitIntegerConstant* is a sequence of one or more decimal digitionsyntaxoptionally preceded by a plus or minus sign. This is an alternate form
specifies a number base (such as base 2 or base 16).

In the following, the explicit integer constants are 0, 115 and 5.

Example	<pre>var count : int := 0 const height := 115</pre>
	 count := height - 5

In current implementations of Turing, the range of the **int** (**integer**) ty -2147483647 to 2147483647. In other words, the maximum size of in 2**31 - 1. This is the range that fits into four bytes, with one pattern l (the largest negative 4-byte number) to represent the uninitialized valu **maxint**.

Values can be written in base 2 or 16 or any other base in the range 2 because there are 10 digits and 26 letters). This form begins with the l as 16, then #, and then the value written in that base, for example, 16[‡] value 10. The letters a, b, c ... represent the digit values 10, 11, 12 ... letters A, B, C ... can be used instead of lower case. Here are some ex

	2#1	= 1		(Base 2)
	2#11	= 3		(Base 2)
Details	16#a	= 10		(Base 16)
	16#FF	= 255	5	(Base 16)
	16#FFFF	= 32767	(Base 16)	
	8#10	= 8		(Base 8)

Here is an example of using these:

<pre>const maxnat1 :=</pre>	16#FF	% Largest 1-byte natur
<pre>const maxint2 :=</pre>	16#7FFF	% Largest 2-byte integ

You should be careful to avoid confusion about patterns such as 16#F tempting to think that this is the value 1, because the bit pattern (2-by)

complement internal representation) for 1 is the same as the bit pattern 16#FFFF = 32767. However, the value (as opposed to the internal representation) of 1 and 32767 are different.

int, **maxint** (the largest integer value), **nat** (positive values only) and byte integers). See also **intstr** and **natstr** which convert integer and nature values to corresponding character strings in any base, for examinister (4, 0, 2) = "100".

See also

explicitRealConstant

real literal

An *explicitRealConstant* consists of an optional plus or minus sign, aSyntax significant digits part, and an *exponent part*.

In the following, the explicit real constants are 0.0 and 2.93e3.

Example var temperature : real := 0.0 const speed := 2.93e3 % Value is 2,930.0

The significant digits part (or *fractional part*) of an explicit real constant consists of a sequence of one or more digits (0 to 9) optionally containing a decimal point (a period). The decimal point is allowed to follow the last digit as in 16. or to precede the first digit, as in .25.

Details The exponent part consists of the letter e or E followed optionally by a plus or minus sign followed by one or more digits. For example, in -9.837e-3 the exponent part is e-3. The value of -9.837e-3 is -9.837 times 0.001.

If the significant figures part contains a decimal point, then the exponent part is not required.

explicitStringConstant

An *explicitStringConstant* is a sequence of characters surrounded bySyntaxquotation marks.

In the following, the explicit string constants are "Hello world", "" and "273 O'Reilly Ave.".

Example

var name : string := "Hello world"
name := "" % Null string, containing zero cha
var address : string := "273 O'Reilly Ave."

Within an explicit string constant (and within an explicit character constant), the back slash \ is used to represent certain other characters as follows:

\"	quotation mark character
n or N	end of line character
t or T	tab character
f or F	form feed character
r or R	return character
\b or \B	backspace character
\e or \E	escape character
d or D	delete character
//	backslash character

For example, **put** "*One**nTwo*" will output *One* on one line and *Two* on the next. In an explicit character constant (which is surrounded by single quotes, as in '*John*'), the backslash is not required before a double quote ", but it is required before a single quote ', as in these two constants:

'John said "Hello" to you' 'Don\'t cry'.

Details You can use the caret ^ to specify ASCII control characters, for example:

'^H' ASCII backspace character

The caret specifies that the top three bits of the character are set to zero. For any character *c*, the following is true:

'^c' = chr (ord ('c') & 2#11111)

However if *c* is the question mark, as in '^?', the bits are not turned off.

Explicit string constants cannot cross line boundaries. To represent a string that is longer than a line, break it into two or more strings on separate lines and use catenation (+) to join the individual strings.

An explicit string constant can contain at most 255 characters (this is in implementation constraint).

String values are not allowed to contain characters with the code values of 0 or 128; these character values are called *eos* (end of string) and *uninitchar* (uninitialized character). These are reserved by the implementation to mark the end of a string value and to see if a string variable has been initialized.

explicitTrueFalseConstant

boolean literal

An *explicitTrueFalseConstant* is one of:

Syntax

(a) true(b) false

The following determines if string *s* contains a period. After the **for** statement, *found* will be **true** if there is a period in *s*.

var	<pre>found : boolean := false</pre>
Example for	<i>i</i> : 1 length (<i>s</i>)
Laumpie	if <i>s</i> = "." then
	found := true
	end if
end	for

Detailstrue/false values are called *boolean* values. A **boolean** variable,
such as *found* in the above example, can have a value of either true
or false.

See also boolean type.

An *expn* is one of:

	(a)	explicitConstant		% e.g.: 25	
	(b)	variableReference		% e.g.: width	
	(C)	constantReference		% e.g.: pi	
	(d)	expn infixOperator expn	% e.g.: 3 + width		
	(e)	prefixOperator expn	% e.g.: - width		
Syntax	(f)	(expn)			% e (wia 7)
	(g)	substring			% е (З
	(h)	functionCall		% e.g.: sqrt (25)	
	(i)	setConstructor		% e.g.: modes (4, 3)	
	(j)	enumeratedValue		% e.g.: color . red	

An expression (*expn*) returns a value; in the general case, this may involve a calculation, such as addition, as in the expression:

Description

3 + width

put "Hello world" % "Hello world" is a

Example	var diameter : realconst pi := 3.14159% 3.14159 is an exprdiameter := pi * r ** 2% pi * r ** 2 isvar x := diameter% diameter is an
Details	In the simplest case, an expression (<i>expn</i>) is simply an explicit consuch as 25 or " <i>Hello world</i> ". A variable by itself is considered to be expression when its value is used. This is the case above, where the of <i>diameter</i> is used to initialize <i>x</i> . More generally, an expression cc an operator such as + and carries out an actual calculation. An expr may also be a substring, function call, set constructor or enumerate value. For details, see the descriptions of these items.
	The Turing infix operators are: +, -, *, /, div , mod , **, <, >, =, <=, not= , not , and , or , =>, in , not in , shr (shift right), shl (shift left), (exclusive or). For details, see <i>infixOperator</i> . The Turing prefix op are +, - and not , ^ (pointer following) and # (see cheat). For details <i>prefix</i> operator.
See also	<i>precedence</i> of operators, as well as the <i>int</i> , <i>real</i> , <i>string</i> and <i>boolea</i> types.

export

An exportList is:

Syntax **export** [howExport] id {, [howExport] id } An **export** list is used to specify those items declared in a module, monitor or class that can be used outside of it. Items that are **Description** declared inside a module, monitor or class but not exported cannot be accessed outside of it. In this example, the procedures names *pop* and *push* are exported from the *stack* module. These two procedures are called from outside the module on the last and third from last lines of the example. Notice that the word *stack* and a dot must precede the use of these names. Since *top* and *contents* were not exported, they can be accessed only from inside the module. module stack export push, pop Example **var** *top* : **int** := 0 var contents : array 1..100 of string procedure push ... end push procedure pop ... end pop end stack stack . push ("Harvey") var name : string stack . pop (name) % This sets name to Harv Procedures, functions, variables, constants and types can be exported. Modules, monitors or classes canot be exported. Parentheses are allowed around the items in an export list, as in:

export (push, pop)

The following syntax specifies that each exported identifier can optionally be preceded by the keywords **var**, **unqualified**, **pervasive** and **opaque**. Of these, only **opaque** is available in Turing proper.

The form of *howExport* is:

{ exportMethod }

The form of *exportMethod* is one of:

- (a) var
- (b) **unqualified**
- (c) **pervasive**
- (d) **opaque**

The keyword **var** means that the exported variable can be changed outside of the exporting module, monitor or class. This keyword applies only to exported variables. For example, if string variable *name* is exported **var** from module *M*, *name* can be changed from outside of *M* by *M.name* := "*Surprise!*".

Details The keyword **unqualified** means that references to the exported item do not need to be prefixed by the name of the exporting item. For example, if module *M* exports procedure *p* unqualified, a call to *p* outside of *M* can be simply *p* instead of the usual *M.p.* A class cannot export variables or dynamic constants unqualified (because each object of the class has its own copies of these). The only things a class can export unqualified are types and compile time constants. The keyword **unqualified** can be abbreviated to ~. which is pronounced as "not dot".

The keyword **pervasive**, which is only meaningful if **unqualified** is also present, specifies that the exported item is to be visible in subsequent scopes, in other words that it is not necessary to import it into internal modules, monitors and classes.

The keyword **opaque**, which can only precede type names, specifies that outside the module, monitor or class, the type is considered to be distinct from all other types. This means, for example, that if the type is an array, it cannot be subscripted outside of the module. See **module** declaration for an example that uses opaque types. In most cases, classes are preferable to opaque types.

	Exported subprograms are considered to be deferred , meaning that expansions are allowed to override these subprograms. See also deferred subprograms. These can be overridden using the keyword body before the resolving subprogram body.
	A class cannot export items from its parent or it parent's ancestors. All exported item must be declared in the current class.
Details	You can export all from a module, monitor or a class. This means that every sibmle that is legal to export is exported. You may also qualify the all, as in export opaque unqualified pervasive all where the qualifiers are added to each export item (if it makes sense).
	If all is specified as the export item, no other item may be specified. Also, and all export affects only the module, monitor or class that it is given in. Any inheriting or implementing module, monitor or class does not export all unless they also specify it.
See also	<u>unit, module, monitor</u> and <u>class</u> . See also <u>import</u> list, <u>inherit</u> clause, <u>implement</u> clause, <u>implement</u> by clause and <u>deferred</u> subprogram.

external

declaration

Dangerous

An *externalDeclaration* is one of:

Syntax(a) external [overrideName] subprogramHeader(b) external [addressSpec] var id [: typeSpec] [:=

An external declaration is used to access variables or subprograms in other languages or which require special linkage. This feature is implementation-dependent and dangerous and may cause arbitrary program corruption. From an interpretive environment such as Tur provides linkage to items that are part of the Turing system. For co of Turing, the linkage would be by means of a standard, operating a linkage editor.

In form (a) the optional *overrideName* must be an explicit string cc "*printf*". If it is omitted, the external name is the name in the *subprose subprogramHeader*.

The current implementation does not support form (b). This form is here in case a future version supports it. The *addressSpec* is a comp expression (its value must fit in the range of the **addressint** type) o time string value. If the *addressSpec* is omitted, the identifier is the external variable. This name represents an implementation-dependence locating a variable. At least one of *typeSpec* or *expn* must be preserved.

Declaring variables at absolute addresses is useful for device mana computer architectures with memory mapped device registers. Externate declared to be **int** or **nat** will by default be checked for initializatio check, declare them to be **int4** or **nat4**.

Place variable *ttyData* at hexadecimal location 9001 and assign it t

Example

Details

external 16#9001 **var** *ttyData* : **char** *ttyData* := 'A' % *Character A is assigned to hex*

Example	Access an external integer variable named ERRFLAG.
	external var ERRFLAG : int if ERRFLAG = 0 then
Example	Access an integer variable which is called <i>y</i> in this program but is externally.
	external "x" var y : int
Example	Declare <i>drawcircle</i> to be a procedure that is externally known as <i>ci</i>
	external "circle" procedure drawcircle (x, y, r,

Syntax false

A boolean (true/false) variable can be either true or false (seeDescriptionboolean type).

	<pre>var found : boolean := false var word : int for i : 1 10 get word</pre>
Example	<pre>found := found or word = "gold" end for if found = true then</pre>
	<pre>put "Found 'gold' in the ten words" end if</pre>
	The line if found= true then can be simplified to if found

DetailsThe line if *found=*true then can be simplified to if *found* thenwith no change to the meaning of the program.

false

fetcharg

fetch argument function

Syntax **fetcharg** (*i* : **int**) : string

The **fetcharg** function is used to access the *i*-th argument that has been passed to a program from the command line. For example, if the program is run from the Turing environment using

r file1 file2:

then **fetcharg**(2) will return "file2". If a program called *prog.x* is run under UNIX using this command:

Description prog.x file1 file2

the value of **fetcharg**(2) will similarly be "file2".

The **nargs** function, which gives the number of arguments passed to the program, is usually used together with the **fetcharg** function. Parameter *i* passed to **fetcharg** must be in the range 0 .. **nargs**.

The 0-th argument is the name of the running program.

This program lists its own name and its arguments.

Exampleput "The name of this program is : ", fetcharg (for i : 1 .. nargs
put "Argument ", i, " is ", fetcharg (i)end for

Execute

See also <u>nargs</u>

File

Details

Description	This unit contains the predefined subprograms that deal with file manipulation on a whole-file basis (as opposed to manipulating the data in the file using open and close , etc.). These routines allow you to rename, copy and delete files, as well as get information about a file and get the free space on disk available for a file. All routines in the File module are exported qualified (and thus must be prefaced with " File. ").		
	Exists	Returns whether a file exists.	
	<u>FullPath</u>	Returns the full absolute path name of a file.	
	Parent	Returns the parent directory of a file or directory.	
D .	<u>Status</u>	Gets information about a file such as size, modification date, etc.	
Entry Points	<u>Copy</u>	Copies a file to another location.	
I Units	<u>Rename</u>	Renames a file or directory.	
	Delete	Deletes a file.	
	<u>DiskFree</u>	Gets the free space on the disk upon which a file or directory resides.	
	On the PC, a path name of a file or a directory can use either the forward slash or backward slash to separate directory names. The drive must be followed by a colon. Thus the following are legal path names:		
	C:/	<pre>\students\west\example.t /turing/test.t est/binary.t (uses the default drive).</pre>	
	On the Maci	ntosh, a path name of a file or directory can use the	

On the Macintosh, a path name of a file or directory can use the standard Macintosh format of Volume Name:Directory Name:Directory Name:File Name or the Unix format of /Volume Name/Directory Name/Directory Name/File Name. Note that the names can have spaces in them. HSA:Applications:Turing Files:example.t /HSA/Applications/Turing Files/example.t

On UNIX systems, the path name must correspond to the UNIX standard of using a forward slash between parts of the path.

/export/home/west/turing/example.t

In general, you can achieve the greatest portability by using the UNIX standard for use in path names, as all Turing systems support it.

File.Copy

Syntax	File.Copy (srcPathName, destPathName : string)
Description	File.Copy copies a file named by the <i>srcPathName</i> parameter to the file named by the <i>destPathName</i> parameter. The copy can be between different disks or file systems.
	The source file name must be an actual file. This procedure will not copy directories.
Details	If the File.Copy call fails, then Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.
	Note that you can use either forward or backward slashes to separate components in a path. If you use backward slashes, you must double them in a string literal. (i.e. "d:\\west\\example.t")
	This program copies the file " <i>d:\west\example.dat</i> " to " <i>new_example.t</i> ". in the current directory.
Example	<pre>File.Copy ("d:/west/example.dat", "new_example.t if Error.Last = eNoError then put "File copied" else put "Did not copy the file." put "Error: ", Error.LastMsg end if</pre>
	An example program is available that copies a file to the current

An example program is available that copies a file to the current directory using **File.Copy**, displays its contents to the run window, and then deletes the file using **File.Delete**.

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
File.Copy, not by calling Copy.

File.Delete

Part of **File** module

Syntax	<pre>File.Delete (filePathName : string)</pre>
Description	File.Delete is used to delete the file specified by the parameter <i>filePathName</i> . This is the equivalent of doing a del in DOS or rm in UNIX.
Details	If the File.Delete call fails, then Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.
Example	<pre>This program deletes the file called information. File.Delete ("information") if Error.Last = eNoError then put "File delete" else put "Did not delete the file." put "Error: ", Error.LastMsg end if</pre>

An example program is available that creates a file in the current directory, and then deletes the file using **File.Delete**.

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
File.Delete, not by calling File.

File.DiskFree

Syntax	File.DiskFree (pathName : string) : int	
Description	File.DiskFree gets the number of bytes for the disk upon which <i>pathName</i> resides. The <i>pathName</i> parameter can specify either a file or a directory. If it is the empty string, then File.DiskFree returns the number of bytes of free disk space on the disk upon which the execution directory resides.	
Details	If the File.DiskFree call fails, then it returns -1. Also Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	If there is more than 2,147,483,647 bytes free on a disk, the File.DiskFree function returns 2,147,483,647.	
	This program prints out the amount of space on the <i>A</i> : drive on a PC and in the execution directory.	
Example	<pre>var bytesFree : int bytesFree := File.DiskFree ("A:\\") if bytesFree = -1 then put "Can't get free space on drive A:." put "Error: ", Error.LastMsg else put "There are ", bytesFree , " bytes free c end if</pre>	
	<pre>bytesFree := File.DiskFree (".") if bytesFree = -1 then put "Can't get free space on default directc put "Error: ", Error.LastMsg else put "There are ", bytesFree , " bytes free c end if</pre>	

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
File.DiskFree, not by calling DiskFree.

File.Exists

Part of **File** module

Syntax	File.Exists (pathName : string) : boolean
Description	File.Exists returns true if a file by the name of <i>pathName</i> exists. It will return false if <i>pathName</i> is a directory.
Details	If the File.Exists returns false , you can examine Error.Last or Error.LastMsg for more information (i.e. whether the path failed or the file was simply not found).
Example	This program loops until the user types in a path name that either doesn't already exist or is allowed to be overwritten. var pathName : string var choice : string loop put "Enter file name to write results to" get pathName if File.Exists (pathName) then put "Overwrite ", pathName, "?" get choice exit when choice = "y" else exit end if end loop

An example program is available that creates a file in the current directory, tests for its existence using **File.Exists** and deletes the file using **File.Delete**.

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
File.Exists, not by calling Exists.

File.FullPath

Part of <u>File</u> module

Syntax	File.FullPath (pathName) : string) : string
Description	File.FullPath returns a string representing the full absolute path name in Turing format (forward slashes) of the path that is passed to the function. The path name passed in does not have to describe an existing file or directory.
Details	The full path name will be in Turing format and include the drive name (for example "d:/turing/examples/games/SpaceGame.t")
	This program obtains a path from the user and then outputs a full path name based on the path.
Example	var pathName : string get pathName put "Full path = ", File.FullPath (pathName)

Execute

Another example is available that checks whether a file and all of the directories in the path to the file exist. This program illustrates the use of **File.FullPath**, **File.Parent**, **File.Exists**, and **Dir.Exists**.

Details

Exported qualified.

StatusThis means that you can only call the function by calling
File.FullPath, not by calling FullPath.

File.Parent

Syntax File.Parent (*pathName* : **string**) : **string**

File.Parent returns a string representing the parent directory in Turing format (forward slashes) of the path passed as a parameter. The path name passed in does not have to describe an existing file

Description or directory.

Attempting to obtain the parent directory of a root directory (for example "c:/") will return the same root directory and will set the value returned by **Error.Last** to a non-zero value.

This program obtains a path from the user and then lists all the parent directories until it reaches the root directory.

Example

```
var pathName : string
put "Enter a path: " ...
get pathName
loop
        pathName := File.Parent (pathName)
        exit when Error.Last not= eNoError
        put pathName
end loop
```

Execute

Another example is available that checks whether a file and all of the directories in the path to the file exist. This program illustrates the use of **File.FullPath**, **File.Parent**, **File.Exists**, and **Dir.Exists**.

Details

Exported qualified.

StatusThis means that you can only call the function by calling
File.Parent, not by calling Parent.

File.Rename

Part of **File** module

Syntax	File.Rename (srcPathName, destName : string)
Description	File.Copy renames a file or directory named by the <i>srcPathName</i> parameter to the <i>destName</i> parameter. The <i>destName</i> parameter must be a name only. In other words File.Rename can't move a file between different directories.
Details	If the File.Rename call fails, then Error.Last will return a non- zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.
	This program renames the file "/usr/west/example" to "testcase" File.Rename ("/usr/west/example", "testcase")
Example	<pre>if Error.Last = eNoError then put "File renamed" else put "Did not rename the file."</pre>
	put "Error: ", Error.LastMsg end if

An example program is available that creates a file in the current directory, renames it to a different name using **File.Rename**, checks the existence of both files using **File.Exists** and then deletes the file using **File.Delete**.

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
File.Rename, not by calling Rename.

File.Status

Syntax	File.Status (<i>pathName</i> : string , var <i>size</i> , <i>attribute</i> , <i>fileTime</i> : int)
	File.Status is used to get assorted information about a file or directory. When the function is called with a specified <i>pathName</i> , it returns the information about the file in the other parameters.
	The <i>size</i> parameter is the size of the file in bytes.
Description	The <i>attribute</i> parameter has its individual bits set as exactly as the <i>attribute</i> parameter in Dir.GetLong subprogram does. See Dir.GetLong for the list of attribute constants.
	The <i>fileTime</i> is the time of last modification of the file. It is returned as the number of seconds since 00:00:00 GMT 1/1/1970. To convert this to a string, use Time.SecDate.
Details	If the File.Status call fails, <i>size</i> , <i>attribute</i> and <i>fileTime</i> are all set to 1. Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.
	This program prints information about the NotePad.exe application in Windows found at " <i>c:\windows\notepad.exe</i> ". (Note on some systems, this will be "c:\winnt\notepade.exe"
	<pre>const pathName : string := "c:/windows/notepad.e var size, attribute, fileTime : int File.Status (pathName, size, attribute, fileTime if Error.Last = eNoError then put " Name: ", File.FullPath (pathName) put " Created: ", Time.SecDate (fileTime) put " Size: ", size, " bytes" put "Attributes: " if (attribute and ootAttrDir) not= 0 then put "Directory " else put "" end if</pre>
Example	<pre>if (attribute and ootAttrRead) not= 0 then put "Readable "</pre>

```
else
        put "" ..
    end if
    if (attribute and ootAttrWrite) not= 0 then
        put "Writable " ...
    else
        put "" ..
    end if
    if (attribute and ootAttrExecute) not= 0 the
        put "Executable", skip
    else
        put skip
    end if
else
    put "Unable to get file information"
   put "Error: ", Error.LastMsg, skip
end if
```

Execute

Another example program is available that uses **File.Status** to determine the status of several files and directories.

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
File.Status, not by calling Status.

flexible array indexType { , indexType } **of** typeSpec **Syntax** The **flexible** keyword allows an array to be resized using **new** at a later point in time. The indices may have compile-time or runtime upper bounds (the lower bound must be compile-time). The upper bounds can be changed by using:

new name , newUpper1 {, newUpper2}

The existing array entries will retain their values, except that any index made smaller will have the corresponding array entries lost. Any index made larger will have the new array entries uninitialized (if applicable).

Additionally, the upper bound (both in the declaration and the **new** statement) can be made one less than the lower bound. This effectively makes an array that contains 0 elements. It can later be increased in size with another **new**.

> In the current implementation (1999), with a multi-dimensional array with a non-zero number of total elements, it is a run-time error to change any but the first dimension (unless one of the new upper bounds is one less than the corresponding lower bound, giving 0 elements in the array) as the algorithm to rearrange the element memory locations has not yet been implemented.

> Currently, only variables can be declared in this form. There is no flexible array parameter type, although a flexible array can be passed to an array parameter with "*" as the upper bound.

See **array** for an example of **flexible**. Example

flexible

Description

Execute

See also array and new.

real-to-integer function

Syntax	floor (<i>r</i> : real) : int
Description	Returns the largest integer that is less than or equal to r .
Details	The floor function is used to convert a real number to an integer. The result is the largest integer that is less than or equal to <i>r</i> . In other words, the floor function rounds down to the nearest integer. For example, floor (3) is 3, floor (2.75) is 2 and floor (-8.43) is -9.
See also	<u>ceil</u> and <u>round</u> functions.

floor

Font

Description	This unit contains the predefined subprograms that deal with fonts. Using these routines, you can display text in a selected font name, size and style on the screen. Note that output in a particular font is treated as graphics output.		
		the Font module are exported qualified (and thus ced with " Font .").	
Details	There is a default font. You can draw in and obtain information about the default font by passing <i>fontDefaultID</i> to Font.Draw , Font.Width and Font.Sizes . The default font is the same font as is used by put in the output window.		
	<u>New</u>	Selects a particular font name, size and style for a new font.	
	<u>Free</u>	Frees up the font created by using New .	
	Draw	Draws text in a given font.	
Entry Points	<u>Width</u>	Gets the width in pixels of a particular piece of text in a specified font.	
	<u>Sizes</u>	Gets the height and various leadings of a specified font.	
	<u>Name</u>	Returns the name of the specified font.	
	<u>StartName</u>	Prepares to list all available fonts,	
	<u>GetName</u>	Gets the next font name.	
	<u>GetStyle</u>	Gets all the available styles for a specified font.	
	<u>StartSize</u>	Prepares to list all available sizes for a specified font and style.	
	<u>GetSize</u>	Gets the next font size.	

Font.Draw

Syntax	Font.Draw (<i>txtStr</i> : string , <i>x</i> , <i>y</i> , <i>fontID</i> , <i>Color</i> : int)
	Font.Draw is used to actually draw text in a specified font. The <i>tex</i> contains the string to be drawn. The <i>x</i> and <i>y</i> parameters are the loca left hand corner of the text to be displayed. The <i>fontID</i> parameter is the font in which the text is to be drawn. The <i>Color</i> parameter is us color in which the text is to appear.
Description	Note that the text that appears is completely unrelated to the text th put. Font.Draw is a graphics command and thus does not use or a: location.
	The text drawn by the Font.Draw procedure does not erase the bac
Details	If Font.Draw is passed an invalid font ID, a fatal error occurs. If the fails for other (non-fatal) reasons, then Error.Last will return a non-indicating the reason for the failure. Error.LastMsg will return a secontains the textual version of the error.
	The program prints out several phrases in a variety of fonts.
Example	<pre>var font1, font2, font3, font4 : int font1 := Font.New ("serif:12") assert font1 > 0 font2 := Font.New ("sans serif:18:bold") assert font2 > 0 font3 := Font.New ("mono:9") assert font3 > 0 font4 := Font.New ("Palatino:24:bold,italic") assert font4 > 0 Font.Draw ("This is in a serif font", 50, 30, fc Font.Draw ("This is in a sans serif font", 50, 8 Font.Draw ("This is in a mono font", 50, 130, fc Font.Draw ("This is in Palatino (if available)", Font.Free (font1) Font.Free (font2) Font.Free (font3) Font.Free (font4)</pre>

Execute

Details	To use the same font as is used by the put statement, use defFontI number. This font does not have to be created or freed by the user, program to quickly place text in any location on the screen.	
	The program draws two strings in the default font (defFontID).	
Example	Font.Draw ("Drawing Here", 100, 120, defFontID , Font.Draw ("and Here", 180, 90, defFontID, brig h	

Execute

Exported qualified.

StatusThis means that you can only call the function by calling Font.Dra
Draw.

Font.Free

Syntax	Font.Free (fontID : int)	
Description	Font.Free is used to release a font that is no longer needed. There number of fonts that may be defined at any one time. By having a l every Font.New , the number of simultaneously defined fonts is ke	
Details	If Font.Free is passed an invalid font ID, a fatal error occurs. If the fails for other (non-fatal) reasons, Error.Last will return a non-zer the reason for the failure. Error.LastMsg will return a string which textual version of the error.	
Example	The program prints out several phrases in a variety of fonts. var font1, font2, font3, font4 : int font1 := Font.New ("serif:12") assert font1 > 0 font2 := Font.New ("sans serif:18:bold") assert font2 > 0 font3 := Font.New ("mono:9") assert font3 > 0 font4 := Font.New ("Palatino:24:Bold,Italic") assert font4 > 0 Font.Draw ("This is in a serif font", 50, 30, fc Font.Draw ("This is in a sans serif font", 50, 2 Font.Draw ("This is in a mono font", 50, 130, fc Font.Draw ("This is in Palatino (if available)", Font.Free (font1) Font.Free (font3) Font.Free (font4)	

Execute

Exported qualified.

StatusThis means that you can only call the function by calling Font.Fre
Free.

Font.GetName

Syntax	Font.GetName : string
	Font.GetName is used to get the next font available on the system. By using Font.StartName and then calling Font.GetName repeatedly, you can get the names of all the fonts available to the program.
Description	Font.StartName must be called before any calls to Font.GetName. After that, Font.GetName returns the list of the font names, one per call. When there are no more sizes, Font.GetName returns the empty string.
	Once the name of a font is known, it's possible to list the available styles (using Font.GetStyle) and the available sizes (using Font.StartSize and Font.GetSize) for that font.
	The program lists all the fonts available on the system.
Example	<pre>var fontName : string Font.StartName loop fontName := Font.GetName exit when fontName = "" put fontName end loop</pre>

Execute

Another example is available that displays full information about the fonts including name, styles, and point sizes.

Example

Exported qualified.

StatusThis means that you can only call the function by calling
Font.GetName, not by calling GetName.

Font.GetSize

Syntax	Font.GetSize : int
	Font.GetSize is used to get the next size in the list of available font sizes for a particular font name and style.
Description	Font.StartSize must be called before any calls to Font.GetSize . After that, Font.GetSize returns the list of sizes, one per call. When there are no more sizes, Font.GetSize returns 0.
Description	Some fonts are "scalable". This means that the computer can scale the fonts to fit any given size. (Under Microsoft Windows and the Apple Macintosh, TrueType and PostScript fonts are scalable with the appropriate utilities.) In this case, Font.GetSize returns -1.
Example	See Font.StartSize for a program that lists all the fonts, styles and sizes available on the system.
	An example is available that displays full information about the fonts including name, styles, and point sizes.
Example	

Exported qualified.

StatusThis means that you can only call the function by calling
Font.GetSize, not by calling GetSize.

Font.GetStyle

Font.GetStyle (fontName : string,Syntaxvar bold, italic, underline : boolean) : string

Font.GetStyle is used to get the styles available on the system for a specified font. *bold*, *italic* and *underline* are set to true if bold, italic or underline versions of the font are available. Once the styles available for a font are known, it's possible to get the sizes available for each style by using Font.StartSize and Font.GetSize.

The program lists all the fonts and their styles available on the system.

var fontName : string var bold, italic, underline : boolean Font.StartName **100**p fontName := Font.GetName exit when fontName = "" Font.GetStyle (fontName, bold, italic, under put fontName : 30 ... Example if bold then put "bold " .. end if if *italic* then put "italic " .. end if **if** underline **then** put "underline " .. end if put "" end loop

Execute

An example is available that displays full information about the fonts including name, styles, and point sizes.

Example

Exported qualified.

StatusThis means that you can only call the function by calling
Font.GetStyle, not by calling GetStyle.

Font.Name

Syntax Font.Name (*fontID* : **int**) *fontName* : **string**

Font.Name is used to get the name of a font that is being used. The string that is returned can be used to determine which font is actually being used for the default fonts "serif", "sans serif" and "mono".

The program prints out the fonts used for "serif", "sans serif" and "mono".

var serifFont, sansSerifFont, monoFont : int serifFont := Font.New ("serif:12") assert serifFont > 0 sansSerifFont := Font.New ("sans serif:12") assert sansSerifFont > 0 monoFont := Font.New ("mono:12") assert monoFont > 0 put "serif = ", Font.Name (serifFont) put "sans serif = ", Font.Name (sansSerifFont) put "mono = ", Font.Name (monoFont) Font.Free (serifFont) Font.Free (sansSerifFont) Font.Free (monoFont)

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Font.Name, not by calling Name.

Font.Nev	NPart of Font module
Syntax	Font.New (fontSelectStr : string) : int
	Font.New is used to obtain a font for drawing. The <i>fontSelectStr</i> pa the name, size and style of the font. Font.New returns a font identiused by the Font.Draw procedure to draw text in the selected font.
	The format for the <i>fontSelectStr</i> parameter is " <i>Family:Size:Style</i> ". E separated by a colon. The " <i>:Style</i> " is optional. If left out, the text ap standard face for the font.
	<i>Family</i> is the name of the font, such as "Times", "Helvetica", etc. T match an existing font on the system. Because one does not necess fonts will be available and names for the same font change between systems (i.e Times, Times-Roman, etc.), Turing defines three famil be mapped as closely as possible to fonts that exist on the system.
Description	"serif" is used for a serifed body font. This will usually be ma Roman. "sans serif" is used for a non-serifed display font. This will u to Arial. "mono" is used for a mono spaced font. This will usually be Courier.
	<i>Size</i> is the point size in which the text should appear. If the number smaller than can be created on a given system, the system will retu largest or smallest size available and set Error.Last .
	Under Turing, the <i>size</i> parameter may also have the form <i>height</i> x <i>height</i> and <i>width</i> are the pixel height and width desired. What is rescaled in order to fit into the <i>width</i> and <i>height</i> requested. The font is scaleable font for this to succeed.
	example <i>fontID</i> := Font.New ("Ariel:18x12:Italic"

Style is the font style in which the text should appear. It can be one or "underline". You can also have "bold,italic" and any other comb

If the **Font.New** call fails, then it returns 0. Also **Error.Last** will revalue indicating the reason for the failure. **Error.LastMsg** will retu contains the textual version of the error.

Details It is quite possible for **Error.Last** to be set, even if the call succeed report success even if unable to successfully match the requested for available resources. A font will be set that matches as closely as porequested font and **Last.Error** will be set to indicate that some sub required.

The program prints out several phrases in a variety of fonts.

	var font1, font2, font3, font4 : int
	font1 := Font.New ("serif:12")
	<pre>font2 := Font.New ("sans serif:18:bold")</pre>
	font3 := Font.New ("mono:9")
	<pre>font4 := Font.New ("Palatino:24:Bold,Italic")</pre>
	assert font1 > 0 and font2 > 0 and font3 > 0 and
Example	Font.Draw ("This is in a serif font", 50, 30, fc
	Font.Draw ("This is in a sans serif font", 50, 8
	Font.Draw ("This is in a mono font", 50, 130, fc
	<pre>Font.Draw ("This is in Palatino (if available)",</pre>
	Font.Free (font1)
	Font.Free (font2)
	Font.Free (font3)
	Font.Free (font4)

Execute

Exported qualified.

Status This means that you can only call the function by calling **Font.Nev New**.

Syntax

Description

Details

If **Font.Sizes** is passed an invalid font ID, a fatal error occurs. If th fails for other (non-fatal) reasons, the metrics for the default font w As well, **Error.Last** will return a non-zero value indicating the rea failure. **Error.LastMsg** will return a string which contains the text error.

Font.Sizes (fontID : **int**, **var** height, ascent, descent,

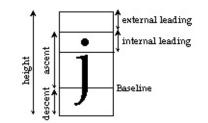
subtracting the ascent and descent from the height.

Font.Sizes is used to get the metrics of a particular font. The various metric are illustrated below. Note that you can calculate the externation

The program gets information about 24pt Bold Italic Palatino.

var fontID, height, ascent, descent, internalLeavar externalLeading: intfontID := Font.New ("Palatino:24:bold,italic")Font.Sizes (fontID, height, ascent, descent, intexternalLeading := height ascent descentput "The height of the font is ", height, " pixeput "The ascent of the font is ", ascent, " pixeput "The descent of the font is ", descent, " piput "The internal leading of the font is ", inteput "The external leading of the font is ", exteFont.Free (fontID)





internalLeading : **int**)

Font.Sizes

Execute

Exported qualified.

StatusThis means that you can only call the function by calling Font.SizeSizes.

Font.StartName

Part of **Font** module

Syntax Font.StartName

Font.StartName is used to start the listing of all the fonts available on the system. This procedure is called before making calls to Font.GetName to get the name of the fonts available.
Description Once the name of a font is known, it's possible to list the available styles (using Font.GetStyle) and the available sizes (using Font.StartSize and Font.GetSize).

The program lists all the fonts available on the system.

var fontName : string
Font.StartName
loop
 fontName := Font.GetName
 exit when fontName = ""
 put fontName
end loop

Example

Execute

Another example is available that displays full information about the fonts including name, styles, and point sizes.

Example

Exported qualified.

StatusThis means that you can only call the function by calling
Font.StartName, not by calling StartName.

Font.StartSize

Part of **Font** module

Syntax Font.StartSize (*fontName*, *fontStyle* : **string**)

Font.StartSize is used to start a listing of all the sizes for a particu

DescriptionThe *fontName* parameter should be an actual font name (as oppose
"serif", etc). You can get a list of the font names by using the **Font**
Font.GetName subprograms. The *fontStyle* parameter should be ir
appear in the **Font.New** procedure.

The program lists all the fonts, styles and sizes available on the sys

var fontName : string var bold, italic, underline : boolean var size : int Font.StartName **100**p fontName := Font.GetName exit when fontName = "" Font.GetStyle (fontName, bold, italic, under for b : false .. bold for i : false .. italic Example for u : false .. underline put fontName : 30, styles (b, i, Font.StartSize (fontName, styles **100p** size := Font.GetSize exit when size = 0 if size = -1 then put "scala else put size, " " ... end if end loop put "" end for end for end for end loop

Execute

Exported qualified.

StatusThis means that you can only call the function by calling Font.StalStartSize.

Font.Width

Syntax	Font.Width (<i>txtStr</i> : string , <i>fontID</i> : int) : int	
Description	Font.Width is used to obtain the width in pixels that a specified state to draw in a specified font. The <i>textStr</i> parameter is the string. <i>fontID</i> parameter is the font in which the string would be drawn.	
Details	If Font.Width is passed an invalid font ID, a fatal error occurs. If t Font.Width call fails for other (non-fatal) reasons, the width for st the default font will be returned. As well, Error.Last will return a value indicating the reason for the failure. Error.LastMsg will retu string which contains the textual version of the error.	
Example	<pre>The program gets information about 24pt Bold Palatino. const testString := "Test String" var width, fontID : int fontID := Font.New ("Palatino:24:Bold") width := Font.Width (testString, fontID) put "The width of \"" + testString + "\" is ", w Font.Draw (testString, 100, 100, fontID, black) Draw.Line (100, 50, 100, 150, brightred) Draw.Line (100 + width, 50, 100 + width, 150, br Font.Free (fontID)</pre>	

Execute

Exported qualified.

Status This means that you can only call the function by calling **Font.Wi** by calling **Width**.

A forStatement is:

Syntax

for [decreasing] [id] : first .. last [by increment] statementsAndDeclarations end for

The statements and declarations in a **for** statement are repeatedly executed. In the first iteration, the identifier is assigned the value of *first*. With each additional iteration, the identifier increases by 1 (or by *increment*, if the **by** clause is present). The loop stops executing when adding 1 (or *increment*) to the identifier would cause the identifier to exceed *last*. *first* and *last* must be integer values (or else enumerated or **char** values). If you specify **decreasing**, then the identifier decreases by 1 (or by *increment*) each time through.

Increment must be a positive integer value. When the **by** clause is present, the **for** loop terminates as soon as the identifier would become greater than *last*, unless **decreasing** is present. If **decreasing** is present, the loop terminates when the identifier would become less than *last*.

The identifier is checked before it is added to (or subtracted from). This means that the loop

Details for *i* : 1 . . maxint will not cause an overflow.

Output 1, 2, 3 to 10.

Example for i : 1 ... 10 put i end for

Output 1, 3, 5, 7 and 9.

Example	for <i>i</i> : 1 10 by 2 put <i>i</i> end for
	Output 10, 9, 8, down to 1.
Example	<pre>for decreasing j : 10 1 put j end for</pre>
	Output 10, 6, and 2.
Example	<pre>for decreasing j : 10 1 by 4 put j end for</pre>
	Output 1.
Example	for j : 1 10 by 20 put j end for
	Output nothing.
Example	for j : 5 2 put j end for
	The for statement declares the counting identifier (a separate declaration should not be given for i or j). The scope of this identifier is restricted to the for statement.
	If <i>first</i> is a value beyond <i>last</i> , there will be no repetitions (and no error message). The counting identifier is always increased (or decreased) by 1 or <i>increment</i> if the by clause is present. Executing an exit statement inside a for statement causes a jump to just beyond end for . You are not allowed to change the counting variable (for example, you are not allowed to write $i := 10$).
Details	The counting identifier can be omitted. In this case, the statement is just as before, except that the program cannot use the value of

the identifier.

If **decreasing** is not present, *first* .. *last* can be replaced by the name of a subrange type, for example by *dozen*, declared by:

```
type dozen : 1..12
```

Procedures, functions and modules cannot be declared inside a **for** statement. Just preceding the statements and declarations, you are allowed to write an "invariant clause" of the form:

```
invariant trueFalseExpn
```

This clause is equivalent to: **assert** *trueFalseExpn*.

fork

statement

Dirty parts

A forkStatement is:

Syntaxfork processId [([expn { , expn }])][: reference [, expn [, reference]]]

DescriptionA fork activates (starts the concurrent execution of) a process
declaration. If the process has parameters, a parenthesized list of
expressions (*expns*) must follow the process' name (*processId*).

This program initiates (forks) two concurrent processes, one of wh repeatedly outputs Hi and the other Ho. The resulting output is an unpredictable sequence of Hi's and Ho's, as *greetings* executes twic concurrently, one instance with its *word* set to Hi and the other witl *word* set to Ho.

Example process greetings (word : string) loop put word end loop end greetings fork greetings ("Hi") fork greetings ("Ho")

See **procedure** declaration for details about parameters. The first optional *reference* in the **fork** statement must be a **boolean** variable reference. The **fork** sets this to **true** if the process is actually activa If this fails to occur (probably because stack space could not be allocated), this *reference* is set to **false**. If the **fork** fails but this reference is omitted, an exception occurs. See exception handlers.

The optional *expn* specifies the number of bytes for the process' sta this overrides the optionally given stack size in the **process** declara The second optional *reference* must be a variable reference with the type **addressint**. See **addressint**. This variable is set to identify the **Details** process activation. This reference has the implementation-depende meaning of locating the process' internal descriptor.

In this explanation of the **fork** statement, we have up to this point ignored the possibility of processes exported from modules. If the process is being forked from outside of a module from which it has been exported, the syntax of the **fork** statement is:

```
fork moduleId . procedureId [ ( expn {, expn } )
```

In other words, the module's name and a dot must precede the proc name.

subprogram declaration

forward

A forwardDeclaration is:

Syntax forward subprogramHeader [import importItem {, importItem }]

Description A procedure or function is declared to be **forward** when you want define its header but not its body. This is the case when one proced or function calls another, which in turn calls the first; this situation called *mutual recursion*. The use of **forward** is necessary in this ca because every item must be declared before it can be used.

This example program evaluates an input expression *e* of the form *t* } where t is of the form $p \{ *p \}$ and *p* is of the form (*e*) or an explicit real expression. For example, the value of 1.5 + 3.0 * (0.5 1.5) halt is 7.5.

```
var token : string
                    forward procedure expn ( var eValue : real )
                    forward procedure term ( var tValue : real )
                    forward procedure primary ( var pValue: real )
                    body procedure expn
                        var nextValue : real
                        term ( eValue )
                                                % Evaluate t
                                                % Evaluate { + t}
                        100p
                            exit when token not= "+"
                            get token
                            term ( nextValue )
                            eValue := eValue + nextValue
                        end loop
                    end expn
Example
                    body procedure term
                        var nextValue : real
                        primary (tValue )
                                                % Evaluate p
                        100p
                                                % Evaluate { * p}
                            exit when token not= "*"
                            get token
```

```
primary ( nextValue )
        tValue := tValue * nextValue
    end loop
end term
body procedure primary
   if token = "(" then
        get token
       expn ( pValue )
                           % Evaluate (e)
       assert token = ")"
                           % Evaluate "explicit
    else
       pValue := strreal ( token )
    end if
    get token
end primary
                       % Start by reading first
get token
var answer : real
                 % Scan and evaluate inpl
expn ( answer )
put "Answer is ", answer
```

Execute

Following a **forward** procedure or function declaration, the **body** (procedure must be given at the same level (in the same sequence of statements and declarations as the **forward** declaration). This is the only use of the keyword **body**. See also **body**.

Any procedure or function that is declared using **forward** requires **import** list. In this list, imported procedures or functions that have yet appeared must be listed as **forward**. For example, the import li *expn* is **import forward** *term* ... Before a procedure or function ca called, before its body appears, and before it can be passed as a parameter, its header as well as headers of procedures or functions imported directly or indirectly by it must have appeared.

The keyword **forward** is also used in **collection** and **type** declarati

See also <u>collection</u> and <u>type</u> declarations.

frealstr	real-to-string function
Syntax	<pre>frealstr (r : real, width, fractionWidth : int) : string</pre>
	The frealstr function is used to convert a real number to a string. For example, frealstr $(2.5e1, 5, 1)="b25.0"$ where <i>b</i> represents a blank. The string is an approximation to <i>r</i> , padded on the left with blanks as necessary to a length of <i>width</i> .
	The number of digits of the fraction to be displayed is given by <i>fractionWidth</i> .
Description	The <i>width</i> must be non-negative. If the <i>width</i> parameter is not large enough to represent the value of r , it is implicitly increased as needed.
	The <i>fractionWidth</i> must be non-negative. The displayed value is rounded to the nearest decimal equivalent with this accuracy. In the case of a tie, the value is rounded to the next larger value. The result string is of the form:
	<pre>{blank} [-]{digit}. {digit}</pre>
	If the leftmost digit is zero, then it is the only digit to the left of the decimal point.
	The frealstr function approximates the inverse of strreal , although round-off errors keep these from being exact inverses.
See also	the <u>erealstr</u> , <u>realstr</u> , <u>strreal</u> , <u>intstr</u> and <u>strint</u> functions.

statement

A freeStatement is:

Syntax free [collectionOrClassId,] pointerVariableReference

A free statement destroys (deallocates) an element that has been all
new statement.

Using a collection, declare a list of records and allocate one of thes deallocate the record.

Example	<pre>var list : collection of record</pre>
	 free list, first % Deallocate the element of % Short form: free first
	The free statement sets the pointer variable to the nil value. See the for examples of allocating elements of classes and values of types. locates a type, the <i>collectionOrClassId</i> in the free statement must t
Details	An imported class can have one of its objects destroyed (by the fre only if the class is imported var .
	The <i>collectionOrClassId</i> is optional in the free statement.
See also	<u>class</u> and <u>collection</u> declarations, the <u>pointer</u> type, the <u>new</u> <u>statem</u> value.

free

function

declaration

A functionDeclaration is:

function id [([paramDeclaration {, paramDeclaration }])]

Syntax

: typeSpec

statementsAndDeclarations

end id

DescriptionA function declaration creates (but does not run) a new function.**Description**The name of the function (*id*) is given in two places, just after
function and just after end.

Example	<pre>function doubleIt (x : real result 2.0 * x end doubleIt</pre>	1):	real	
	<pre>put doubleIt (5.3)</pre>	% Th:	is outputs	10.6

The set of parameters declared with the function are called *formal* parameters. For example, in the *doubleIt* function, *x* is a formal parameter. A function is called (invoked) by a *function call* which consists of the function's name followed by the parenthesized list of *actual* parameters (if any). For example, *doubleIt* (5.3) is a call having 5.3 as an actual parameter. If there are no parameters and no parentheses, the call does not have parentheses. The keyword **function** can be abbreviated to **fcn**. See also *functionCall* and *procedureDeclaration*.

Each actual non-**var** parameter must be assignable to the type of its corresponding formal parameter. See also *assignability*.

A function must finish by executing a **result** statement, which produces the function's value. In the above example, the **result** statement computes and returns the value 2.0 * x.

In principle, a function (1) should not change any variables outside of itself (global variables) or (2) should not have **var** parameters. In other words, it should have no *side effects*. The original implementation prevented (1) and (2) and thereby prevented function side effects. Current implementations of Turing do not enforce this restriction.

The upper bounds of arrays and strings that are parameters may be declared to be an asterisk (*), meaning the bound is that of the actual parameter. See *paramDeclaration* for details about parameters.

Details

Procedures and functions cannot be declared inside other procedures and functions.

The syntax of a *functionDeclaration* presented above has been simplified by leaving out the optional result identifier, **import** list, **pre** condition, **init** clause, **post** condition and exception handler. The full syntax is

```
function [ pervasive ] id
  [ ( [ paramDeclaration {, paramDeclaration }
      [ resultId ] : typeSpec
  [ pre trueFalseExpn ]
  [ init id := expn {, id := expn } ]
  [ post trueFalseExpn ]
  [ exceptionHandler ]
  statementsAndDeclarations
end id
```

The *resultId* is the name of the result of the function and can be used only in the **post** condition.

A function must be declared before being called; to allow for mutually recursive procedures and functions, there are **forward** declarations with later declaration of the procedure or function **body**. See **forward** and **body** declarations for explanations.

You declare parameterless functions using an empty parameter list. When this is done, a call to the function must include an empty parameter list. **import** list, **pre** condition, **init** clause, **post** condition and *exceptionHandler* for explanations of these additional features.

See also

See also **<u>pervasive</u>**.

functionCall

A functionCall is:

Syntax

functionId [([expn { , expn }])]

Description A function call is an expression that calls (invokes or activates) a **function**. If the function has parameters, a parenthesized list of expressions (*expns*) must follow the function's name (*functionId*).

This function takes a string containing a blank and returns the first word in the string (all the characters up to the first blank).

Example

Details

<pre>function firstWord (str : string): string for i : 1 length (str) if str (i) = " " then</pre>
result <i>str</i> (1 <i>i</i> - 1)
end if
end for
end firstWord
<pre>put "The first word is: ", firstWord ("Henry Huc</pre>
% The output is Henry.

The parameter declared in the header of a function, is a *formal* parameter, for example, *str* above is a formal parameter. Each expression in the call is an *actual* parameter, for example, *sample* above is an actual parameter.

Each actual parameter passed to its non-**var** formal parameter must be assignable to that parameter (see *assignability* for details). See also *functionDeclaration* and *procedureDeclaration*.

In this explanation of *functionCall*, we have up to this point ignore the possibility of functions exported from modules. If the function is being called from outside of a module from which it has been exported, the syntax of the *functionCall* is:

moduleId . functionId [(expn {, expn })]

In other words, the module or monitor name and a dot must preced

the function's name. If the function is being called from outside of class from which it has been exported, the syntax of the *functionCall* is one of:

(a) classId (p) . functionId [([expn {, exp (b) p -> functionId [([expn {, expn }])]

In these *p* must be a pointer value that locates an object in the class Form (b) is a short form for form (a).

See also <u>class</u>.

file statement

A getStatement is:

Syntax get [: streamNumber ,] getItem { , getItem } The **get** statement inputs each of the *getItems*. Ordinarily, the output comes from the keyboard. However, if the *streamNumber* is presen the input comes from the file specified by the stream number (see t **open** statement for details). Also, input can be redirected so it is tal from a file rather than the keyboard. Check the documentation on t environment for instructions on doing so. The syntax of a *getItem* is one of: (a) variableReference (b) skip (c) variableReference : * Description (d) variableReference : widthExpn These items are used to support three kinds of input: (1) token and character oriented input: supported by forms (a) and (2) line oriented input: supported by form (c), and (3) character oriented input: supported by form (d). Examples of these will be given, followed by detailed explanations Token-oriented input. var name, title : string var weight : real Example get name% If input is Alice, it is inputget title% If input is "A lady", A lady ivar weight% If input is 9.62, it is input

Line-oriented input.

Example

var query : string

get query : * % Entire line is input into quer

Character-oriented input.

Example

```
var code : string
get code : 2 % Next 2 characters are inpu
```

A *token* is a sequence of characters surrounded by *white space*, wh *white space* is defined as the characters: blank, tab, form feed, new and carriage return as well as end-of-file. The sequence of characte making up the token are either all non-white space or else the token must be a quoted string (an explicit string constant). When the *variableReference* in form (a) is a string, integer, real, **int***n*, **nat***n*, o **real***n*. Turing skips white space, reads a token into the *variableReference*, and then skips white space (stopping at the beginning of the next line).

If the *variableReference* is a string, the token is assigned to the var (if the token is quoted, the quotation marks are first removed). See examples involving *name* and *title* above. If the *variableReference* integer or a real, the token is converted to be numeric before being assigned to the variable. See the example involving *weight* above.

Details

When the input is coming from the keyboard, no input is done unti Return is typed. The line that is input may contain more than one to Any tokens that are not input by one **get** statement will remain to b input by the next **get** statement.

Turing has been modified so that token-oriented input now also ski white space following the token, but does not skip beyond the beginning of the next line. This change implies that form (b) is usu not needed, as **skip** was used to skip white space after the token.

Form (a) supports **char** and **char**(n). If the type is **char**, exactly on character is read, with no skipping of white space before or after. T character may be, for example, a blank or a carriage return. If the t is **char**(n), exactly n characters are read, with no skipping of white space.

Inputting **char** and **char**(*n*) types using form (a). The statement *ge*

is not legal, because length specification is not allowed with charac variables.

Example var c : char var d : char (3) get c get d % Read one character. % Read three characters get d Form (a) supports enumerated types. If the type is an enumerated t then the token read in must be one of the elements of the enumerate Details type. Inputting an enumerated type using form (a). The statement *get c:1* not legal, because length specification is not allowed with enumera variables. Example type colors : enum (red, blue, green) **var** c : colors **get** c % Read one of red, green or blue Form (a) supports **boolean**. If the type is an **boolean** type, then the Details token read in must be one of "true" or "false" Inputting a **boolean** type using form (a). The statement *qet c*:1 is n legal, because length specification is not allowed with **boolean** variable. Example var *tf* : boolean % Read one of true or false qet tf In form (b) of *getItem*, **skip** causes white space in the input to be skipped until non-white space (a token) or the end-of-file is reache This is used when the program needs to determine if there are more tokens to be input. To determine if there are more tokens to be read Details program should first **skip** over any possible white space (such as a new line character) and then test to see if **eof** (end-of-file) is true. T is illustrated in this example: Using token-oriented input, input and then output all tokens. This example gives what used to be the standard way of reading tokens

end of file. With the new meaning of form (a) for reading tokens, tl

get skip line can be omitted. This omission is possible because the **get** *word* now automatically skips white space following the input value, up to the beginning of the next line.

	<pre>var word : string loop</pre>	
Example	get skip exit when eof get word put word	% Skip over any white space % Are there more characters? % Input next token % Output the token
	end loop	
	In the above and the payt exam	pla if the input has been redirected

In the above and the next example, if the input has been redirected that it is from a file, **eof** becomes true exactly when there are no me characters to be read. If the input is coming from the keyboard, you signal **eof** by typing control-Z (on a PC) or control-D (on UNIX).

In form (c) of *getItem*, the *variableReference* is followed by :* whi implies line-oriented input. This form causes the entire line (or the remainder of the current line) to be read. In this case the variable m be a string (not an integer or real). The new line character at the en the line is discarded. It is an error to try to read another line when y are already at the end of the file. The following example shows how use line-oriented input to read all lines in the input.

Using line-oriented input, input and then output all lines.

var line : string loop exit when eof % Are there more characters? get line : * % Read entire line put line end loop

In form (d) of getItem, the variableReference is followed by

: widthExpn

Details

Details

Example

which specifies character-oriented input. This form causes the spec number (*widthExpn*) of characters to be input (or all of the remaini characters if not enough are left). If no characters remain, the null string is read and no warning is given. In this form, the new line character is actually input into the *variableReference* (this differs fi line-oriented input which discards new line characters). The follow example shows how to use character-oriented input to read each character of the input. Form (d) can be used with **string** and **char**(*t* variables, but not with **char**, **int** or any other type.

Using character-oriented input, input and then output all characters

	var ch : string (1) loop
Example	exit when eof % Are there more characters? get ch : 1 % Read one character put ch % Output the character, whic % may be a new line character
	end loop
	Using character-oriented input, input two characters.
Example	var d : char (3) := 'abc' get d : 2 % Read two character (replac

See also read statement, which provides binary file input.

getch	get character procedure		
Syntax	getch (var <i>ch</i> : string (1))		
Description	The getch procedure is used to input a single character without waiting for the end of a line. The parameter <i>ch</i> is set to the next character in the keyboard buffer (the oldest not-yet-read character).		
	This program contains a procedure called <i>getKey</i> which causes the program to wait until a key is pressed.		
	<pre>setscreen ("graphics")</pre>		
Example	<pre>procedure getKey var ch : string (1) getch (ch) end getKey</pre>		
	<pre>for i : 1 1000 put i : 4, " Pause till a key is pressed" getKey end for</pre>		
	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. See the setscreen procedure for details. If the screen is not in one of these modes, it will automatically be set to " <i>screen</i> " mode.		
Details	On IBM PC's some keys, such as the left arrow key, insert key, delete key, and function keys do not produce ordinary character values. These keystrokes are returned by getch as their "scan code" with 128 added to them, unless the scan code already has a value of 128 or greater. This provides a unique value for every key on the keyboard. See Appendix D for these codes.		
See also	hasch (has character) which is used to see if a character has been typed but <u>not</u> yet <u>read</u> .		
See also	See also predefined unit Input.		

getchar

get character function

Syntax getchar : char

Description The **getchar** function is used to input a single character without waiting for the end of a line. The next character in the keyboard buffer (the oldest not-yet-read character) is returned.

This program contains a procedure called *getKey* which causes the program to wait until a key is pressed.

setscreen ("graphics")
procedure getKey
var ch : char
ch := getchar
end getKey
for i : 1 .. 1000
put i : 4, " Pause till a key is pressed"
getKey
end for

The screen should be in a "*screen*" or "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in one of these modes, it will automatically be set to "*screen*" mode.

On IBM PC's some keys, such as the left arrow key, insert key, delete key, and function keys do not produce ordinary character values. These keystrokes are returned by **getchar** as their "scan code" with 128 added to it, unless the scan code already has a value of 128 or greater. This provides a unique value for every key on the keyboard. See Appendix D for these codes.

hasch (has character) which is used to see if a character has been typed but <u>not</u> yet <u>read</u>.

See also

See also predefined unit Input.

getenv	get environment function
Syntax	getenv (symbol : string) : string
Description	The getenv function is used to access the environment string whose name is <i>symbol</i> . These strings are determined by the shell (command processor) or the program that caused your program to run. See also the nargs and fetcharg functions.
Example	<pre>On a PC, this retrieves the environment variable USERLEVEL and prints extra instructions if USERLEVEL had been set to NOVICE. USERLEVEL can be set to NOVICE with the command SET USERLEVEL = NOVICE in the autoexec.bat file or in any batch file. const userLevel := string userLevel := getenv ("USERLEVEL") if userLevel = "NOVICE" then</pre>
See also	See also predefined unit <u>Sys</u> .

getpid

get process id function

Syntaxgetpid : intDescriptionThe getpid function is used to determine the I.D. (number) that
identifies the current operating system task (process). Beware that
there are processes, activated by the fork statement, that are
independent of the operating systems tasks.DescriptionUnder UNIX, the number is used, for example, for creating a
unique name of a file.See alsosee also predefined unit Sys.

getpriority

function

Syntax	getpriority : nat
Description	The getpriority function returns the priority of an executing process in a concurrent program. A smaller value means a faster speed.
See also	setpriority, fork and monitor. See also predefined unit <u>Concurrency</u> .

GUI

This unit contains the predefined subprograms for creating and using a GUI (**G**raphical **U**ser **I**nterface). Elements of the GUI include buttons, check boxes, text boxes, scroll bars, menus, etc.

Description

For a general introduction to the the GUI module, see <u>Introduction to Graphical User Interfaces</u>.

GUI.AddLine

Part of **<u>GUI</u>** module

Syntax GUI.AddLine (*widgetID* : **int**, *text* : **string**)

GUI.AddLine adds text and a newline to the current line of the text box specified by *widgetID*. It is essentially equivalent to put *text* in the text box. GUI.AddLine scrolls the text box (if necessary) so that the added text is now visible. The *widgetID* parameter must be the widget id of a text box. The *text* parameter is the text to be added to the text box.

The following creates a text box and puts the numbers from 1 to 25 in it.

Example

```
import GUI
var boxID : int := GUI.CreateTextBox (50, 50, 20
for i : 1 .. 25
    GUI.AddLine (boxID, intstr (i))
end for
loop
    exit when GUI.ProcessEvent
end loop
```

Execute

Exported qualified.

StatusThis means that you can only call the function by callingGUI.AddLine, not by calling AddLine.

See also GUI.CreateTextBox.

GUI.AddText

GUI.AddText (widgetID : int, text : string)		
GUI.AddText adds text to the current line of the text box specified by <i>widgetID</i>. It does not add a newline after the text. It is essentially equivalent to put <i>text</i> in the text box.GUI.AddLine scrolls the text box (if necessary) so that the added text is now visible. The <i>widgetID</i> parameter must be the widget id of a text box. The <i>text</i> parameter is the text to be added to the text box.		
To force a text box to scroll to the end of the text without adding any extra text, call GUI.AddText with "" (the null string) for the <i>text</i> parameter.		
<pre>The following creates a text box and puts the numbers from 1 to 26 followed by the appropriate letter of the alphabet in it. import GUI var boxID : int := GUI.CreateTextBox (50, 50, 20 for i : 1 26 GUI.AddText (boxID, intstr (i)) GUI.AddText (boxID, " ") GUI.AddLine (boxID, chr (64 + i)) end for loop exit when GUI.ProcessEvent end loop</pre>		

Execute

Exported qualified.

- StatusThis means that you can only call the function by calling
GUI.AddText, not by calling AddText.
- See also GUI.CreateTextBox.

GUI.Alert[2,3,Full]

One of four procedures:

Syntax

GUI.Alert (title, msg : string) GUI.Alert2 (title, msg1, msg2 : string) GUI.Alert3 (title, msg1, msg2, msg3 : string) GUI.AlertFull (title : string, msg : array 1 .. * of string, button : string)

Displays a dialog box with the string specified by *msg* in it. There is a single button labelled *OK* which dismisses the dialog and resumes execution. The *title* parameter specifies the window title under Microsoft Windows. On the Apple Macintosh, there is no title, so do not assume the user will see the title. The dialog box is centered on the screen.

DescriptionThe GUI.Alert2 and GUI.Alert3 procedures allow the user to
specify a two or three line message respectively. The
GUI.AlertFull procedure allows the user to specify any number
of lines of text in the string array specified by *msg* as well as the
text in the dismissal button. Any empty strings at the end of the
array are not displayed.

Note: This function is **not** available in the current version of the GUI Procedure Library (shipping with Turing 4.0 and MacOOT 1.5). It is documented here for use with future shipping version of Turing. It is likely to be implemented in the version of Turing released in September 2002. Check the release notes that are found in the on-line help to find out if this function is now available.

The following program asks the user for the name of a file puts up an alert dialog box if it fails.

import GUI in "%oot/lib/GUI"

```
var fileName : string
var streamNumber : int

Example
loop
    fileName := GUI.SaveFile ("Save As")
    open : streamNumber, fileName, put
    exit when streamNumber > 0
    GUI.Alert ("Open Failure", "\"" + fileName +
        "\" could not be opened")
end loop
```

The following program asks the user for the name of a file puts up a more complete alert dialog box if it fails.

```
import GUI in "%oot/lib/GUI"
var fileName : string
var streamNumber : int
loop
    fileName := GUI.SaveFile ("Save As")
    open : streamNumber, fileName, put
    exit when streamNumber > 0
    GUI.Alert2 ("Open Failure",
        "\"" + fileName + "\" could not be opene
        "Reason: " + Error.LastMsg)
end loop
```

Example

The following program fragment displays an alert with four lines of text and a button that says "Abort".

```
var message : array 1 .. 10 of string
for i : 1 .. 10
    message (i) := ""
end for
...
message (1) := "The program must now quit"
message (2) := "becasue of an unrecoverable errc
message (3) := "A Read Error occurred while reac
message (4) := "file \"" + fileName + "\"."
message (5) := Error.LastMsg
GUI.AlertFull ("Error", message, "Abort")
```

Exported qualified.

Status This means that you can only call the function by calling

Example

GUI.Alert, not by calling **Alert**.

Part of **GUI** module

GUI.Choose[Full]

One of two procedures:

Syntax

GUI.Choose (title, msg1, msg2, msg3 : string, btn1, btn2, btn3 : string) : int GUI.ChooseFull (title : string, msg : array 1 .. * of string, btn1, btn2, btn3 : string, defaultBtn : int) : int

Displays a dialog box with text and from one to three buttons. The button to dismiss the dialog. The number of the button pressed is refunction. The dialog box is centered on the screen.

The *title* parameter specifies the title in the window bar of the dialc Apple Macintosh does not have a title bar, so do not assume that th the string in the *title* parameter. The message is specified by strings and *msg3* for **GUI.Choose** and the string array *message* for **GUI.C** each case, empty strings at the end of the list of strings are ignored and *btn3* parameters specify the text to appear in the buttons. If the string (""), the button is not displayed.

Description

The function returns the button number from one to three that was

The *defaultBtn* parameter in **GUI.ChooseFull** specifies which, if a should be the default button. The default button is selected if the us **Enter**. If the default button is 0, then no button is highlighted as th

Note: This function is **not** available in the current version of the G^T Library (shipping with Turing 4.0 and MacOOT 1.5). It is document with future shipping version of Turing. It is likely to be implemented of Turing. Check the release notes that are found in the on-line help this function is now available.

The following program asks if the user wants coffee or tea and set

appropriately.

import GUI in "%oot/lib/GUI"
var wantsCoffee : boolean
var choice : int := GUI.Choose ("Beverage Choice
 "Do you want coffee or tea?", "", "", "Coffe
if choice = 1 then
 wantsCoffee := true
else
 wantsCoffee := false
end if

The following program asks the user whether they want to save the save their work or Cancel.

import GUI in "%oot/lib/GUI" % Returns false if cancelling operation procedure CheckUnsavedWork : boolean var message : array 1 .. 3 of string message (1) := "Changes to " + fileName + " message (2) := "saved. Unsaved work will be Example message (3) := "want to save before quitting var choice : int := GUI.ChooseFull ("Save Be message, "Save", "Don't Save", "Cancel", **if** choice = 1 **then** SaveWork elsif choice = 3 then return false end if return true end CheckUnsavedWork

Exported qualified.

Status This means that you can only call the function by calling **GUI.Ch**c calling **Choose**.

GUI.ClearText

Part of **<u>GUI</u>** module

Syntax GUI.ClearText (widgetID : int)

Description Clears all the text in a text box specified by *widgetID*. The *widgetI*. must be the widget id of a text box.

The program lists 25 numbers in a text box. Every time the button clears the text box and prints the next 25 numbers.

```
import GUI
                    var boxID, buttonID, start : int
                    start := 1
                    procedure PrintTwentyFive
                        GUI.ClearText (boxID)
                        for i : start .. start + 24
                            GUI.AddLine (boxID, intstr (i))
Example
                        end for
                        start += 25
                    end PrintTwentvFive
                    boxID := GUI.CreateTextBox (50, 50, 200, 200)
                    buttonID := GUI.CreateButton (50, 5, 0, "Next 25
                    PrintTwentyFive
                    100p
                        exit when GUI.ProcessEvent
                    end loop
```

Execute

Exported qualified.

Status This means that you can only call the function by calling GUI.Clear by calling ClearText.

See also GUI.CreateTextBox.

GUI.CloseWindow

Syntax GUI.CloseWindow (*window* : **int**)

Closes a window with widgets in it. This procedure automatically disposes of any widgets in the window and makes certain that the (Library recognizes that the window no longer exists. This procedur call *Window.Close*, so there is no need for the user to do so.

The program opens up a window with two buttons. If the button lal "Close and Open" is pressed, the window is closed and a new wind with two buttons is opened in a random location on the screen.

import GUI const screenWidth : int := Config.Display (cdScr const screenHeight : int := Config.Display (cdSc const titleBarHeight : int := 32 const windowEdgeSize : int := 13 const windowWidth : int := 150 const windowHeight : int := 100 var windowID, windowNumber, closeButton, quitBut procedure CloseAndOpen if windowID not= 0 then GUI.CloseWindow (windowID) end if windowNumber += 1 Example var xPos : int := Rand.Int (0, screenWidth windowEdgeSize) var yPos : int := Rand.Int (0, screenHeight *titleBarHeight*) windowID := Window.Open ("title:Window #" + intstr (windowNumber) + ",graphics:" +
intstr (windowWidth) + ";" + intstr (win ", position:" + intstr (xPos) + ";" + int closeButton := GUI.CreateButton (10, 60, 130 "Close And Open", CloseAndOpen) quitButton := GUI.CreateButton (10, 10, 130, end CloseAndOpen CloseAndOpen **100**p exit when GUI.ProcessEvent end loop

Execute

Exported qualified.

- StatusThis means that you can only call the function by calling
GUI.CloseWindow, not by calling CloseWindow.
- GUI.ShowWindowand GUI.HideWindow, for showing and hidiSee alsowindows with widgets in them.

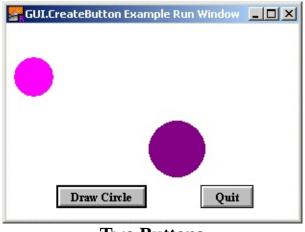
GUI.CreateButton[Full]

GUI.CreateButton (*x*, *y*, *width* : **int**, *text* : **string**, *actionProc* : **procedure** *x* ()) : **int**

Syntax GUI.CreateButtonFull (*x*, *y*, *width* : **int**, *text* : **string**, *actionProc* : **procedure** *x* (), *height* : **int**, *shortcut* : **char**, *default* **int**

Creates a button and returns the button's widget ID.

The button widget is used to implement a textual button. When you button, the button's *action procedure* is called. If a button is given a then entering the keystroke will cause the *action procedure* to be canot visibly cause the button to depress.







The *x* and *y* parameters specify the lower-left corner of the button. parameter specifies the width of the button. If *width* is less than the necessary to display the button, the button is automatically enlarge text. The *text* parameter specifies the text to appear in the button. T parameter is the name of a procedure that is called when the button

For **GUI.CreateButtonFull**, the *height* parameter specifies the hei button. If *height* is less than the size necessary to display the buttor automatically enlarged to fit the text. The *shortcut* parameter is the be used as the button's shortcut. The *default* parameter is a boolean whether the button should be the default button. If there is already button, and *default* is set to true, then this button becomes the new

The following program creates two buttons, one which draws a ran the screen and one which quits the program.

import GUI procedure DrawRandomCircle var r : int := Rand.Int (20, 50) **var** x : **int** := **Rand.Int** (r, maxx - r)**var** y : **int** := **Rand.Int** (r, maxy - r)var c : int := Rand.Int (0, maxcolor) **Draw.FillOval** (x, y, r, r, c)Example % In case we drew over the buttons, redraw t **GUI**.Refresh end DrawRandomCircle View.Set ("graphics:300;200,nobuttonbar ") var draw : int := GUI.CreateButtonFull (50, 10, DrawRandomCircle, 0, '^D', true) var quitBtn : int := GUI.CreateButton (200, 10, **100**p exit when GUI.ProcessEvent end loop

Execute

	When GUI.CreateButton or GUI.CreateButtonFull is called, the created button will be displayed immediately unless GUI.Display has been called with the <i>display</i> parameter set to false.
	If a button's width or height is set to zero (or not specified at all), the shaped to fit the text.
Details	A button can be the default button for a window. The default button with a thicker border around it. If the user presses ENTER in a win default button, the default button's <i>action procedure</i> is called.

When a button is not enabled, the text in the button is grayed out an no longer responds to any mouse clicks or keystrokes until the butt again.

The following GUI subprograms can be called with a button as the parameter:

Details GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable, GUI.Dis GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize, GUI.SetLabel, GUI.SetDefault

Exported qualified.

Status This means that you can only call the function by calling **GUI.Cre** not by calling **CreateButton**.

GUI.SetLabelfor changing the button's text, GUI.SetDefaultSee alsodefault button in a window and GUI.SetColor

GUI.CreateCanvas[Full]

Syntax

Part of **<u>GUI</u>** module

GUI.CreateCanvas (*x*, *y*, *width*, *height* : **int**) : **int**

GUI.CreateCanvasFull (x, y, width, height : int, border : int, mouseDown : procedure x (mx, my : int), mouseDrag : procedure x (mx, my : int), mouseUp : procedure x (mx, my : int)) : int

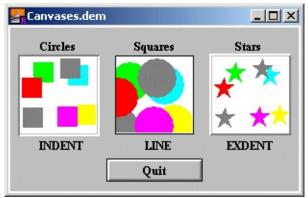
Creates a canvas and returns the canvas' widget ID.

A canvas is a drawing surface for use by the program. It differs from using the window surface to draw on in that (0, 0) represents the lo left corner of the canvas and all drawing is clipped to the canvas. (" means that if you accidently attempt to draw outside of the canvas, will not actually draw beyond the border of the canvas.)

Canvases have procedures that emulate all the procedures in the D_i module as well as a procedure to emulate Font.Draw, Pic.Draw, Pic Pic.ScreenLoad and Pic.ScreenSave.

You can get mouse feedback from a canvas. Using the *CreateCanv* method, you can specify three routines that are called when the mo button is depressed while pointing in a canvas. One routine will be when the user presses the mouse button down in a canvas. Another routine will be called while the user drags the mouse with the mouse button down. This routine is repeatedly called whenever the mouse changes position while the mouse button is down. The last routine called when the mouse button is released. All three routines take ar and *y* parameter, which is the location of the mouse with respect to canvas (i.e. (0, 0) is the lower-left corner of the canvas).

Description



Output of Canvases.dem

The *x* and *y* parameters specify the lower-left corner of the canvas. *width* and *height* parameters specify the width and height of the can

For **GUI.CreateCanvasFull**, the *border* parameter specifies the ty border that surrounds the canvas and is one of 0, *GUI.LINE*, *GUI.INDENT* or *GUI.EXDENT*. A border of 0 is the default and is same as *GUI.LINE*. *GUI.INDENT* and *GUI.EXDENT* only display properly if the background colour has been set to *gray* using **GUI.SetBackgroundColor**. *GUI.INDENT* makes the canvas appeindented or recessed. *GUI.EXDENT* makes the canvas appear to sta out from the window.

The *mouseDown* parameter is a procedure called when the user pre the mouse button in the canvas. The *mouseDrag* parameter is a procedure called when the user drags the mouse while the mouse b is still pressed. The *mouseUp* parameter is a procedure called wher user releases the mouse button. The parameters to all three are the 1 y location of the mouse where the button was pressed (dragged/released). The coordinates are given with respect to the ca (i.e. (0, 0) is the lower-left corner of the canvas).

The following program draws 10 random stars in the canvas.

	import GUI
,	<pre>var canvas : int := GUI.CreateCanvas (10, 10, ma</pre>
	for i : 1 10
Example	<pre>var x : int := Rand.Int (0, maxx - 20)</pre>
-	var y : int := Rand.Int (0, maxy - 20)
	<pre>var c : int := Rand.Int (0, maxcolor)</pre>
	GUI.DrawFillOval (canvas, x, y, 20, 20, c)

end for

	When GUI.CreateCanvas or GUI.CreateCanvasFull is called, th newly created canvas will be displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i> paran set to false.
Details	The border of the canvas is just outside the drawing surface, so GUI.GetWidth and GUI.GetHeight will return slight larger value <i>width</i> and <i>height</i> .
	When the canvas is disabled, clicking the mouse in the canvas does call any of the <i>mouseDown</i> , <i>mouseDrag</i> , or <i>mouseUp</i> procedures. The appearance of the canvas does not change.
	The following GUI subprograms can be called with a button as the <i>widgetID</i> parameter:
Details	GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable, GUI.Dig GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize,
	GUI.Draw, GUI.FontDraw, GUI.Pic, GUI.SetXOR
	GUI.Draw, GUI.FontDraw, GUI.Pic, GUI.SetXOR Exported qualified.
Status	

GUI.CreateCheckBox[Full]

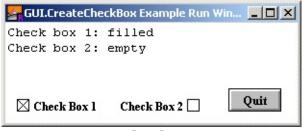
Part of **GUI** module

GUI.CreateCheckBox (*x*, *y* : **int**, *text* : **string**, *actionProc* : **procedure** *x* (*filled* : **boolean**)) : **int**

SyntaxGUI.CreateCheckBoxFull (x, y : int, text : string,
actionProc : procedure x (filled : boolean), alignment : int, sho
int

Creates a check box (with accompanying text) and returns the chec ID.

The check box widget is used to implement a check box that can be When you click on a check box, the status of the check box flips fr unset and back again and the check box's *action procedure* is called status as a parameter. If a check box is given a short cut, then enter keystroke will cause the check box to change status and the *action* be called. The new status will be displayed immediately.



Two Check Boxes

Description The *x* and *y* parameters specify the lower-left corner of the check b *alignment* is set to *GUI.RIGHT*, in which case they specify the low of the check box). The *text* parameter specifies the text (or label) be check box. The *actionProc* parameter is the name of a procedure the when the status of the check box changes. The *actionProc* procedur one boolean parameter which is the new status of the check box. In **GUI.CreateCheckBox**, the check box's text is always to the right of the check box. In **GUI.CreateCheckBoxFull**, the text can be set to the of the check box with the *alignment* parameter.

For **GUI.CreateCheckBoxFull**, the *alignment* parameter specifies of the check box in relation to the text as well as the meaning of the

parameters. The *alignment* parameter is one of 0, *GUI.LEFT*, or *Gi alignment* of 0 is the default and is the same as *GUI.LEFT*. *GUI.LI* actual box in the check box appears to the left of the check box's la specifies the lower-left corner. An *alignment* of *GUI.RIGHT* means actual box appears to the right of the check box's label and (*x*, *y*) sp lower-right corner of the check box. The *shortcut* parameter is the be used as the button's shortcut. The *default* parameter is a boolean whether the button should be the default button. If there is already button, and *default* is set to true, then this button becomes the new

A check box's size is not specified during creation. It is determined size of the text. Instead the user specifies the lower-left corner of th (or the lower-right if the check box is right justified).

The following program creates two buttons, one which draws a ran the screen and one which quits the program

import GUI procedure DoNothing (status : boolean) end DoNothing View.Set ("graphics:300;100, nobuttonbar") var cb1 : int := GUI.CreateCheckBox (10, 10, "Ch DoNothing) var cb2 : int := GUI.CreateCheckBoxFull (200, 10) DoNothing, GUI.RIGHT, '2') **GUI.SetCheckBox** (*cb2*, **true**) var quitBtn : int := GUI.CreateButton (230, 10, Example **loop** exit when GUI.ProcessEvent end loop var cb1Status : boolean := GUI.GetCheckBox (cb1) var cb2Status : boolean := GUI.GetCheckBox (cb2) if cb1Status then put "Check box 1: filled" else put "Check box 1: empty" end if if cb2Status then put "Check box 2: filled" else put "Check box 2: empty" end if

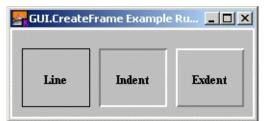
Details	 When GUI.CreateButton or GUI.CreateButtonFull is called, the created check box will be displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i> paran false. When a check box is not enabled, the label beside the check box is
	and the check box no longer responds to any mouse clicks or keyst check box is enabled again.
	The following GUI subprograms can be called with a check box as parameter:
Details	GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable, GUI.Di
Details	GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize, GUI.SetLabel, GUI.GetCheckBox, GUI.SetCheckBox
Details	GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize,
Status	GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize, GUI.SetLabel, GUI.GetCheckBox, GUI.SetCheckBox

GUI.CreateFrame

Syntax GUI.CreateFrame (*x*1, *y*1, *x*2, *y*2, *kind* : **int**) : **int**

Creates a frame and returns the frame's widget ID.

A frame is a box drawn around other GUI widgets to make the win look better and help organize the GUI elements.



Three Types of Frames With a Label in Each Frame

Description Frames are the only GUI widgets that can have other widgets place within them. Frames are passive widgets, meaning that they do not respond to button clicks or keystrokes.

The *x*1 and *y*1 parameters specify the lower-left corner of the frame the *x*2 and *y*2 parameters specify the upper-right corner of the fram *kind* parameter specifies the type of frame. This is one of 0, *GUI.L*. *GUI.INDENT*, or *GUI.EXDENT*. A *kind* of 0 is the default and is th as *GUI.LINE*.

GUI.INDENT and *GUI.EXDENT* only display properly if the back[†] colour has been set to *gray* using **GUI.SetBackgroundColor**. *GUI.INDENT* makes the contents frame appear indented or recesse *GUI.EXDENT* makes the contents of the frame appear to stand out the window.

The following program draws three frames in the window and drav label in each one.

import GUI
View.Set ("graphics:250;90,nobuttonbar")
GUI.SetBackgroundColor (gray)
var lineFrame, indentFrame, exdentFrame : int

	var lineLabel, indentLabel, exdentLabel : int
Example	<pre>lineFrame := GUI.CreateFrame (10, 10, 80, 70, 0)</pre>
•	<pre>indentFrame := GUI.CreateFrame (90, 10, 160, 70,</pre>
	<pre>exdentFrame := GUI.CreateFrame (170, 10, 240, 70</pre>
	% Label the lines.
	<pre>lineLabel := GUI.CreateLabelFull (10, 10, "Line"</pre>
	<i>GUI.CENTER</i> + <i>GUI.MIDDLE</i> , 0)
	<pre>indentLabel := GUI.CreateLabelFull (90, 10, "Inc</pre>
	<i>GUI.CENTER</i> + <i>GUI.MIDDLE</i> , 0)
	<pre>exdentLabel := GUI.CreateLabelFull (170, 10, "Ex</pre>
	GUI.CENTER + GUI.MIDDLE, 0)

Details	When GUI.CreateFrame is called, the newly created frame will b displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i> parameter set to false.
	A frame widget is a passive widget and cannot be enabled or disab
	The following GUI subprograms can be called with a frame as the <i>widgetID</i> parameter:
Details	GUI.Show, GUI.Hide, GUI.Dispose, GUI.GetX, GUI.Get` GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.Se GUI.SetPositionAndSize
Details	GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.Se

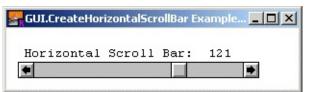
GUI.CreateHorizontalScrollBar[Full] Part of GUI module

GUI.CreateHorizontalScrollBar (*x*, *y*, *size* : **int**, *min*, *max*, *start*: **int**, *actionProc* : **procedure** *x* (*value* : **i**

Syntax GUI.CreateHorizontalScrollBarFull (x, y, size : int, min, max, start : int, actionProc : procedure x (value : int), arrowInc, pageInc, thumbSize : int) : int

Creates a horizontal (left-right) scroll bar and returns the scroll bar

A scroll bar is a widget that allows users to see a piece of a docume cannot be displayed on the screen in its entirety. The picture below horizontal scroll bar. To control a scroll bar, there are a few choices can click on the thumb (the box in the scroll bar) and slide it left or user can click in the scroll bar itself to the left or right of the thumb case the thumb is moved up or down one "page"), or the user can c left or right arrows at the ends of the scroll bar (in which case the th moved left or right one "arrow increment").



A Horizontal Scroll Bar

The programmer defines a page or an arrow increment. When the v scroll bar changes, the *action procedure* of the scroll bar is called v value as a parameter. The *action procedure* should then redraw the using the new value of the scroll bar.

Description

The range of values that the scroll bar will give is determined by th *max* parameters in the *Create* call. The left side of the scroll bar reprinimum value, while the right represents the maximum value. Th the "thumb size". This represents the range of values that can be se the screen.

By default, the arrow increment (the amount the value is changed v scrolling arrows are pressed) is set to one. The page increment (the value is changed when the user clicks in the bar to the right or left is set to one quarter the difference between the minimum and the n The "thumb size" is set to zero (see the description of scroll bars for explanation of the thumb size).

The *x* and *y* parameters specify the lower-left corner of the scroll be parameter specifies the length of the scroll bar (including the arrow The *min* and *max* parameters are the minimum and maximum value by the scroll bar. The *start* parameter is the initial value of the scrol should be between *min* and *max* inclusive. The *actionProc* parameter name of a procedure that is called when the value of the scroll bar i The parameter to the *action procedure* is the current value of the sc

The following program creates a horizontal scroll bar. Whenever the value is changed, a message is displayed in the window.

import GUI View.Set ("graphics:300;60,nobuttonbar") var scrollBar : int procedure ScrollBarMoved (value : int) Text.Locate (2, 3) put "Horizontal Scroll Bar: ", value : 4 end ScrollBarMoved scrollBar := GUI.CreateHorizontalScrollBar (10, 50, 150, 50, ScrollBarMoved) loop exit when GUI.ProcessEvent end loop

Execute

For GUI.CreateHorizontalScrollBarFull, the arrowInc parameter

the arrow increment (the amount the scroll bar's value is changed w scroll arrows are pressed). The *pageInc* specifies the page increme amount the scroll bar's value is changed when the user clicks in the left/right section of the scroll bar). The *thumbSize* parameter specif "thumb size". (See the scroll bar explanation for more detail on a so "thumb size").

Description

For example, if you have a window that can display 20 lines of text there are 100 lines of text, you would set *min* to 1, *max* to 100 and 20. The value returned by the scroll bar would then be the line num first line on the screen to be displayed. When the scroll bar was at i value, it would return 81, since by doing so, lines 81-100 would be

Here is an example program that scrolls a large picture over a smal

% The "ScrollPic" program. import GUI var h, v : int %
var canvas : int
var pic : int % % The scroll bars. % The canvas. % The picture. const width : int := 220 % The width of the c procedure ScrollPic (ignore : int) % Get the current value of the scroll bars. var x : int := GUI.GetSliderValue (h) var y : int := GUI.GetSliderValue (v) **GUI.PicDraw** (canvas, pic, -x, -y, picCopy) end ScrollPic pic := Pic.FileNew ("Forest.jpg") if pic <= 0 then put "Error loading picture: ", Error.LastMsg Example return end if View.Set ("graphics:265;265") canvas := GUI.CreateCanvas (15, 15 + GUI.GetScro width, width) % Note the frame of the canvas is: (14, 14 + ScrollbarWidth) - (235, 235 + Scrc % *h* := GUI.CreateHorizontalScrollBarFull (14, 14, 221, 0, Pic.Width (pic) , 0, ScrollPic, 3, 1 v := GUI.CreateVerticalScrollBarFull (235, 14 + GUI.GetScrollBarWidth, 221, 0, Pic.Heig

```
Pic.Height (pic), ScrollPic, 3, 100, width)
ScrollPic (0) % Draw the picture initially
```

loop
 exit when GUI.ProcessEvent
end loop

	In some instances, you will want the the minimum and maximum v scroll bar to be reversed (right/top is minimum). In that case, call th GUI.SetSliderReverse procedure to flip the values of the scroll ba Scroll bars always have a fixed height (for horizontal scroll bars) o vertical scroll bars). To get a scroll bar's width, use the GUI.GetScrollBarWidth function.
Details	When GUI.CreateHorizontalScrollBar or GUI.CreateHorizontalScrollBarFull is called, the newly created will be displayed immediately unless GUI.DisplayWhenCreated called with the <i>display</i> parameter set to false.
	When a scroll bar is not enabled, the gray in the bar is set to white thumb is not displayed. The scroll bar no longer responds to any m until the scroll bar is enabled again.
	The following GUI subprograms can be called with a scroll bar as parameter:
Details	GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable, GUI.Di GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize, GUI.GetSliderValue, GUI.SetSliderValue, GUI.SetSlider GUI.SetSliderSize, GUI.SetSliderReverse, GUI.SetScroll/

Exported qualified.

See also

StatusThis means that you can only call the function by callingGUI.CreateHorizontalScrollBar, not by calling CreateHorizont

GUI.GetSliderValue and **GUI.SetSliderValue** for reading and set value of a scroll bar, **GUI.SetSliderMinMax** for changing the min maximum values of a scroll bar, and **GUI.SetScrollAmount** for ch scrolling increments and thumb size of a scroll bar. See also **GUI.S** for setting the <u>length</u> of a scroll bar and **GUI.SetSliderReverse** for the sense of a scroll bar.

GUI.CreateHorizontalSlider

Syntax

Part of **<u>GUI</u>** module

GUI.CreateHorizontalSlider (*x*, *y*, *length* : **int**,

min, max, start : int, actionProc : procedure x (value :
int)) : int

Creates a horizontal (left-right) slider and returns the slider's widget ID.

A slider is a widget that allows the user to set a continuous set of values. It has a real-life equivalent in things such as a stereo volume control.

GUI. Cr	ateHorizontalSlider Example Ru 💻 🗖 🔀
Horiz	ontal Slider: 111
<u> </u>	
	1 1

A Horizontal Slider

To control a slider, the user clicks on the slider box and drags it back and forth. Every time the value changes, a procedure is called with the new value as a parameter.

Description The range of values that the slider will give is determined by the *min* and *max* parameters in the *Create* call. The left side of the slider represents the minimum value, while the right represents the maximum value.

The *x* and *y* parameters specify the lower-left corner of the slider track. This means that the slider actually extends above and below this point (and slightly to the left of it to take into account the rounded end of the track). The *length* parameter specifies the length of the track in pixels. (You can use **GUI.GetX**, **GetY**, **GetWidth**, and **GetHeight** to get the exact dimensions of the slider.) The *min* and *max* parameters are the minimum and maximum values returned by the slider. The *start* parameter is the initial value of the slider and should be between *min* and *max* inclusive. The *actionProc* parameter is the name of a procedure

that is called when the value of the slider is changed. The parameter to the *action procedure* is the current value of the slider.

The following program creates a horizontal slider. Whenever the slider's value is changed, a message is displayed in the window.

```
import GUI
View.Set ("graphics:300;60,nobuttonbar")
var slider : int
procedure SliderMoved (value : int)
Text.Locate (2, 3)
put "Horizontal Slider: ", value : 4
end SliderMoved
slider := GUI.CreateHorizontalSlider (10, 10, 25
50, 150, 50, SliderMoved )
loop
exit when GUI.ProcessEvent
end loop
```

	In some instances, you will want the the minimum and maximum values of the slider to be reversed (right is minimum). In that case, call the GUI.SetSliderReverse procedure to flip the values of the slider.
	Sliders always have a fixed height (for horizontal sliders) or width (for vertical sliders).
Details	When GUI.CreateHorizontalSlider or GUI.CreateHorizontalSliderFull is called, the newly created slider will be displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i>

parameter set to false.

When a slider is not enabled, the appearance does not change. However, the slider no longer responds to any mouse clicks until it is enabled again.

The following GUI subprograms can be called with a slider as the *widgetID* parameter:

GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable,
GUI.Dispose, GUI.GetX, GUI.GetY, GUI.GetWidth,
GUI.GetHeight, GUI.SetPosition, GUI.SetSize,
GUI.SetPositionAndSize, GUI.GetSliderValue,
GUI.SetSliderValue, GUI.SetSliderValue, GUI.SetSliderSize, GUI.SetSliderReverse

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.CreateHorizontalSlider, not by calling
CreateHorizontalSlider.

GUI.GetSliderValueand GUI.SetSliderValuefor reading andsetting the value of a slider, GUI.SetSliderMinMaxfor changingthe minimum and maximum values of a slider. See alsoGUI.SetSliderSizefor setting the length of a slider andGUI.SetSliderReversefor reversing the sense of a slider.

GUI.CreateLabel[Full]

GUI.CreateLabel (*x*, *y* : **int**, *text* : **string**) : **int**

Syntax GUI.CreateLabelFull (*x*, *y* : **int**, *text* : **string**, *width*, *height*, *alignment*, *fontID* : **int**) : **int**

Creates a label and returns the label's widget ID.

The label widget is used to display text. It can be used to display te of fonts and sizes. Label widgets can also be aligned in a variety of

GUI.CreateLabel Examp	le Run Window 📃 🗖 🗙
	Upper-Right
Сет	nter
Lower-Left	
Three	Labels

The *x* and *y* parameters specify the lower-left corner of the area in will be drawn. For **GUI.CreateLabel**, this is the lower-left corner *text* parameter specifies the text of the label.

Description For **GUI.CreateLabelFull**, the *width* and *height* parameters specif which the label is to appear. This is used for alignment purposes. S below for an example of aligning the text to different corners of the *alignment* parameter specifies the alignment of the text in the text a is the sum of horizontal alignment and the vertical alignment. The alignment is one of 0, *GUI.LEFT*, *GUI.CENTER*, or *GUI.RIGHT*. . alignment of 0 is the default and is the same as the alignment of *Gl* vertical alignment is one of 0, *GUI.TOP*, *GUI.MIDDLE*, or *GUI.B* horizontal alignment of 0 is the default and is the same as the align *GUI.BOTTOM*. These alignments align the text in various ways in The *fontID* parameter specifies the font ID of the font to be used in The font ID is received from a *Font.New* call. Do not call *Font.Fre* ID until the label has been disposed of by calling **GUI.Dispose**.

By using the *fondID* parameter, labels can be have any size or type

Labels are passive widgets, meaning that they do not respond to bu keystrokes.

The following program creates three labels, one with the default all other two aligned to appear in the center and upper-right corner of

import GUI
View.Set ("graphics:300;100,nobuttonbar ")
var lowerLeft : int := GUI.CreateLabel (0, 0, "L
var center : int := GUI.CreateLabelFull (0, 0, "
GUI.MIDDLE + GUI.CENTER, 0)
var upperRight : int := GUI.CreateLabelFull (0,
maxx, maxy, GUI.RIGHT + GUI.TOP, 0)

Details	When GUI.CreateLabel or GUI.CreateLabelFull is called, the nelabel will be displayed immediately unless GUI.DisplayWhenCre called with the <i>display</i> parameter set to false.
	A frame widget is a passive widget and cannot be enabled or disable
	The following GUI subprograms can be called with a label as the <i>v</i> parameter:
Details	GUI.Show, GUI.Hide, GUI.Dispose, GUI.GetX, GUI.Get' GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.Se GUI.SetPositionAndSize, GUI.GetSliderValue, GUI.SetSl GUI.SetSliderMinMax, GUI.SetLabel
	Exported qualified.
Status	This means that you can only call the function by calling GUI.Cre by calling CreateLabel .

GUI.SetLabel for changing the label's text.

GUI.CreateLabelledFrame

Part of **<u>GUI</u>** module

GUI.CreateLabelledFrame (x1, y1, x2, y2, kind : int,Syntaxtext : string) : int

Creates a labelled frame and returns the frame's widget ID.

A labelled frame is a box with a text label drawn around other GUI make the window look better and help organize the GUI elements.

GUI.CreateLa	abelledFrame	Еха 💶 🗵 🗙
Line —	Indent —	Exdent

Three Types of Labelled Frames

Description Frames and labelled frames are the only GUI widgets that can have widgets placed within them. Labelled frames are passive widgets, I that they do not respond to button clicks or keystrokes.

The *x1* and *y1* parameters specify the lower-left corner of the frame *x2* and *y2* parameters specify the upper-right corner of the frame. (' will extend above the frame.) The *kind* parameter specifies the type This is one of 0, *GUI.LINE*, *GUI.INDENT* or *GUI.EXDENT*. A *kin* the default and is the same as *GUI.LINE*.

GUI.INDENT and *GUI.EXDENT* only display properly if the back colour has been set to *gray* using **GUI.SetBackgroundColor**. *GUI* makes the contents frame appear indented or recessed. *GUI.EXDE*. the contents of the frame appear to stand out from the window.

The following program draws three frames in the window.

import GUIView.Set ("graphics:250;90,nobuttonbar")GUI.SetBackgroundColor (gray)var lineFrame, indentFrame, exdentFrame : intlineFrame := GUI.CreateLabelledFrame (10, 10, 80)

indentFrame := GUI. CreateLabelledFrame (90, 10, GUI.INDENT, "Indent") exdentFrame := GUI. CreateLabelledFrame (170, 10 GUI.EXDENT, "Exdent")

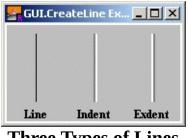
Details	When GUI.CreateLabelledFrame is called, the newly created lab frame will be displayed immediately unless GUI.DisplayWhenCr been called with the <i>display</i> parameter set to false.A labelled frame widget is a passive widget and cannot be enabled disabled.
	The following GUI subprograms can be called with a labelled fram <i>widgetID</i> parameter:
Details	GUI.Show, GUI.Hide, GUI.Dispose, GUI.GetX, GUI.Get GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.Se GUI.SetPositionAndSize, GUI.SetLabel
	Exported qualified.
Status	This means that you can only call the function by calling GUI.CreateLabelledFrame , not by calling CreateLabelledFram
See also	<u>GUI.SetLabel</u> for changing the frame's text.

GUI.CreateLine

Syntax GUI.CreateLine (*x*1, *y*1, *x*2, *y*2, *kind* : **int**) : **int**

Creates a line and returns the line's widget ID.

Lines are generally used to separate parts of a window. A line is us look better and help organize the GUI elements.



Three Types of Lines

Description

Lines are passive widgets, meaning that they do not respond to but

The *x*1 and *y*1 parameters specify one end-point of the line and the specify the other end point. The line must either be horizontal or ve x2 or y1 must equal y2). The *kind* parameter specifies the type of li *GUI.LINE*, *GUI.INDENT* or *GUI.EXDENT*. A *kind* of 0 is the defa *GUI.LINE*.

GUI.INDENT and *GUI.EXDENT* only display properly if the back set to *gray* using **GUI.SetBackgroundColor**. *GUI.INDENT* makes or recessed. *GUI.EXDENT* makes the line appear to stand out from

The following program draws three lines with three labels in the w

import GUI View.Set ("graphics:180;100,nobuttonbar") GUI.SetBackgroundColor (gray) var line, indentLine, exdentLine : int var lineLabel, indentLabel, exdentLabel : int line := GUI.CreateLine (30, 20, 30, 90, 0) indentLine := GUI.CreateLine (90, 20, 90, 90, GL exdentLine := GUI.CreateLine (150, 20, 150, 90, lineLabel := GUI.CreateLabelFull (30, 15, "Line" GUI.CENTER + GUI.TOP, 0)
indentLabel := GUI.CreateLabelFull (90, 15, "Inc
GUI.CENTER + GUI.TOP, 0)
exdentLabel := GUI.CreateLabelFull (150, 15, "Ex
GUI.CENTER + GUI.TOP, 0)

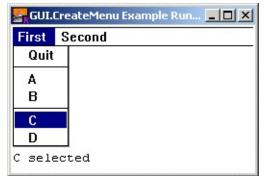
Details	When GUI.CreateLine is called, the newly created line will be dis unless GUI.DisplayWhenCreated has been called with the <i>displa</i>
Details	A line widget is a passive widget and cannot be enabled or disabled
	The following GUI subprograms can be called with a line as the wi
Details	GUI.Show, GUI.Hide, GUI.Dispose, GUI.GetX, GUI.Get` GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetP
Details	

GUI.CreateMenu

Syntax **GUI.CreateMenu** (*name* : string) : int

Creates a menu and returns the menu's widget ID. The menu will b after the other menus in the menu bar. If there are no previous men menu bar is automatically created and the menu added.

The *name* parameter specifies the text that appears in the menu bar



Description

A Menu With an Item Selected

Menus are used in most modern interfaces. In order to create a full menus, you must create the menu and then create the menu items in menu. The menus are automatically added to the menu bar of the so menu.

As of the v1.0 release of the GUI Library, it is an error to create a r without having created a menu first. In future releases it will be por create menus and attach and remove them from menu bars when de

The following program creates a series of menus with menu items It then disables the second menu.

import GUI in "%oot/lib/GUI"
View.Set ("graphics:250;150,nobuttonbar")
var first, second : int % The menus.
var item : array 1 .. 12 of int % The menu items
var name : array 1 .. 12 of string (20) :=
 init ("Quit", "---", "A", "B", "---", "C", "
 "Disable B Menu Item", "Enable B Menu Item",

"Disable Second Menu", "Enable Second Menu") procedure MenuSelected for i : 1 .. 12 **if** *item* (i) = **GUI.GetEventWidgetID then Text.Locate** (*maxrow*, 1) put name (i) + " selected end if end for end MenuSelected procedure DisableB **GUI.Disable** (*item* (4)) end DisableB procedure EnableB Example **GUI.Enable** (*item* (4)) end EnableB procedure DisableFirst **GUI.Disable** (first) end DisableFirst procedure EnableFirst **GUI.Enable** (first) **end** *EnableFirst* % Create the menus first := GUI.CreateMenu ("First") item (1) := GUI.CreateMenuItem (name (1), GUI.Qu for cnt : 2 .. 7 item (cnt) := GUI.CreateMenuItem (name (cnt)) *MenuSelected*) end for second := GUI.CreateMenu ("Second") item (8) := GUI.CreateMenuItem (name (8), Disabl item (9) := GUI.CreateMenuItem (name (9), Enable item (10) := GUI.CreateMenuItem (name (10), Menu item (11) := GUI.CreateMenuItem (name (11), Disa item (12) := GUI.CreateMenuItem (name (12), Enat **100**p exit when GUI.ProcessEvent end loop

Details	When a menu is not enabled, the text in the menu bar is grayed out clicking on the menu does not cause the menu to appear.
Details	The following GUI subprograms can be called with a menu as the parameter:
	GUI.Show, GUI.Hide, GUI.Dispose, GUI.Enable, GUI.Disable
Status	Exported qualified.
	This means that you can only call the function by calling GUI.CreateMenu , not by calling CreateMenu .
See also	<u>GUI.CreateMenuItem</u> for adding items to a menu. See also <u>GUI.ShowMenuBar</u> and <u>GUI.HideMenuBar</u> for showing and hid menu bar.

GUI.CreateMenuItem[Full]

Part of **GUI** module

GUI.CreateMenuItem (name : string		
actionProc : procedure x ()) : int		

Creates a menu item and returns the menu item's widget ID.

Menu items are the individual entries of a menu. To create menus for a window, you must create a menu, then create the menu items for that menu, then create the next menu, etc. All menu items are automatically added to the last menu and after the last menu item of the currently selected (not active!) window.

The menu item will be added to the last menu after the other menu items in the menu. If there are no menus defined, an error results.

Description

The *name* parameter specifies the text that is to appear. A *name* of three dashes ("---") creates a separator across the menu. The *actionProc* parameter specifies the name of a procedure to be called when user the selects the menu item from the menu.

For **GUI.CreateMenuItemFull**, the *shortCut* parameter specifies the keystroke to be used as the menu item's shortcut. If no shortcut is desired, then '\0' can be used. The *addNow* parameter has no effect in the current version of the GUI Library. In future versions, it will allow you to create menu items that can then be added to a menu later in the program.

Examples See the example for <u>**GUI.CreateMenu</u></u>.</u>**

Details	When a menu item is not enabled, the text of the menu item is grayed out and clicking on the menu item does not cause the menu to appear.
Details	The following GUI subprograms can be called with a menu as the <i>widgetID</i> parameter:
	GUI.Show, GUI.Hide, GUI.Dispose, GUI.Enable, GUI.Disable
	Exported qualified.
Status	This means that you can only call the function by calling GUI.CreateMenuItem , not by calling CreateMenuItem .

GUI.CreatePicture

Part of **<u>GUI</u>** module

GUI.CreatePicture (*x*, *y*, *picture* : **int**, *mergePic* : **boolean**) : **int**

Creates a picture and returns the picture's widget ID.

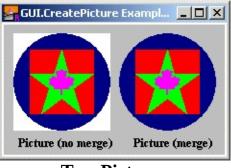
The picture widget is used to display a picture. It can be used to dismerged into the background or not.

The *x* and *y* parameters specify the lower-left corner of the picture. parameter specifies the picture ID of the picture. The picture ID is **Pic.New** or **Pic.FileNew** call. Do not call **Pic.Free** for this picture has been disposed of by calling **GUI.Dispose**. The *mergePic* paran specifies whether anything that was the background colour in the p 0) should be set to the background colour of the window.

Description

Syntax

A picture widget is a passive widget and cannot be enabled or disal



Two Pictures

The following program draws two pictures, merged and not merged

import GUI View.Set ("graphics:230;135,nobuttonbar") % We'll need to create a picture for our Picture % an external file (and Pic.FileNew) would be us Draw.FillOval (50, 50, 50, 50, blue) Draw.FillBox (17, 17, 83, 83, brightred) Draw.FillStar (17, 17, 83, 83, brightgreen) Draw.FillMapleLeaf (37, 37, 63, 63, brightpurple var pic : int := Pic.New (0, 0, 100, 100)

Example

var picture1, picture2 : int var label1, label2 : int GUI.SetBackgroundColor (gray) label1 := GUI.CreateLabel (15, 5, "Picture (no m picture1 := GUI.CreatePicture (10, 25, pic, fals label2 := GUI.CreateLabel (135, 5, "Picture (mer picture2 := GUI.CreatePicture (120, 25, pic, tru

Details	When GUI.CreatePicture is called, the newly created picture will immediately unless GUI.DisplayWhenCreated has been called we parameter set to false.
	A picture widget is a passive widget and cannot be enabled or disal
	The following GUI subprograms can be called with a picture as the
Details	GUI.Show, GUI.Hide, GUI.Dispose, GUI.GetX, GUI.Get` GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetP
	Exported qualified.

GUI.CreatePictureButton[Full]

Description

Part of **<u>GUI</u>** module

GUI.CreatePictureButton (*x*, *y*, *picture* : **int**, *actionProc* : **procedure** x ()) : **int**

Syntax GUI.CreatePictureButtonFull (*x*, *y*, *picture* : **int**, *actionProc* : **procedure** x (), *width*, *height* : **int**, *shortcut* : **char**, *mergePic* : **boolean**) : **int**

Creates a picture button and returns the button's widget ID.

Picture buttons behave like buttons (see **GUI.CreateButton**) except of text on the button, a picture specified by the user is displayed on The picture button widget responds to mouse clicks and keystrokes manner as a regular button widget.

The picture must be created by the program beforehand using **Pic.I Pic.FileNew**. The resulting picture can then be used as a parameter **GUI.CreatePictureButton**. In general, pictures should be a maxin 50 pixels high and wide, although there is no built-in limit in the G

The *x* and *y* parameters specify the lower-left corner of the picture *picture* parameter specifies the picture ID of the picture to be displabutton. (Note that, in general, this picture should be fairly small.) T is received from a **Pic.New** or **Pic.FileNew** call. Do not call **Pic.Fr** picture ID until the button has been disposed of by calling **GUI.Di***actionProc* parameter specifies the name of a procedure that is call picture button is pressed.

For **GUI.CreatePictureButtonFull**, the *width* and *height* paramete width and height of the button. If they are set to 0, then the picture automatically sized to fit the picture. If you need to know the preci button, use the **GUI.GetWidth** and **GUI.GetHeight** functions. If *v height* are larger than the picture, the picture is centered in the buttor *shortCut*parameter is the keystroke to be used as the button's shortc *mergePic* parameter specifies whether anything that was the backg in the picture (usually colour 0) should be set to the background co button (which is usually gray). This defaults to true for *CreatePict*



Two Picture Buttons

The following program displays five picture buttons which output when pressed.

import GUI View.Set ("graphics:100;70") const size : int := 25 % The buttons size. const border : int := 3 **var** starButton, mapleButton, starPic, mapleLeafF procedure StarPressed Text.Locate (1, 1) п **put** "Star Pressed end StarPressed procedure MaplePressed **Text.Locate** (1, 1) put "Maple Pressed " end MaplePressed % Create the pictures. % The star. Draw.Star (border, border, border + size, border Example **Draw.Star** (border + 1, border + 1, border + size border + size - 1, black) Draw.FillStar (border + 2, border + 2, border + border + size - 2, brightred) starPic := **Pic.New** (0, 0, 2 * border + size, 2 * % The mapleleaf. Draw.FillBox (border, border, border + size, bor **Draw.MapleLeaf** (border, border, border + size, k Draw.MapleLeaf (border + 1, border + 1, border + border + size - 1, black) **Draw.FillMapleLeaf** (border + 2, border + 2, borc border + size - 2, brightred) mapleLeafPic := **Pic.New** (0, 0, 2 * border + size % Create the picture buttons. Draw.Cls

loop
 exit when GUI.ProcessEvent
end loop

Execute

Details	When GUI.CreatePictureButton or GUI.CreatePictureButtonF the newly created picture will be displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i> paran false. When a picture button is not enabled, the picture button is grayed c
	picture button no longer responds to any mouse clicks or keystroke button is enabled again.
Details	The following GUI subprograms can be called with a picture butto widgetID parameter:

Status Exported qualified.

This means that you can only call the function by calling **GUI.CreatePictureButton**, not by calling **CreatePictureButton**.

GUI.CreatePictureRadioButton[Full] Part of GUI module

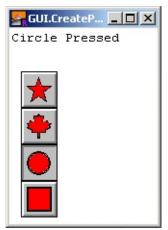
GUI.CreatePictureRadioButton (*x*, *y*, *picture* : **int**, *joinID* : **int**, *actionProc* : **procedure** *x* ()):**int**

Syntax GUI.CreatePictureRadioButtonFull (*x*, *y* : **int**, *picture*, *joinID* : **int**, *actionProc* : **procedure** *x* (), *width,height* : *shortcut* : **char**, *mergePic* : **boolean**) : **int**

Creates a picture radio button and returns the button's widget ID.

Picture radio buttons behave like picture buttons (see **GUI.CreatePictureButton**) except that they have the "radio" prop one of the buttons in the radio group is always selected, and if anot the group is selected, the previously selected button is unselected.

A common example is the buttons on a paint program that indicate shape being painted. The maple leaf button is currently selected. If is selected by the user, the maple leaf button becomes unselected. I buttons, the selected button appears depressed.



Four Picture Radio Buttons with the Maple Leaf Selec

A radio group is created by first creating a single radio button. To a button to the group, a second radio button is created specifying the button in the *joinID* parameter. Subsequent radio buttons are added specifying a previous member of the group in the *joinID* parameter

Description

The picture must be created by the program beforehand using **Pic.I Pic.FileNew**. The resulting picture can then be used as a parameter **GUI.CreatePictureButton**. In general, pictures should be a maxin 50 pixels high and wide, although there is no built-in limit in the G

The *x* and *y* parameters specify the lower-left corner of the picture If these are both 1 and *joinID* is not zero, then the button will be pl below the previous picture radio button in the group. The *picture* p specifies the picture ID of the picture to be displayed on the button general, this picture should be fairly small.) The picture ID is receient **Pic.New** or **Pic.FileNew** call. Do not call **Pic.Free** for this picture button has been disposed of by calling **GUI.Dispose**. The *joinID* p specifies a member of the radio group that this widget should join. sepecifies this radio button is not a member of any group. The *actic* parameter specifies the name of a procedure that is called when the button is pressed.

For **GUI.CreatePictureRadioButtonFull**, the *width* and *height* pa specify the width and height of the button. If they are set to 0, then radio button is automatically sized to fit the picture. If you need to precise size of the button, use the **GUI.GetWidth** and **GUI.GetHe** functions. If *width* and *height* are larger than the picture, the picture the button. The *shortCut*parameter is the keystroke to be used as th shortcut. The *mergePic* parameter specifies whether anything that v background colour in the picture (usually colour 0) should be set to background colour of the button (which is usually gray). This defau *CreatePictureRadioButton*.

The following program creates and displays for picture radio butto

```
import GUI
View.Set ("graphics:150;200, nobuttonbar")
const size : int := 25 % The buttons size.
const border : int := 3
var starButton, mapleButton, circleButton, squar
var starPic, mapleLeafPic, circlePic, squarePic
procedure StarPressed
    Text.Locate (1, 1)
    put "Star Pressed "
```

end StarPressed **procedure** *MaplePressed* Text.Locate (1, 1) put "Maple Pressed " end MaplePressed **procedure** *CirclePressed* **Text.Locate** (1, 1) put "Circle Pressed" end CirclePressed procedure SquarePressed Text.Locate (1, 1) put "Square Pressed" end SquarePressed % Create the pictures. % The star. **Draw.Star** (border, border, border + size, border **Draw.Star** (border + 1, border + 1, border + size border + size - 1, black) **Draw.FillStar** (border + 2, border + 2, border + border + size - 2, brightred) starPic := **Pic.New** (0, 0, 2 * border + size, 2 * % The mapleleaf. Draw.FillBox (border, border, border + size, bor Example **Draw.MapleLeaf** (border, border, border + size, *k* Draw.MapleLeaf (border + 1, border + 1, border + border + size - 1, black) **Draw.FillMapleLeaf** (border + 2, border + 2, bora border + size - 2, brightred) mapleLeafPic := **Pic.New** (0, 0, 2 * border + size % The circle. **const** radius : **int** := size **div** 2 **Draw.FillBox** (border, border, border + size, bor **Draw.Oval** (border + radius, border + radius, rad Draw.Oval (border + radius, border + radius, rad black) **Draw.FillOval** (border + radius, border + radius, radius - 2, brightred) circlePic := Pic.New (0, 0, 2 * border + size, 2 % The square. **Draw.FillBox** (border, border, border + size, bor **Draw.Box** (border, border, border + size, border **Draw.Box** (border + 1, border + 1, border + size border + size - 1, black) **Draw.FillBox** (border + 2, border + 2, border + s

```
border + size - 2, brightred)
squarePic := Pic.New (0, 0, 2 * border + size, 2
% Create the picture buttons.
Draw.Cls
starButton := GUI. CreatePictureRadioButton (10,
    starPic, 0, StarPressed)
mapleButton := GUI. CreatePictureRadioButton (-1
    mapleLeafPic, starButton, MaplePressed)
circleButton := GUI.CreatePictureRadioButton (-1
    circlePic, mapleButton, CirclePressed)
squareButton := GUI.CreatePictureRadioButton (-1
    squarePic, circleButton, SquarePressed)
```

```
exit when GUI.ProcessEvent
end loop
```

Details	When GUI.CreatePictureRadioButton or GUI.CreatePictureRadioButtonFull is called, the newly created displayed immediately unless GUI.DisplayWhenCreated has bee the <i>display</i> parameter set to false.
	When a picture radio button is not enabled, the picture radio buttor and the picture button no longer responds to any mouse clicks or ke the button is enabled again.
	The following GUI subprograms can be called with a picture radio <i>widgetID</i> parameter:
Details	GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable, GUI.Di GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize, GUI.SelectRadio

Exported qualified.

StatusThis means that you can only call the function by callingGUI.CreatePictureRadioButton, not by calling CreatePictureRadioButton

See alsoGUI.SelectRadiofor selecting a picture radio button in a program.
GUI.CreatePictureButton and GUI.CreateRadioButton
for info
picture buttons and radio buttons.

GUI.CreateRadioButton[Full]

GUI.CreateRadioButton (*x*, *y* : **int**, *text* : **string**, *joinID* : **int**, *actionProc* : **procedure** *x* ()) : **int**

Syntax GUI.CreateRadioButtonFull (*x*, *y* : **int**, *text* : **string**, *joinID* : **int**, *actionProc* : **procedure** *x* (), *alignment* : **int**, *shortCut* : **char**) : **int**

Creates a radio button and returns the radio button's widget ID.

A slider is a widget that allows the user to select one of a set of values. It has a real-life equivalent in the old car stereos where a single station is selected at a time. That is, one of the buttons in the radio group is always selected, and if another button in the group is selected, the previously selected button is unselected.

GUI.CreateRadio	Button Example Run Window	
Radio Button 2	Selected	
O Radio Button 1 Radio Button A (Shortcut: 5		'a) 🖲
Radio Button 2	Radio Button B (Shortcut:	b) O
🔿 Radio Button 3	Radio Button C (Shortcut:	6) ()

Six Radio Buttons in Two Groups

A radio group is created by first creating a single radio button. To add another button to the group, a second radio button is created specifying the first radio button in the *joinID* parameter. Subsequer radio buttons are added, each specifying a previous member of the group in the *joinID* parameter.

The *x* and *y* parameters specify the lower-left corner of the radio button (unless *alignment* is set to *GUI.RIGHT*, in which case they specify the lower-right corner of the radio button). If these are both 1 and *joinID* is not zero, then the button will be placed directly below the previous radio button in the group. The *text* parameter specifies the text (or label) beside the radio button. The *joinID* parameter specifies a member of the radio group that this widget should join. A *joinID* of 0 sepecifies this radio button is not a

member of any group. The *actionProc* parameter is the name of a procedure that is called when the radio button is selected. In **GUI.CreateRadioButton**, the radio button's text is always to the right of the actual radio button. In **GUI.CreateRadioButtonFull**, the text can be set to the right or left of the radio button with the *alignment* parameter.

For **GUI.CreateRadioButtonFull**, the *alignment* parameter specifies the position of the radio button in relation to the text as well as the meaning of the *x* and *y* parameters. The *alignment* parameter is one of 0, *GUI.LEFT*, or *GUI.RIGHT*. An *alignment* of 0 is the default and is the same as *GUI.LEFT*. *GUI.LEFT* means th actual box in the check box appears to the left of the check box's label and (*x*, *y*) specify the lower-left corner. An *alignment* of *GUI.RIGHT* means that the actual box appears to the right of the radio button's label and (*x*, *y*) specify the lower-right corner of the radio button. The *shortcut* parameter is the keystroke to be used as the button's shortcut.

A radio button's size is not specified during creation. It is determined based on the size of the text. Instead the user specifies the lower-left corner of the radio button (or the lower-right if the radio button is right justified).

The following program creates six radio buttons in two groups.

```
import GUI
View.Set ("graphics:350;80")
var radio : array 1 .. 6 of int % The radio butt
procedure RadioPressed
Text.Locate (1, 1)
put "Radio Button " ..
for i : 1 .. 6
    if radio (i) = GUI.GetEventWidgetID ther
        put i ..
    end if
end for
put " Selected"
end RadioPressed
radio (1) := GUI.CreateRadioButton (15, maxy 35)
```

"Radio Button 1", 0, RadioPressed) Example radio (2) := GUI.CreateRadioButton (1, 1, "Radi radio (1), RadioPressed) radio (3) := GUI.CreateRadioButton (1, 1, "Radi radio (2), RadioPressed) radio (4) := GUI.CreateRadioButtonFull (maxx 15 "Radio Button A (Shortcut: 'a')", 0, RadioPr GUI.RIGHT, 'a') radio (5) := GUI.CreateRadioButtonFull (1, 1, "Radio Button B (Shortcut: 'b')", radio (4), GUI.RIGHT, 'b') radio (6) := GUI.CreateRadioButtonFull (1, 1, "Radio Button C (Shortcut: 'c')", radio (5), GUI.RIGHT, 'c') **loop**

exit when GUI.ProcessEvent end loop

Execute

Details	When a group of radio buttons is selected, the first radio button created in the group will be the selected one. You can change this by using the GUI.SelectRadio procedure to select a different one. When GUI.CreateRadioButton or GUI.CreateRadioButtonFull is called, the newly created picture will be displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i> parameter set to false.
	When a radio button is not enabled, the radio button is grayed out and the radio button no longer responds to any mouse clicks or keystrokes until the button is enabled again.
	The following GUI subprograms can be called with a radio button as the <i>widgetID</i> parameter:

Details	GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable, GUI.Dispose, GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize, GUI.SetLabel, GUI.SelectRadic	
	Exported qualified.	
Status	This means that you can only call the function by calling GUI.CreateRadioButton , not by calling CreateRadioButton .	
See also	<u>GUI.SelectRadio</u> for selecting a radio button in a program. See also <u>GUI.SetLabel</u> for changing the radio button's text.	

GUI.CreateTextBox[Full]

Part of **<u>GUI</u>** module

GUI.CreateTextBox (*x*, *y*, *width*, *height* : **int**) : **int**

Syntax GUI.CreateTextBoxFull (*x*, *y*, *width*, *height* : **int**, *border*, *fontID* : **int**) : **int**

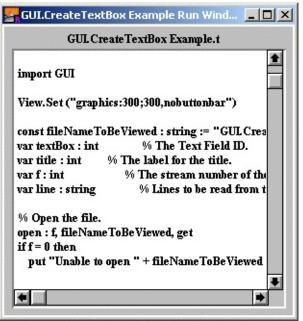
Creates a text box and returns the text box's widget ID.

A text box is a box used for displaying text. It has scroll bars that a text appears outside the border of the text box. The user cannot dire edit or modify the text in the text box.

The *x* and *y* parameters specify the lower-left corner of the area in text will be drawn. The *width* and *height* parameters specify the wi height of the text drawing area The text box border is just outside t drawing area. Because of this, **GUI.GetX** and **GUI.GetY** will retu slightly smaller than *x* and *y* and **GUI.GetWidth** and **GUI.GetHei** return values slightly larger than *width* and *height*.

For GUI.CreateTextBox, the border around the text box is always
 GUI.CreateTextBoxFull, the type of border is specified by the border parameter. The border parameter is one of 0, GUI.LINE, GUI.INDI
 GUI.EXDENT. A border of 0 is the default and is the same as GUI.
 GUI.INDENT and GUI.EXDENT only display properly if the back colour has been set to gray using GUI.SetBackgroundColor. GUI
 makes the text box appear indented or recessed. GUI.EXDENT mal box appear to stand out from the window. The fontID parameter sp font ID of the font to be used in the text box. The font ID is receive Font.New call. Do not call Font.Free for this font ID until the label disposed of by calling GUI.Dispose.

By using the *fondID* parameter, text boxes can have any size or typ



A text box displaying the contents of a file.

The following program displays the contents of a file in a text box.

```
import GUI
                    View.Set ("graphics:300;300")
                    const fileNameToBeViewed : string := "TextBxs.DE
                                            % The Text Field ID.
                    var textBox : int
                    var title : int
                                        % The label for the title.
                    var f : int
                                        % The stream number of the f
                    var line : string
                                            % Lines to be read from
                    % Open the file.
                    open : f, fileNameToBeViewed, get
                    if f = 0 then
                        put "Unable to open " + fileNameToBeViewed +
                        return
                    end if
                    % Set background color to gray for indented text
                    GUI.SetBackgroundColor (gray)
Example
                    % Create the title label and text box.
                    title := GUI.CreateLabelFull (20, 280, fileName1
                        GUI.CENTER, 0)
                    textBox := GUI.CreateTextBoxFull (10, 10, 280, 2
                        GUI.INDENT, 0)
                    % Read the file and place it in the text box.
```

loop exit when eof (f) get : f, line : * GUI.AddLine (textBox, line) end loop close : f % Close the file. loop exit when GUI.ProcessEvent end loop

Execute

Details	 When GUI.CreateTextBox or GUI.CreateTextBoxFull is call created picture will be displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i> particles Is false. 		
	A text box widget is a passive widget and cannot be enabled or disa		
	The following GUI subprograms can be called with a text box as the parameter:		
Details	GUI.Show, GUI.Hide, GUI.Dispose, GUI.GetX, GUI.Get GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.Se GUI.SetPositionAndSize, GUI.AddLine, GUI.AddText, GUI.ClearText, GUI.SetTopLine, GUI.SetScrollOnAdd		
	Exported qualified.		
Status	This means that you can only call the function by calling GUI.Cre not by calling CreateTextBox .		
	GUI.AddLine, GUI.AddText for adding text to the text box. See a		

See also GUI.SetTopLine to set the top line of text in the text box. See also **GUI.SetScrollOnAdd** to set stop the text box from scrolling when to the text box. See also **GUI.ClearText** for clearing the text box.

GUI.CreateTextBoxChoice

Part of **<u>GUI</u>** module

GUI.CreateTextBoxChoice (x, y, width, height : int,Syntaxborder, fontID : int, actionProc : procedure x (line : int)

Creates a text box that can be used for selecting individual items ar box's widget ID.

A text box is a box used for displaying text. By using **GUI.AddLir** the program can then wait for the user to double click on lines in th highlights when the user clicks the line once, and calls the action p user clicks the highlighted line a second time. Like a text box, the t scroll bars that activate when text appears outside the border of the cannot directly edit or modify the text in the text box choice.

The *x* and *y* parameters specify the lower-left corner of the area in ^y drawn. The *width* and *height* parameters specify the width and heig drawing area The text box border is just outside the text drawing an **GUI.GetX** and **GUI.GetY** will return a value slightly smaller than **GUI.GetWidth** and **GUI.GetHeight** will return values slightly lar *height*.

Description For **GUI.CreateTextBox**, the border around the text box is always **GUI.CreateTextBoxFull**, the type of border is specified by the *bol border* parameter is one of 0, *GUI.LINE*, *GUI.INDENT*, or *GUI.EX* is the default and is the same as *GUI.LINE*. *GUI.INDENT* and *GUI* display properly if the background colour has been set to *gray* usin **GUI.SetBackgroundColor**. *GUI.INDENT* makes the text box app recessed. *GUI.EXDENT* makes the text box appear to stand out fro *fontID* parameter specifies the font ID of the font to be used in the is received from a *Font.New* call. Do not call *Font.Free* for this for has been disposed of by calling **GUI.Dispose**.

By using the *fontID* parameter, text boxes can have any size or type

	GUI.CreateTextB	oxChoice R	tun Window	
The	e user chose	7		
1	Choose D			
	Choose E			
	Choose F			
	Choose G			
	Choose H		*	
	會		m	

A text box choice after a user double clicked a

The following program displays the contents of a file in a text box.

import GUI
procedure ChoseLine (line : int)
 location (1, 1)
 put "The user chose ", line
end ChoseLine
var x : int := GUI.CreateTextBoxChoice (20, 20,
GUI.AddLine (x, "Choose A")
GUI.AddLine (x, "Choose B")
GUI.AddLine (x, "Choose B")
GUI.AddLine (x, "Choose C")
GUI.AddLine (x, "Choose E")
GUI.AddLine (x, "Choose F")
GUI.AddLine (x, "Choose F")
GUI.AddLine (x, "Choose H")
GUI.AddLine (x, "Choose I")
GUI.Add

Execute

Details	When GUI.CreateTextBox or GUI.CreateTextBoxFull is called, text box choice will be displayed immediately unless GUI.Display been called with the <i>display</i> parameter set to false.		
	The following GUI subprograms can be called with a text box as the parameter:		
Details	GUI.Show, GUI.Hide, GUI.Dispose, GUI.GetX, GUI.Get GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPo GUI.AddLine, GUI.AddText, GUI.ClearText, GUI.SetTo GUI.SetScrollOnAdd		
	Exported qualified.		
Status	This means that you can only call the function by calling GUI.Cre not by calling CreateTextBoxChoice .		
See also	GUI.AddLine for adding text to the text box. See also GUI.SetTo line of text in the text box. See also GUI.SetScrollOnAdd to set st scrolling when text is added to the text box. See also GUI.ClearTe text box.		

GUI.CreateTextField[Full]

GUI.CreateTextField (*x*, *y*, *width* : **int**, *text* : **string**, *actionProc* : **procedure** *x* (*text* : **string**)) : **int**

Syntax GUI.CreateTextFieldFull (*x*, *y*, *width* : **int**, *text* : **string**, *actionProc* : **procedure** *x* (*text* : **string**), *border*, *fontID*, *inputKind* : **int**) : **int**

Creates a text field and returns the text field's widget ID.

A text field is used to create a line of text that can be edited by the user can use the mouse to select part of the text and can enter text i field.

If one or more text fields are enabled in a window, then one of the will be active. This means that when any keystrokes are entered int window, the active text field will receive the keystrokes. The active can be changed using the **GUI.SetActive** procedure.

The *x* and *y* parameters specify the lower-left corner of the area in text will be drawn. The text field border is just outside the text draw The *width* parameter specifies the width of the text drawing area. T the text field is determined by the height of the font used by the tex border of the text field is just outside the text drawing area, so **GU**I will return values slightly larger than *width*. The *actionProc* param specifies the name of the procedure to be called when the user pres (RETURN on a Macintosh) when the text field is active. The parar current text in the text field.

DescriptionFor **GUI.CreateTextField**, the border around the text field is alway
For **GUI.CreateTextFieldFull**, the type of border is specified by tl
parameter. The *border* parameter is one of 0, *GUI.LINE*, *GUI.INDI*
GUI.EXDENT. A border of 0 is the default and is the same as GUI.
GUI.INDENT and *GUI.EXDENT* only display properly if the back
colour has been set to *gray* using **GUI.SetBackgroundColor.** *GUI*
makes the text field appear indented or recessed. *GUI.EXDENT* ma
field appear to stand out from the window. The *fontID* parameter specified appear indented or recessed.

font ID of the font to be used in the text field. The font ID is receiv *Font.New* call. Do not call *Font.Free* for this font ID until the label disposed of by calling **GUI.Dispose**. The *inputKind* parameter spectype of input accepted by the text field. This is one of 0, *GUI.ANY*, or *GUI.REAL*. An input type of 0 is the default and is the same as (*GUI.ANY* allows any type of input in the text field. *GUI.INTEGER* positive integer input in the text field. *GUI.REAL* allows any real n in the text field. Note that using

GUI.INTEGER or *GUI.REAL* does not guarantee that the text field be converted to an integer or a real. The text could be a null string, *GUI.REAL* could be part of a number such as the string "" or "1.25 which are illegal numbers. (To check the conversion, use the *strint strrealok* functions before calling *strint* or *strreal*.)



Two Text Fields

The following program creates a text field and echoes it on the scre the user presses ENTER.

```
import GUI
View.Set ("graphics:200;100")
var nameTextField, addressTextField : int % Th
procedure NameEntered (text : string)
GUI.SetSelection (addressTextField, 0, 0)
GUI.SetActive (addressTextField)
end NameEntered
procedure AddressEntered (text : string)
GUI.SetSelection (nameTextField, 0, 0)
GUI.SetActive (nameTextField, 0, 0)
GUI.SetActive (nameTextField)
end AddressEntered
Example
GUI.SetBackgroundColor (gray)
var quitButton := GUI.CreateButton (52, 5, 100,
```

```
nameTextField := GUI.CreateTextFieldFull (50, 70)
    NameEntered, GUI.INDENT, 0, 0)
addressTextField := GUI.CreateTextFieldFull (50,
    AddressEntered, GUI.INDENT, 0, 0)
var nameLabel := GUI.CreateLabelFull (45, 70, "N
    GUI.RIGHT, 0)
var addressLabel := GUI.CreateLabelFull (45, 40,
    GUI.RIGHT, 0)
100p
    exit when GUI.ProcessEvent
end loop
GUI.Dispose (quitButton)
colorback (gray)
Text.Locate (maxrow - 1, 1)
put "Name = ", GUI.GetText (nameTextField)
put "Address = ", GUI.GetText (addressTextField)
```

Execute

Only one text field is active at a time. The active text field has a bli cursor (or its selection highlighted). If a keystroke occurs when a w an active text field in it, the keystroke will be directed to the active You can change which text field is active with the **GUI.SetActive**] by simply clicking on another text field with the mouse.

When multiple text fields are created in a window, the first text fiel active when the program begins.

Details The current version of the text field does not support cut and paste commands to extend the selection.

Because strings are a maximum of 255 characters, this is the maximum of characters in a text field.

The TAB character cycles between different text fields in a window through the text fields in the order in which they were created. BA(

	(shift+TAB) cycles through the fields in reverse order.
Details	When GUI.CreateTextField or GUI.CreateTextFieldFull is calle newly created picture will be displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i> paran false.
	When a text field is not enabled, the text field cannot be made activitext in the field cannot be edited.
	The following GUI subprograms can be called with a text box as the parameter:
Details	GUI.Show, GUI.Hide, GUI.Dispose, GUI.GetX, GUI.Get
	GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.Se GUI.SetPositionAndSize, GUI.GetText, GUI.SetText, GUI.SetSelection, GUI.SetActive, GUI.SetEchoChar
	GUI.SetPositionAndSize, GUI.GetText, GUI.SetText,
Status	GUI.SetPositionAndSize, GUI.GetText, GUI.SetText, GUI.SetSelection, GUI.SetActive, GUI.SetEchoChar

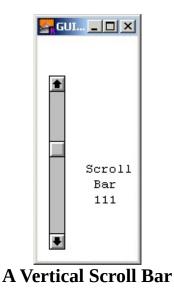
GUI.CreateVerticalScrollBar[Full] Part of GUI module

GUI.CreateVerticalScrollBar (x, y, size : int, min, max, start : int, actionProc : procedure x (value : int)) : int

Syntax GUI.CreateVerticalScrollBarFull (x, y, size : int, min, max, start : int, actionProc : procedure x (value : int), arrowInc, pageInc, thumbSize : int) : int

Creates a vertical (up-down) scroll bar and returns the scroll bar's widget ID.

A scroll bar is a widget that allows users to see a piece of a document that cannot be displayed on the screen in its entirety. The picture below shows a vertical scroll bar. To control a scroll bar, there are a few choices: the user can click on the thumb (the box in the scroll bar) and slide it up and down, or the user can click in the scroll bar itself above or below the thumb (in which case the thumb is moved up or down one "page"), or the user can click on the up or down arrows at the ends of the scroll bar (in which case the thumb is moved up or down one "arrow increment" or "line").



Description The programmer defines a page or an arrow increment. When the value of the scroll bar changes, the *action procedure* of the scroll bar is called with the new value as a parameter. The *action procedure* should then redraw the contents using the new value of the scroll bar.

The range of values that the scroll bar will give is determined by the *min* and *max* parameters in the *Create* call. The left side of the scroll bar represents the minimum value, while the right represents the maximum value. There is also the "thumb size". This represents the range of values that can be seen at once on the screen.

By default, the arrow increment (the amount the value is changed when the scrolling arrows are pressed) is set to one. The page increment (the amount the value is changed when the user clicks in the bar to the right or left of the thumb) is set to one quarter the difference between the minimum and the maximum. The "thumb size" is set to zero (see the description of scroll bars for an explanation of the thumb size).

The *x* and *y* parameters specify the lower-left corner of the scroll bar. The *size* parameter specifies the length of the scroll bar (including the arrows) in pixels. The *min* and *max* parameters are the minimum and maximum valies returned by the scroll bar. The *start* parameter is the initial value of the scroll bar and should be between *min* and *max* inclusive. The *actionProc* parameter is the name of a procedure that is called when the value of the scroll bar is changed. The parameter to the *action procedure* is the current value of the scroll bar.

The following program creates a vertical scroll bar. Whenever the scroll bar's value is changed, a message is displayed in the window.

```
import GUI
View.Set ("graphics:125;200,nobuttonbar")
var scrollBar : int
```

Example procedure ScrollBarMoved (value : int) Text.Locate (9, 7) put "Scroll" Text.Locate (10, 8) put "Bar" Text.Locate (11, 8) put value : 3 end ScrollBarMoved ScrollBar := GUI.CreateVerticalScrollBar (10, 10 50, 150, 50, ScrollBarMoved) loop exit when GUI.ProcessEvent end loop

Execute

For **GUI.CreateVerticalScrollBarFull**, the *arrowInc* parameter specifies the arrow increment (the amount the scroll bar's value is changed when the scroll arrows are pressed). The *pageInc* specifies the page increment (the amount the scroll bar's value is changed when the user clicks in the page left/right section of the scroll bar). The *thumbSize* parameter specifies the "thumb size". See the scroll bar explanation for more detail on a scroll bar's "thumb size".

Description

For example, if you have a window that can display 20 lines of text at once and there are 100 lines of text, you would set *min* to 1, *max* to 100 and *thumbSize* to 20. The value returned by the scroll bar would then be the line number of the first line on the screen to be displayed. When the scroll bar was at its maximum value, it would return 81, since by doing so, lines 81-100 would be displayed.

For an example program that scrolls a large picture over a smallerwindow, see GUI.CreateHorizontalScrollBar.

In some instances, you will want the the minimum and maximum values of the scroll bar to be reversed (right/top is minimum). In that case, call the **GUI.SetSliderReverse** procedure to flip the values of the scroll bar.

Scroll bars always have a fixed height (for horizontal scroll bars) or width (for vertical scroll bars). To get the scroll bar's width, use the **GUI.GetScrollBarWidth** function.

DetailsWhen GUI.CreateVerticalScrollBar or
GUI.CreateVerticalScrollBarFull is called, the newly created
scroll bar will be displayed immediately unless
GUI.DisplayWhenCreated has been called with the display
parameter set to false.

When a scroll bar is not enabled, the gray in the bar is set to white and the thumb is not displayed. The scroll bar no longer responds to any mouse clicks until the scroll bar is enabled again.

The following GUI subprograms can be called with a scroll bar as the *widgetID* parameter:

GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable,
GUI.Dispose, GUI.GetX, GUI.GetY, GUI.GetWidth,DetailsGUI.GetHeight, GUI.SetPosition, GUI.SetSize,
GUI.SetPositionAndSize, GUI.GetSliderValue,
GUI.SetSliderValue, GUI.SetSliderValue, GUI.SetSliderSize, GUI.SetSliderSize, GUI.SetSliderReverse,
GUI.SetScrollAmount

Exported qualified.

StatusThis means that you can only call the function by callingGUI.CreateVerticalScrollBar, not by callingCreateVerticalScrollBar.

<u>GUI.GetSliderValue</u> and <u>GUI.SetSliderValue</u> for reading and setting the value of a scroll bar, <u>GUI.SetSliderMinMax</u> for

See alsochanging the minimum and maximum values of a scroll bar, and
GUI.SetScrollAmount for changing the scrolling increments and
thumb size of a scroll bar. See also GUI.SetSliderSize for setting
the length of a scroll bar and GUI.SetSliderReverse for reversing
the sense of a scroll bar.

GUI.CreateVerticalSlider

Part of <u>GUI</u> module

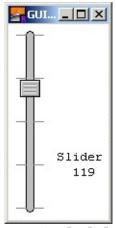
GUI.CreateVerticalSlider (*x*, *y*, *length* : **int**,

Syntax

min, max, start : int, actionProc : procedure x (value :
int)) : int

Creates a vertical (up-down) slider and returns the slider's widget ID.

A slider is a widget that allows the user to set a continuous set of values. It has a real-life equivalent in things such as a stereo volume control.



A Vertical Slider

Description To control a slider, the user clicks on the slider box and drags it back and forth. Every time the value changes, a procedure is called with the new value as a parameter.

The range of values that the slider will give is determined by the *min* and *max* parameters in the *Create* call. The left side of the slider represents the minimum value, while the right represents the maximum value.

The *x* and *y* parameters specify the lower-left corner of the slider track. This means that the slider actually extends above and below this point (and slightly to the left of it to take into account the rounded end of the track). The *length* parameter specifies the

length of the track in pixels. (You can use **GUI.GetX**, **GetY**, **GetWidth**, and **GetHeight** to get the exact dimensions of the slider.) The *min* and *max* parameters are the minimum and maximum valies returned by the slider. The *start* parameter is the initial value of the slider and should be between *min* and *max* inclusive. The *actionProc* parameter is the name of a procedure that is called when the value of the slider is changed. The parameter to the *action procedure* is the current value of the slider.

The following program creates a vertical slider. Whenever the slider's value is changed, a message is displayed in the window.

```
import GUI
View.Set ("graphics:125;200, nobuttonbar")
var slider : int
procedure SliderMoved (value : int)
Text.Locate (9, 7)
put "Slider"
Text.Locate (10, 9)
put value : 3
end SliderMoved
slider := GUI.CreateVerticalSlider (20, 10, 180,
50, 150, 50, SliderMoved )
loop
exit when GUI.ProcessEvent
end loop
```

Execute

In some instances, you will want the the minimum and maximum values of the slider to be reversed (right is minimum). In that case, call the **GUI.SetSliderReverse** procedure to flip the values of the slider.

	Sliders always have a fixed height (for horizontal sliders) or width (for vertical sliders).
Details	When GUI.CreateVerticalSlider or GUI.CreateVerticalSliderFull is called, the newly created slider will be displayed immediately unless GUI.DisplayWhenCreated has been called with the <i>display</i> parameter set to false.
	When a slider is not enabled, the appearance does not change. However, the slider no longer responds to any mouse clicks until it is enabled again.
	The following GUI subprograms can be called with a slider as the <i>widgetID</i> parameter:
Details	GUI.Show, GUI.Hide, GUI.Enable, GUI.Disable, GUI.Dispose, GUI.GetX, GUI.GetY, GUI.GetWidth, GUI.GetHeight, GUI.SetPosition, GUI.SetSize, GUI.SetPositionAndSize, GUI.GetSliderValue, GUI.SetSliderValue, GUI.SetSliderMinMax, GUI.SetSliderSize, GUI.SetSliderReverse
	Exported qualified.
Status	This means that you can only call the function by calling GUI.CreateVerticalSlider , not by calling CreateVerticalSlider .
	<u>GUI.GetSliderValue</u> and <u>GUI.SetSliderValue</u> for reading and

GUI.Disable

Syntax **GUI.Disable** (*widgetID* : int)

Disables a widget specified by *widgetID*.

Used in conjunction with **GUI.Enable** to enable and disable widge

Description Disabled widgets generally are "grayed out" to visually depict their disabled status.

Disabled widgets do not respond to keystrokes or mouse clicks.

The three color radio buttons are enabled only when the color chec selected.

import GUI in "%oot/lib/GUI" View.Set ("graphics:100;100") var colorCheckBox, redRadio, greenRadio, blueRac procedure DoNothing end DoNothing procedure ColorCheckBoxProc (filled : boolean) if filled then **GUI.Enable** (*redRadio*) **GUI.Enable** (greenRadio) **GUI.Enable** (*blueRadio*) else **GUI.Disable** (*redRadio*) Example **GUI.Disable** (greenRadio) **GUI.Disable** (*blueRadio*) end if end ColorCheckBoxProc colorCheckBox := GUI.CreateCheckBox (10, 80, "Use Color", ColorCheckBoxProc) redRadio := GUI.CreateRadioButton (33, 60, "Red" greenRadio := GUI.CreateRadioButton (1, 1, "Gree redRadio, DoNothing) blueRadio := GUI.CreateRadioButton (1, 1, "Blue" greenRadio, DoNothing) ColorCheckBoxProc (false)

	loop exit when GUI.ProcessEvent end loop
	The following types of widgets can be enabled or disabled:
Details	Buttons, Check Boxes, Radio Buttons, Picture Buttons,
Details	Picture Radio Buttons, Horizontal Scroll Bars, Horizontal Sliders, Canvases, Text Fields, Vertical Scroll Bars, Vertical Sliders
	Exported qualified.
Status	This means that you can only call the procedure by calling GUI.Di not by calling Disable .
See also	<u>GUI.Enable</u> .

GUI.Dispose

Syntax	GUI.Dispose (<i>widgetID</i> : int)		
	Eliminates the widget specified by <i>widgetID</i> .		
Description	If the widget is visible, it is immediately made invisible before bein deleted. It should be called in order to free up any memory that the widget might have allocated. Note that you cannot use the widget after it has been disposed of. If you wish to temporarily get rid of a widget, consider using the <i>Hide</i> method and then the <i>Show</i> method when you want to use it again.		
	The following program waits for the Quit button to be pressed. Wh it is, the Quit button is deleted and a message is displayed in the center of the screen.		
	<pre>import GUI in "%oot/lib/GUI" View.Set ("graphics:150;100")</pre>		
	var button, message : int		
Example	<pre>button := GUI.CreateButton (20, 40, 0, "Quit", C</pre>		
	loop exit when GUI.ProcessEvent end loop		
	<pre>GUI.Dispose (button) message := GUI.CreateLabelFull (0, 0, "Done", ma GUI.CENTER + GUI.MIDDLE, 0)</pre>		

Exported qualified.

StatusThis means that you can only call the procedure by calling
GUI.Dispose , not by calling Dispose .

GUI.Draw...

Syntax

GUI.DrawArc (*widgetID*, *x*, *y*, *xRadius*, *yRadius* : **int**, *initialAngle*, *finalAngle*, *Color* : **int**)

GUI.DrawBox (*widgetID*, *x*1, *y*1, *x*2, *y*2, *Color* : **int**)

GUI.DrawCls (*widgetID* : **int**)

GUI.DrawDot (*widgetID*, *x*, *y*, *Color* : **int**)

GUI.DrawFill (*widgetID*, *x*, *y* : **int**, *fillColor*, *borderColor* : **int**)

GUI.DrawFillArc (*widgetID*, *x*, *y* : **int**, *xRadius*, *yRadius* : **int**, *initialAngle*, *finalAngle*, *Color* : **int**)

GUI.DrawFillBox (*widgetID*, *x*1, *y*1, *x*2, *y*2 : **int**, *Color* : **int**)

GUI.DrawFillMapleLeaf (*widgetID*, *x*1, *y*1 : **int**, *x*2, *y*2, *Color* : **int**)

GUI.DrawFillOval (widgetID, x, y : int, xRadius, yRadius : int, Color : int)

GUI.DrawFillPolygon (*widgetID* : **int**, *x*, *y* : **array** 1 .. * **of int**, *n* : **int**, *Color* : **int**)

GUI.DrawFillStar (*widgetID*, *x*1, *y*1, *x*2, *y*2 : **int**, *Color* : **int**)

GUI.DrawLine (*widgetID*, *x*1, *y*1, *x*2, *y*2, *Color* : **int**)

GUI.DrawMapleLeaf (*widgetID*, *x*1, *y*1, *x*2, *y*2 : **int**, *Color* : **int**)

GUI.DrawOval (*widgetID*, *x*, *y* : **int**,

xRadius, yRadius, Color : **int**)

	GUI.DrawPolygon (<i>widgetID</i> : int , <i>x</i> , <i>y</i> : array 1 * of int , <i>n</i> : int , <i>Color</i> : int)		
	GUI.DrawStar (<i>widgetID</i> , <i>x</i> 1, <i>y</i> 1, <i>x</i> 2, <i>y</i> 2, <i>Color</i> : int)		
	GUI.DrawText (widgetID : int , textStr : string , x, y : int , fontID, Color : int)		
	Performs a <i>Draw</i> command to the canvas specified by <i>widgetID</i> .		
Description	All of these routines are essentially the same as the similarly- named procedures in the <i>Draw</i> module. All coordinates are based on the canvas and all drawing is clipped to the canvas drawing surface. If the canvas is in "xor mode", all the drawing will be done with "xor" set. (See View.Set for more information about "xor".)		
	The <i>widgetID</i> must specify a canvas widget.		
Example	See GUI.CreateCanvas for an example of GUI.DrawFillOval .		
	Exported qualified.		
Status	This means that you can only call the procedures by calling GUI.Draw , not by calling Draw .		
See also	GUI.CreateCanvas.		

GUI.Enable

Part of **GUI** module

Syntax	GUI.Enable (widgetID : int)		
Description	Enables a disabled widget specified by <i>widgetID</i> .		
	Used in conjunction with GUI.Disable to enable and disable widgets.		
	Disabled widgets generally are "grayed out" to visually depict their disabled status.		
	Disabled widgets do not respond to keystrokes or mouse clicks.		
Example	See GUI.Disable for an example of GUI.Enable .		
Details	The following types of widgets can be enabled or disabled:	Buttons, Check Boxes, Radio Buttons, Picture Buttons,	

Picture Radio Buttons, Horizontal Scroll Bars, Horizontal Sliders, Canvases, Text Fields, Vertical Scroll Bars, Vertical Sliders

Status Exported qualified.

This means that you can only call the procedure by calling **GUI.Enable**, not by calling **Enable**.

See also <u>GUI.Disable</u>.

GUI.FontDraw

Syntax	GUI.FontDraw (widgetID : int , textStr : string , x, y, fontID, Color : int)
	Performs a <i>Font.Draw</i> command to the canvas specified by <i>widgetID</i> .
Description	This routine is essentially the same as the <i>Font.Draw</i> procedure in the <i>Font</i> module. All coordinates are based on the canvas and all drawing is clipped to the canvas drawing surface. If the canvas is in "xor mode", all the drawing will be done with "xor" set. (See View.Set for more information about "xor".)
	The <i>widgetID</i> must specify a canvas widget.
	Exported qualified.
Status	This means that you can only call the procedure by calling GUI.FontDraw , not by calling FontDraw .
See also	GUI.CreateCanvas.

GUI.GetCheckBox

Part of <u>GUI</u> module

Syntax	GUI.GetCheckBox (<i>widgetID</i> : int) : boolean
Description	Returns the status of the check box specified by <i>widgetID</i> . If the check box is set (has an X in it), <i>GetCheckBox</i> returns true, otherwise it returns false.
Example	See GUI.CreateCheckBox for an example of GUI.GetCheckBox .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.GetCheckBox , not by calling GetCheckBox .
See also	GUI.CreateCheckBox.

GUI.GetEventTime

Description

Syntax GUI.GetEventTime : int

Returns the time in milliseconds when the event (mouse button or l place. This value is the same value as *Time.Elapsed* returns if calle was processed. This function should only be called in an *action prc* default mouse, keystroke, or null event handler, as it will return 1 v event being processed.

This event can be used as a timer for various functions such as dete a single click or a double click of the mouse button took place or for keyboard input.

The following program times the interval between two button press

import GUI in "%oot/lib/GUI" View.Set ("graphics:300;100") var startTime, startButton, finishButton : int procedure Start startTime := GUI.GetEventTime end Start procedure Finish Example Text.Locate (1, 1) put "The time between button pressed is ", GUI.GetEventTime startTime, " msecs" GUI.Quit end Finish startButton := GUI.CreateButton (10, 10, 110, "C finishButton := GUI.CreateButton (180, 10, 110, **loop** exit when GUI.ProcessEvent end loop Exported qualified.

StatusThis means that you can only call the function by calling GUI.Get
by calling GetEventTime .

See also <u>GUI.ProcessEvent</u>.

GUI.GetEventWidgetID

Description

Syntax GUI.GetEventWidgetID : int

Returns the widget ID of the widget that was activated by the mouse button press or the keystroke. This function should only be called in an *action procedure*, as it will return 1 when there is no event that activated a widget being processed.

This function is used when a several buttons use the same action procedure to determine which button was pressed.

The following program prints a message stating which button was selected.

import GUI in "%oot/lib/GUI" View.Set ("graphics:150;210") var buttonNames : array 1 .. 5 of string := init "Blue", "Yellow", "Purple") var buttons : array 1 .. 5 of int procedure ButtonPush for i : 1 .. 5 if GUI.GetEventWidgetID = buttons (i) th Example Text.Locate (1, 1) put buttonNames (i), " selected" end if end for end ButtonPush for *i* : 1 .. 5 buttons (i) := GUI.CreateButton (10, 210 40 buttonNames (i), ButtonPush) end for **100**p exit when GUI.ProcessEvent end loop

Exported qualified.

StatusThis means that you can only call the function by callingGUI.GetEventWidgetID, not by calling GetEventWidgetID.

See also <u>GUI.ProcessEvent</u>.

GUI.GetEventWindow

Part of **<u>GUI</u>** module

Syntax GUI.GetEventWindow : int

Returns the window ID of the window in which the event (mouse t keystroke) took place. This function should only be called in an *ac* or in a default mouse or keystroke event handler, as it will return 1 no event being processed.

Description

This function is commonly used when several windows share the s The same buttons in each window point to the same *action procedu* determine which button was actually pressed, the function is called window.

The following program creates four windows in a row, each with a when pressed, causes a star to be drawn in that window.

import GUI in "%oot/lib/GUI" procedure DrawStar var windowID : int := GUI.GetEventWindow Window.Select (windowID) Draw.FillStar (25, 40, 175, 190, Rand.Int (1 end DrawStar Example for i : 0 .. 3 var window : int := Window.Open ("graphics:2 % Place window above task bar, across from p Window.SetPosition (window, 220 * i, 27) var button : int := GUI.CreateButton (5, 5, DrawStar) end for **loop** exit when GUI.ProcessEvent end loop Exported qualified.

StatusThis means that you can only call the function by callingGUI.GetEventWindow, not by calling GetEventWindow.

See also <u>GUI.ProcessEvent</u>.

Syntax	GUI.GetHeight (widgetID : int) : int
	Returns the actual height of a widget. Note that this may be different from the height specified in the <i>Create</i> call (especially since many widgets do not specify a height. The GUI module determines the actual height).
Description	This function is used in conjunction with GUI.GetX , GUI.GetY and GUI.GetWidth to determine the bounds of a widget. The entin widget should always fit in the box (GUI.GetX , GUI.GetY) - (GUI.GetX + GUI.GetWidth 1, GUI.GetY + GUI.GetHeight 1)
	The position and size of a widget is known only after it has been drawn to the screen. Attempting to get the location or dimesions of the widget may cause an uninitialized variable error.
	The following procedure draws a red box around the widget specified by <i>widgetID</i> .
	<pre>import GUI in "%oot/lib/GUI"</pre>
Example	<pre>procedure BoxWidget (widgetID : int) var x, y, width, height : int x := GUI.GetX (widgetID) y := GUI.GetY (widgetID) width := GUI.GetWidth (widgetID) height := GUI.GetHeight (widgetID) Draw.Box (x 1, x 1, x + width, y + height, Draw.Box (x 2, x 2, x + width + 1, y + hei end BoxWidget</pre>
	var title : int := GUI.CreateLabel (20, 20, "Fra BoxWidget (title)
	Exported qualified.
Status	This means that you can only call the function by calling GUI.GetHeight , not by calling GetHeight .

See also GUI.GetX, GUI.GetY, and <u>GUI.GetWidth</u>.

GUI.GetMenuBarHeight

Syntax	GUI.GetMenuBarHeight : int
Description	Returns the height of the menu bar. Useful when drawing or placing widgets to make certain that they don't overlap the menu bar.
	The following program draws a red box in the window just belowe the menu bar.
	<pre>import GUI in "%oot/lib/GUI"</pre>
Example	<pre>var menu : int := GUI.CreateMenu ("File") var item : int := GUI.CreateMenuItem ("Quit", GU</pre>
	Draw.FillBox (0, 0, maxx, maxy GUI.GetMenuBarHε brightred)
	loop exit when GUI.ProcessEvent end loop
	Exported qualified.
Status	This means that you can only call the function by calling GUI.GetMenuBarHeight , not by calling GetMenuBarHeight .
See also	<u>GUI.CreateMenu</u> .

GUI.GetScrollBarWidth

Syntax	GUI.GetScrollBarWidth : int
Description	Returns the width of a scroll bar. Useful when placing a scroll bar widget beneath or beside another widget or object.
Example	See the <i>ScrollPic</i> program in GUI.CreateHorizontalScrollBarFull for an example of GUI.GetScrollBarWidth .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.GetScrollBarWidth , not by calling GetScrollBarWidth .
See also	GUI.CreateHorizontalScrollBar and GUI.CreateVerticalScrollBar.

GUI.GetSliderValue

Syntax	GUI.GetSliderValue (<i>widgetID</i> : int) : int
Description	Returns the current value of a slider or scroll bar specified by <i>widgetID</i> . The <i>widgetID</i> must specify either scroll bar or a slider (horizontal or vertical).
Example	See the <i>ScrollPic</i> program in GUI.CreateHorizontalScrollBarFull for an example of GUI.GetSliderValue .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.GetSliderValue , not by calling GetSliderValue .
See also	GUI.SetSliderValue for setting a slider <u>or</u> scroll bar's value. See also GUI.CreateHorizontalScrollBar , GUI.CreateVerticalScrollBar , <u>GUI.CreateHorizontalSlider</u> , and <u>GUI.CreateVerticalSlider</u> .

GUI.GetText

Syntax	GUI.GetText (widgetID : int) : string
Description	Returns the current text of a text field specified by <i>widgetID</i> . The <i>widgetID</i> must specify a text field widget.
Example	See GUI.CreateTextField for an example of GUI.GetText .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.GetText , not by calling GetText .
See also	<u>GUI.SetText</u> for setting the text in a text field. See also GUI.CreateTextField .

GUI.GetVersion

Syntax	GUI.GetVersion : int
Description	Returns the current version of the GUI Procedure Library. Because the GUI Procedure Library is expected to grow, new versions will probably be made available at our web site http://www.holtsoft.com/turing . If you wish to use features that do not appear in earlier versions of the library, you can have your program check that the current available version meets the programs needs. GUI.GetVersion returns an integer from 100 - 999 and is read as 1.00 to 9.99.
Example	<pre>The following program fragment immediately exits if OOT does not support version 1.1 of the GUI Procedure Library import GUI in "%oot/lib/GUI" if GUI.GetVersion < 110 then put "You must update to at least version 1.1 put "GUI Procedure Library to use this progr return end if</pre>
Details	In version 1.00 (shipped with MacOOT 1.5), GUI.GetVersion did not exist.
	Exported qualified.
Status	This means that you can only call the function by calling GUI.GetVersion , not by calling GetVersion .

GUI.GetWidth

Syntax	GUI.GetWidth (widgetID : int) : int
	Returns the actual width of a widget. Note that this may be differer from the width specified in the <i>Create</i> call (especially since some widgets do not specify a width. The GUI module determines the actual width).
Description	This function is used in conjunction with GUI.GetX , GUI.GetY and GUI.GetHeight to determine the bounds of a widget. The entire widget should always fit in the box (GUI.GetX , GUI.GetY) - (GUI.GetX + GUI.GetWidth 1, GUI.GetY + GUI.GetHeight 1
	The position and size of a widget is known only after it has been drawn to the screen. Attempting to get the location or dimesions of the widget may cause an uninitialized variable error.
	The following procedure draws a red box around the widget specified by <i>widgetID</i> .
Example	<pre>import GUI in "%oot/lib/GUI" procedure BoxWidget (widgetID : int) var x, y, width, height : int x := GUI.GetX (widgetID) y := GUI.GetY (widgetID) width := GUI.GetWidth (widgetID) height := GUI.GetHeight (widgetID) Draw.Box (x 1, x 1, x + width, y + height, Draw.Box (x 2, x 2, x + width + 1, y + hei end BoxWidget</pre>
	var title : int := GUI.CreateLabel (20, 20, "Fra BoxWidget (title)
	Exported qualified.
Status	This means that you can only call the function by calling GUI.GetWidth , not by calling GetWidth .
See also	GUI.GetX, GUI.GetY, and GUI.GetHeight.

GUI.Get{X,Y}

GUI.GetX (*widgetID* : **int**) : **int**

Syntax GUI.GetY (*widgetID* : **int**) : **int**

Returns the x coordinate of the left edge of a widget. Note that this may be different from the x coordinate specified in the widget's *Create* call. For example, if a radio button is created with right justification, the x coordinate in the *Create* method specifies the right edge while **GUI.GetX** will return the x coordinate of the left edge.

DescriptionThis function is used in conjunction with **GUI.GetWidth** and**DescriptionGUI.GetHeight** to determine the bounds of a widget. The entire
widget should always fit in the box (**GUI.GetX**, **GUI.GetY**) -
(**GUI.GetX + GUI.GetWidth -** 1, **GUI.GetY + GUI.GetHeight -**
1)

The position and size of a widget is known only after it has been drawn to the screen. Attempting to get the location or dimesions of the widget may cause an uninitialized variable error.

The following procedure draws a red box around the widget specified by *widgetID*.

import GUI in "%oot/lib/GUI"
procedure BoxWidget (widgetID : int)
 var x, y, width, height : int
 x := GUI.GetX (widgetID)
 y := GUI.GetY (widgetID)
 width := GUI.GetWidth (widgetID)
 height := GUI.GetHeight (widgetID)
 Draw.Box (x 1, x 1, x + width, y + height,
 Draw.Box (x 2, x 2, x + width + 1, y + hei
end BoxWidget
var title : int := GUI.CreateLabel (20, 20, "Fra
BoxWidget (title)

Exported qualified.

- StatusThis means that you can only call the function by calling
GUI.GetX, not by calling GetX.
- See also <u>GUI.GetHeight</u> and <u>GUI.GetWidth</u>.

GUI.Hide

Syntax	GUI.Hide (widgetID : int)
	Hides a widget specified by <i>widgetID</i> . Used in conjunction with <i>Show</i> to hide and show widgets. Hidden widgets cannot get events (i.e. respond to keystrokes or mouse clicks). If an active text field (see text field) is hidden, then any keystrokes in the window will be ignored.
Description	In most cases where a widget is to appear, then disappear, then appear again, it is advised to create the widget once and hide it until it is to appear, whereupon GUI.Show is called. When the user is finished with the widget, the widget is hidden using GUI.Hide . This saves the overhead of creating and disposing of the same widget several times.
Example	See GUI.SetDisplayWhenCreated for an example of GUI.Hide .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.Hide , not by calling Hide .
See also	GUI.Show.

GUI.HideMenuBar

Syntax	GUI.HideMenuBar
Description	Hides the menu bar in the selected window. No menu items can be selected when the menu bar is hidden. (Menu item shortcuts are ignored while the menu bar is hidden.)
Example	See GUI.SetMouseEventHandler for an example of GUI.HideMenuBar .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.HideMenuBar , not by calling HideMenuBar .
See also	GUI.ShowMenuBar. See also GUI.CreateMenu.

GUI.HideWindow

Part of **<u>GUI</u>** module

Syntax GUI.HideWindow (window : int)

DescriptionHides a window with widgets in it. This procedure makes certain the
Library recognizes that the window no longer visible. This procedu
Window.Hide, so there is no need for the user to do so.

The program opens up a window with two buttons. If the button lal and Open" is pressed, the window is closed and a new window wit opened in a random location on the screen.

import GUI var windowA, windowB : int var buttonA, buttonB, buttonQuit : int var backToMainA, backToMainB : int procedure ShowWindowA GUI.HideWindow (defWinID) **GUI.ShowWindow** (windowA) end ShowWindowA procedure ShowWindowB GUI.HideWindow (defWinID) **GUI.ShowWindow** (windowB) end ShowWindowB procedure ShowMain Example GUI.HideWindow (Window.GetActive) GUI.ShowWindow (defWinID) end ShowMain View.Set ("graphics:280;100, nobuttonbar") % Place the buttons in the main window buttonA := GUI.CreateButton (10, 10, 0, "Show Wi buttonB := GUI.CreateButton (150, 10, 0, "Show W buttonQuit := GUI.CreateButton (100, 60, 0, "Qui windowA := Window.Open ("title:Window A, graphic backToMainA := GUI.CreateButton (20, 20, 0, "Bac Window.Hide (windowA) windowB := Window.Open ("title:Window B, graphic backToMainB := GUI.CreateButton (20, 20, 0, "Bac Window.Hide (windowB)

loop
 exit when GUI.ProcessEvent
end loop

Execute

Exported qualified.

StatusThis means that you can only call the function by calling GUI.Hid
by calling HideWindow.

GUI.ShowWindowfor showing a window with widgets in it. SeeSee alsoGUI.CloseWindowfor closing a window with widgets in it.

Syntax GUI.OpenFile (*title* : **string**) : **string**

Displays an "Open File" dialog box to obtain the name of an already existing file. The caption (a window title under MS Windows, a string in a Macintosh dialog box) is specified by the *title* parameter. The function uses a dialog box specific to the operating system the program is being run on.

If the user did not choose a file (i.e. hit the *Cancel* button in the dialog), the function returns "" (the empty string).

Description

Note: This function is **not** available in the current version of the GUI Procedure Library (shipping with Turing 4.0 and MacOOT 1.5). It is documented here for use with future shipping version of Turing. It is likely to be implemented in the version of Turing released in September 2002. Check the release notes that are found in the on-line help to find out if this function is now available.

The following program asks the user for the name of a file and then echoes the contents of it.

	<pre>import GUI in "%oot/lib/GUI"</pre>
	var fileName, line : string var streamNumber : int
Example	<pre>fileName := GUI.OpenFile ("Choose a Text File")</pre>
Liumpic	<pre>open : streamNumber, fileName, get assert streamNumber > 0 loop exit when eof (streamNumber) get : streamNumber, line : * put line end loop close : streamNumber</pre>

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.OpenFile, not by calling OpenFile.

Part of <u>GUI</u> module

GUI.OpenFileFull

GUI.OpenFileFull (title, filter: string,SyntaxstartDir : string) : string

Displays an "Open File" dialog box to obtain the name of an alread The caption (a window title under MS Windows, a string in a Maci box) is specified by the *title* parameter. The list of files shown is sp *filter* parameter. The initial directory to be displayed is specified by parameter. The function uses a dialog box specific to the operating program is being run on.

The *filter* parameter is a file name suffix that should be displayed. I suffixes can be specified by separating them with commas. If the u the empty string for *filter*, then all the files in the directory are disp *startDir* parameter is empty, or if it specifies a non-existent directo current directory is displayed in the "Open File" dialog box.

If the user did not choose a file (i.e. hit the *Cancel* button in the dia function returns "" (the empty string).

Note: This function is **not** available in the current version of the G¹ Library (shipping with Turing 4.0 and MacOOT 1.5). It is document use with future shipping version of Turing. It is likely to be implement version of Turing released in September 2002. Check the release net found in the on-line help to find out if this function is now available

The following program asks the user to select a file ending in ".txt' directory is the root directory of the C drive.

Example	<pre>import GUI in "%oot/lib/GUI"</pre>
	var fileName : string fileName := GUI.OpenFileFull ("Choose a Text Fil
Details	If a suffix is placed in single quotes, it will be ignored on all but the Macintosh, where it will specify a Macintosh file type.

The example makes the dialog box display all files ending in ".txt"

Example	all systems but the Macintosh. On the Apple Macintosh, only files 'TEXT' will be displayed.
	<pre>fileName := GUI.OpenFileFull ("Open", "txt,text,</pre>
	Exported qualified.
Status	This means that you can only call the function by calling GUI.Op e by calling OpenFileFull .

GUI.Pic...

	GUI.PicDraw (<i>widgetID</i> : int , <i>picID</i> , <i>x</i> , <i>y</i> , <i>mode</i> : int)
	GUI.PicNew (<i>widgetID</i> : int , <i>x</i> 1, <i>y</i> 1, <i>x</i> 2, <i>y</i> 2 : int) : int
Syntax	GUI.PicScreenLoad (widgetID : int , fileName : string , x, y, mode : int)
	GUI.PicScreenSave (widgetID : int , x1, y1, x2, y2 : int , fileName : string)
	Performs a <i>Pic</i> command to the canvas specified by <i>widgetID</i> .
Description	All of these routines are essentially the same as the similarly- named procedures in the <i>Pic</i> module. All coordinates are based on the canvas and all drawing is clipped to the canvas drawing surface.
Example	See the <i>ScrollPic</i> program in GUI.CreateHorizontalScrollBarFull for an example of GUI.PicDraw .
	Exported qualified.
Status	This means that you can only call the procedures by calling GUI.Pic , not by calling Pic .
See also	GUI.CreateCanvas.

GUI.ProcessEvent

Syntax GUI.ProcessEvent : boolean

This function processes a single event (a mouse button press or a k event activates a widget, then the *action procedure* of the widget is

The function returns **false** until **GUI.Quit** is called. It then returns

In order for the widgets to function once placed, the GUI.ProcessE called continually. Without a call to **GUI.ProcessEvent**, the widge but will not react to mouse clicks or keystrokes.

Description Almost all programs involving the GUI Procedure Library have the fragment in the program. This code fragment is often called the *eve*

loop
 exit when GUI.ProcessEvent
end loop

The loop runs continuously until **GUI.Quit** is called, whereupon **GUI.ProcessEvent** will return **true** and the loop will exit. The rest is reached through the *action procedures* that are called when the u various widgets.

DetailsIt is usually desirable to allow the user some way of quitting the pr
having to abort it. This can be done most simply by adding a Quit I
placing it in an appropriate location.

Here is program that does nothing but wait for the user to press the

Example import GUI in "%oot/lib/GUI" var quitButton : int := GUI.CreateButton (10, 10 loop exit when GUI.ProcessEvent end loop

To find out which widget was activated and called the *action proce* if several widgets have the same *action procedure*), you can call **GUI.GetEventWidgetID**. To get the exact time that the event occi

call GUI.GetEventTime. To get the window in which the event to	
can call GUI.GetEventWindow .	

If a mouse click occured, but did not activate any widget, then the event handler is called. By default, this does nothing. However, if y program to respond to mouse events that do not affect a widget, cal **GUI.SetMouseEventHandler** to specify your own default mouse

Details

If a keystroke occurred, but did not activate any widget (i.e. it wası a widget and there are no text fields in the window) then the defaul handler is called. By default, this does nothing. However, if you wa to respond to keystroke events that do not affect a widget, call **GUI.SetKeyEventHandler** to specify your own default key event

If no event occurred, then the null event handler is called. By defau nothing. However, if you want your program to perform some actic when it is not doing anything else, then call **GUI.SetNullEventHa** your own null event handler. The null event handler is often used for updating a clock and making certain that music is playing in the ba

Exported qualified.

StatusThis means that you can only call the procedures by calling GUI.P
not by calling ProcessEvent.

GUI.GetEventWidgetID, GUI.GetEventTime, and GUI.GetEve
obtaining information about an event in an action procedure. See a
GUI.SetMouseEventHandler, GUI.SetKeyEventHandler and
GUI.SetNullEventHandler for handling mouse, keyboard an d nu
event loop. See also GUI.Quit for information on exitting the even

GUI.Quit

Syntax GUI.Quit

This procedure causes **GUI.ProcessEvent** to return **true**. If the prc structured properly with a

Description	loop exit when GUI.ProcessEvent end loop
	at the end of the program, then the program will exit the loop after current <i>action procedure</i> . This procedure is usually called from the <i>procedure</i> of a Quit button or Exit menu item.
	Here is program that does nothing but wait for the user to press the type the letter 'Q', 'q', 'X', or 'x'.
	<pre>import GUI</pre>
Example	<pre>procedure KeyHandler (ch : char) if ch = 'Q' or ch = 'q' or ch = 'X' or ch = GUI.Quit end if end KeyHandler</pre>
	<pre>var quitButton : int := GUI.CreateButton (10, 10 GUI.SetKeyEventHandler (KeyHandler)</pre>
	loop exit when GUI.ProcessEvent end loop
	<pre>put "Done!"</pre>
	Exported qualified.
Status	This means that you can only call the procedures by calling GUI.Ç calling Quit .
See also	GUI.ProcessEvent . See also <u>GUI.ResetQuit</u> for resetting the "quit second event loop can be used in the same program.

GUI.Refresh

Syntax GUI.Refresh

This routine redraws all the widgets in the currently-selected window. This is used when some form of drawing may have overwritten the widgets in a window.

Description It is often used when there is some possibility that the widgets may have been drawn over. For example, a program that places buttons on top of a background image should call **GUI.Refresh** when the image is changed.

GUI.Refresh is used by the GUI Library to redraw all the
widgets when the background colour of a window has changed.

Exported qualified.

StatusThis means that you can only call the procedures by calling
GUI.Refresh, not by calling Refresh.

GUI.ResetQuit

Syntax GUI.ResetQuit

The **GUI.ResetQuit** procedure resets the "quitting" flag that is set The **GUI.ResetQuit** can be used whenever you want to have a promain processing loop, does some work, and then reenters the proce

Description

Without **GUI.ResetQuit**, the program will exit the second loop im **GUI.ProcessEvent** will return **true**.

This program allows the user to select the color that stars are to be presses the button, the program exits the first event loop and draws another button and entering a second event loop, it calls **GUI.Rese**

import GUI var radio : array 1 .. 4 of int var button : int var starColor : int := red % Action procedures for the radio buttons procedure Red starColor := brightred end Red procedure Green starColor := brightgreen end Green procedure Blue starColor := brightblue end Blue procedure Cyan starColor := brightcyan end Cyan % Create the radio buttons radio (1) := GUI.CreateRadioButton (15, maxy - 3 radio (2) := GUI.CreateRadioButton (-1, -1, "Gre radio (3) := GUI.CreateRadioButton (-1, -1, "Blι radio (4) := GUI.CreateRadioButton (-1, -1, "Cya % Create the push button

button := GUI.CreateButton (100, maxy - 70, 0, " Example % Process events until the "Draw Stars" button i **100**p exit when GUI.ProcessEvent end loop % Dispose of all the radio buttons and the push for *i* : 1 ... 4 **GUI.Dispose** (*radio* (i)) end for **GUI.Dispose** (button) % Draw a bunch of stars - No more handling of ev for *i* : 1 .. 100 **var** x : int := Rand.Int (0, maxx - 20) **var** y : int := Rand.Int (0, maxy - 20) **Draw.FillStar** (x, y, x + 20, y + 20, starCol end for % Create a new button button := GUI.CreateButton (300, 10, 0, "Quit", % Reset the guit flag. Without this statement, t % exit immediately because the quit flag was set GUI.ResetQuit % Process events until the "Quit" button is pres **100**p exit when GUI.ProcessEvent end loop % Close the window GUI.CloseWindow (defWinID)

Execute

Exported qualified.

StatusThis means that you can only call the function by calling GUI.ResResetQuit.

GUI.SaveFile

Syntax	GUI.SaveFile (title : string) : string
	Displays an "Save File" dialog box to obtain the name of a file. The caption (a window title under MS Windows, a string in a Macintosh dialog box) is specified by the <i>title</i> parameter. The function uses a dialog box specific to the operating system the program is being run on.
Description	If the user did not choose a file (i.e. hit the <i>Cancel</i> button in the dialog), the function returns "" (the empty string).
	Note : This function is not available in the current version of the GUI Procedure Library (shipping with Turing 4.0 and MacOOT 1.5). It is documented here for use with future shipping version of Turing. Check the release notes that are found in the on-line help to find out if this function is now available.
	The following program asks the user for the name of a file and then writes the numbers 1 to 10 in it.
	<pre>import GUI in "%oot/lib/GUI"</pre>
	var fileName : string var streamNumber : int
Example	<pre>fileName := GUI.SaveFile ("Choose a Text File")</pre>
	<pre>open : streamNumber, fileName, put assert streamNumber > 0 for i : 1 10 put : streamNumber, i end loop close : streamNumber</pre>
	Exported qualified.
C 4 - 4	This was and that some some solls soll the founding has calling

StatusThis means that you can only call the function by calling
GUI.SaveFile, not by calling SaveFile.

GUI.SaveFileFull

GUI.SaveFileFull (title, filter: string,SyntaxstartDir : string) : string

Displays an "Save File" dialog box to obtain the name of an already existing file. The caption (a window title under MS Windows, a string in a Macintosh dialog box) is specified by the *title* parameter. The list of files shown is specified by the *filter* parameter. The initial directory to be displayed is specified by the *startDir* parameter. The function uses a dialog box specific to the operating system the program is being run on.

DescriptionThe *filter* parameter is a file name suffix that should be displayed.
Multiple suffixes can be specified by separating them with
commas. If the user specifies the empty string for *filter*, then all
the files in the directory are displayed. If the *startDir* parameter is
empty, or if it specifies a non-existent directory, then the current
directory is displayed in the "Open File" dialog box.

If the user did not choose a file (i.e. hit the *Cancel* button in the dialog), the function returns "" (the empty string).

Note: This function is **not** available in the current version of the GUI Procedure Library (shipping with Turing 4.0 and MacOOT 1.5). It is documented here for use with future shipping version of Turing. Check the release notes that are found in the on-line help to find out if this function is now available.

The following program segment asks the user for the name of a file ending in ".txt". The initial directory is the root directory of the C drive.

Example

var fileName : string := GUI.SaveFileFull ("Choc "txt", "C:\\")

If a suffix is placed in single quotes, it will be ignored on all butthe Apple Macintosh, where it will specify a Macintosh file type.

The following program segment asks the user for the name of a file. It displays files of type 'TEXT'. The initial directory is the "Turing Programs" directory on the "Macintosh HD" volume.

Example

var fileName : string := GUI.SaveFileFull ("Choc "'TEXT'", "Macintosh HD:Turing Programs")

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SaveFileFull, not by calling SaveFileFull.

GUI.SelectRadio

Part of **GUI** module

Syntax GUI.SelectRadio (widgetID : int)

Description Selects a radio button specified by *widgetID*. The previously-select button is "de-selected". The *action procedure* of the radio button is

The following program creates siz radio buttons. Selecting one of t cause the bottom two radio buttons to become selected.

```
import GUI in "%oot/lib/GUI"
                    View.Set ("graphics:350;110")
                    var radio : array 1 .. 6 of int % The radio
                    procedure RadioPressed
                        Text.Locate (1, 1)
                        for i : 1 .. 6
                            if radio (i) = GUI.GetEventWidgetID ther
                                put "Radio Button " , i, " Selected"
                            end if
                        end for
                    end RadioPressed
                    procedure Select
                        GUI.SelectRadio (radio (3))
                        GUI.SelectRadio (radio (6))
                    end Select
Example
                    radio (1) := GUI.CreateRadioButton (15, maxy
                                                                   35
                        "Radio Button 1", 0, RadioPressed)
                    radio (2) := GUI.CreateRadioButton ( 1, 1, "Radi
                        radio (1), RadioPressed)
                    radio (3) := GUI.CreateRadioButton ( 1, 1, "Radi
                        radio (2), RadioPressed)
                    radio (4) := GUI.CreateRadioButtonFull (maxx 15
                        "Radio Button 4", 0, RadioPressed, GUI.RIGH7
                    radio (5) := GUI.CreateRadioButtonFull ( 1, 1, "
                        radio (4), RadioPressed, GUI.RIGHT, GUI.NONE
                    radio (6) := GUI.CreateRadioButtonFull ( 1, 1, "
                        radio (5), RadioPressed, GUI.RIGHT, GUI.NONE
                    var selectButton : int := GUI.CreateButton (15,
                        "Select Bottom Buttons", Select)
                    var guitButton : int := GUI.CreateButton (maxx
                        "Quit", GUI.Quit)loop
```

exit when GUI.ProcessEvent end loop

Exported qualified.

StatusThis means that you can only call the procedures by calling
GUI.SelectRadio, not by calling SelectRadio.

See also GUI.CreateRadioButton and GUI.CreatePictureRadioButton.

GUI.SetActive

Syntax	GUI.SetActive (widgetID : int)
Description	Makes a text field specified by <i>widgetID</i> active. If the text field is not in an active window, then the text field will become active when the window does. If another text field was active in the window, it is deactivated.
Example	See GUI.CreateTextField for an example of GUI.SetActive .
	Exported qualified.
Status	This means that you can only call the procedures by calling GUI.SetActive , not by calling SetActive .
See also	GUI.CreateTextField.

GUI.SetBackgroundColor

Syntax	GUI.SetBackgroundColor (Color : int)
Description	Changes the background colour of the currently-selected window to the color specified by <i>Color</i> . This does <i>not</i> change the value of color 0 in the window. Instead it fills the entire window with the new background color and then redraws all the widgets.
	For indented and extended items, the background color is assumed to be set to <i>gray</i> .
	The alternate spelling is GUI.SetBackgroundColour
Example	See GUI.CreateFrame for an example of GUI.SetBackgroundColour .
	Exported qualified.
Status	This means that you can only call the procedures by calling GUI.SetBackgroundColor , not by calling SetBackgroundColor .

GUI.SetCheckBox

Syntax	GUI.SetCheckBox (<i>widgetID</i> : int , <i>status</i> : boolean)
Description	Sets the status of a check box specified by <i>widgetID</i> . If <i>status</i> is true , the check box is filled (marked with an 'X'). If <i>status</i> is false , the check box is set empty. GUI.SetCheckBox calls the check box's <i>action procedure</i> with the new status and redraws the widget with the new status.
Example	See GUI.CreateCheckBox for an example of GUI.SetCheckBox .
	Exported qualified.
Status	This means that you can only call the procedures by calling GUI.SetCheckBox , not by calling SetCheckBox .
See also	GUI.CreateCheckBox.

GUI.SetColor

Part of **<u>GUI</u>** module

GUI.SetColor (*widgetID* : **int**, *clrNumber* : **int**) **Syntax** The **GUI.SetColor** procedure is used buttons to change the color o button. The color is specified by the *clrNumber* parameter. The edg **Description** the button are a darker version of the color specified by *clrNumber* general you should use brighter colors for buttons. This program displays three buttons with different colors. import GUI procedure DoNothing end DoNothing var b1 := GUI.CreateButton (100, 100, 0, "Push № GUI.SetColor (b1, brightred) Example var b2 := GUI.CreateButton (100, 140, 0, "Push № **GUI.SetColor** (*b2*, **brightgreen**) var b3 := GUI.CreateButton (100, 180, 0, "Push № **GUI.SetColor** (*b3*, **brightblue**) **100**p exit when GUI.ProcessEvent end loop

Execute

Exported qualified.

Status This means that you can only call the function by calling **GUI.Set(** not by calling **SetColor**.

GUI.SetDefault

Syntax	GUI.SetDefault (widgetID : int, default : boolean)
Description	Sets the "default status" of a button specified by <i>widgetID</i> . If a button is the default button, then it is drawn with a heavy outline and it is activated when the user presses ENTER.
	Only one button can be the default button per window. If a button is set to be the default button, then the previous default button has its "default status" removed.
	Exported qualified.
Status	This means that you can only call the procedures by calling GUI.SetDefault , not by calling SetDefault .
See also	<u>GUI.CreateButton</u> .

GUI.SetDisplayWhenCreated

Part of **<u>GUI</u>** module

Syntax **GUI.SetDisplayWhenCreated** (*display* : **boolean**)

By default, whenever a widget is created with a *GUI.Create...* proc the widget instantly appears. Sometimes, this is not the desired beh For example, if several widgets are to occupy the same location wi one being displayed at a time, then it is desirable not to have the w appear when first created.

Description If a widget is not displayed when created, then *GUI.Show* must be to make the widget visible.

If the *display* parameter is **true**, then widgets are displayed immed upon creation. If the *display* parameter is set to **false**, then the widg not made visible on creation and *GUI.Show* must be called to displ widget.

The following program toggles the visiblility of the frame when the button is pushed. The frame starts out invisible.

import GUI in "%oot/lib/GUI" View.Set ("graphics:150;100") var visible : boolean := false var button, frame : int procedure Toggle if visible then GUI.Hide (frame) else Example GUI.Show (frame) end if visible := **not** visible end Toggle button := GUI.CreateButton (25, 40, 0, "Toggle F GUI.SetDisplayWhenCreated (false) frame := GUI.CreateFrame (10, 10, 140, 90, 0) **loop** exit when GUI.ProcessEvent end loop

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SetDisplayWhenCreated, not by calling SetDisplayWhenC

GUI.SetEchoChar

Syntax	GUI.SetEchoChar (widgetID : int, echoChar : char)
Description	The GUI.SetEchoChar procedure is used with text fields, especial when using a text field to input a password. When the GUI.SetEchoChar is called with a text field, any character entered into the text field will appear as the character specified by <i>inputCh</i> .
	This allows you to use the text field to enter a password. The chara that the user types will be echoed with the character specified by <i>inputChar</i> (often an asterisk).
	This program displays a text field. As the user enters characters, ea character is represented as a '*'. When the user enters return, the program exits after displaying the actual text entered into the text f
	<pre>import GUI</pre>
Example	<pre>procedure EchoString (s : string) put "You entered \"", s, "\"" GUI.Quit end EchoString</pre>
	<pre>var tf := GUI.CreateTextField (10, 100, 100, "", GUI.SetEchoChar (tf, '*') loop exit when GUI.ProcessEvent end loop</pre>

Execute

DetailsNote that the *echoChar* argument to GUI.SetEchoChar must be a
character, not a string. This means the character should be enclosed
single quote marks ('), not double quotes ('').

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SetEchoChar, not by calling SetEchoChar.

GUI.SetKeyEventHandler

Description

GUI.SetKeyEventHandler (SyntaxkeyEventHandler : procedure x (ch : char))

Sets the new default keystroke event handler. The *keyEventHandle* the name of a procedure that is called every time **GUI.ProcessEve** there is a keystroke which is not handled by any widget. The *ch* pa *keyEventHandler* is the keystroke typed by the user.

This procedure is often used to allow for more than one shortcut ch given button.

The following program draws a star or quits depening on the button Star button can be activated by clicking on it or typing 'D', 'd', 'S', 's The Quit button can be activate by typing 'Q', 'q' or Ctrl+Q. The Dr is also the default button. It is activated whenever the user presses

```
import GUI
                     View.Set ("graphics:220;200")
                     procedure DrawStar
                         Draw.FillStar (25, 40, 175, 190, Rand.Int (1
                     end DrawStar
                     procedure KeyHandler (ch : char)
                         if ch = 'q' or ch = '^Q' then
                             Draw.Cls
Example
                             GUI.Quit
                         elsif ch = 'd' or ch = '^d' or ch = 'S' or c
                             DrawStar
                         end if
                     end KeyHandler
                     GUI.SetKeyEventHandler (KeyHandler)
                     var button : int := GUI.CreateButtonFull (5, 5,
                     DrawStar, 0, 'D', true)
var quitButton : int := GUI.CreateButtonFull (11
                         GUI.Quit, 0, 'Q', false)
                     loop
                         exit when GUI.ProcessEvent
                     end loop
```

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SetKeyEventHandler, not by calling SetKeyEventHandler.

See also <u>GUI.ProcessEvent</u>.

Syntax **GUI.SetLabel** (*widgetID* : **int**, *text* : **string**)

Changes the text of a widget specified by *widgetID* to *text*. This procedure can accept a button, check box, radio button, label, or a labelled frame widget as the *widgetID* parameter.

Description In most cases, if the text will not fit in the widget's current size, the widget will be resized to fit the text. If the widget was made larger to fit the text and then the text is changed, the widget will be resize as appropriate for the original *width* specified and the new text.

The following program changes the text in the button whenever a keystroke occurs. When the text is changed back to "Quit", the button assumes a width of 100 again.

```
import GUI in "%oot/lib/GUI"
                     View.Set ("graphics:220;50")
                     var short : boolean := true
                     var button : int
                     procedure KeyHandler (ch : char)
                         if short then
                             GUI.SetLabel (button, "Press This Buttor
Example
                         else
                             GUI.SetLabel (button, "Quit")
                         end if
                         short := not short
                     end KeyHandler
                     GUI.SetKeyEventHandler (KeyHandler)
                     button := GUI.CreateButton (10, 5, 100, "Quit",
                     1000
                         exit when GUI.ProcessEvent
                     end loop
            Exported qualified.
            This means that you can only call the function by calling
Status
```

GUI.SetLabel, not by calling **SetLabel**.

	GUI.CreateButton, GUI.CreateCheckBox,
See also	GUI.CreateRadioButton, GUI.CreateLabel, and
	GUI.CreateLabelledFrame .

GUI.SetMouseEventHandler

Syntax

GUI.SetMouseEventHandler (

mouseEventHandler : **procedure** x (mx, my : **int**))

Sets the new default mouse event handler. The mouseEventHandleparameter is the name of a procedure that is called every time**GUI.ProcessEvent** is called and there is a mouse button down whinot handled by any widget. The mx and my parameters in themouseEventHandler are the location of mouse when the button waspressed.

This procedure is used by programs to allow for mouse input in a program that uses widgets.

This is a program that allows the user to place stars on the screen. The menu bar allows the user to quit the program at any time. The user also toggle the appearance of the menu bar by pressing any key.

```
import GUI
var starX, starY, starColor : array 1 .. 100 of
var numStars : int := 0
var menuVisible : boolean := true
procedure DrawStar (i : int)
    if menuVisible then
        View.ClipSet (0, 0, maxx,
            maxy GUI.GetMenuBarHeight)
    end if
    Draw.FillStar (starX (i) 20, starY (i) 20,
        starY (i) + 20, starColor (i)) View.C
end DrawStar
procedure Redraw
    for i : 1 .. numStars
        DrawStar (i)
    end for
    Text.Locate (maxrow, 1)
    put "Press any key to toggle menu bar" ...
end Redraw
procedure KeyHandler (ch : char)
```

```
if menuVisible then
Example
                            GUI.HideMenuBar
                                                else
                            GUI.ShowMenuBar
                        end if
                        menuVisible := not menuVisible
                        Redraw
                    end KeyHandler
                    procedure MouseHandler (x, y : int)
                        if numStars = 100 then
                            Text.Locate (maxrow, 1)
                            put "Maximum number of stars exceeded!"
                            return
                        end if
                        numStars += 1
                        starX (numStars) := x
                        starY (numStars) := y
                        starColor (numStars) := Rand.Int (9, 15)
                        DrawStar (numStars)
                    end MouseHandler
                    var menu : int := GUI.CreateMenu ("File")
                    var menuItem : int := GUI.CreateMenuItemFull ("C
                        GUI.Quit, '^Q', false)
                    GUI.SetKeyEventHandler (KeyHandler)
                    GUI.SetMouseEventHandler (MouseHandler)
                    Redraw
                    100p
                        exit when GUI.ProcessEvent
                    end loop
            Exported qualified.
            This means that you can only call the function by calling
Status
            GUI.SetMouseEventHandler, not by calling
            SetMouseEventHandler.
```

See also <u>GUI.ProcessEvent</u>.

GUI.SetNullEventHandler

GUI.SetNullEventHandler (*nullHandler* : **procedure** *x* Syntax ())Sets the new null event handler. The *nullHandler* parameter is the name of a procedure that is called every time **GUI.ProcessEvent** is called and there are no mouse button presses or keystrokes to be processed. Description This is used by programs that need to call subprograms often, but do not wish to interrupt the action of user widgets. The following program has a Quit button. When no widgets are being processed, a clock in the corner is updated. import GUI View.Set ("graphics:220;50") var oldTime : string := "" var button : int procedure NullHandler var newTime : string := Time.Date newTime := newTime (11 .. *) if newTime not= oldTime then Example Text.Locate (maxrow, maxcol 9) put newTime ... oldTime := newTime end if end NullHandler **GUI.SetNullEventHandler** (NullHandler) button := GUI.CreateButton (10, 5, 100, "Quit", **loop** exit when GUI.ProcessEvent end loop Exported qualified. **Status** This means that you can only call the function by calling

GUI.SetNullEventHandler, not by calling SetNullEventHandler

See also <u>GUI.ProcessEvent</u>.

GUI.SetPosition

Syntax GUI.SetPosition (*widgetID*, *x*, *y* : **int**)

Description Moves a widget specified by *widgetID* to the location (x, y). If the widget is visible, it is moved immediately to the new location. If the widget is hidden, it will appear at the new location when the *Show* procedure is called. Note that the *x* and *y* specified here are the same as in the *Create* method. For example, if you had specified a check box to be right justified in the *CreateCheckBoxFull* function, then (x, y) in a call to *SetPosition* would specify the lower-right corner as opposed to the lower-left corner.

The following program moves the button every time the button is pressed.

import GUI in "%oot/lib/GUI"
var button : int
procedure MoveButton
var newX, newY : int
newX := Rand.Int (0, maxx GUI.GetWidth (but
newY := Rand.Int (0, maxy GUI.GetHeight (bu
GUI.SetPosition (button, newX, newY)
end MoveButton
button := GUI.CreateButton (100, 100, 0, "Move E
MoveButton)
loop
exit when GUI.ProcessEvent
end loop

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SetPosition, not by calling SetPosition.

GUI.SetPositionAndSize

Part of **GUI** module

GUI.SetPositionAndSize (*widgetID*, *x*, *y* : **int**,

Syntax width, height : **int**)

Changes the position and size of the widget specified by *widgetID* simultaneously. The *x*, *y*, *width* and *height* parameters have the same meaning as in the **GUI.Create** function for that widget. Any widget except a menu or a menu item can be resized, although for some widgets, the *width* or *height* parameter may be ignored.

GUI.SetPositionAndSize works the same way as the **GUI.SetPosition** and **GUI.SetSize** procedures.

The following program moves and resizes the button every time the button is pressed.

import GUI in "%oot/lib/GUI" **var** button, minWidth, minHeight : **int** procedure MoveButton var newX, newY, newWidth, newHeight : int newWidth := max (minWidth, Rand.Int (0, 200) newHeight := max (minHeight, Rand.Int (0, 10) newX := Rand.Int (0, maxx newWidth) Example newY := Rand.Int (0, maxy newHeight) **GUI.SetPositionAndSize** (button, newX, newY, *newWidth*, *newHeight*) end MoveButton button := GUI.CreateButton (100, 100, 0, "Move E *MoveButton*) minHeight := GUI.GetHeight (button) minWidth := GUI.GetWidth (button) 1000 exit when GUI.ProcessEvent end loop

Exported qualified.

Status This means that you can only call the function by calling

GUI.SetPositionAndSize, not by calling SetPositionAndSize.

GUI.SetScrollAmount

GUI.SetScrollAmount (*widgetID* : **int**,

Syntax arrowInc, pageInc, thumbSize : int)

 Sets a scroll bar's arrow increment, page increment and thumb size The *widgetID* specifies the scroll bar to be changed. The *arrowInc* value is changed when the scroll arrows are pressed). A value of 1 *pageInc* parameter specifies the new page increment (the amount tl up/down section of the scroll bar). A value of 1 means to use the pi specifies he new thumb size. See the scroll bar explanation for mor previously-defined thumb size.

> The following program displays an image in a canvas in a window. left are used to allow the user to see the entire image. A text field a button is pressed.

> > % The "ScrollPic2" program. import GUI in "%oot/lib/GUI" % The maximum width/height of the canvas. const maxSize : int := 220 **const** leftBorder : **int** := 15 % Left margin. **const** bottomBorder : **int** := 25 % Bottom margin. var h, v : int %
> > var canvas : int
> > var pic : int % The scroll bars. % The canvas. var pic : int % The picture.
> > var fileNameField : int % The file name text
> > var errorLabel : int % The error message
> > var loadButton : int % The "Load Picture" procedure ScrollPic (ignore : int) % Get the current value of the scroll bars. var x : int := GUI.GetSliderValue (h) var y : int := GUI.GetSliderValue (v) **GUI.PicDraw** (canvas, pic, x, y, picCopy) end ScrollPic procedure LoadFile (fileName : string) **var** picWidth, picHeight, canvasWidth, canvas var newPic : int := Pic.FileNew (fileName) if newPic <= 0 then</pre> **GUI.SetLabel** (errorLabel,

"Error loading picture: " + Error.La **GUI.SetSelection** (*fileNameField*, 1, 1) return else **GUI.SetLabel** (errorLabel, "") pic := newPic end if picWidth := Pic.GetWidth (pic) picHeight := Pic.GetHeight (pic) canvasWidth := min (picWidth, maxSize) canvasHeight := min (picHeight, maxSize) % Hide the canvas and the three items, readj % and then show them. **GUI.Hide** (canvas) GUI.Hide (h) GUI.Hide (v) **GUI.SetSize** (canvas, canvasWidth, canvasHeig **GUI.SetSliderSize** (*h*, *canvasWidth* + 1) **GUI.SetPosition** (v, 15 + canvasWidth, bottomBorder + GUI.GetScrollBarWidth - 1 Example **GUI.SetSliderSize** (v, canvasHeight + 1) **GUI.SetSliderMinMax** (h, 0, picWidth 1) **GUI.SetSliderMinMax** (v, 0, picHeight 1) **GUI.SetScrollAmount** (*h*, 3, 100, *canvasWidth*) **GUI.SetScrollAmount** (*v*, 3, 100, *canvasHeight* **GUI.SetSliderValue** (*h*, 0) **GUI.SetSliderValue** (*v*, *picHeight*) **GUI.Show** (canvas) **GUI.Show** (*h*) GUI.Show (v) ScrollPic (0) end LoadFile procedure LoadFileButton var fileName : string := GUI.GetText (fileNa LoadFile (fileName) end LoadFileButton **View.Set** ("graphics:265;295") % We place the canvas first and everything else % relative to the canvas. canvas := GUI.CreateCanvas (leftBorder, bottomBorder + GUI.GetScrollBarWidth, maxSiz **GUI.GetY** (canvas) **GUI.GetScrollBarWidth**, GUI.GetWidth (canvas), 0, 100, 0, ScrollPic, v := GUI.CreateVerticalScrollBarFull (**GUI.GetX** (canvas) + **GUI.GetWidth** (canvas), **GUI.GetY** (canvas), **GUI.GetHeight** (canvas), *C* 100, ScrollPic, 3, 100, maxSize)

fileNameField := GUI.CreateTextField (GUI.GetX (**GUI.GetY** (canvas) + **GUI.GetHeight** (canvas) + LoadFile) loadButton := GUI.CreateButton (GUI.GetX (fileNa **GUI.GetWidth** (fileNameField) + 20, **GUI.Gety** (*fileNameField*), 0, "Load File", *Lc* errorLabel := GUI.CreateLabel (GUI.GetX (canvas) % Set the initial picture and return if it is no **GUI.SetText** (*fileNameField*, "Forest.bmp") LoadFileButton if pic = 0 then return end if **100**p exit when GUI.ProcessEvent end loop

Exported qualified.

Status This means that you can only call the function by calling **GUI.Set**§

See also GUI.CreateHorizontalScrollBar and GUI.CreateVerticalScroll1

GUI.SetScrollOnAdd

GUI.SetScrollOnAdd (widgetID : int, scrollOnAdd : boo

Syntax

Description

Example

The **GUI.SetScrollOnAdd** procedure allows you to specify wheth scrolls to the bottom of the text when new text is added (the defaul By calling this procedure with *scrollOnAdd* set to **false**, the text bo scroll unless the user manipulates the text box's scroll bars.

This program displays the contents of a file in a text box. After the displayed, the top line in the text box will be the first line in the file

import GUI View.Set ("graphics:300;300") **const** fileNameToBeViewed : **string** := "TextBxs.DE var textBox : int % The Text Field ID. var title : int % The label for the title. var f : int % The stream number of the f % Lines to be read from var line : string % Open the file. open : f, fileNameToBeViewed, get if f = 0 then put "Unable to open " + fileNameToBeViewed + return end if % Set background color to gray for indented text **GUI.SetBackgroundColor** (gray) % Create the title label and text box. title := GUI.CreateLabelFull (20, 280, fileName1 GUI.CENTER, 0) textBox := GUI.CreateTextBoxFull (10, 10, 280, 2 GUI.INDENT, 0) **GUI.SetScrollOnAdd** (*textBox*, **false**) % Read the file and place it in the text box. **loop** exit when eof (f) get : f, line : **GUI.AddLine** (*textBox*, *line*) end loop

close : f % Close the file.
loop
 exit when GUI.ProcessEvent
end loop

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SetScrollOnAdd, not by calling SetScrollOnAdd.

GUI.SetSelection (widgetID, fromSel, toSel : int) Syntax Sets the selected text in the text field specified by *widgetID*. The va the *fromSel* and *toSel* parameters indicate the characters where the will begin and end. For example, if the text was "Hello there", setti fromSel to 2 and toSel to 5 would select "ell". Setting fromSel and t automatically selects the entire text. The *fromSel* parameter specifies the start of the selection. This range 1 (before the first character) to the number of characters in the text Description the last character). A value of 1 for both *fromSel* and *toSel* selects t text. The *toSel* parameter specifies the end of the selection. This ranges (before the first character) to the number of characters in the text + the last character). A value of 1 for both *fromSel* and *toSel* selects t text. The following program allows the user to type into a text field. Wh user presses ENTER, it searches for any non-lowercase text and if any, selects it to make it easy for the user to correct it. If all the inp lower-case text, the program terminates. import GUI in "%oot/lib/GUI" var textField, lbl : int procedure CheckInput (s : string) **for** i : 1 .. **length** (s) if (s(i) < 'a' or 'z' < s(i)) and s(i)**GUI.SetSelection** (*textField*, *i*, *i* + return end if Example end for GUI.Quit end CheckInput textField := GUI.CreateTextField (100, 100, 200, lbl := GUI.CreateLabelFull (100 + GUI.GetWidth (100 + GUI.GetHeight (textField),

"Only Allows Lower Case Letters", 0, 0, GUI.CENTER + GUI.BOTTOM, 0) loop exit when GUI.ProcessEvent end loop GUI.SetLabel (1b1, "Program Finished!") Exported qualified.

- StatusThis means that you can only call the function by calling GUI.SetS
not by calling SetSelection.
- See also GUI.CreateTextField.

GUI.SetSize

Syntax GUI.SetSize (*widgetID*, *width*, *height* : **int**)

Changes the size of the widget specified by *widgetID*. If the widget is visible, its size is changed immediately, otherwise the widget will appear in its new size when the widget is next made visible. Note that the *width* and *height* parameters are no

Description necessarily the actual width and height of the widget. For example, the *TextField* widget ignores the *height* parameter, calculating the widget's actual height from the height of the text in the *TextField*.

The following program resizes the button every time the button is pressed.

import GUI in "%oot/lib/GUI"
var button : int
procedure ResizeButton
var newWidth, newHeight : int
newWidth := Rand.Int (0, 200)
GUI.SetSize (button, newWidth, newHeight)
end ResizeButton
button := GUI.CreateButton (100, 100, 0, "Resize
ResizeButton)
loop
exit when GUI.ProcessEvent
end loop

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SetSize, not by calling SetSize.

GUI.SetSliderMinMax

Part of <u>GUI</u> module

Syntax	GUI.SetSliderMinMax (widgetID, min, max : int)
Description	Sets the minimum and maximum values of the slider or scroll bar specified by <i>widgetID</i> . The <i>min</i> parameter specifies the new minimum value of the slider or scroll bar. The <i>max</i> parameter specifies the new maximum value of the slider or scroll bar. The <i>max</i> parameter must be greater than the <i>min</i> parameter. GUI.SetSliderMinMax redraws the thumb to take into account the new minimum and maximum. If the current value of the slider is outside the new minimum/maximum, then the value is adjusted
	appropriately. See GUI.SetScrollAmount for an example of
Example	GUI.SetSliderMinMax.
	Exported qualified.
Status	This means that you can only call the function by calling GUI.SetSliderMinMax , not by calling SetSliderMinMax .
See also	GUI.CreateHorizontalScrollBar, GUI.CreateVerticalScrollBar, <u>GUI.CreateHorizontalSlider</u> , and <u>GUI.CreateVerticalSlider</u> .

GUI.SetSliderReverse

Syntax GUI.SetSliderReverse (widgetID : int)

Description
 Sets a slider or scroll bar specified by *widgetID* into (or out of, if a into) "reverse mode". Normally, a slider or scroll bar is at its minin value when the thumb is on the left hand side (bottom for a vertica. This reverses it, so the minimum value is when the thumb is at the hand side (top for vertical sliders) of the track. Calling this routine second time reverses it back to normal. This procedure redraws the to move the thumb to its new location.

The following program creates two sliders, one of which is reverse

import GUI in "%oot/lib/GUI" View.Set ("graphics:300;70") var sBar, sBarLabel, reverseSBar, reverseSBarLat **procedure** SBarMoved (value : **int**) **GUI.SetLabel** (*sBarLabel*, **intstr** (*value*)) end SBarMoved procedure ReverseSBarMoved (value : int) **GUI.SetLabel** (reverseSBarLabel, intstr (valu end ReverseSBarMoved Example sBar := GUI.CreateHorizontalScrollBar (10, 10, 2 50, 150, 50, *SBarMoved*) sBarLabel := GUI.CreateLabel (**GUI.GetX** (*sBar*) + **GUI.GetWidth** (*sBar*) + 10, reverseSBar := GUI.CreateHorizontalScrollBar (10 50, 150, 50, ReverseSBarMoved) **GUI.SetSliderReverse** (reverseSBar) reverseSBarLabel := GUI.CreateLabel (GUI.GetX (**GUI.GetWidth** (*reverseSBar*) + 10, 40, "50") **loop** exit when GUI.ProcessEvent end loop

Exported qualified.

Status This means that you can only call the function by calling

GUI.SetSliderReverse, not by calling **SetSliderReverse**.

GUI.CreateHorizontalScrollBar, GUI.CreateVerticalScrollBarSee alsoGUI.CreateHorizontalSlider, and GUI.CreateVerticalSlider.

GUI.SetSliderSize

Part of <u>GUI</u> module

Syntax	GUI.SetSliderSize (widgetID, length : int)
Description	Changes the length of a slider or scroll bar specified by <i>widgetID</i> to the value specified by the <i>length</i> parameter. Redraws the slider or scroll bar and changes the position of the thumb to take into account the new size of the slider or scroll bar.
Example	See GUI.SetScrollAmount for an example of GUI.SetSliderSize .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.SetSliderSize , not by calling SetSliderSize .
See also	GUI.CreateHorizontalScrollBar, GUI.CreateVerticalScrollBar, <u>GUI.CreateHorizontalSlider</u> , and <u>GUI.CreateVerticalSlider</u> .

GUI.SetSliderValue

Part of <u>GUI</u> module

Syntax	GUI.SetSliderValue (widgetID, value : int)
Description	Sets the value of a slider or scroll bar specified by <i>widgetID</i> to <i>value</i> . It moves the thumb on the slider or scroll bar to the appropriate location and calls the slider's <i>action procedure</i> with the new value.
Example	See GUI.SetScrollAmount for an example of GUI.SetSliderValue .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.SetSliderValue , not by calling SetSliderValue .
See also	GUI.GetSliderValue for reading a slider <u>or</u> scroll bar's value. See also GUI.CreateHorizontalScrollBar , GUI.CreateVerticalScrollBar , <u>GUI.CreateHorizontalSlider</u> , and <u>GUI.CreateVerticalSlider</u> .

Part of **<u>GUI</u>** module

Syntax **GUI.SetText** (*widgetID* : **int**, *text* : **string**)

Description Sets the text of a text field specified by *widgetID* to *text*. The select to 1, 1 (i.e. the cursor is at the beginning of the text).

Exported qualified.

StatusThis means that you can only call the function by calling GUI.Setby calling SetText.

The following program converts all lower case input in the text fiel upper case when the user presses ENTER.

import GUI var textField, lbl : int procedure CheckInput (s : string) var newString : string := "" for $i : 1 \dots$ length (s)if 'a' <= s (i) and s (i) <= 'z' then newString += chr (ord (s (i)) 32) else newString += s(i)Example end if end for **GUI.SetText** (textField, newString) **GUI.SetSelection** (*textField*, 1, 1) end CheckInput textField := GUI.CreateTextField (100, 100, 200, lbl := GUI.CreateLabelFull (100 + GUI.GetWidth (100 + **GUI.GetHeight** (*textField*), "Converts 0, 0, GUI.CENTER + GUI.BOTTOM, 0)1000 exit when GUI.ProcessEvent end loop

Exported qualified.

StatusThis means that you can only call the function by calling GUI.Setby calling SetText.

See also GUI.CreateTextField.

GUI.SetTopLine

Part of **<u>GUI</u>** module

Syntax	GUI.SetTopLine (widgetID : int, lineNum : int)
Description	The GUI.SetTopLine procedure scrolls the text in a text box so as to place the line number specified by <i>lineNum</i> at the top of the text box (if possible). This is often used to scroll the text back to the beginning of the text box by assigning a value of 1 to <i>lineNum</i> .
	This program displays 100 lines of text in a text box, then sets the text box to display line 50 at the top of the box.
	import GUI
	<pre>var textBox : int % The Text Field ID. textBox := GUI.CreateTextBoxFull (10, 10, 180, 2 GUI.INDENT, 0)</pre>
Example	GUI.SetBackgroundColor (gray)
	% Read the file and place it in the text box. for i : 1 100 GUI.AddLine (<i>textBox</i> , intstr (i)) end for GUI.SetTopLine (<i>textBox</i> , 50)
	loop exit when GUI.ProcessEvent end loop

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SetTopLine, not by calling SetTopLine.

GUI.SetXOR

Syntax GUI.SetXOR (*widgetID* : **int**, *xor* : **boolean**)

Description Sets the "xor mode" of the canvas specified by *widgetID*. If the *xor* parmeter is set to **true**, the canvas is set to *xor mode*. When in *xor mode*, all the *Draw*... procedures of a canvas are treated as if the **View.Set** ("xor") statement had been executed before the *Draw* procedure.

See **GUI.SetDisplayWhenCreated** for an example of **GUI.Show**.

import GUI in "%oot/lib/GUI" View.Set ("graphics:400;300") var canvas1, label1, canvas2, label2 : int canvas1 := GUI.CreateCanvas (10, 20, maxx div 2 label1 := GUI.CreateLabelFull (10, 2, "XOR", max GUI.CENTER, 0)canvas2 := GUI.CreateCanvas (maxx div 2 + 10, 20 maxx div 2 20, maxy 30) label2 := GUI.CreateLabelFull (maxx div 2 + 10, maxx div 2 20, 0, GUI.CENTER, 0) **GUI.SetXOR** (canvas1, true) Example for i : 1 .. 20 var x : int := Rand.Int (0, maxx div 2 20) var y : int := Rand.Int (0, maxy 20) var c : int := Rand.Int (1, 15) GUI.DrawFillStar (canvas1, x 20, y 20, x + end for **GUI.SetXOR** (*canvas2*, **false**) for *i* : 1 .. 20 var x : int := Rand.Int (0, maxx div 2 20) var y : int := Rand.Int (0, maxy 20) var c : int := Rand.Int (1, 15) GUI.DrawFillStar (canvas2, x 20, y 20, x + end for

Exported qualified.

StatusThis means that you can only call the function by calling
GUI.SetXOR, not by calling SetXOR.

See also GUI.CreateCanvas.

GUI.Show

Syntax	GUI.Show (widgetID : int)
	Shows a widget specified by <i>widgetID</i> . Used in conjunction with GUI.Hide to show and hide widgets. Hidden widgets cannot get events (i.e. respond to keystrokes or mouse clicks). If an active text field (see text field) is hidden, then any keystrokes in the window will be ignored.
Description	In most cases where a widget is to appear, then disappear, then appear again, it is advised to create the widget once and hide it until it is to appear, whereupon GUI.Show is called. When the user is finished with the widget, the widget is hidden using GUI.Hide . This saves the overhead of creating and disposing of the same widget several times.
Example	See GUI.SetDisplayWhenCreated for an example of GUI.Show .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.Show , not by calling Show .
See also	<u>GUI.Hide</u> .

GUI.ShowMenuBar

Part of <u>GUI</u> module

Syntax	GUI.ShowMenuBar
Description	Shows the menu bar in the selected window.
Example	See GUI.SetMouseEventHandler for an example of GUI.HideMenuBar .
	Exported qualified.
Status	This means that you can only call the function by calling GUI.ShowMenuBar , not by calling ShowMenuBar .
See also	GUI.HideMenuBar. See also GUI.CreateMenu.

GUI.ShowWindow

Part of **GUI** module

Syntax GUI.ShowWindow (window : int)

DescriptionHides a window with widgets in it. This procedure makes certain the
Library recognizes that the window no longer visible. This procedu
Window.Hide, so there is no need for the user to do so.

The program opens up a window with two buttons. If the button lal and Open" is pressed, the window is closed and a new window wit opened in a random location on the screen.

import GUI var windowA, windowB : int var buttonA, buttonB, buttonQuit : int var backToMainA, backToMainB : int procedure ShowWindowA GUI.HideWindow (defWinID) **GUI.ShowWindow** (windowA) end ShowWindowA procedure ShowWindowB GUI.HideWindow (defWinID) **GUI.ShowWindow** (windowB) end ShowWindowB procedure ShowMain Example GUI.HideWindow (Window.GetActive) GUI.ShowWindow (defWinID) end ShowMain View.Set ("graphics:280;100, nobuttonbar") % Place the buttons in the main window buttonA := GUI.CreateButton (10, 10, 0, "Show Wi buttonB := GUI.CreateButton (150, 10, 0, "Show W buttonQuit := GUI.CreateButton (100, 60, 0, "Qui windowA := Window.Open ("title:Window A, graphic backToMainA := GUI.CreateButton (20, 20, 0, "Bac Window.Hide (windowA) windowB := Window.Open ("title:Window B, graphic backToMainB := GUI.CreateButton (20, 20, 0, "Bac Window.Hide (windowB)

loop
 exit when GUI.ProcessEvent
end loop

Execute

Exported qualified.

StatusThis means that you can only call the function by calling GUI.Sho
by calling ShowWindow.

GUI.HideWindowfor hiding a window with widgets in it. See alsoSee alsoGUI.CloseWindowfor closing a window with widgets in it.

handler

exception handler

Dirty

A *exceptionHandler* is:

Syntax

handler (*id*) statementsAndDeclarations end handler

An exception handler is an optional block of statements and declarations in a subprogram (or process). It is activated when the Description program (or process) fails. This occurs, for example when dividing by zero.

> This program parses the input stream using a stack. If the stack overflows (its top exceeds its maximum), a **quit** statement in the *push* procedure aborts the parsing and gives control to the exception handler in the *parse* procedure. The *parse* procedure calls *parseExpn* which calls *push*. If *push* overflows the stack, it executes a **quit** and control is passed to the exception handler in the *parse* procedure. The interrupted procedures (*parseExpn* and *push*) are terminated and their local variables are deleted.

```
const stackOverflow := 500
const maxTop := 100
var top : 0 .. maxTop := 0
var stack : array 1 .. maxTop of int
procedure push ( i : int )
    if top = maxTop then
        quit : stackOverflow
    end if
    top := top + 1
    stack ( top ) := i
end push
procedure parse
    handler ( exceptionNumber )
    put "Failure number ", exceptionNumber
         case exceptionNumber of
```

Example

```
label stackOverflow :
    put "Stack has overflowed!!"
    ... other exceptions handled here ...
    label : % Unexpected failures
    quit > % Pass exception further
    end case
    end handler
    parseExpn % Eventually push is cal
end parse
```

See the **quit** statement for an explanation of its *quitReason* (*stackOverflow* in the first **quit** statement above) and its *guiltyParty* (> in the second **quit** statement, meaning the exception is due to causes outside of this handler).

An exception handler can appear only in the body of a subprogram (or process), just preceding the declarations and statements. The form of a procedure is:

```
procedure [ pervasive ] id
  [ ( [ paramDeclaration {, paramDeclaration }
     [ import [ [var] id {, [var] id } ] ]
     [ pre trueFalseExpn ]
     [ init id := expn {, id := expn } ]
     [ post trueFalseExpn ]
     [ exceptionHandler ]
     statementsAndDeclarations
end id
```

Exactly the same declarations and statements can appear in a handler as can appear in the subprogram body following the handler. In the absence of exceptions, handlers have no observable effect. A particular handler is activated (it becomes ready to handle an exception) when it is encountered during execution. It remains active until the subprogram (or process) containing it has completed, or the handler is given control. Activation of a handler when a previous handler is already active will cause exceptions to be passed to the newly-activated handler. In other words, handlers have a dynamic scope that begins when the exception handler is encountered and ends when the subprogram (or process) containing the handler has terminated or the handler is given control. When a handler is given control, it becomes, in effect, a replacement for the declarations and statements following it. If the handler is in a function, it must terminate with a **result** statement or with a **quit**. If the handler is in a procedure (or process), the handler must terminate with a **return**, a **quit**, or by encountering the end of the handler (which is equivalent to a **return**).

Details

When a handler terminates with a **result** or **return** statement (or by reaching the end of a procedure's handler), the subprogram's **post** condition (if any) must be true. A **quit** statement does not need to establish the **post** condition.

Programming with exception handlers easily leads to incomprehensible software, due to the difficulty of keeping track of the flow of control. One of the most insidious situations is when an exception occurs in a module, class or monitor and is propagated outside of the unit. This can leave the contained data in an inconsistent state; in the case of a monitor, it is left locked forever. To avoid this possibility, you can use a handler in each exported subprogram. If an exception in a process is not handled, the entire program is aborted. If an implementation allocates dynamic arrays on the heap, an exception may prevent the deallocation of such an array.

Without exception handling, a program executes according to the language definition or else is aborted. If an exception handler is active, instead of aborting, control is given to the handler. The *quitNumber* for a system-detected failure is implementation-dependent. There is a file "*%exceptions*" which lists these numbers. The user program can simulate a system exception by doing a **quit** with the corresponding number.

If the user turns off checking explicitly, the system may not detect failures. In some cases the failure may yield incorrect data or arbitrary behavior.

Some exceptions are unpredictable or implementation-dependent. For example, in x := 24 div i + 24 / i, if *i* is zero, the exception

could be either an integer or a real division by zero, because the order or evaluation is implementation-dependent.

has character function

hasch

Syntax hasch : boolean

Description The **hasch** procedure is used to determine if there is a character that has been typed but not yet been read.

This program simulates the rolling of a pair of dice until the user pressed the keyboard.

put "Press any key to stop the dice rolling"
var die1, die2 : int
var ch : string (1)
loopExampleexit when hasch
randint (die1, 1, 6)
randint (die2, 1, 6)
locate (1, 1)
put "You rolled ", die1 + die2
end loopgetch (ch)% Discard the character

Execute

The screen should be in a "*graphics*" mode. See the **setscreen** procedure for details. If the screen is not in "*graphics*" mode, it will automatically be set to "*graphics*" mode.

Details If there is the possibility that there are already keystrokes in the keyboard buffer, the **Input.Flush** command can be used to flush the keyboard buffer (remove all keystrokes from the buffer) before entering the loop to check for input.

getch and getchar.

See also the Input.Flush command for flushing the keyboardSee alsobuffer.

See also predefined unit <u>Input</u>.

Variables, constants, types, procedures, etc. in Turing programs are given names such as *incomeTax*, *x*, and *height*. These names are called identifiers (*ids*).

An identifier must start with a letter (large or small) or an underscore (_) and can contain up to 50 characters, each of which must be a letter, a digit (0 to 9) or an underscore (_). Large and small letters are considered distinct, so that *A* and *a* are different names. This differs from Pascal where large and small letters in names are equivalent.

Description Every character in a name is significant in distinguishing one name from another.

By convention, words that make up an identifier are capitalized (except the first one), as in *incomeTax* and *justInTime*.

An item in a Turing program cannot be given the same name as a keyword such as **get** or as a reserved word such as **index**. See Appendix A for a list of keywords and reserved words. As well, there are some identifiers that are used by the Turing error recovery procedures and are thus unavailable for use as identifiers. Specifically, they are: *endif, elseif, endloop* and *endfor*.

A conditional compilation **#if** has the form:

	# if expn then
Syntax	any source text
	{ #elsif expn then
	any source text }
	[#else
	any source text]
	#end if
	An #if construct supports compile time selection

An **#if** construct supports compile time selection of sections of source text to make up a program (or unit of a program), in other words *conditional compilation*. Any arbitrary source text (characters) can be selected.

Each of the selecting expressions (*expns*) have the form of a boolean expression, with the use of the operators and, or and not (but not =>) and parentheses. The short forms & and ~ are supported. The operands of the expressions must be *preprocessor flags*, which are set by a system- dependent mechanism not described here. A flag is considered to be true if it is explicitly set. If it is not explicitly set, it is considered false.

Unlike other parts of the language, the **#if**, **#elsif**, **#else** and **#end if** constructs are not free format. Specifically, they must be placed by themselves on a single line.

A pair of declarations is chosen if both *stats* and *debug* are set, otherwise the **put** statement is selected. The selected part becomes part of the program and the other parts are ignored.

Example #if stats and debug then var count : array 1 .. 5 of real var message : string #else put "Debugging message"
#end if

statement

An *ifStatement* is:

Syntax

statementsAndDeclarations
{ elsif trueFalseExpn then
 statementsAndDeclarations }
[else

if trueFalseExpn then

statementsAndDeclarations]

end if

An **if** statement is used to choose among a set of statements (and declarations). One set (at most) is chosen and executed and then execution continues just beyond **end if**.

The expressions (the *trueFalseExpressions*) following the keyword **if** and each **elsif** are checked one after the other until one of them is found to be true, in which case the statements (and declarations) following the corresponding **then** are executed. If none of these expressions evaluates to true, the statements following **else** are executed. If no **else** is present and none of the expressions are true, no statements are executed and execution continues following the **end if**.

Output a message based on value of mark.

Example

```
if mark >= 50 then
    put "You pass"
else
    put "You fail"
end if
```

Output A, B, C, D or F depending on mark.

if mark >= 80 then
 put "A"
elsif mark >= 70 then

Example	<pre>put "B" elsif mark >= 60 then put "C" elsif mark >= 50 then put "D" else put "F" end if</pre>
	If <i>x</i> is negative, change its sign.
Example	if x < 0 then
	If x is less than zero or greater than maxx, put a message.
Example	<pre>if x < 0 or x > maxx then put "Out of bounds!" end if</pre>
	If the boolean <i>flag</i> is true and <i>name</i> is "stop", put a message and return.
Example	<pre>if flag and name = "stop" then put "Exiting routine" return end if</pre>
Details	Several statements and declarations can appear after a particular then .
See also	<u>case</u> statements for another way to select among statements.

implement by

An *implementByClause* is:

Syntax implement by *implementByItem* An implement-by clause is used to specify that a module, monitor or class *C* is to be automatically implemented by the *implementByItem*. *C* is called the *interface* and the *implementByItem*, which must contain an implement clause, is called the *implementation*. See **implement** clause for details and an example. The implement-by clause can only be used in a **unit**. See **unit** for the definition of a unit. An implementByItem is one of: (a) *id* **Description** (b) *id* **in** *fileName* The second form is used when the implement-by clause is for a separate **unit** and the imported item is in a file whose name is different from the item's name, as in: implement by ledgerBody in "ledgbod.t" The fileName must be an explicit character string, e.g.,

"ledgbod.t". See also **unit**. Parentheses are allowed around the items in an **implement-by** clauses, as in:

implement by (ledgerBody in "ledgbod.t")

clause

implement

Syntax

An *implementClause* is:

implement implementItem

	An implement clause is used to specify that the modul			
	monitor or class containing the clause is to be the			
	implementation of another module, monitor or class.			
	This implementation is a special kind of expansion. The			
Description	module, monitor or class containing the clause gains			
	access to (inherits) all the declarations inside the target			
	item. See inherit clause for rules about expansions,			
	which are also rules for implementations.			

The **implement** clause can only be used in a **unit**. See **unit** for the definition of a unit.

Here is a *stack* module which defers all of its exported subprograms. This module is an interface but not an implementation. Following *stack* is the *stackBody* module that implements the *stack* module, giving the bodies for *stack*'s subprograms. Any call to *stack's push* or *pop* procedures, such as *stack.push(*"Ed"), will actually call the procedures given in *stackBody*.

> module stack % Interface implement by stackBody % stackBody has implemer. export push, pop deferred procedure push (s : string) deferred procedure pop (var s : string) end stack

Example Next comes the expansion which gives the bodies for the deferred procedures *push* and *pop*. The *stackBody* body also adds declarations for the *top* and *contents* variables.

module stackBody	% Implementation
<pre>implement stack</pre>	% stack has interface

clause

```
var top : int := 0
var contents : array 1 .. 100 of string
body procedure push % ( s : string )
    top := top + 1
    contents ( top ) := s
end push
body procedure pop % ( var s : string )
    s := contents ( top )
    top := top - 1
end pop
end stackBody
```

Module, monitor or class D can be in C's implement-by
clause if, and only if, C is in D's implement clause. In
other words, an interface must apply to exactly one
implementation and vice versa. A module can implement
only a module, a monitor only a monitor, and a class only
a class. Classes (but not modules and monitors) can
(a)
contain inherit clauses. A class cannot contain both an
id
inherit and an implement clause.

An implementItem is one of:

(b) *id* **in** *fileName*

Details

The second form is used when the implement clause is for a separate **unit** and the imported item is in a file whose name is different from the item's name, as in:

implement ledger in "ledg.t"

The *fileName* must be an explicit character string, e.g., "ledg.t". See also **unit**. Parentheses are allowed around the items in **implement** clauses, as in:

```
implement ( ledger in "ledg.t" )
```

There is no restriction on the declarations that an interface may contain. In particular, an interface (any module, monitor or class containing an implementby clause), can contain subprogram bodies and variable declarations, exactly as is the case in expansions. This is different from languages such as C++ in which there are strict rules limiting what you can put in an interface. Even though *D* contains an **implement** clause, *D* can also contain an implementby clause, which implies further implementation by further automatic expansion.

Suppose class *D* is in class *C*'s implement-by clause and that *p* is a pointer to class *C*:

var *p* : ^ *C*

Even though *C* is implemented by *D*, *p* remains a pointer to class *C*. Each creation of an object of class *C* actually creates an object of type *D*, for example:

new p % Creates object of class D

Class *D*, which implements *C*, could also have an implement-by clause, which causes its implementation to be automatically created and so on. If another class *E* inherits *C*, this expansion does not include *D*.

If the **new** statement contains an explicit class name *E* that is a descendant of *C* (but not actually C), as in

new E, p

the object of the explicit class is created. If *E* has an implement-by clause, the expansion is created.

See also <u>unit</u>, <u>module</u>, <u>monitor</u> and <u>class</u>. See also <u>implement by</u> clause, <u>inherit</u> clause, <u>export</u> list, and <u>import</u> list. See also <u>deferred</u> subprograms.

import

An *importList* is:

Syntax import [howImport] importItem {, [howImport] importItem }

An import list is used to specify those items that a procedure, function, module, monitor, or a class uses from outside of itself. Note that a function or procedure is not allowed to have an import list and thus automatically imports whichever functions or procedures are used by the function or procedure. The compiler determines the list automatically by looking to see what items are actually used.

In this example, the type *T* is imported into the *stack* **module** and used as the type that can be pushed onto or popped off the stack. Since no other items are imported, the only identifiers from outside of *stack* that can be used in it must be predefined, such as **sqrt**, or declared to be **pervasive**.

type T : string

Example

module stack
 import T
 export push, pop
 var top : int := 0
 var contents : array 1..100 of T
 procedure push ... end push
 procedure pop ... end pop
end stack

The *importItem* is one of:

(a) *id*(b) *id* **in** *fileName*

The second form is used in OOT when the list is the import list for a separate **unit** (or the main program), and the imported item

is in a file whose name is different from the item's name, for example:

import ledger in "newledg.t"

The *fileName* must be an explicit character string. See also **unit**.

Parentheses are allowed around the items in an import lists, as in:

import (ledger in "newledg.t")

There are various ways to import items, as determined by *howImport*. The form of *howImport* is one of:

(a) var(b) const(c) forward

Details

Commonly the *howImport* is omitted, which means the default access for the item is the same access as the item has. In other words, a read-write item that is imported without a *howImport* is imported read-write. A read-only symbol that is imported without a *howImport* is imported read-only.

If the *importItem* is **forward**, the import list is part of a **forward** procedure or function declaration and the imported item is itself necessarily a procedure or function. See **forward** declarations for details and an example.

If the **import** list of a **module**, **monitor** or **class** is omitted, the implementation assumes that the list is **import**(), meaning that no items are imported. For example, a **module** must explicitly import any global identifiers that are not predefined or **pervasive**.

Circular (recursive) imports are not allowed. For example, if unit A imports B then B cannot import A. However, circular usage of separately compiled units is possible by separating the units into interfaces and bodies and having the bodies import the interfaces. For example, if C is the parent class of D, D can import C, but not vice versa.

	In an expansion (or implementation), the import list of the expansion augments the import list of the parent.
	An overriding subprogram (in an expansion) ignores the import list of the target subprogram and uses its own import list.
	Turing initializes modules and monitors in order of importation. Initialization begins with the main program, which first initializes its imports in the order given in its import list, and then initializes itself.
See also	unit , module , monitor and class . See also export list, inherit clause, implement clause and implement by clause.

Syntax	in		
Description	The in operator determines if an element is in a set.		
Example	type rankSet : set of 0 10 var rankings : rankSet := rankSet (0) % The s if 5 in rankings then % Is 5 in the rankin		
	The not in operator is exactly the opposite of in . For example, 7 not in <i>rankings</i> means the same as not (7 in <i>rankings</i>).		
Description	The element is required to be in the set's index type. In the above example this is satisfied because element 5 is in the index type 0 10.		
	The keyword in is also used in lists such as import lists. See import list.		
See also	the <u>set</u> type, <i>infix operators</i> , and <i>precedence</i> of operators.		

source files

include

An includeConstruct is:

Syntax include *fileName*

An include is used to copy parts of files so that they become part of the Turing program. This copying is temporary, that is, no files are changed. The file name must be an explicit string constant such as "stdstuff".

On IBM PC compatible computers, there are arrow keys that produce character values such as 200 and 208. Let us suppose that a file called *arrows* contains definitions of these values:

const upArrow := 200
const downArrow := 208
const rightArrow := 205
const leftArrow := 203

These definitions can be included in any program in the following manner:

t	i nclude "arrows"			
Example				
F \	/ar ch : string (1)			
<u>(</u>	getch (ch)	% Reac	l one	character
(case ord (ch) of			
	label upArrow :			
	handle up arrow			
	label downArrow :			
	handle down arrow			
	<pre>label rightArrow :</pre>			
	handle right arrow.			
	label leftArrow :			
	…handle left arrow…			
	label :			
	handle any other ke	<i>Ξγ</i>		
(end case	2		

An include file can itself contain **include** constructs. This can continue to any level, although a circular pattern of includes

Details	would be a mistake, as it would lead to an infinitely long program.
	It is common to save procedures, functions and modules in separate files. The files are collected together using include .
Details	If the filename in the include statement starts with a "%", then Turing searches the system directory for the file. See the editor reference for the environment to see how to set the system directory. This method can be used to allow the system administrator to easily supply a set of routines in a file to a large number of users by placing it in one easy-to-find location.
	If the system directory is set to "C:\TURING", then the line
Example	<pre>include "%sorting.t"</pre>
-	
	will include the file "C:\TURING\SORTING.T" in the program.
Details	will include the file "C:\TURING\SORTING.T" in the program. Under OOT, there are several system directories available. The "%oot" directory is the directory where all the OOT system files are located. The "%home" directory is the user's home directory.
Details	Under OOT, there are several system directories available. The "%oot" directory is the directory where all the OOT system files
Details Example	Under OOT, there are several system directories available. The "%oot" directory is the directory where all the OOT system files are located. The "%home" directory is the user's home directory.

index	find pattern in string function			
Syntax	<pre>index (s , patt : string) : int</pre>			
Description	The index function is used to find the position of <i>patt</i> within string <i>s</i> . For example, index ("chair", "air") is 3.			
Example	This program outputs 2, because "ill" is a substring of "willing", starting at the second character of "willing".			
	<pre>var word : string := "willing" put index (word, "ill")</pre>			
Details	If the pattern (<i>patt</i>) does not appear in the string (<i>s</i>), index returns 0 (zero). For example, here is an if statement that checks to see if string <i>s</i> contains a blank:			
	if index (s , " ") not= 0 then			
	The index is sometimes used to efficiently determine if a character is one of a given set of characters. For example, here is an if statement that checks to see if <i>ch</i> , which is declared using var <i>ch</i> : string (1), is a digit:			
	if index ("0123456789", <i>ch</i>) not= 0 then			
	If a string contains more that one occurrence of the pattern, the leftmost location is returned. For example, index ("pingpong", "ng") returns 3.			
	If <i>patt</i> is the null string, the result is 1.			

indexType

An *indexType* is one of:

		subrangeType enumeratedType	
Syntax	(c)	namedType	% Which is a subrange or enumerated type
	(d)	char	
	(e)	boolean	

DescriptionAn index type defines a range of values that can be used as an **arra**
subscript, as a case selector, as a selector (tag) for a **union** type, or
base type of a **set** type.

Evample	<pre>var z : array 1 9 of real</pre>	% 09 is an inc
Example	type <i>smallSet</i> : set of 0 2	% 02 is an inc

indirection

operator (@)

Dangerous

Syntax	targetType @ (expn)		
Description	The indirection operator @ is used to access values that lie at absolute machine addresses in the computer's memory. This is dangerous and implementation-dependent and can cause arbitrary corruption of data and programs.		
	Copy the byte value at memory location 246 into b and then set tha memory byte to zero.		
Example	var b : nat1 % One byte natural numbe b := nat1 @ (246) nat1 @ (246) := 0		
	The form of <i>targetType</i> must be one of:		
	<pre>(a) [id .] typeId (b) int, int1, int2 or int4 (c) nat, nat1, nat2 or nat4 (d) boolean (e) char [(numberOfCharacters)] (f) string [(maximumLength)] (g) addressint</pre>		
Details	In form (a) the beginning identifier <i>id</i> must be the name of a module, monitor or class that exports the <i>typeId</i> . Each of <i>numberOfCharacters</i> and <i>maximumLength</i> must be compile time integer expressions. These are the same target types as in type cheats.		
	The indirection operator @ takes an integer as an address. This value must fit in the range of addressint . See addressint . See also pointer types and the ^ operator (which accesses objects located by pointers).		
See also	<u>cheat</u> . See also <i><u>explicitIntegerConstant</u></i> (which explains how to write hexadecimal constants, which are often used for addresses).		

infix

	(a)	+ % string catenation	% Integer and real addition; set union;
	(b)	-	% Integer and real subtraction; set difference
	(c)	*	% Integer and real multiplication; set intersection
	(d)	/	% Real division
	(e)	div	% Truncating integer division
	(f)	mod	% Modulo
	(g)	rem	% Remainder
	(h)	**	% Integer and real exponentiation
	(i)	<	% Less than
Syntax	(j)	>	% Greater than
	(k)	=	% Equal
	(l)	<=	% Less than or equal; subset
	(m)	>=	% Greater than or equal; superset
	(n)	not=	% Not equal
	(0)	and	% And (boolean conjunction)

An *infixOperator* is one of:

(p)	or	% Or (boolean
		disjunction)
(q)	=>	% Boolean implication
(r)	in	% Member of set
(s)	not in	% Not member of set
(t)	shr	% Shift right
(u)	shl	% Shift left
(v)	xor	% Exclusive OR

An *infix operator* is placed between two values or *operands* to produce a third value. For example, the result of 5 + 7 is 12. In some cases the meaning of the operator is determined by its operands. For example, in "pine" + "apple", the + operator means string catenation while in 5 + 7 it means integer addition. There are also *prefix operators* (-, + and **not**), which are placed in front of a single value. See *prefix operator*.

In expressions with several operators, such as 3 + 4 * 5, the *precedence* rules determine the order in which the operation is done (see *precedence* for a listing of these rules). In this example, the multiplication is done before the addition, so the expression is equivalent to 3 + (4 * 5).

The numerical (integer or real) operators are +, -, *, /, **div**, **mod**, and **. All of these except **div** produce a **real** result when at least one of their operands is **real**. If both operands are integers, the result is an integer except in the case of **real** division (/) which always produces a **real** result regardless of the operands.

The **div** operator is like **real** division (/), except that it always produces an integer result, truncating any fraction to produce the nearest integer in the direction of zero.

The **mod** operator is the *modulo* and the **rem** operator is the *remainder*. The sign of the result of **mod** operator is the same as the sign of the second operand. The **rem** operator operates like the **mod** operator in Turing (and in most other languages). It

Description produces the remainder, which is the difference between **real** division (/) and integer division (**div**). When both operands are positive, this is the *modulo*. For example, 14 **mod** 10 is 4. If one of the operands is negative, a negative answer may result, for example, 7 **mod** 2 is 1. See also the **int** and **real** types.

The comparison operators (<, >, =, <=, >=, **not=**) can be applied to numbers as well as to enumerated types. They can also be applied to strings to determine the *ordering* between strings (see **string** type for details). Arrays, records, unions and collections cannot be compared. Boolean values (**true** and **false**) can be compared only for equality (= and **not=**); the same applies to **pointer** values. Set values can be compared using <= and >=, which are the subset and superset operators. The **not=** operator can be written as ~=.

Strings are manipulated using catenation (+) as well as substring expressions (see *substring*) and the **index** function (see **index**). See also the **string** type.

The operators to combine true/false values are **and**, **or**, and => (implication), as well as equality (= and **not=**). See also the **boolean** type.

The set operators are union (+), intersection (*), set difference (-), subset (<=), superset (>=), and membership (**in** and **not in**). See also the **set** type.

The **shr** (shift right), **shl** (shift left) and **xor** (exclusive OR) operators accept and produce natural numbers. See **shr**, **shl**, and **xor**.

inherit

inheritance clause

An inheritClause is:

Syntax inherit inheritItem

DescriptionAn **inherit** clause specifies that the class containing the clause is to
expansion of another class. This expansion is called *inheritance*. The
containing the clause gains access to (inherits) all the declarations is
the target item. Expansions are used to add new declarations and examples and to support *polymorphism* (overriding subprograms).

Here is an example of a stack class. Following it, we show another called *stackWithDepth*, that inherits *stack* by adding a function call *depth*.

class stack export push, pop var top : int := 0 var contents : array 1 .. 100 of string procedure push (s : string) top := top + 1 contents (top) := s end push procedure pop (var s : string) s := contents (top) top := top - 1 end pop end stack

Next comes an expansion, which inherits the internal declarations (stack class and adds the *depth* function.

class stackWithDepth
 inherit stack
 export depth
 function depth : int
 result top
 end push
end stackWithDepth

Example

Objects of the inherited class *stackWithDepth* are like objects of the class *stack*, except there is an additional exported function named *c*

An inheritItem is one of:

(a) *id*(b) *id* **in** *fileName*

The second form is used when the inherit clause is for a separate **u** the imported item is in a file whose name is different from the item for example:

inherit ledger in "newledg.t"

The *fileName* must be an explicit character string, e.g., "*newledg.t*" Parentheses are allowed around the item in an inherit clause, as in:

inherit (ledger in "newledg.t")

There is a special form of **inherit** clause, called an **implement clau** is used to separate an interface from an implementation. Modules a monitors, as well as classes, use these clauses. See **implement** clau **implement by** clause.

If class *D* inherits class *C*, we say that *C* is the *parent* and *D* is the *C* class *B* is said to be an *ancestor* of class *D* (and *D* is the *descendar* if *B* and *D* are the same class, or if *B* is the parent of *D*, or if *B* is th of the parent of *D*, etc. We write this as follows:

B <= D % B is an ancestor of D

If *B* is an ancestor of *D* but not the same as *D*, we say *B* is a *strict* ϵ of *D*. We write this as:

B < D % B is a strict ancestor of D

We also use the notations $D \ge B$, $D \ge B$ and D = B with the obvior meanings. All of these notations can be used in a program. Their m is in conjunction with **objectclass**, which determines the class of an located by a pointer. For example, if p is declared to be a pointer to we can write the following to see if p currently locates an object w

Details

depth operation:

% Does the object located by p have the depth op if stackWithDepth <= objectclass(p) then</pre>

A pointer that locates an object created as class E can be assigned t pointer to class B, only if B is an ancestor of E. For example, a point an object that is a *stackWithDepth* can be assigned to a pointer to *st* not vice versa. The pointer **nil** can be assigned to any pointer varial the value **nil**(C) can only be assigned to a pointer to an ancestor of

An object (located by a pointer) can be assigned to another object c they were created as objects of the same class. However, assignmen objects that are monitors or that contain dynamic arrays or collectic not allowed.

Circular (recursive) inherits are not allowed. For example, if unit *B A* then *A* cannot inherit *B*. Only one item is allowed in an inherit cl other words, Turing supports *single* inheritance but not *multiple* inheritance.

See **implement** clause for a special kind of expansion that separate module, monitor or class' interface from its implementation. See **cl** an example of polymorphism, in which an inheriting class override subprograms of its parent class.

The initialization of a module, a monitor or an object is immediatel preceded by the initialization of the item that it inherits or impleme any). Correspondingly, if the item has an **implement by** clause, the implementation is initialized immediately after the initialization of current item.

Within a class *C*, with ancestor *B*, you can force a call to exported subprogram *p* using the form *C*.*p* (or *B*.*p*). This calls the subprogra declared in *C* (or in *B* in the case of *B*.*p*), regardless of the actual cl the object and any overriding of *p*. This is similar to the notation *C* the C++ language. This notation can only be used inside class *C*.

<u>unit</u>, <u>module</u>, <u>monitor</u> and <u>class</u>. See also <u>export</u> list, <u>import</u> list, <u>implement</u> clause, <u>implement by</u> clause and <u>deferred</u> subprogram

See also

also <u>objectclass</u>.

init	array initialization			
Syntax	init			
Description	The init (initialization) keyword is used for two different purposes in Turing. The most common is for initializing arrays, records and unions. The less common is for recording parameter values in subprograms for later use in post conditions.			
Example	<pre>var mensNames : array 1 3 of string := init ("Tom", "Dick", "Harry") put mensNames (2) % This outputs C var names : array 1 2, 1 3 of string := init ("Tom", "Dick", "Harry",</pre>			
Details	The order of initializing values for multi-dimensional arrays is based on varying the right subscripts (indexes) most rapidly. This is called <i>row major order</i> . Initialization of records and unions is analogous to initializing arrays. Values are listed in the order in which they appear in the type. See array , record , and union types.			
Example	This procedure is supposed to set integer variable <i>i</i> to an integer approximation of its square root. The init clause records the initial value of <i>i</i> as <i>j</i> so it can be used in the post condition to make sure that the approximation is sufficiently accurate. The name <i>j</i> can be used only in the post condition and nowhere else in the procedure.			
	<pre>procedure intSqrt (var i : int) pre i >= 0 init j := i post abs (i - sqrt (j)) <= 1 statements to approximate square root end intSqrt</pre>			
See also	pre and post assertions and procedure and process declarations.			

Input

Description	This unit contains the predefined procedures that deal with handling input on a character-by-character basis.			
	All routines in the Input module are exported unqualified. (This means you can call the entry points directly.)			
Entry Points		<u>getch</u>	Gets the next character in the keyboard buffer (procedure with a string (1) argument).	
	T .	<u>hasch</u>	Returns true if there are characters waiting in the keyboard buffer.	
	5	<u>getchar</u>	Gets the next character in the keyboard buffer (function returning a char).	
	Pause	Waits for a key to be pressed.		
		<u>KeyDown</u>	Detect which keys are currently pressed.	
		<u>Flush</u>	Empty the keyboard buffer.	

Input.getch

Syntax	getch (var <i>ch</i> : string (1))			
Description	The getch procedure is used to input a single character without waiting for the end of a line. The parameter <i>ch</i> is set to the next character in the keyboard buffer (the oldest not-yet-read character).			
	This program contains a procedure called <i>getKey</i> which causes the program to wait until a key is pressed.			
	<pre>View.Set ("graphics")</pre>			
Example	<pre>procedure getKey var ch : string (1) getch (ch) end getKey</pre>			
	<pre>for i : 1 1000 put i : 4, " Pause till a key is pressed" getKey end for</pre>			
	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. See the View.Set procedure for details. If the screen is not in one of these modes, it will automatically be set to " <i>screen</i> " mode.			
Details	Some keys, such as the left arrow key, insert key, delete key, and function keys do not produce ordinary character values. These keystrokes are returned by getch as special values. See Appendix D for these values.			
Status	Exported unqualified.			
	This means that you can call the function by calling getch or by calling Input.getch .			
See also	hasch (has character) procedure which is used to see if a character has been typed but not yet read .			

Input.getchar

Syntax getchar : char

Description The **getchar** function is used to input a single character without waiting for the end of a line. The next character in the keyboard buffer (the oldest not-yet-read character) is returned.

This program contains a procedure called *getKey* which causes the program to wait until a key is pressed.

View.Set ("graphics")
procedure getKey
var ch : char
ch := getchar
end getKey
for i : 1 .. 1000
put i : 4, " Pause till a key is pressed"
getKey
end for

The screen should be in a "*screen*" or "*graphics*" mode. See the **View.Set** procedure for details. If the screen is not in one of these modes, it will automatically be set to "*screen*" mode.

Details Some keys, such as the left arrow key, insert key, delete key, and function keys do not produce ordinary character values. These keystrokes are returned by **getch** as special values. See Appendix D for these values.

Exported unqualified.

Status This means that you can call the function by calling **getchar** or by calling **Input.getchar**.

hasch (has character) procedure which is used to see if a character has been typed but not yet read.

Input.hasch

Syntax	hasch : boolean	
Description	The hasch procedure is used to determine if there is a character that has been typed but not yet been read.	
	The <i>flush</i> procedure gets rid of any characters that have been typed but not yet read.	
Example	<pre>procedure flush var ch : string (1) loop exit when not hasch getch (ch) % Discard this character end loop end flush</pre>	
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. See the View.Set procedure for details. If the screen is not in one of these modes, it will automatically be set to " <i>screen</i> " mode.	
	Exported unqualified.	
Status	This means that you can call the function by calling hasch or by calling Input.hasch .	
See also	getch and getchar.	

Input.Flush

Syntax Input.Flush

DescriptionThe **Input.Flush** procedure empties the keyboard buffer. It is
often used to avoid accidentally reading multiple keystrokes
because the user pressed a key for too long, causing autorepeat.

This program echoes input from the keyboard at a rate of 10 characters per second. If you hold down a key, the echoing will stop as soon as you stop pressing the key. Without the **Input.Flush**, the program continues to echo many more keystrokes that have accumulated in the keyboard buffer.

Example

loop if hasch then put getchar .. Input.Flush delay (100) end if end loop

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Input.Flush, not by calling Flush.

Input.KeyDown

Syntax	Input.KeyDown (var <i>chars</i> : array char of boolean)
Description	The Input.Keydown procedure allows a program to read which keys are currently being pressed. This procedure is different from getch in that it allows a program to detect when a user presses and releases a button. As such, it is not to be used in conjunction with getch .
	The procedure can be used in games where an action takes place while a key is depressed.
	Determine if the T key is pressed. Note that we check for the lower case letter.
Example	<pre>var chars : array char of boolean Input.KeyDown (chars) if chars ('t') then put "The T key is pressed" end if</pre>
Details	The array returned is a list of all the characters. A key is currently pressed if the array element with the corresponding character is true . For example, the <i>a</i> key is pressed is <i>chars</i> ('a') is true . Note that each key is individually represented, so if the user has pressed Shift+a to get a 'A', then Input.KeyDown would register Shift and 'a' as pressed, but not 'A'.
	A full list of all the possible characters that can be set with Input.KeyDown can be found in the Key Codes appendix.
Details	The array returned is a list of all the characters. A key is currently pressed if the array element with the corresponding character is true. For example, the <i>a</i> key is pressed is <i>chars</i> ('a') is true . Note that each key is individually represented, so if the user has pressed Shift+a to get a 'A', then Input.KeyDown would register Shift and 'a' as pressed, but not 'A'.

DetailsThe number of keys that can be pressed simultaneously is
hardware dependent. Most keyboards can detect a minimum of
two keys + Shift + Control + Alt. This means that if you are
designing a two player game to be played at a single keyboard
and you wish to make certain that players cannot "hog the
keyboard" by holding down keys, you should not use more than 6
different keys and three of them should be the Shift, Control and
Alt keys.

The program reads the keyboard and displays a message while the arrow keys are pressed. It can detect up to all four arrow keys pressed at once.

```
var chars : array char of boolean
loop
    Input.KeyDown (chars)
    locate (1, 1)
    if chars (KEY_UP_ARROW) then
        put "Up Arrow Pressed " ...
    else
                                " . .
        put "
    end if
    if chars (KEY_RIGHT_ARROW) then
        put "Right Arrow Pressed " ...
    else
                                   "..
        put "
    end if
    if chars (KEY_LEFT_ARROW) then
        put "Left Arrow Pressed " ..
    else
        put "
                                  н
                                    . .
    end if
    if chars (KEY_DOWN_ARROW) then
        put "Down Arrow Pressed " ...
    else
                                "..
        put "
    end if
end loop
```

Example

Execute

Another example is available that checks for all possible keys.

Example

Exported qualified.

StatusThis means that you can only call the function by calling
Input.KeyDown, not by calling KeyDown.

Input.Pause

Syntax Input.Pause

The **Input.Pause** procedure simply waits for a key to be pressed and then returns. It echoes the key pressed if **echo** mode is set. (See **View.Set** for setting **echo** mode)

Description

This subprogram helps avoid having to declare a variable declaration and then make a call to **getch**or **getchar**.

This program pauses after every name read from the file.

var f : int var line : string open : f, "data.txt", get loop exit when eof (f) get : f, line : * put name Input.Pause end loop close : f

Example

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Input.Pause, not by calling Pause.

Syntax	int	
Description	The int (integer) type has the values 2, 1, 0, 1, 2 Integers can various operators such as addition (+) and multiplication (*). Integer combined with real numbers, in which case the result is generally a integer can always be assigned to a real variable, with implicit com-	
Example	<pre>var counter, i : int var j : int := 9 var tax := 0 % The type is implicitly int</pre>	
	See also <i>explicitIntegerConstant</i> . The real type is used instead of i have fractional parts as in 16.837. See the real type for details.	
	The operators on integers are +, -, * (multiply), div (truncating inte (integer remainder), ** (exponentiation), as well as comparisons (+ <=). The operators and , or and xor can be applied to non-negative The bit-wise boolean result is produced as an integer (actually, as a The shr (shift right) and shl (shift left) operators are also introduce	
	Real numbers can be converted to integers using ceil (ceiling), floo descriptions of these functions). Integers can be converted to real n intreal , but in practice this is rarely used, because an integer value real value will be automatically converted to real.	
Details	Integers can be converted to strings and back using intstr and strin converted to corresponding ASCII (or EBCDIC) characters using c the descriptions of these functions.	
	Pseudo-random sequences of integers can be generated using rand	
	In current implementations of Turing, the range of integers is from 2147483647. In other words, the maximum size of integer is 2**31 This range exists because integers are stored in 4 bytes. The remain value, -2147483648 records uninitialization. The types int1 , int2 a integers that fit into 1, 2 or 4 bytes. The int <i>n</i> types (int1 , int2 and i	

int

checked for initialization and allow all their bit patterns as number:

The natural number type **nat** allows only the non-negative values: number values can be used whenever integer values are expected a given that the value does not exceed the range of the expected type

See also nat and **int***n*.

n-byte integer type

Dirty

	(a)	int1	% 1-byte integer
Syntax	(b)	int2	% 2-byte integer
	(c)	int4	% 4-byte integer
Description	occupy a s principle a overflows	specified number a machine-indepe	ypes are machine-dependent types that c of bytes. By contrast, the int type is in endent and mathematical type (it the value is too large or small, that is, t into 4 bytes).
Example	١	var counter2 :	int1 % Range is -128 127 int2 % Range is -32768 32767 int4 % Range is -2147483648 21
Details	In current implementations of Turing, the range of the int is -2147483647 to 2147483647, which means that the int4 type allows one more value, -2147483648. This extra value is used in int to represent the state of being initialized. The int <i>n</i> types allow use of all possible values that fit into <i>n</i> bytes and thereby cannot check for initialization.		
	The int <i>n</i> types are like the C language types short int, int, and long int, except that the number of bytes occupied by the C types depends on the particular C compiler.		
See also	the <u>nat</u> n ty also <u>int</u> ar		byte natural (non-negative) values. See

int*n*

intreal

integer-to-real function

Syntax	<pre>intreal (i : int) : real</pre>
Description	The intreal function is used to convert an integer to a real number. This function is rarely used, because in Turing, an integer value can be used where ever a real value is required. When the integer value is used where a real value is required, the intreal function is implicitly called to do the conversion from int to real .
See also	floor, ceil and round functions.

intstr	integer-to-string function
Syntax	<pre>intstr (i : int [, width : int [, base : int]]) : string</pre>
	The intstr function is used to convert an integer to a string. The string is equivalent to <i>i</i> , padded on the left with blanks as necessary to a length of <i>width</i> , written in the given number <i>base</i> . For example, intstr $(14, 4, 10)="bb14"$ where <i>b</i> represents a blank. The <i>width</i> and <i>base</i> parameters are both optional. If they are omitted, the string is made just long enough to hold the value, and the number base is 10. For example, intstr $(14, 4) = "bb14"$ and intstr $(23) = "23"$.
	The <i>width</i> parameter must be non-negative. If <i>width</i> is not large enough to represent the value of <i>i</i> , the length is automatically increased as needed.
	The string returned by intstr is of the form:
Description	{blank}[-]digit{digits}
	where {blank} means zero or more blanks, [-] means an optional minus sign, and digit{digit} means one or more digits. The leftmost digit is either non-zero or else a single zero digit. In other words, leading zeros are suppressed.
	The letters A, B, C are used to represent the digit values 10, 11, 12, The <i>base</i> must be in the range 2 to 36 (36 because there are ten digits and 26 letters). For example, intstr (255, 0, 16) = "FF".
	The intstr function is the inverse of strint , so for any integer <i>i</i> ,
	<pre>strint (intstr (i)) = i.</pre>
See also	chr , ord and strint functions. See also the natstr and strnat functions. See also <i>explicitIntegerConstants</i> for the way to write non base 10 values in a program .

invariant

Syntax

assertion

An invariantAssertion is:

invariant trueFalseExpn

An **invariant** assertion is a special form of an **assert** statement that is used only in **loop** and **for** statements and in modules, monitors, and classes. It is used to make sure that a specific requirement is met. This requirement is given by the

Description *trueFalseExpn*. The *trueFalseExpn* is evaluated. If it is true, all is well and execution continues. If it is false, execution is terminated with an appropriate message. See **assert**, **loop** and **for** statements and the **module** declarations for more details.

This program uses an invariant in a **for** loop. The invariant uses the function *nameInList* to specify that a key has not yet been found in an array of names.

```
var name : array 1 .. 100 of string
                    var key : string
                    ... input name and key ...
                    function nameInList ( n : int) : boolean
                         for i : 1 . . n
                             if key = name ( i ) then
                                 result true
Example
                             end if
                         end for
                         result false
                    end nameInList
                    for j : 1 ... 100
                         invariant not nameInList ( j - 1)
                         if key = name ( j) then
                             put "Found name at ", j
                             exit
                         end if
                    end loop
```

Joystick

Description	This unit contains the predefined subprogram that deal with reading the joystick in a Turing program. The routines allow you to get the current joystick position and whether either one of the two buttons are pressed.		
	All routines in the Joystick module are exported qualified (and thus must be prefaced with " Joystick. "). All the constants are exported unqualified and thus do not need the Joystick prefix.		
Entry Points	joystick1, joystick2 <u>GetInfo</u>	joystick name contants (exported unqualified) Reads the current value of a joystick and status of the joystick buttons.	

Joystick.GetInfo

Syntax	Joystick.GetInfo (joystick : int, var xPos, yPos : int, btn1Pressed, btn2Pressed : boolean)	
	Reads the position and button status of the joystick specified by the parameter. The <i>x</i> and <i>y</i> parameter are returned in the <i>xPos</i> and <i>yPos</i> . If button 1 or button 2 on the joystick are currently pressed, <i>btn1Pr btn2Pressed</i> will be set to true . The <i>joystick</i> parameter can be eithe or joystick2 .	
Description	The x and y positions vary from joyMin to joyMax . To use them v to a screen, the coordinates returned from Joystick.GetInfo must t into screen coordinates. The following formula can be used:	
	screenX = round (maxx * (xPos joyMin) / (joyMax screenY = round (maxy * (yPos joyMin) / (joyMax	
Details	The Joystick module contains undocumented subprograms for the to access more than two buttons or axes on a joystick. Contact Holy you need more information.	
	The following program outputs the current location of joystick #1 a cursor on the screen to point out where it is showing.	
	<pre>var jx, jy, x, y, ox, oy : int := 1 var b1, b2, oB1, oB2 : boolean := false loop</pre>	
Example	Joystick.GetInfo (joystick1, jx, jy, b1, b2) % Convert joystick coordinates into screen c x = round (maxx * (jx joyMin) / (joyMax jc y = round (maxy * (jy joyMin) / (joyMax jc if x not= ox or y not= oy or b1 not= oB1 or Text.Locate (1, 1) put "x = ", x, " y = ", y, " b1 = ", k View.Set ("xor") Draw.Line (ox 10, oy, ox + 10, oy, brig Draw.Line (ox , oy 10, ox , oy + 10, br Draw.Line (x 10, y, x + 10, y, brightre Draw.Line (x, y 10, x, y + 10, brightre ox := x oy := y	

Exported qualified.

StatusThis means that you can only call the function by calling Joystick.
by calling GetInfo.

Keyboard

This unit contains all the constants representing both the characters and their ordinal values for all possible keystrokes that can be read by Turing.

Constants beginning with "KEY_" are **char** values. These are the values returned by **getch** and used as an index into the **Input.KeyDown** array. The constants that the constants KEY_KEYPAD_5, KEY_SHIFT, KEY_CTRL and KEY_ALT can only be used in conjunction with **Input.KeyDown**. The **getch** and **getchar** subprograms do not return these values. All other constants with with SHIFT, CTRL and ALT as part of the name cannot be checked for in the **Input.KeyDown** array.

Constants beginning with "ORD_" KEY_H are **int** values and represent the ordinal values (values returned by the **ord** function) for all the possible keystrokes.

Here is a list of most of the constants

KEY_F1	KEY_CTRL_A
KEY_F12	 KEY CTRL Z
KEY_SHIFT_F1	KEY ALT A
	KEY_ALT_Z

Description

KEY_SHIFT_F12 KEY_CTRL_F1 .. KEY_CTRL_F12 KEY_ALT_0

KEY_ALT_F1 .. KEY_ALT_F12

KEY_UP_ARROW	KEY_C
KEY_PGUP	KEY_C
KEY_LEFT_ARROW	KEY_C
KEY_RIGHT_ARROW	KEY_C
KEY_END	KEY_C
KEY_DOWN_ARROW	KEY_C
KEY_PGDN	KEY_C
KEY_INSERT	KEY_C
KEY_DELETE	KEY_C
KEY_BACKSPACE	KEY_K
KEY_TAB	KEY_SI
KEY_ENTER	KEY_C
KEY_ESC	KEY_A
KEY_CTRL_OPEN_BRACKET	KEY_C
KEY_CTRL_CLOSE_BRACKET	KEY_C
KEY_CTRL_UNDERSCORE	KEY_C
KEY_ALT_MINUS	KEY_A
KEY_BACK_TAB	KEY_SI

Above constants with "ORD_" instead of "KEY_"

ORD_A ORD_Z	ORD_0 ORD_9
ORD_LOWER_A ORD_LOWER_Z	
ORD_SPACE	ORD_EXCALAMATION_MARK
ORD_QUOTATION_MARK	ORD_HAS_MARK
ORD_DOLLAR_SIGN	ORD_PERCENT_SIGN

- ORD_AMPERSAND ORD_OPEN_PARENTHESIS ORD_ASTERISK ORD_COMMA ORD_PERIOD ORD_SLASH ORD_SEMICOLON ORD_EQUALS ORD_QUESTION_MARK ORD_OPEN_BRACKET ORD_CLOSE_BRACKET ORD_OPEN_BRACE ORD_OPEN_BRACE
- ORD_SINGLE_QUOTE ORD_CLOSE_PARENTHESIS ORD_PLUS ORD_MINUS ORD_DOT ORD_COLON ORD_COLON ORD_LESS_THAN ORD_GREATER_THAN ORD_AT_SIGN ORD_BACKSLASH ORD_CARET ORD_CARET ORD_APOSTROPHE ORD_BAR ORD_TILDE

Characters marked with an asterisk (*) are keys that can only be detected using **Input.KeyDown**.

All constants in the Keyboard module are exported unqualified. (This means you can use the constants directly without having to use the qualifier "**Keyboard.**".)

See also getch, getchar, and Input.KeyDown.

length

Syntax	length (s : string) : int	
Description	The length function returns the number of characters in the string. The string must be initialized. For example, length ("table") is 5.	
	This program inputs three words and outputs their lengths.	
Example	<pre>var word : string for i : 1 3 get word put length (word) end for If the words are "cat", "robin" and "crow", the program will</pre>	
	output 3, 5 and 4.	
Details	The length function gives the current length of the string. To find the maximum length of a string, use upper . For example, given the declaration var s : string (10), upper (<i>s</i>) returns 10.	
See also	<u>upper</u> .	

Limits

Entry Points This unit contains constants and functions used in determining the mathematical accuracy of the language.

Description All routines in the Limits module are exported qualified (and thus must be prefaced with "**Limits.**") except **maxint, maxnat, minint** and **minnat**, which are exported unqualified (this means you can call those entry points directly).

DefaultFW	Default fraction width used in printing using the "put" statement.
DefaultEW	Default exponent width used in printing using the "put" statement.
<u>minint</u>	The minimum integer in Turing (exported unqualified).
maxint	The maximum integer in Turing (exported unqualified).
<u>minnat</u>	The minimum natural number in Turing (exported unqualified).
maxnat	The maximum natural number in Turing (exported unqualified).
Real number 0	s are represented in Turing as: f * (radix ** e) or

```
where for non-zero f:
(1 / radix) <= abs (f) and abs (f) < 1.0
minexp <= e and e <= maxexp.
```

Radix	The "radix" (usually 2).
NumDigits	The number of radix digits in f.
MinExp	"minexp" (the smallest exponent allowed).
MaxExp	"maxexp" (the largest exponent allowed).
GetExp	Function that returns the value of "e".
SetExp	Procedure that sets the value of "e".

Rreb The relative round-off error bound.

Syntax	In (<i>r</i> : real) : real
Description	The In function is used to find the natural logarithm (base e) of a number. For example, In (1) is 0.
	This program prints out the logarithms of 1, 2, 3, up to 100.
Example	for i : 1 100 put "Logarithm of ", i, " is ", ln (i) end for
Details	See also the exp (exponential) function. You cannot take the logarithm of zero or a negative number.
Note	$\log n(i) = \ln (i) / \ln (n)$
	exp (the exponentiation function).
See also	See also predefined unit <u>Math</u> .

ln

locate (*row*, *column* : **int**) **Syntax** The **locate** procedure is used to move the cursor so that the next output from **put** will be at the given row and column. Row 1 is Description the top of the screen and column 1 is the left side of the screen. This program outputs stars of random colors to random locations

on the screen. The variable *colr* is purposely spelled differently from the word *color* to avoid the procedure of that name (used to set the color of output). The row number is purposely chosen so that it is one less than **maxrow**. This avoids the scrolling of the screen which occurs when a character is placed in the last column of the last row.

setscreen ("screen")

Example

var row, column, colr : int **loop** randint (row, 1, maxrow - 1) randint (column, 1, maxcol) randint (colr, 0, maxcolor) color (colr) **locate** (row, column) put "*"`.. % Use dot-dot to avoid clearing end loop

The **locate** procedure is used to locate the next output based on row and column positions. See also the **locatexy** procedure which is used to locate the output based x and y positions, where x=0, v=0 is the left bottom of the screen.

Details

The screen should be in a "screen" or "graphics" mode. See the **setscreen** procedure for details. If the screen is not in one of these modes, it will automatically be set to"screen" mode.

setscreen and drawdot.

See also See also predefined unit **Text**.

locate

procedure

locatexy

graphics procedure

Syntax	locatexy (x , y : int)	
Description	The locatexy procedure is used to move the cursor so that the next output from put will be at approximately (x, y) . The exact location may be somewhat to the left of x and below y to force alignment to a character boundary.	
	This program outputs "Hello" starting at approximately (100, 50) on the screen.	
Example	setscreen ("graphics") locatexy (100, 50) put "Hello"	
Details	The locatexy procedure is used to locate the next output based on x and y positions, where the position $x=0$, $y=0$ is the left bottom of the screen. See also the locate procedure which is used to locate the output-based row and column positions, where row 1 is the top row and column 1 is the left column.	
	The screen should be in a " <i>graphics</i> " mode. See the setscreen procedure for details. If the screen is not in a " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode.	
See also	setscreen and drawdot.	
	See also predefined unit <u>Text</u> .	

statement

A loopStatement is:

Syntax	loop statementsAndDeclarations end loop
Description	A loop statement causes the statements (and declarations) in it to be repeatedly executed. This continues until terminated by one of its enclosed exit statements (or by an enclosed return or result statement).
	Output on separate lines: Happy, Happy, Happy, etc.
Example	loop put "Happy" end loop
	Read words up to the word Stop.
Example	<pre>var word : string loop get word exit when word = "Stop" end loop</pre>
	A loop statement can contain more than one exit , or none at all (in which case it is an infinite loop). When the exit when is at the beginning of the loop, the loop works like Pascal's do while ; when at the end, the loop works like Pascal's repeat until .
Details	Just preceding the statements and declarations, you are allowed to write an "invariant clause" of the form:
	invariant trueFalseExpn
	This clause is equivalent to: assert <i>trueFalseExpn</i> .

loop

lower

Syntax	lower (reference [, dimension]): int
Description	The lower attribute is used to find the lower bound of an array, string, char (n) or non-opaque subrange type. Since the lower bound is necessarily known at compile time, lower is rarely used.
See also	<u>upper</u> which finds the <u>upper</u> bound.

Math

This unit contains all the mathematical routines. There are three routines that are part of the language, but are conceptually part of the **Math** unit.

All routines in the **Math** unit except **Math.Distance** and **Math.DistancePointLine** are exported unqualified. (This means you can call the entry points directly.) **Math.Distance** and **Math.DistancePointLine** are exported qualified. All constants in the **Math** unit are exported qualified. (Both **Math.PI** and **Math.E** must be prefaced with "**Math.**".)

Descriptions of all the subprograms in the **Math** module can be found in this chapter.

	Math.PI	The constant p (3.14).
Constants	Math.E	The natural base <i>e</i> (2.718).

	<u>abs</u> *	The absolute value function.
	<u>arccos</u>	The arccos function (radians).
	<u>arccosd</u>	The arccos function (degrees).
	<u>arcsin</u>	The arcsin function (radians).
	<u>arcsind</u>	The arcsin function (degrees).
	<u>arctan</u>	The arctan function (radians).
	<u>arctand</u>	The arctan function (degrees).
	<u>COS</u>	The cosine function (radians).
	<u>cosd</u>	The cosine function (degrees).
Entry Points	<u>exp</u>	The exponentiation function.
	<u>ln</u>	The natural logarithm function.
	<u>max</u> *	The maximum value function.
	<u>min</u> *	The minimum value function.
	<u>sign</u>	The sign function.
	<u>sin</u>	The sine function (radians).
	<u>sind</u>	The sine function (degrees).

<u>tan</u>	The tangent function (radians).	
<u>tand</u>	The tangent function (degrees).	
<u>sqrt</u>	The square root function.	
Distance	The distance between two points.	
DistancePointLine	The distance between a point and a line segment.	

* Part of the language, conceptually part of the **Math** unit.

Math.Distance

- **Syntax** Math.Distance (*x*1, *y*1, *x*2, *y*2 : real) : real
- **Description** Math.Distance is used to calculate the distance between two points. (x1, y1) is the location of the first point, and (x2, y2) is the location of the second point.
- **Details** The parameters to **Math.Distance** are real, but you can pass integer variables and constants.
- **Example** This progam draws two circles with radius 75 on the screen and outputs whether they touch.

var x1, y1, x2, y2 : int const RADIUS : int := 75 x1 := Rand.Int (RADIUS, maxx - RADIUS) y1 := Rand.Int (RADIUS, maxy - RADIUS) x2 := Rand.Int (RADIUS, maxy - RADIUS) y2 := Rand.Int (RADIUS, maxy - RADIUS) Draw.Filloval (x1, y1, RADIUS, RADIUS, brightred Draw.Filloval (x2, y2, RADIUS, RADIUS, brightgre if Math.Distance (x1, y1, x2, y2) < RADIUS * 2 t put "The two circles touch" else put "The two circles do not touch" end if

Execute

Status Exported qualified.

This means that you can only call the function by calling **Math.Distance**, not by calling **Distance**.

Math.DistancePointLine

Part of <u>Math</u> module

Syntax	Math.DistancePointLine (<i>xp</i> , <i>yp</i> , <i>x1</i> , <i>y1</i> , <i>x2</i> , <i>y2</i> : real) : real	
Description	Math.DistancePointLine is used to calculate the distance between and a line segment. It is often used in games to determine if a collis occurred. (xp , yp) is the location between the point. ($x1$, $y1$) and ($x2$ are the end points of the line segment.	
Details	The parameters to Math.DistancePointLine are real, but you can j integer variables and constants. The parameters are in the same ord subprogram name (that is point, then line).	
Example	This progam draws a line segment and a circle with radius 75 on th and outputs whether they touch. <pre>var xp, yp, x1, y1, x2, y2 : int const RADIUS : int := 75 xp := Rand.Int (RADIUS, maxx - RADIUS) yp := Rand.Int (RADIUS, maxy - RADIUS) x1 := Rand.Int (0, maxx) y1 := Rand.Int (0, maxy) x2 := Rand.Int (0, maxy) y2 := Rand.Int (0, maxy) Draw.Filloval (xp, yp, RADIUS, RADIUS, brightred Draw.Line (x1, y1, x2, y2, brightgreen) if Math.DistancePointLine (xp, yp, x1, y1, x2, y put "The circle touches the line" else</pre>	

Execute

Status Exported qualified.

This means that you can only call the function by calling **Math.DistancePointLine**, not by calling **DistancePointLine**.

maximum function

Syntax	max (expn , expn)	
Description	The max function is used to find the maximum of two numbers (the two <i>expn</i> 's). For example, max (5, 7) is 7. If both numbers are int , the result is int . If both numbers are nat (natural numbers), the result is nat . But if one or both of the numbers are real , the result is real . See also the min function.	
Example	This program outputs 85.72. var x : real := 74.61 var y : real := 85.72 put max (x, y) % Outputs 85.72	
Example	This program inputs 10 numbers and outputs their maximum. var m, t : real get m % Input first number for i : 2 10 % Handle remaining 9 numbers get t m := max (m, t) end for put "The maximum is ", m	
See also	See also predefined unit Math.	

max

maxcol

maximum column function

Syntax	maxcol : int
Description	The maxcol function is used to determine the number of columns on the screen.
Example	This program outputs the maximum column number. <pre>put "Number of columns on the screen is ", maxrc</pre>
Details	For IBM PC compatibles as well as most UNIX dumb terminals, ir " <i>text</i> " or " <i>screen</i> " mode, maxcol = 80. For the default IBM PC compatible " <i>graphics</i> " mode (CGA), maxcol = 40.
See also	locate procedure for an example of the use of maxcol .

maxcolor

graphics function

Syntax	maxcolor : int
Description	The maxcolor function is used to determine the maximum color number for the current mode of the screen. The alternate spelling is maxcolour .
	This program outputs the maximum color number.
Example	<pre>setscreen ("graphics")</pre>
	 put "The maximum color number is ", maxcolor
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. If it is not, it will automatically be set to " <i>screen</i> " mode. See setscreen for details.
	For IBM PC compatibles in " <i>screen</i> " mode, maxcolor = 15. For the default IBM PC compatible " <i>graphics</i> " mode (CGA), maxcolor = 3.
See also	drawdot and palette for examples of the use of maxcolor . See the color procedure which is used for setting the currently active <u>color</u> .

maximum integer function

Syntax	maxint : int
Description	The maxint function is used to determine the largest integer (int) t be used in a program.
Example	This program outputs the maximum integer. <pre>put "The largest integer that can be used is ",</pre>
	In current Turing and OOT implementations, int values are stored bytes, i.e., 32 bits. This determines the maximum int value, which 2**311, equaling 2147483647.
Details	There is an anomaly in computer arithmetic in that the absolute val the largest negative integer is one larger than maxint . Turing reservextra value to represent the uninitialized integer. This value can be computed but any attempt to assign it to an int variable is detected overflow. You can use this extra value by using the int4 type instea int , but this type has no initialization checking.
	maxnat and minint.
See also	See also OOT predefined unit <u>Math</u> .

maxint

maximum natural number function

Syntax	maxnat : nat
Description	The maxnat function is used to determine the largest natural numb used in a program.
Example	This program outputs the maximum natural number.
	put "The largest natural number that can be usec
	In current implementations, natural numbers are stored in 4 bytes, i This determines the maximum natural number, which is 2**32 - 2, 4294967294.
Details	In four bytes it is possible to represent one more value, namely, 2*: 4294967295. This extra value is used in Turing to represent the uni natural number. Although it can be computed, any attempt to assign variable is detected as an overflow. You can use this extra value by type instead of nat , but this type has no initialization checking.
	maxint and minnat.
See also	See also predefined unit Limits.

maxnat

maxrow

maximum row function

Syntax	maxrow : int
Description	The maxrow function is used to determine the number of rows on the screen.
Example	This program outputs the maximum row number. <pre>put "Number of rows on the screen is ", maxrow</pre>
Details	For IBM PC compatibles, maxrow = 25. For many UNIX dumb terminals, maxrow = 24.
See also	locate procedure for an example of the use of maxrow .

graphics function

Syntax	maxx : int		
Description	The maxx function is used to determine the maximum value of x for the current graphics mode.		
	This program outputs the maximum x value.		
Example	<pre>setscreen ("graphics")</pre>		
	 put "The maximum x value is ", maxx		
Details	The screen should be in a " <i>graphics</i> " mode. If it is not, it will automatically be set to " <i>graphics</i> " mode. See setscreen for details.		
	For the default IBM PC compatible graphics mode (CGA), maxx = 319.		
See also	<u>drawdot</u> for an example of the use of $maxx$ and for a diagram illustrating x and y positions.		

maxx

graphics function

Syntax	maxy : int		
Description	The maxy function is used to determine the maximum value of y for the current graphics mode.		
	This program outputs the maximum y value.		
Example	<pre>setscreen ("graphics")</pre>		
	 put "The maximum y value is ", maxy		
Details	The screen should be in a " <i>graphics</i> " mode;. If it is not, it will automatically be set to " <i>graphics</i> " mode. See setscreen for details.		
	For the default IBM PC compatible graphics mode (CGA), maxy = 199.		
See also	drawdot for an example of the use of maxy and for a diagram illustrating x and y positions.		

maxy

minimum function

Syntax	min (expn , expn)		
Description	The min function is used to find the minimum of two numbers (the two <i>expn</i> 's). For example, min (5, 7) is 5. If both numbers are int , the result is int . If both numbers are nat (natural numbers), the result is nat . But if one or both of the numbers are real , the result is real . See also the max function.		
Example	This program outputs 74.61. var x : real := 74.61 var y : real := 85.72 put min (x, y) % Outputs 74.61		
Example	This program inputs 10 numbers and outputs their minimum. var m, t : real get m % Input first number for i : 2 10 % Handle remaining 9 num get t m := min (m, t) end for put "The minimum is ", m		
See also	See also predefined unit <u>Math</u> .		

min

minint

Syntax	minint : int		
Description	The minint function is used to determine the smallest integer (int) be used in a program.		
	This program outputs the maximum integer.		
Example	put "The smallest integer that can be used is ",		
	In current implementations, int values are stored in 4 bytes, i.e., 32 This determines the minimum int value, which is -2**31-1, equalin -2147483647.		
Details	There is an anomaly in computer arithmetic in that the absolute val the largest negative integer is one larger than maxint . Turing reservextra value to represent the uninitialized integer. This value can be computed but any attempt to assign it to an int variable is detected overflow. You can use this extra value by using the int4 type instea but this type has no initialization checking.		
	minnat and maxint.		
See also	See also predefined unit Limits.		

minnat : nat **Syntax** The **minnat** function is used to determine the smallest natural num **Description** used in a program. This program outputs the smallest natural number. Example put "The smallest natural number that can be use In current Turing and OOT implementations, natural numbers are s i.e., 32 bits. However, the minimum natural number in all impleme minnat is provided for purposes of symmetry with minint, maxim In four bytes it is possible to represent one more value, namely, 2* Details 4294967295. This extra value is used in Turing to represent the uni number. Although it can be computed, any attempt to assign it to a detected as an overflow. You can use this extra value by using the **I** of **nat**, but this type has no initialization checking. minint and maxnat. See also See also predefined unit **Limits**.

minimum natural number function

minnat

mod	modulo operator		
Syntax	mod		
Description	The mod (<i>modulo</i>) operator produces the modulo of one number with another. In other words, the result is always a number between 0 and the second operand. If both operands are positive, the result is identical to the remainder operator. For example, 7 mod 2 produces 1 and 12 mod 5 produces 3.		
	<pre>In this example, hours is the current time. It is moved back and forth by a random amount, but the final result must always be between 1 and 12 (the mod operation produces a number between 0 and 11 and then 0 becomes 12). var hours : int := 12 var hoursPassed : int put "The time is now ", hours, " o'clock" loop randint (hoursPassed, -12, 12)</pre>		
Example	<pre>exit when hoursPassed = 0 if hoursPassed < 0 then put hoursPassed, " hours before " else put hoursPassed, " hours later " end if put hours, " o'clock" hours := (hours + hoursPassed) mod 12 if hours = 0 then hours = 12 end if put " it was ", hours, " o'clock" end loop</pre>		
Details	If the second operand is positive, then the result is always non- negative. Likewise, if the second operand is negative, then the result is always non-positive. If both operands are negative, the result is the same as the remainder operator.		
See also	<u>infix</u> operators, <u>precedence</u> of operators and the <u>rem</u> and <u>div</u> operators.		

module

declaration

A moduleDeclaration is:

end id

A module declaration creates a package of variables, constants, typ subprograms, etc. The name of the module (*id*) is given in two plac **module** and just after **end**. Items declared inside the module can be outside of the module only if they are exported. Items from outside that are to be used in the module need to be imported (unless they a predefined or pervasive).

This module implements a stack of strings.

module stack% Implements a LIFO list
export push, popvar top : int := 0
var contents : array 1 .. 100 of stringprocedure push (s : string)
top := top + 1
contents (top) := s
end pushExampleprocedure pop (var s : string)
s := contents (top)
top := top - 1
end pop
end stack

Syntax

```
stack . push ( "Harvey" )
var name : string
stack . pop ( name ) % This sets name to
```

Outside of the *stack* module, the procedures *push* and *pop* can be c the notation *stack.push* and *stack.pop*. This access is allowed becau *pop* are *exported* from the module. Other items declared in the moc *contents*) cannot be accessed from outside because they are not exp

In some other programming languages, a module is called a *packag object*.

A module declaration is executed (it is initialized) by executing its and statements. For example, the *stack* module is initialized by sett variable to 0. This initialization executes all the statements and dec the module that are not contained in procedures or functions. The i is completed before any procedure or function of the module can b outside the module. An exported subprogram must not be called ur initialization of the module is complete.

A call to an exported procedure or function from outside the modul the body of that procedure or function (the module is *not* initialized such call). See also **monitor** and **class** declarations.

The **import** list gives the names of items declared outside the mode be accessed inside the module. Since *stack* has no **import** list, it is to access any names declared outside of it. See also **import** lists. See compiled units that are imported are initialized before the importin

The **export** list is used to implement *information hiding*, which iso implementation details inside the module. The **export** list gives the items declared inside the module that can be used outside the module example, *push* and *pop* are exported from *stack*. Each such use of *a* item must be preceded by the module name and a dot, for example, (See **unqualified** for advice on how to avoid using the prefix "*stac* that are not exported, such as *top* and *contents*, cannot be accessed module.

Details

Procedures, functions, variables, constants and types can be export monitors or classes cannot be exported. A class is essentially a template for creating individual modules (ol **class** for details. A **monitor** is essentially a module in which only c can be active at a time. See **monitor** and **process** for details.

The **opaque** keyword is used (only) in export lists to precede exponent names that have declarations in the module. Outside of the module will be distinct from all others types. This means, for example, that opaque type is a record, its fields cannot be accessed outside of the Opaque types are used to guarantee that certain items are inspected manipulated in only one place, namely, inside the module. These ty sometimes called *abstract data types*. See also **export** lists, which describes **unqualified** and **pervasive** exports.

Implement and i**mplement-by** lists are used to separate a module's from its body. This allows only a part of a module (its interface) to its users (its importers), while hiding its implementation. See **imple implement by** lists.

Use an **opaque** type to implement complex arithmetic.

Example

```
module complex
    export opaque value, constant, add,
                    ... other operations ...
    type value :
        record
            realPt, imagPt : real
        end record
    function constant (realPt, imagPt: real ) :
        var answer : value
        answer . realPt := realPt
        answer . imagPt := imagPt
        result answer
    end constant
    function add (L, R : value ) : value
        var answer : value
        answer . realPt := L . realPt + R . real
        answer . imagPt := L . imagPt + R . imag
        result answer
    end add
```

... other operations for complex arithmetic go her **end** complex

var c,d : complex .value :=complex.constant (1, % c and d become the complex number (1,5 var e : complex .value := complex.add (c, d) % e becomes the complex number (2,10)

Module declarations can be nested inside other modules but cannot inside procedures or functions. A module must not contain a **bind** a (outermost) declarations. A **return** statement cannot be used as one (outermost) statements in a module.

The syntax of a *moduleDeclaration* presented above has been simpleaving out **pre**, **invariant** and **post** clauses; the full syntax is:

```
module id
  [ implement implementItem ]
  [ implement by implementByItem ]
  [ import [ var ] importItem {, [ var ] impor
  [ export [ howExport ] id {, [ howExport ] i
  [ pre trueFalseExpn ]
  statementsAndDeclarations
  [ invariant trueFalseExpn ]
  statementsAndDeclarations
  [ post trueFalseExpn ]
end id
```

The true/false expression in the **pre** and **post** clauses must be true v initialization reaches each of them. After that, these have no effect. true/false expression in the **invariant** must be true any time the mc exited (when finishing initialization or when returning from an externa exported subprogram) or called (via an exported subprogram). I (**pre,post** and **invariant**) are not inherited by expansions. For exan module *B* inherits *A*, the subprograms of *B* are bound by *B*'s clause *A*'s.

unit, monitor and class. See also export list, import list, implement
 See also implement by list, inherit list and deferred subprogram.

Details

monitor

declaration

A monitor Declaration is:

monitor id [**implement** *implementItem*] [**implement by** *implementByItem*] [**import** [**var**] *importItem* {, [**var**] *importItem* }] [**export** [*howExport*] *id* {,[*howExport*] *id* }] *statementsAndDeclarations*

end id

A monitor is a special purpose module (see **module**) that is used with concurrent processes (see **process**). At most, one concurrent process (see **process**) can be active in a monitor at a time. This means that a process will be blocked if it calls a monitor that is Description already active. The process will not be allowed to proceed until the monitor is inactive. The monitor provides *mutually exclusive* access to the monitor's internal data.

> This monitor controls access to the *count* variable so it can be updated by two processes (the observer and the reporter) without being corrupted by this concurrent access. Generally, it is not safe to have one process update a variable that other processes are simultaneously accessing. The *observer* process repeatedly increments the *counter* when it observes an event. The *reporter* process repeatedly writes out the number of events that have occurred since the last report, resetting the *counter* to zero.

> > monitor controller export observe, report var counter : int := 0 procedure observe counter := counter + 1

Syntax

```
end observe
    procedure report (var n : int )
         n := counter
         counter := 0
    end report
end controller
process observer
    100p
         ... observe one event ...
         controller . observe
    end loop
end observer
process reporter
    var n : int
    1000
         controller.report ( n )
         ... report n events ...
    end loop
end reporter
fork observer% Activate the observerfork reporter% Activate the reporter
```

Example

A **monitor** is essentially a module in which only one process can be active at a time. See **module** declarations for details about initialization. Initialization is the same for modules and monitors.

A monitor can contain **wait** statements (that put processes to sleep) and **signal** statements (that wake them up again). These statements operate on **condition** variables, which are essentially queues of sleeping processes.

A class is essentially a template for creating individual modules (objects). See **class** for details. If the class declaration is preceded by the keyword **monitor**, the created modules are actually monitors. Monitor classes can only inherit (inherit from) other monitor classes.

The body of a monitor has the same form as that of a module, except that modules, monitors and processes cannot be declared inside monitors, and certain statements (**wait** and **signal**) are allowed in monitors.

The syntax of a *monitorDeclaration* presented above has been simplified by leaving out **pre**, **invariant** and **post** clauses. See **module** for an explanation of these extra features. There is also an optional *compilerTimeIntegerExpression* in the first line, which is explained below. The full syntax is:

```
monitor id [ : compileTimeIntegerExpn ]
  [ implement implementItem ]
  [ implement by implementByItem ]
  [ import [ var ] importItem {, [ var ] impor
  [ export [ howExport ] id {, [ howExport ] i
  [ pre trueFalseExpn ]
  statementsAndDeclarations
  [ invariant trueFalseExpn ]
  statementsAndDeclarations
  [ post trueFalseExpn ]
end id
```

Details

If the optional *compileTimeIntegerExpression* is present, this is a *device monitor*. Its exclusive access is enforced by an implementation-dependent trick, such as executing it at a hardware priority level given by the expression. A device monitor is restricted from calling monitors (directly or indirectly). This restriction is imposed to eliminate the possibility of blocking a process with a non-zero hardware priority (as this would inadvertently allow multiple entry into a device monitor). It is the programmer's responsibility to meet this restriction; the compiler will not in general enforce the restriction. The current (1999) implementation ignores this *compileTimeIntegerExpression*.

An unexported parameterless procedure in a monitor can be specified to be an *interrupt handling procedure* by specifying a device in its header, using the form:

procedure id [: deviceSpecification]

The *deviceSpecification* is a compile time natural number that designates, to the implementation, the class of interrupts that effectively call this procedure. Interrupt handling procedures cannot be called explicitly within the program.

Details	There are two restrictions that the programmer must follow when using interrupt handling procedures; these restrictions will not necessarily be enforced by the software. The first is that an interrupt handling procedure must not execute a wait , either directly or indirectly, by calling another procedure. The second is that the interrupt handling procedure must not directly or indirectly cause an exception, unless the exception will be caught by an exception handler that is activated directly or indirectly by the interrupt handling procedure.
	Declarations of monitors within monitors are disallowed. This would be redundant anyway, as only one process can be inside the outer monitor, so the inner monitor is guaranteed to be successful.
Details	Declarations of classes within monitors are also disallowed.
	Any subprogram declared within a subprogram is now allowed to be assigned to a subprogram variable, nor passed as a parametric subprogram.
See also	unit, module and class. See also export list, import list, implement list, implement by list and deferred subprogram.

Mouse

DescriptionThis unit contains the predefined subprograms that deal with
using the mouse in a Turing program. The routines allow you to
get the current mouse cursor position, check if a button has been
pressed and get the information if it has. There are also routines to
hide and show the mouse on systems where it makes sense. (On
GUI based systems like the Macintosh, the mouse can't be hidden
as it may be needed by other applications running at the same
time.)

All routines in the Mouse module are exported qualified (and thus must be prefaced with "**Mouse.**").

Entry Points	<u>Where</u>	Gets the current location of the mouse cursor and status of the mouse buttons.
	ButtonMoved	Checks to see if a mouse button has been pressed.
	<u>ButtonWait</u>	Gets information about a mouse button being pressed such as where it was pressed, which button was pressed, etc.
	ButtonChoose	Selects the mode for the mouse (either single button mode or multi-button mode).

Mouse.ButtonChoose

Part of Mouse module

Syntax Mouse.ButtonChoose (choice : string)

The **Mouse.ButtonChoose** procedure is used to change the mode of the mouse. In Turing, the mouse can either be in "*single-button mode*" or in "*multi-button mode*". In "*single-button mode*" the mouse is treated as a one button mouse. A button is considered pressed when any button is pressed and released only when all buttons have been released.

Description

In Turing, the mouse starts in "single-button mode".

The parameter *choice* can be one of "singlebutton", "onebutton" (which switch the mouse into "*single-button mode*") or "multibutton" (which switches the mouse into "*multi-button mode*").

A program that displays the status of the mouse at the top left corner of the screen.

Mouse.ButtonChoose ("multibutton") **var** x, y, button, left, middle, right : **int Mouse.Where** (*x*, *y*, *button*) left := button mod 10 % left = 0 or 1 middle := (button - left) mod 100 % middle = 6 right := button - middle - left % right = 0 Example if left = 1 then put "left button down" end if **if** *middle* = 10 **then** put "middle button down" end if **if** right = 100 **then** put "right button down" end if

Exported qualified.

StatusThis means that you can only call the function by calling
Mouse.ButtonChoose, not by calling ButtonChoose.

See alsoMouse.ButtonMoved and Mouse.ButtonWait to get mouse
events saved in a queue. See also Mouse.Where to get the current
status of mouse button(s).

Mouse.ButtonMoved

Syntax Mouse.ButtonMoved (motion : string) : boolean

The **Mouse.ButtonMoved** function indicates whether there is a mouse event of the appropriate type on the mouse queue. Events are either "up", "down", "updown" or "downup" events (although the "downup" and "updown" are the same event).

The parameter *motion* must be one of "up", "down", "updown" or "downup". If an event of the type requested is in the queue, **Mouse.ButtonMoved** returns **true**. If the event is not in the queue, then **Mouse.ButtonMoved** returns **false**.

Description

In "*single-button mode*" (where the mouse is treated like a onebutton mouse), a "down" event occurs whenever all the buttons are up and a button is pressed. An "up" event takes place when the last button is released so that no buttons remain pressed.

In "*multi-button mode*", a "down" event occurs whenever any button is pressed, and an "up" event occurs whenever any button is released.

This program draws random circles on the screen until the user clicks the mouse button, whereupon is starts drawing random boxes. Clicking the mouse button switches between the two.

```
var circles: boolean := true
loop
var x, y, radius, clr: int
if Mouse.ButtonMoved ("down") then
var buttonnumber, buttonupdown : int
Mouse.ButtonWait ("down", x, y, buttonnu
buttonupdown)
circles := not circles
end if
x := Rand.Int (0, maxx)
y := Rand.Int (0, maxy)
radius := Rand.Int (0, 100)
clr := Rand.Int (0, maxcolor)
if circles then
```

Draw.FillOval (x, y, radius, radius, clr else Draw.FillBox (x, y, x + radius, y + radi end if end loop

This is an example demonstrating how to check for both character and mouse input at the same time.

Example	<pre>var ch : string (1) var x, y, btnnum, btnupdown : int loop if hasch then getch (ch) Text.Locate (1, 1) put "The character entered is a: ", ch end if if Mouse.ButtonMoved ("down") then Mouse.ButtonWait ("down", x, y, btnnum, Text.Locate (1, 1) put "The button was clicked at position: end if end loop</pre>
Details	Mouse.ButtonMoved can be thought of as the mouse equivalent of hasch in that they both check for something in a queue and both return immediately.
	Exported qualified.
Status	This means that you can only call the function by calling Mouse.ButtonMoved , not by calling ButtonMoved .
See also	Mouse.ButtonMoved to get mouse events saved in the queue. See also Mouse.ButtonChoose to switch between " <i>single-button mode</i> " and " <i>multi-button mode</i> ".

Mouse.ButtonWait

Syntax	<pre>Mouse.ButtonWait (motion : string, var x, y, buttonNumber, buttonUpDown : int)</pre>		
	The Mouse.ButtonWait procedure gets information about a mouse event and removes it from the queue.		
Description	The parameter <i>motion</i> must be one of "up", "down", "updown" or "downup". If an event of the type requested is in the queue, Mouse.ButtonWait returns instantly. If there isn't such an event, Mouse.ButtonWait waits until there is one and then returns (much like getch handles keystrokes).		
	In " <i>single-button mode</i> " (where the mouse is treated like a one- button mouse), a "down" event occurs whenever all the buttons are up and a button is pressed. An "up" event takes place when the last button is released so that no buttons remain pressed.		
	In " <i>multi-button mode</i> ", a "down" event occurs whenever any button is pressed, and an "up" event occurs whenever any button is released.		
	The parameters <i>x</i> and <i>y</i> are set to the position of the mouse cursor when the button was pressed. The parameter <i>buttonnumber</i> is set to 1 when in " <i>single-button mode</i> ". In " <i>multi-button mode</i> ", it is set to 1 if the left button was pressed, 2 if the middle button was pressed, and 3 if the right button was pressed. The parameter <i>buttonupdown</i> is set to 1, if a button was pressed and 0 if a button was released.		
	This program draws lines. It starts a line where the user presses down and continues to update the line while the mouse button is held down. When the button is released, the line is permanently draw and the user can draw another line.		
	<pre>var x, y, btnNumber, btnUpDown, buttons : int var nx, ny : int loop Mouse ButterWait ("down", x, y, btnNumber, buttor</pre>		

Mouse.ButtonWait ("down", x, y, btnNumber, k

```
nx := x
Example
                         ny := y
                         loop
                             Draw.Line (x, y, nx, ny, 0) % Erase prev
                             exit when Mouse.ButtonMoved ("up")
                             Mouse.Where (nx, ny, buttons)
                             Draw.Line (x, y, nx, ny, 1) % Draw line
                         end loop
                         Mouse.ButtonWait ("up", nx, ny, btnNumber, k
                         Draw.Line (x, y, nx, ny, 2) % Draw line to f
                    end loop
            This is an example demonstrating how to check for both character
            and mouse input at the same time.
                    var ch : string (1)
                    var x, y, btnNum, btnUpDown : int
                    loop
```

if hasch then

end if

getch (ch)

Text.Locate (1, 1)

Text.Locate (1, 1)

if Mouse.ButtonMoved ("down") then

put "The character entered is a: ", ch

Mouse.ButtonWait ("down", x, y, btnNum,

put "The button was clicked at position:

```
end if
end loop
```

Example

	Mouse.ButtonWait can be thought of as the mouse equivalent of
Details	getch in that they both read something in a queue and both wait
	until they get the thing they're looking for.

Exported qualified.

StatusThis means that you can only call the function by calling
Mouse.ButtonWait, not by calling ButtonWait.

See alsoMouse.ButtonWaitto see if an appropriate event is in the queue.See alsoMouse.ButtonChooseto switch between "single-button
mode" and "multi-button mode".

Syntax Mouse.Where (var *x*, *y*, *button* : int)

The **Mouse.Where** procedure is used to get current information about the status of the mouse. The parameters *x* and *y* are set to the current location of the mouse cursor. If the program is running on a system using windows, the cursor may be outside the window. This means that *x* and *y* may be set to values outside of the bounds of 0 to **maxx** and 0 to **maxy**.

Description The parameter *button* is set depending on the current mode. In "*single-button mode*" (where the mouse is treated like a one-button mouse), *button* is set to 0 if all the mouse buttons are up, and 1 if any of the mouse buttons are down. In "*multi-button mode*", *button* is assigned the sum of 1 if the left button is down, 10 if the middle button is down, and 100 if the right button is down. Thus if *button* has the value of 101, then it means that the left and right mouse buttons were depressed.

A program that displays the status of the mouse at the top left corner of the screen.

```
var x, y, button : int
loop
Mouse.Where (x, y, button)
Text.Locate (1, 1)
if button = 0 then
        put x : 4, " ", y : 4, " button up"
else
        put x : 4, " ", y : 4, " button down"
end if
end loop
```

Exported qualified.

StatusThis means that you can only call the function by calling
Mouse.Where, not by calling Where.

Mouse.ButtonMoved and **Mouse.ButtonWait** to get mouse

See also events saved in a queue. See also **Mouse.ButtonChoose** to switch between "*single-button mode*" and "*multi-button mode*".

mousewhere

Syntax mousewhere (var *x*, *y*, *button* : int)

The **mousewhere** procedure is used to get current information about the status of the mouse. The parameters *x* and *y* are set to the current location of the mouse cursor. If the program is running on a system using windows, the cursor may be outside the window. This means that *x* and *y* may be set to values outside of the bounds of 0 to **maxx** and 0 to **maxy**.

Description The parameter *button* is set depending on the current mode. In "*single-button mode*" (where the mouse is treated like a one-button mouse), *button* is set to 0 if all the mouse buttons are up, and 1 if any of the mouse buttons are down. In "*multi-button mode*", *button* is assigned the sum of 1 if the left button is down, 10 if the middle button is down, and 100 if the right button is down. Thus if *button* has the value of 101, then it means that the left and right mouse buttons were depressed.

A program that displays the status of the mouse at the top left corner of the screen.

var x, y, button : int loop mousewhere (x, y, button) locate (1, 1) if button = 0 then put x : 4, " ", y : 4, " button up" else put x : 4, " ", y : 4, " button down" end if end loop

See alsobuttonmovedandbuttonwaitto get mouse events saved in aqueue. See alsobuttonchooseto switch between "single-buttonmode" and "multi-button mode".

Example

Music

Description	This unit contains the predefined subprograms that deal with sound and music. Some of these routines have not been implemented at the time of the writing of this manual and will be implemented in future releases. All routines in the Music module are exported qualified (and thus must be prefaced with " Music .").	
Entry Points	<u>Play</u>	Plays a series of notes.
	<u>PlayFile</u>	Plays music from a file, returning when the file is finished playing. File must be in an allowable format.
	<u>PlayFileReturn</u>	Plays music from a file, returning as soon as the music starts. File must be in an allowable format.
	<u>PlayFileLoop</u>	Plays music from a file, looping over and over. Returns as soon as the music starts. File must be in an allowable format.
	<u>PlayFileStop</u>	Immediately terminates any playing music files.
	Sound	Plays a specified frequency for a specified duration.
	SoundOff	Immediately terminates any sound playing.

Music.Play

Syntax	Music.Play (music : string)		
	The Music.Play procedure is used to sound musical notes on the computer.		
Description	Sounds are produced synchronously on a per process basis. This means that when a process executes a Music.Sound or Music.Play command, it stops until the command is finished. However, other processes will continue to executing.		
Example	This program sounds the first three notes of the C scale.		
	Music.Play ("cde")		
Example	This program plays from middle C to one octave above middle C and down again in 8th notes.		
	Music.Play ("8cdefgab>c") Music.Play (" <bagfedc")<="" td=""></bagfedc">		

Execute

The syntax of the play string may be enhanced in the future.

DetailsThe **Music.Play** procedure takes strings containing characters that
specify notes, rests, sharps, flats and duration. The notes are the
letters a to g (or A to G). A rest is p (for pause). A sharp is + and a
flat is -. The durations are 1 (whole note), 2 (half note),
4 (quarter note), 8 (eight note) and 6 (sixteenth note). The
character > raises to the next octave and < lowers. For example,
this is the way to play C and then C sharp one octave above

	middle C with a rest between them, all in sixteenth notes: Music.Play (">6cpc+"). Blanks can be used for readability and ar ignored by Music.Play .	
	The Music.Play procedure requires that the machine have a sound card in order to play tones.	
	Exported qualified.	
Status	This means that you can only call the function by calling Music.Play , not by calling Play .	
See also	the Music.Sound procedure, which makes a sound of a given frequency (Hertz) and duration (milliseconds).	

Music.PlayFile

Syntax	Music.PlayFile (<i>fileName</i> : string)		
	The Music.PlayFile procedure is used to play a file of music. The must be in one of the acceptable formats and the machine, must har appropriate hardware.		
	The <i>fileName</i> parameter must give the format of the file:		
Description	WAV files "WAV:filename" or "filename.WAV"MP3 files "MP3:filename" or "filename.MP3"MIDI files "MIDI:filename" or "filename.MIDI"		
	Sounds are produced synchronously on a per process basis. This m that when a process executes a Music.Sound , Music.Play or Music.PlayFile command, it stops until the command is finished. However, other processes will continue executing.		
Details	To play music while performing any other activity, the call to Music.PlayFile must be executed in its own process . The process then called using the fork command. When a fork command is giv execution starts on the process (like a procedure call) <i>and</i> continue following the fork command <i>at the same time</i> .		
	This program plays the music in the file " <i>branden3.wav</i> " while dra ovals on the screen.		
Example	<pre>process DoMusic loop Music.PlayFile ("branden3.wav") end loop end DoMusic fork DoMusic var x, y, clr : int loop</pre>		
	<pre>loop x := Rand.Int (0, maxx) y := Rand.Int (0, maxy) clr := Rand.Int (0, maxcolor)</pre>		

Draw.FillOval (x, y, 30, 30, clr) end loop

Details	To play a sound file requires that the computer be equipped with a card and speakers.	
Details	The Turing 4.1 software can play files in the following audio forma WAVE (.wav) files, MIDI files (.midi or .mid), and MP3 files (.mp general, MIDI files are the most efficient and thus are the preferred for longer music pieces like background music. WAVE files can re- anything, not just music, so are often used for sound effects.	
Details	The Turing 4.1 software can also play music on a compact disk. To the complete contents of the compact disk, the filename is "cd". To a single track from a compact disk, the filename is "cd:[track numt Music.PlayFile ("cd") % Play the entire CL Music.PlayFile ("cd:3") % Play the third tra	
Details	On the PC, different formats of music can play simultaneously. Thi means that a program might use a MIDI file as a background sound and then use WAVE files for individual sound effects. The sound erwould not interfere with the background music. Playing a second n file with the same format as an already playing piece immediately I the first piece and starts the second. This can be used to stop a sing type of music by playing a short silent piece of music.	
	This program bounces a maple leaf around the screen with backgrc music and a sound effect when the maple leaf hits an edge. When t user presses any key, the program immediately exits.	

const STAR_SIZE : int := 80 **var** pic, x, y, dx, dy : **int** var finished : boolean := false % Play sound effect once. process Boing Music.PlayFile ("boing.wav") end Boing % Loop playing background music until 'finished' **process** BackgroundMusic **100p exit** when finished Music.PlayFile ("background.mid") end loop **end** *BackgroundMusic* % Get the original picture Draw.FillStar (3, 3, STAR_SIZE - 3, STAR_SIZE pic := Pic.New (0, 0, STAR_SIZE, STAR_SIZE) cls Example % Set the initial location and direction of move x := **Rand.Int** (0, maxx - STAR_SIZE) y :=**Rand.Int** (0, maxy - STAR_SIZE) dx := 1dv := 1**fork** BackgroundMusic % Start background music **loop** if x + dx < 0 or $x + dx > maxx - STAR_SIZE$ t dx := -dxfork Boing end if if y + dy < 0 or $y + dy > maxy - STAR_SIZE$ t dy := -dyfork Boing end if x += dxv += dv**Pic.Draw** (*pic*, *x*, *y*, *picCopy*) exit when hasch Time.Delay (5) end loop % Stop the background music. finished := true Music.PlayFileStop

Execute

See also	Music.PlayFileStop to halt a music file that is current playing, Music.PlayFileReturn for a procedure that starts a music file play and returns immediately (meaning that you do not need to use proc and fork statements), and Music.PlayFileLoop that starts a music playing continuously until it is stopped without the use of process fork .
	Exported qualified.
Status	This means that you can only call the function by calling

StatusThis means that you can only can the function by caMusic.PlayFile, not by calling PlayFile.

Music.PlayFileLoop

Part of <u>Music</u> module

Syntax	Music.PlayFileLoop (fileName : string)	
	The Music.PlayFileLoop procedure is used to play a file of music looping until the program is halted or the Music.PlayFileStop con The file must be in one of the acceptable formats and the machine, appropriate hardware.	
	The <i>fileName</i> parameter must give the format of the file:	
Description	WAV files "WAV:filename" or "filename.WAV"MP3 files "MP3:filename" or "filename.MP3"MIDI files "MIDI:filename" or "filename.MIDI"	
	The Music.PlayFileLoop procedure is used to provide continuous music for a program. When called, the music starts playing, and the returns immediately.	
Details	Unlike Music.PlayFile , the Music.PlayFileLoop procedure does r called in a separate process. However, the music will not stop play program that calls this procedure terminate) until Music.PlayFileS	
	This program continuously plays the music in the file " <i>branden3.w</i> drawing ovals on the screen.	
Example	<pre>Music.PlayFileLoop ("branden3.wav") var x, y, clr : int loop</pre>	

Details	To play a sound file requires that the computer be equipped with a speakers.	
Details	The Turing 4.1 software can play files in the following audio forma (.wav) files, MIDI files (.midi or .mid), and MP3 files (.mp3). In ge files are the most efficient and thus are the preferred form for longe like background music. WAVE files can record anything, not just n often used for sound effects.	
	The Turing 4.1 software can also play music on a compact disk. To complete contents of the compact disk, the filename is "cd". To pla from a compact disk, the filename is "cd:[track number]".	
Details	Music.PlayFileLoop ("cd")% Play the entirMusic.PlayFileLoop ("cd:3")% Play the third	
Details	On the PC, different formats of music can play simultaneously. Thi program might use a MIDI file as a background soundtrack and the files for individual sound effects. The sound effects would not inter background music. Playing a second music file with the same form playing piece immediately halts the first piece and starts the second used to stop a single type of music by playing a short silent piece of	
	This program bounces a maple leaf around the screen with backgrc sound effect when the maple leaf hits an edge. When the user press program immediately exits.	
	<pre>const STAR_SIZE : int := 80 var pic, x, y, dx, dy : int</pre>	
	% Get the original picture Draw.FillStar (3, 3, STAR_SIZE - 3, STAR_SIZE -	

pic := Pic.New (0, 0, STAR_SIZE, STAR_SIZE) cls % Set the initial location and direction of move x := Rand.Int (0, maxx - STAR_SIZE) y := **Rand.Int** (0, maxy - STAR_SIZE) dx := 1dy := 1Music.PlayFileLoop ("background.mid") % Start ba Example **loop** if x + dx < 0 or $x + dx > maxx - STAR_SIZE$ t dx := -dxMusic.PlayFileReturn ("boing.wav") end if if y + dy < 0 or $y + dy > maxy - STAR_SIZE$ t dy := -dvMusic.PlayFileReturn ("boing.wav") end if x += dxy += dy**Pic.Draw** (*pic*, *x*, *y*, *picCopy*) exit when hasch Time.Delay (5) end loop % Stop the background music.

Music.PlayFileStop

See also	Music.PlayFileStop to halt a music file that is current playing.
	Exported qualified.
Status	This means that you can only call the function by calling Music.Pl by calling PlayFileLoop .

Music.PlayFileReturn

Part of Music module

Music.PlayFileReturn (fileName : string) **Syntax**

The **Music.PlayFileReturn** procedure is used to play a file of mus be in one of the acceptable formats and the machine, must have the hardware.

The *fileName*parameter must give the format of the file:

	WAV files	"WAV:filename" or "filename.WAV"
Description	MP3 files	"MP3:filename" or "filename.MP3"
	MIDI files	"MIDI:filename" or "filename.MIDI"

Unlike Music.PlayFile, the Music.PlayFileReturn procedure show in a separate process. Instead, the procedure returns immediately. T Music.PlayFileReturn easier to use, but makes it unsuitable for pl files sequentially.

This program plays the music in the file "*branden3.wav*" once whil on the screen. If the music has not finished when the user presses a using Music.PlayFileStop

	<pre>Music.PlayFileReturn ("branden3.wav")</pre>
	var x, y, clr : int
	loop
Example	<pre>x := Rand.Int (0, maxx)</pre>
	y := Rand.Int (0, maxy)
	clr := Rand.Int (0, maxcolor)
	Draw.FillOval (<i>x</i> , <i>y</i> , 30, 30, <i>clr</i>)
	exit when hasch
	end loop
	Music.PlayFileStop

Details	To play a sound file requires that the computer be equipped with a speakers.	
Details	The Turing 4.1 software can play files in the following audio forma (.wav) files, MIDI files (.midi or .mid), and MP3 files (.mp3). In go files are the most efficient and thus are the preferred form for longo like background music. WAVE files can record anything, not just n often used for sound effects.	
D . 1	The Turing 4.1 software can also play music on a compact disk. To complete contents of the compact disk, the filename is "cd". To pla from a compact disk, the filename is "cd:[track number]".	
Details	Music.PlayFileReturn ("cd")% Play the entMusic.PlayFileReturn ("cd:3")% Play the thi	
Details	On the PC, different formats of music can play simultaneously. Thi program might use a MIDI file as a background soundtrack and the files for individual sound effects. The sound effects would not inter background music. Playing a second music file with the same form playing piece immediately halts the first piece and starts the second used to stop a single type of music by playing a short silent piece o	
	This program bounces a maple leaf around the screen with backgrc sound effect when the maple leaf hits an edge. When the user press program immediately exits.	
	<pre>const STAR_SIZE : int := 80 var pic, x, y, dx, dy : int</pre>	
	% Get the original picture Draw.FillStar (3, 3, STAR_SIZE - 3, STAR_SIZE -	

pic := Pic.New (0, 0, STAR_SIZE, STAR_SIZE) cls % Set the initial location and direction of move x :=**Rand.Int** (0, *maxx* - *STAR_SIZE*) y :=**Rand.Int** (0, maxy - STAR_SIZE) dx := 1dy := 1Music.PlayFileLoop ("background.mid") % Start ba Example **loop** if x + dx < 0 or $x + dx > maxx - STAR_SIZE$ t dx := -dxMusic.PlayFileReturn ("boing.wav") end if if y + dy < 0 or $y + dy > maxy - STAR_SIZE$ t dy := -dyMusic.PlayFileReturn ("boing.wav") end if x += dxv += dv**Pic.Draw** (*pic*, *x*, *y*, *picCopy*) exit when hasch Time.Delay (5) end loop % Stop the background music.

Music.PlayFileStop

Execute

 See also
 Music.PlayFileStop to halt a music file that is current playing. Exported qualified.
 Status
 This means that you can only call the function by calling Music.Pl not by calling PlayFileReturn.

Music.PlayFileStop

Syntax Music.PlayFileStop

DescriptionThe Music.PlayFileStop procedure is used to to stop all music file
playing. This includes processes that are executing the Music.Play
procedure (they exit immediately and start executing the next state
process), and the Music.PlayFileReturn and Music.PlayFileLoop
statements, which simply stop playing the music.

In Turing, a program will not halt execution until all processes hav terminated. This means that if you are playing background music, t program will not terminate, even if execution returns from the main unless the background music is halted.

This program plays the background music for 30 seconds and then terminates. Note that it is important to set the flag (*finished*) before **Music.PlayFileStop**. If **Music.PlayFileStop** comes first, it is poss process to return from **Music.PlayFile**, loop around, skip over the and call **Music.PlayFile** again before the *finished* flag is set.

	<pre>var finished : boolean := false</pre>
Example	<pre>% Loop playing background music until 'finished' process BackgroundMusic loop exit when finished Music.PlayFile ("background.mid") end loop end BackgroundMusic</pre>
	fork BackgroundMusic % Start the background n Time.Delay (30000) % Wait for 30 seconds
	% Stop the background music. finished := true % The flag must be set f Music.PlayFileStop % Music.PlayFile will return

Execute

See also	Music.PlayFile for playing music files and a larger example.
	Exported qualified.
Status	This means that you can only call the function by calling Music.PlayFileStop , not by calling PlayFileStop .

Music.Sound

Syntax Music.Sound (*frequency*, *duration* : **int**)

The **Music.Sound** statement is used to cause the computer to soun of a given frequency for a given time. The frequency is in cycles posecond (Hertz). The time duration is in milliseconds. For example, A on a piano is 440 Hertz, so **Music.Sound**(440, 1000) plays midd one second.

Description

Sounds are produced synchronously on a per process basis. This m that when a process executes a **Music.Sound** or **Music.Play** comm stops until the command is finished. However, other processes will continue executing.

This program plays a siren sound in the background.

process siren loop for i : 100 .. 3000 by 100 Music.Sound (i, 50) % Sound note end for for decreasing i : 2900 .. 200 by 100 Music.Sound (i, 50) % Sound note end for end for end loop end siren ... the rest of the program goes here while the si

Example

Execute

Exported qualified.

- StatusThis means that you can only call the function by calling Music.So
not by calling Sound.
- Music.Playstatement, which plays notes based on musical notationSee alsoexample, Music.Play("8C") plays an eighth note of middle C.

Music.SoundOff

Syntax	Music.SoundOff
Description	The Music.SoundOff procedure stops any sound or music that is currently playing or is waiting to play.
	Exported qualified.
Status	This means that you can only call the function by calling Music.SoundOff , not by calling SoundOff .
See also	Music.Play , Music.PlayFile , and Music.Sound procedures, which make sounds that can be turned off with Music.SoundOff .

A *namedType* is one of:

Syntax (a) typeId (b) moduleId . typeId

Description A type can be given a name (*typeId*) and later this name can be used instead of writing out the type.

In this example, *phoneRecord* is a named type.

Exampletype phoneRecord :
record
name : string (20)
phoneNumber : int
address : string (50)
end recordExample...
record
var oneEntry : phoneRecord
var phoneBook : array 1 ... 100 of phoneRecordDetailsForm (a) is the most common kind of named type. Form (b) is
used when the type name has been exported from a module.DetailsArrays whose bounds are not known at compile time cannot be
named.

number of arguments

Syntax	nargs : int
	The nargs function is used to determine the number of arguments that have been passed to a program from the command line. For example, if the program is run from the Turing environment using
	:r file1 file2
Description	then nargs will return 2. If a program called <i>prog.x</i> is run under UNIX using this command:
Description	prog.x file1 file2
	the value of nargs will similarly be 2.
	The nargs function is usually used together with the fetcharg function to access the arguments that have been passed to the program.
See also	fetcharg for an example of the use of <u>nargs</u> .

nargs

natural number type

nat **Syntax** The **nat** (natural number) type has the values 0, 1, 2, 3 ... Natural numbers can be combined by various operators, such as addition (+) and multiplication (*). Natural numbers can be combined with integers (type **int**), in which case the result is an integer. Natural **Description** numbers can also be combined with **real** numbers, in which case the result is generally a real number. Natural numbers can always be assigned to real variables, with implicit conversion to real. var counter : nat Example **var** *j* : **nat** := 9 See also *explicitIntegerConstant*. The **nat** type is used instead of **int** when the values are known to be non-negative. The Turing operators on natural numbers are the same as those for integers: +, -, * (multiply), **div** (truncating integer division), **mod** (integer remainder), ****** (exponentiation), as well as comparisons (+, **not=**, >, >=, <, <=). The operators **and**, **or** and **xor** to be applied to natural number values. The bit-wise boolean result is produced as a natural number. The **shr** (shift right) and **shl** (shift left) operators are also introduced. In the current implementation, the range of natural numbers is from 0 to 4294967294. In other words, the maximum value of a natural number is 2**32 - 2. This range exists because natural numbers are stored in 4 bytes. The types **nat1**, **nat2** and **nat4** specify natural numbers that fit into 1, 2 or 4 bytes. Explicit constants such as 213 and 0 are considered to be integers. As a result the type of *tax* in this declaration is **int**: **var** tax := 0 % The type is **int**

Natural number values can be used whenever integer values are

nat

expected and vice versa, given that the value does not exceed the range of the expected type.

Details When integer and natural numbers are combined using a binary operator such as +, the result is an integer. This means, for example, that if *counter* is a natural number, *counter* + 1 is considered to be an integer. As long as the result fits into the range that is the intersection of the ranges of **int** and **nat**, the result will be as expected. Anomalies occur when the result is (or would be) greater than the largest integer (**maxint**=2147483647). For example, if natural number *n* is greater than **maxint**, the expression *n* + 1 will overflow, because its result is an **int** (because 1 is an **int**). To avoid this problem, you must be careful that both operands are natural numbers.

Suppose we have this declaration:

const natOne : nat := 1

We can safely compute *n* + *natOne* because both operands have type **nat**.

Natural numbers can be converted to real numbers using **natreal**, but in practice this is rarely used, because a natural value used in place of a real value will be automatically converted to real.

Natural numbers can be converted to strings and back using **natstr** and **strnat**.

In the C language, a natural number is said to be "unsigned".

See also <u>maxnat</u>, <u>int</u>, <u>nat</u>*n*, <u>int</u>*n*, <u>natstr</u>, <u>strnat</u> and <u>natreal</u>.

n-byte natural number type

Dirty

	(a)	nat1	% 1-by	te natural number
Syntax	(b)	nat2	% 2-by	te natural number
	(C)	nat4	% 4-by	te natural number
Description	types that nat type is type (it ov	occupy a speci s in principle a	fied number machine-in ver, when th	pes are machine-dependent r of bytes. By contrast, the dependent and mathematical ne value is too large or small, nto 4 bytes).
Example	١	var counter1 var counter2 var counter4	: nat2	% Range is 0 255 % Range is 0 65536 % Range is 0 42949672
Details	In Turing, the range of the nat is 0 to 4294967294, which means that the nat4 type allows one more value, 4294967295. This extra value is used in nat to represent the state of being uninitialized. The nat <i>n</i> types allow use of all possible values that fit into n bytes and thereby eliminates checking for initialization.			
	The natn types are like the C language types <i>short unsigned</i> , <i>unsigned</i> , and <i>long unsigned</i> , except that the number of bytes occupied by the C types depends on the particular C compiler.			
See also	the <u>int</u> n ty <u>int</u> .	pes which are	n byte integ	er values. See also <u>nat</u> and

nat*n*

natreal

natural number to real function

Syntax	natreal (<i>n</i> : nat) : real
Description	The natreal function is used to convert a natural number to a real number. This function is rarely used, because in Turing, a natural number can be used anyplace a real value is required. When this is done, the natreal function is implicitly called to do the conversion from nat to real . The natreal function is similar to intreal , except that natreal handles values that are larger than int values and does not handle negative values.
See also	<u>nat</u> . See also the <u>intreal</u> , <u>floor</u> , <u>ceil</u> and <u>round</u> functions.

natural-number-to-string function natstr **natstr** (*n* : **nat** [, width : **int** [, base : **int**]]) : **string Syntax** The **natstr** function is used to convert a natural number to a string. The string is equivalent to *n*, padded on the left with blanks as necessary to a length of *width*, written in the given number *base*. For example, **natstr** (14, 4, 10)="*bb*14" where *b* represents a blank. The *width* and *base* parameters are both optional. It they are omitted, the string is made just long enough to hold the value and the number base is 10. For example, **natstr** (23) = "23". The *width* parameter must be non-negative. If *width* is not large enough to represent the value of *i*, the length is automatically increased as needed. The string returned by **natstr** is of the form: Description {blank}digit{digits} where {blank} means zero or more blanks and digit{digit} means one or more digits. The leftmost digit is either non-zero, or a single zero digit; in other words, leading zeros are suppressed. The letters A, B, C ... are used to represent the digit values 10, 11, 12, ... The *base* must be in the range 2 to 36 (36 because there are ten digits and 26 letters). For example, **natstr** (255, 0, 16) = "FF". The **natstr** function is the inverse of **strnat**, so for any natural number *n*, **strnat** (**natstr**(*n*)) = n. chr, ord and strnat functions. See also the intstr and strint functions. See also *explicitIntegerConstant* for the way to write See also values in base 2 and base 16 in a program.

Net

Entry Points The Net module allows TCP/IP equipped machines to communicate. In the current implementation (WinOOT 3.0), this is available only under Win32 (Windows 95, 98, NT and later).

To allow two machines to communicate, there must be a server
(which calls Net,WaitForConnection) and a client (which calls
Net.OpenConnection). The server waits until a client connects
and then starts communication between the two. When a
connection is established, a net stream is returned that can be used
in the same fashion as a file stream (i.e. using puts and gets).DescriptionNet.CloseConnection.

For ease of reading web pages, the **Net.OpenURLConnection** opens up a URL for reading with the **get** statement. It is up to the user program to interpret the HTML or file located at the URL.

All subprograms in the **Net** unit are exported qualified (and thus must be prefaced with "**Net**.").

WaitForConnection	Waits until a client connects to a specified port.
OpenConnection	Opens a connection to a specified machine.
OpenURLConnection	Opens a connection to a file specified by a URL.
<u>CloseConnection</u>	Closes a specified connection.
BytesAvailable	Returns the number of bytes available to be read from a net stream.
<u>CharAvailable</u>	Returns true if there is a character available to be read from a net stream.
LineAvailable	Returns true if there is a line of text available to be read from a net

	stream.
TokenAvailable	Returns true if there is a token available to be read from a net stream.
HostAddressFromName	Returns a host's address given its host name.
HostNameFromAddress	Returns a host's name given its address.
LocalAddress	Returns the host name of the local machine.
LocalName	Returns the TCP/IP address of the local machine.

Net.BytesAvailable

Syntax	Net.BytesAvailable (<i>netStream</i> : int) : int
Description	Returns the number of bytes available for reading from the net stream specified by the <i>netStream</i> parameter.
Details	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
Status	Exported qualified.
	This means that you can only call the function by calling Net.BytesAvailable , not by calling BytesAvailable .
See also	Net.CharAvailable, Net.LineAvailable, and Net.TokenAvailable.

Net.CharAvailable

Syntax	Net.CharAvailable (<i>netStream</i> : int) : boolean
Description	Returns true if a character is waiting to be read from the net stream specified by the <i>netStream</i> parameter. If Net.CharAvailable returns true , then a single character can be read from the stream without blocking.
Details	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
Example	The following program fragment reads a character from <i>netStream</i> only if there is one waiting to be read.
	<pre>if Net.CharAvailable (netStream) then var ch : char get : netStream, ch put ch end if</pre>
Status	Exported qualified.
	This means that you can only call the function by calling Net.CharAvailable , not by calling CharAvailable .
See also	<u>Net.BytesAvailable, Net.LineAvailable</u> , and <u>Net.TokenAvailable</u> .

Net.CloseConnection

Syntax	Net.CloseConnection (netStream : int)		
Description	Closes a network connection made with Net.OpenConnection or Net.WaitForConnection . After the connection is closed, the net st cannot be used for any purpose on either side of the connection.		
Details	The Net module requires a TCP/IP stack to be installed and operati order to function. It does not communicate using any other protoco		
	It is possible for Firewalls to interfere with the actions of the Net n preventing connections from taking place.		
	The following program fragment connects to port 5300 on the mac specified by <i>netAddress</i> , sends the work OK to it and closes the connection.		
Example	<pre>netStream := Net.OpenConnection (netAddress, cha if netStream <= 0 then put "Unable to connect to ", netAddress return end if put : netStream, "OK" Net.CloseConnection (netStream)</pre>		
	Exported qualified.		
Status	This means that you can only call the function by calling Net.CloseConnection , not by calling CloseConnection .		
See also	Net.OpenConnection and Net.WaitForConnection.		

Net.HostAddressFromName

Syntax	Net.HostAddressFromName (hostName : string) : string
Description	Returns the numeric TCP/IP address of the machine whose hostname is specified by the <i>hostName</i> parameter.
Details	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
Example	The following program prints out the hostname of the current machine.
	<pre>var hostName : string := "www.holtsoft.com" put "The machine address of ", hostName , " is " Net.HostAddressFromName (hostName)</pre>
Status	Exported qualified.
	This means that you can only call the function by calling Net.HostAddressFromName , not by calling HostAddressFromName .
See also	Net.HostNameFromAddress.

Net.HostNameFromAddress

Syntax	Net.HostNameFromAddress (hostAddr : string) : string
Description	Returns the TCP/IP hostname of the machine whose numeric address is specified by the <i>hostAddr</i> parameter.
Details	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
Example	The following program prints out the hostname of the machine whose TCP/IP numeric address is "128.100.5.1".
	<pre>var hostAddr : string := "128.100.5.1" put "The machine name of ", hostAddr, " is ", Net.HostNameFromAddress (hostAddr)</pre>
Status	Exported qualified.
	This means that you can only call the function by calling Net.HostNameFromAddress , not by calling LocalName .
See also	Net.HostAddressFromName.

Net.LineAvailable

Syntax	Net.LineAvailable (<i>netStream</i> : int) : boolean
Description	Returns true if a line of input is waiting to be read from the net stream specified by the <i>netStream</i> parameter. If Net.LineAvailable returns true , then a line of input can be read from the stream without blocking.
Details	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
Example	The following program fragment reads a character from <i>netStream</i> only if there is one waiting to be read.
	<pre>if Net.LineAvailable (netStream) then var line : string get : netStream, line : * put line end if</pre>
Status	Exported qualified.
	This means that you can only call the function by calling Net.LineAvailable , not by calling LineAvailable .
See also	Net.BytesAvailable, Net.CharAvailable, and Net.TokenAvailable.

Net.LocalAddress

Syntax	Net.LocalAddress : string
Description	Returns the TCP/IP numeric address of the machine the program is running on. The numeric address is of the form <i>xxx.yyy.zzz.www</i> where each segment is a number from 0 to 255.
Details	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
Example	The following program prints out the TCP/IP numeric address of the current machine.
	<pre>put "Your machine address is ", Net.LocalAddress</pre>
Status	Exported qualified.
	This means that you can only call the function by calling Net.LocalAddress , not by calling LocalAddress .
See also	<u>Net.LocalName</u> .

Net.LocalName

Syntax	Net.LocalName : string
Description	Returns the TCP/IP hostname of the machine the program is running on.
Details	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
Example	The following program prints out the hostname of the current machine.
	<pre>put "Your machine name is ", Net.LocalName</pre>
Status	Exported qualified.
	This means that you can only call the function by calling Net.LocalName , not by calling LocalName .
See also	Net.LocalAddress.

Net.OpenConnection

Syntax	Net.OpenConnection (<i>netAddr</i> : string , <i>port</i> : int) : int
	Attempts to open a connection to port specified by the <i>port</i> parameter on the machine specified by <i>netAddr</i> parameter. There must be a program listening to that port for the connection to be made. In OOT, this is done using the Net.WaitForConnection function.
Description	If successful, Net.OpenConnection returns a network stream descriptor which can be used with the put , get , read , and write statements and eof function to send and receive data to the listening program. It is also the parameter used for the Net.CloseConnection , Net.BytesAvailable , Net.CharAvailable , Net.LineAvailable , and Net.TokenAvailable functions.
	The <i>netAddr</i> parameter is a string specifying the net address of the machine to be connected to. This can either be the full hostname or the numerical address.
	In general, system program listen in on ports with numbers below 1024. Port numbers above 1024 are generally available for use by user created programs.
Details	The program will wait for an indeterminate amount of time to mak the connection. If it fails, it will return a non-positive value.
	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
	The following program implements a "Chat" program. One user runs the program on their machine as a server, which waits for

another machine to connect to it. The second user specifies the machine to connect to and then connects. The two can then type at each other.

% The "Chat" program const chatPort : int := 5055 var choice : int **100**p put "Enter 1 to run chat server" put "Enter 2 to run chat session" put "Choice: " ... **aet** choice **exit** when choice = 1 or choice = 2 end loop var netStream : int var netAddress : string **if** choice = 1 **then** netStream := Net.WaitForConnection (chatPort else put "Enter the address to connect to: " ... get netAddress netStream := Net.OpenConnection (netAddress, if netStream <= 0 then</pre> put "Unable to connect to ", netAddress return end if end if Draw.Cls put "Connected to ", netAddress var localRow : int := 2 **var** localCol : **int** := 1 var remoteRow := maxrow div 2 var remoteCol : int := 1 var ch : char View.Set ("noecho") **loop** if hasch then ch := getchar put : netStream, ch if $ch = ' \ then$ localRow := localRow mod (maxrow div localCol := 1 **Text.Locate** (localRow, localCol) put "" % Clear to end of line **Text.Locate** (localRow, localCol) else

Example

```
Text.Locate (localRow, localCol)
            put ch ..
            localCol += 1
        end if
    end if
    if Net.CharAvailable (netStream) then
        get : netStream, ch
        if ch = ' \ n' then
            remoteRow := remoteRow mod (maxrow d
                1 + (maxrow div 2)
            remoteCol := 1
            Text.Locate (remoteRow, remoteCol)
            Text.Locate (remoteRow, remoteCol)
        else
            Text.Locate (remoteRow, remoteCol)
            put ch ..
            remoteCol += 1
        end if
    end if
end loop
```

Exported qualified.

- StatusThis means that you can only call the function by calling
Net.OpenConnection, not by calling OpenConnection.
- See also <u>Net.WaitForConnection</u> and <u>Net.CloseConnection</u>.

Net.OpenURLConnection

Syntax	Net.OpenURLConnection (<i>urlAddr</i> : string) : int
	Attempts to open a http connection to pthe URL (Universal Resource Locator) specified by the <i>urlAddr</i> .
Description	If successful, Net.OpenURLConnection returns a network stream descriptor which can be used with the get statement and eof function to read the web page located at the URL.
	The program will wait for an indeterminate amount of time to make the connection. If it fails, it will return a non-positive value.
Details	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
	The following program prints out the contents of the file specified by the user.
	var <i>url</i> : string put "Enter the URL to load: " get <i>url</i>
Example	<pre>var netStream : int var line : string</pre>
	<pre>netStream := Net.OpenURLConnection (url) if netStream <= 0 then put "Unable to connect to ", url return end if loop exit when eof (netStream) get : netStream, line put line end loop</pre>
	Net.CloseConnection (<i>netStream</i>)

Exported qualified.

- StatusThis means that you can only call the function by calling
Net.OpenURLConnection, not by calling
OpenURLConnection.
- See also <u>Net.CloseConnection</u>.

Net.TokenAvailable

Syntax	Net.TokenAvailable (<i>netStream</i> : int) : boolean
Description	Returns true if a line of input is waiting to be read from the net stream specified by the <i>netStream</i> parameter. If Net.TokenAvailable returns true , then a single token (character surrounded by whitespace) can be read from the stream without blocking.
	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
Details	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
	The following program fragment reads a character from <i>netStream</i> only if there is one waiting to be read.
Example	<pre>if Net.TokenAvailable (netStream) then var token : string get : netStream, token put token end if</pre>
Status	Exported qualified.
	This means that you can only call the function by calling Net.TokenAvailable , not by calling TokenAvailable .
See also	<u>Net.BytesAvailable, Net.CharAvailable</u> , and <u>Net.LineAvailable</u> .

Net.WaitForConnection

	Net.WaitForConnection (port : int,
Syntax	var netAddr : string) : int
Description	Listens for a connection at the port specified by the <i>port</i> parameter. When another program connects to the port, then the function returns. The address of the connecting machine is specified in the <i>netAddr</i> parameter and the Net.WaitForConnection returns a network stream descriptor which can be used with the put , get , read , and write statements and eof function to send and receive data to the connecting program. It is also the parameter used for the Net.CloseConnection , Net.BytesAvailable , Net.CharAvailable , Net.LineAvailable , and Net.TokenAvailable functions.
	In OOT, the connection to a port is made with the Net.OpenConnection function.
	The <i>netAddr</i> parameter is a string specifying the net address of the machine that connected to the port. It is the machines numerical address.
	In general, system program listen in on ports with numbers below 1024. Port numbers above 1024 are generally available for use by user created programs.
	The program will wait for indefinitely for a connection to made to the port.
	The Net module requires a TCP/IP stack to be installed and operating in order to function. It does not communicate using any other protocols
	It is possible for Firewalls to interfere with the actions of the Net module, preventing connections from taking place.
	See Net.OpenConnection for an example of

ExampleNet.WaitForConnection.Exported qualified.StatusThis means that you can only call the function by calling
Net.WaitForConnection, not by calling WaitForConnection.See alsoNet.OpenConnection and Net.CloseConnection.

new	statement
	A newStatement is:
Syntax	new [collectionOrClassId ,] pointerVariableReferen
Description	A new statement creates (allocates) a new element and assigns its l pointer variable. This element can be an object of a collection or cl type. If the <i>collectionOrClassId</i> is omitted, the choice of element is the type of the pointer. For example, if the pointer is to class <i>C</i> , an will be allocated.
	Using a collection, declare a list of records and allocate one record
Example	<pre>var list : collection of record contents : string (10) next : pointer to list % Short form: ne end record var first : pointer to list % Short form: va new list, first % Short form: new first</pre>
Example	Using a class, create an object of that class. The object is located by class node export var next, var name name : string (25) next : pointer to node % Short form: next : end node var start : pointer to node % Short form: var st new node, start % Short form: new start
Example	Using a record type, declare a list of records and allocate one recor type item: record

	As the examples in this section show, a pointer can locate one of th object of a collection, an object of a class or a value of a type.
	In the new statement, the <i>collectionOrClassId</i> can be omitted. If th type, it must be omitted. The free statement is used to deallocate a
	An imported class can have one of its objects created (by the new s the class is imported var .
	If there is no more space to allocate an element, new will set the povalue, and the program will continue executing.
Details	If the pointer locates class <i>C</i> and <i>C</i> contains an implement by list, by new is the inherited object (through any number of levels of im pointer, however, remains a pointer to <i>C</i> .
	The form new p is a short form for new C , p when C is the class or in p 's type.
	If p is a pointer to class C and C has a descendant (expansion) class statement can be used to allocate an object of type D , as in:
	new D, p % Allocates an object of class D
	If D has an implement by clause, the expansion is created.
	The new statement can also be used to resize a flexible array . If a declared flexible using the syntax .
	<pre>var name : flexible array indexType { , indexTyp</pre>
	The indices may have compile-time or run-time upper bounds (the be compile-time). The upper bounds can be changed by using:
	new name , newUpper1 {,newUpper2}
	The existing array entries will retain their values, except that any in will have the corresponding array entries lost. Any index made larg new array entries uninitialized (if applicable).
Details	Additionally, the upper bound (both in the declaration and the new

made one less than the lower bound. This effectively makes an arra elements. It can later be increased in size with another **new**.

In the current implementation (1999), with a multi-dimensional arr zero number of total elements, it is a run-time error to change any l dimension (unless one of the new upper bounds is one less than the lower bound, giving 0 elements in the array) as the algorithm to rea element memory locations has not yet been implemented.

Currently, only variables can be declared in this form. There is no f parameter type, although a flexible array can be passed to an array "*" as the upper bound.

Example See **array** for an example of **flexible arrays**.

<u>class</u> and <u>**collection**</u> declarations, <u>**pointer**</u> type, <u>**free**</u> <u>statement</u>, <u>**nil**</u> <u>**implement**</u> by list.

See also

For <u>flexible</u> arrays, see also <u>array</u> and <u>flexible</u>.

nil	pointer to a collection
Syntax	nil [(collectionOrClassId)]
Description	The nil pointer does not locate any element (object). Pointers locate items in collections, classes and types. The <i>collectionOrClassId</i> is optional.
	This nil pointer is distinct from pointers to actual elements, and it can be compared to these pointers. It is also distinct from the uninitialized pointer value.
Example	In this example, the pointer called <i>first</i> is set to the nil pointer of collection c , that is, to nil (c).
	<pre>var c : collection of record name : string (50) next : pointer to c end record var first : pointer to c := nil (c)</pre>
Details	See also collection, class and pointer. When nil is written without the <i>collectionOrClassId</i> , it can be assigned to a pointer to any collection, class or type.
	The type of nil without the <i>collectionOrClassId</i> is effectively a pointer to <i>everyClass</i> , an imaginary class that has no objects and is the descendant of all classes. This implies that it can be assigned to any other class pointer, because it is a descendant of all classes.
	Turing allows you to write nil (<i>id</i>) after a forward declaration of <i>id</i> (the name of a collection, class or type) before (and after) the resolution of the <i>id</i> .

Syntax	not
Description	The not (<i>boolean negation</i>) operator produces the opposite of a tru value. For example, not ($x > y$) is equivalent to $x \le y$.
Example	<pre>var error : boolean := false var success : boolean success := not error % success becomes the op</pre>
Details	The not operator takes true and produces false and takes false and produces true . The not operator can be written as ~. See also the b type, <i>prefix operators</i> , and <i>precedence</i> of operators. The not operator can be applied to sets.

not

objectclass

Syntax	objectclass (<i>pointerExpn</i>)			
Description	The objectclass attribute is used to find the class of an object located by a pointer. The <i>pointerExpn</i> must be an expression that is a pointer to a class.			
Example	See class for an example of classes and inheritance, in which a class called <i>TextFile</i> is inherited by a class called <i>Device</i> . The <i>Device</i> class adds a new exported procedure called <i>ioCtl</i> . In the present example, objectclass is used to test to make sure that the <i>textFilePtr</i> currently locates an object that was created as a <i>Device</i> (or as a descendant of <i>Device</i>). The notation <i>Device(textFilePtr)</i> converts the pointer to be a pointer to a <i>Device</i> so that <i>ioCtl</i> can be called.			
	var textFilePtr : ^ TextFile			
	 if objectclass (textFilePtr) >= Device then % Can safely treat object as a Device Device (textFilePtr) . ioCtl			
	end if			
	This example uses the class comparison operator >= which means "is a descendant of". See class .			
Details	You can only use objectclass in class comparisons. In particular, objectclass cannot be used to declare pointers. For example, this:			
	<pre>var p : ^objectclass (q)</pre>			
	is not allowed.			

opaque

Description	When a type T is exported from module, monitor or class M using the keyword opaque , the type $M.T$ is distinct from all other types. Opaque types are used to guarantee that updates to values of the type are done within M .
See also	module declarations for an example of an <u>opaque</u> type used to <u>implement</u> complex arithmetic. See also <u>equivalence</u> of types for the definition of the type matching rules for <u>opaque</u> types.

An openStatement is one of:

Syntax

(a) open : fileNumberVar, fileName, ioCapability

ioCapability }
open : fileNumberVar, argNum, ioCapability

ioCapability }

The **open** statement connects the program to a file so the program operations such as **read** on the file. In form (a), the **open** statement *fileName*, such as "Master", to a file number such as 5. Form (b), w commonly used, opens a file whose name is given by a program ar described below.

Description

The **read** statement uses the file number, not the file name, to accert the program is finished using the file, it disconnects from the file u statement. Each *ioCapability* is the name of an operation, such as **r** performed on the file.

This programs illustrates how to open, read and then close a file.

var fileName : string := "Master" % Name of fi
var fileNo : int % Number of file
var inputVariable : string (100)
open : fileNo, fileName, read
...
read : fileNo, inputVariable
...
close : fileNo

The **open** statement always sets the *fileNumber* to a positive number fails (generally because the file does not exist), the *fileNumber* is so positive number. It is wise to check that the stream number is great before using it further.

An *ioCapability* is one of:

get, put, read, write, seek, mod

open

A file can be accessed using only the statements corresponding to t capabilities with which it was opened. Note: **tell** is allowed only if **seek**.

The **open** statement truncates the file to length zero if the *ioCapab* **put** or **write** but not **mod** (which stands for **mod**ify). In all other cathe existing file intact. The **mod** *ioCapability* specifies that the file modified without being truncated. Each **open** positions to the begin There is no mechanism to delete a file.

To open for appending to the end of the file, one has to **open** for **se write** or **put** and then **seek** to the end of the file. See the **seek** state:

Mixed mode files, which combine **get** and **read** (or **put** and **write**), by some operating systems, such as UNIX, but not by others, such Windows.

On Microsoft Windows, one should note that opening files in other the backslash character. This is because the backslash is a special c Turing (as in \t for tab and \n for a newline). To get a single backsla

e.g. open : f, "C:\\STUDENTS\\SMITH\\ACCT.DAT",

Form (b) of the syntax allows you to open a file whose name is giv argument on the command line. For example, under UNIX, the cor

Details

prog.x infile outfile

specifies to execute *prog.x* with program arguments *infile* and *outfi* the Turing programming environment, the **run** command can accel arguments. The *argNumber* is the position of the argument on the c (The first argument is number 1.) The name of the file to be opened corresponding program argument. If there is no such argument, or the opened successfully, *fileNumberVariable* is set to zero. See also gives the number of arguments, and **fetcharg**, which gives the *n*-th

Program argument files referenced by argument number and used i or **write** statements need not be explicitly opened, but are implicitly the capability corresponding to the input/output statement in which used. (The *fileNumber* gives the number of the argument.) The operating system standard files (error, output and input) are ac numbers 0, -1, and -2, respectively (although this may be subject to files are not opened explicitly, but are used simply by using form (l number. Beware of the anomalous case of a failed open that gives y 0. A subsequent use of this number in a **put** will produce output tha standard error stream, with no warning that the file you attempted t actually being used.

To append to a file, the file must be opened with the **mod** and **seek** then there must be a seek to the end of file. For example:

var streamnumber : int
open : streamnumber, "myfile", put, mod, seek
seek : streamnumber, *
put : streamnumber, "This appears at the end of

There is an older and still acceptable version of **open** that has this s

open (var fileNumber : int, fileName : string,

The *mode* must be "r" (for **get**) or "w " (for **put**).

The path name specified in the open statement and elsewhere can a UNIX format (i.e. with forward slashes, an initial forward slash inc absolute directory). On the PC, absolute paths would have the form

a:/dir1/dir2/filename

On the Macintosh, they would have the form:

Details

/volume name/directory1/directory2/file name

Note that in addition to the UNIX path format, on the PC, you can standard PC path notation and on the Macintosh, you can use stand path notation. On the Macintosh volume, directory and file names (in them.

All routines (such as the File and Dir module routines) will return f UNIX format, regardless of the machine the program is run on.

See also <u>close</u>, <u>get</u>, <u>put</u>, <u>read</u>, <u>write</u>, <u>seek</u> and <u>tell</u> statements.

Syntax	A or B			
Description	The or (boolean) operator yields a result of true if at least one (or both) of the operands is true. or is a short circuit operator. For example, if <i>A</i> is true in <i>A</i> or <i>B</i> then <i>B</i> is not evaluated.			
Example	<pre>var success : boolean := false var continuing := true % the type is boolean continuing := continuing or success</pre>			
	<i>continuing</i> is set to false , if and only if, both <i>continuing</i> and <i>success</i> are false . Since Turing uses short circuit operators, once <i>continuing</i> is true, <i>success</i> will not be looked at.			
Details	The or operator can be applied to natural numbers. The result is the natural number that is the bit-wise or of the operands. See nat (natural number).			
See also	boolean (which discusses true / false values), <u>explicitTrueFalseConstant</u> (which discusses the values true and <u>false</u>), <u>precedence</u> and <u>expn</u> (expression).			

Syntax	ord (<i>ch</i> : char) : int
Description	The ord function accepts an enumerated value, char , or a string of length 1, and returns the position of the value in the enumeration, or of the character in the ASCII (or EBCDIC for IBM mainframes) sequence. Values of an enumerated type are numbered left to right starting at zero. For example, ord ("A") is 65. The ord function is the inverse of chr , so for any character <i>c</i> , chr (ord (<i>c</i>)) = <i>c</i> .
See also	<u>chr</u> , <u>intstr</u> and <u>strint</u> functions.

parallelget

parallel port function

Syntax parallelget : int

Description The **parallelget** procedure is used on a PC to read the value of cert parallel port. This port corresponds to the MS-DOS device "LPT1" can be used to control robots and peripherals.

This program reads and prints the values of the five data pins of the port.

 Example
 const val : int := parallelget % Read in the se

 put "Pin 10 is: ", (val div 64) mod 2

 put "Pin 11 is: ", (val div 128) mod 2

 put "Pin 12 is: ", (val div 32) mod 2

 put "Pin 13 is: ", (val div 16) mod 2

 put "Pin 15 is: ", (val div 8) mod 2

The five pins that are used for parallel input are pins 10-15. The **pa** procedure returns the sum of

	64	Pin 10 high
	128	Pin 11 high
Details	32	Pin 12 high
	16	Pin 13 high
	8	Pin 15 high

The **mod** and **div** operators can be used to determine which pins ar

the parallelput procedure for a diagram of the pins. That procedurSee alsovalues on the parallel port.

parallelput

parallel port procedure

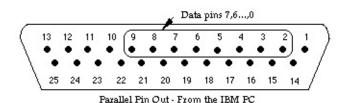
Syntax parallelput (*p* : int)

Description The **parallelput** procedure is used on a PC to set the values on the pins on the parallel port. This port corresponds to the MS-DOS dev "LPT1". This procedure can be used to control robots and peripher

This program sets data bit 0, data bit 1 and so on to data bit 7.

for i : 0 .. 7Exampleparallelput (2 ** i)%Set data bit i on tput "Data bit ", i, " or Pin ", i + 2, "hasend for

The **parallelput** procedure is used to set the eight data bits on the I parallel port. These data bits 0 - 7 correspond to pins 2 - 9 on the pport.



The value sent to **parallelput** is the sum of the following:

1	Data bit 0	16	Data bit 4
2	Data bit 1	32	Data bit 5
4	Data bit 2	64	Data bit 6
8	Data bit 3	128	Data bit 7

For example, the command **parallelput** (97) sets data bits 0, 5 and high (97 = 1 + 32 + 64) and sets the other data pins low. Because the are only 8 data pins in the parallel port, the value passed to **paralle** must be from 0 to 255.

The **parallelput** procedure is not meant for sending a stream of characters to the parallel port (for example, if you want to send the "Hello" to the printer). If you want to do this, open the file "LPT1"

Details

the **open** statement and **put** to the file.

See alsothe parallelget function, which is used to read the values of pins of
parallel port.

paramDeclaration

Details

parameter declaration

A *paramDeclaration* is one of:

Syntax	(a)	[var] <i>id</i> {, <i>id</i> } : <i>typeSpec</i>
	(b)	subprogramHeader

A parameter declaration, which is part of the header of a procedure specifies a formal parameter (see also **procedure** and **function** dec**Description**Form (a) above is the most common case. Form (b) specifies proce functions that are themselves passed as parameters.

	<pre>procedure putTitle (title : string) % The parameter declaration is: title : stri put title end putTitle</pre>
Example	<pre>procedure x (var s : array 1 * of string (*)) % Set each element of s to the null string for i : 1 upper (s) s (i) := "" end for end x</pre>

Parameters to a procedure may be declared using **var**, which mean parameter can be changed inside the procedure. For example, *s* is c procedure. If a parameter is declared without **var**, it cannot be char differs from Pascal, where non-**var** parameters can be changed.) Pa functions cannot be declared to be **var**.

Parameters declared **var** are passed by reference, which means that the value is passed to the procedure, rather than passing the actual implies that in the call *p* (*a* (*i*)), in which array element *a*(*i*) is pass *p*, a change to *i* in *p* does not change the element referred to by *p*'s parameter. Every non-scalar (not integer, subrange, real, boolean, e pointer or the **char** type) parameter is passed by reference whether declared **var**. In all other cases (scalar non-**var** parameters) the par passed by value (the actual value is copied to the procedure). The upper bound of an array or string that is a formal parameter mass an asterisk (*), as is done above for parameter *s* in procedure *x*. ' that the size of the upper bound is inherited from the corresponding parameter. Parameters declared using star are called *dynamic* parameters are called *dynamic* parameters.

The names of the formal parameters must be distinct from each oth procedure or function name, and from pervasive identifiers. Howev not be distinct from names outside of the procedure or function.

Find the zero of function f. This example illustrates form (b), which that is a function. See also *subprogramHeader*.

```
function findZero ( function f ( x : real) : rea
                                    left, right, accuracy : real ) :
                        pre sign ( f ( left ) ) not= sign ( f ( righ
                                and accuracy > 0
                        var L : real := left
                        var R : real := right
                        var M : real
                        const signLeft := sign ( f ( left ) )
Example
                        100p
                            M := (R + L) / 2
                            exit when abs (R - L) \leq accuracy
                            if signLeft =sign ( f ( M ) ) then
                                L := M
                            else
                                R := M
                            end if
                        end loop
                        result M
                    end findZero
```

Form (b) of *paramDeclaration* is used to specify formal parameter themselves procedures or functions. For example, in the *findZero* f formal parameter that is itself a function. The subprogram type can replace form (b). In particular, the header to the *findZero* function c by the following with no change in the action. The names *g* and *x* s purpose, except as place holders in the declaration of *f*.

Details

Parameters that are declared non var should, in principle, be consta

Unfortunately, there is an anomalous situation in which these can c occurs when the parameter is passed by reference, because it is a ne as a string. If the actual parameter is changed while the subprogram the formal parameter will change as well.

You can also optionally use the **register** keyword to request that the placed in a machine register. This changes form (a) to allow the op **register** keyword. The syntax for form (a) is actually:

Details

[var] [register] id {, id } : [cheat] type

In the current (1999) implementation, programs are run interpretive pseudo-code, which has no machine registers, and the **register** key ignored. See **register** for restrictions on the use of register paramet

The optional keyword **cheat** means that the parameter has a type cl Any variable or constant non scalar (in other words, items passed t can be passed to a type cheat parameter. The internal representation interpreted as a value of the specified type. This is dangerous as it j unconstrained access to the underlying computer memory.

This procedure outputs the values of *n* bytes starting at the address parameter *a*, using a parameter type cheat.

Example

procedure dump (a : cheat array 0 10000	of na
for i : 0 n - 1	
put $i, a (i) : 4$	
end for end <i>dump</i>	
<pre>var s : string := "abc" dump (s, 5) % Dumps 5 bytes, starting</pre>	ng wi

statement

pause

A pauseStatement is:

Syntax

pause expn

The **pause** statement blocks the program (or just the process in the case of a concurrent program) for a given number of simulated time units. The *expn* must be a non-negative **int** value giving the number of time units. This is analogous to the **delay** statement, which causes blocking for a given amount of real time (actual physical time).

The interpreter maintains a counter which it considers to be simulated time. The only execution that causes this counter to increase is the **pause** statement. The process executing the **pause** is blocked until the counter has counted forward the number of units given by *expn*. All other statements (except **wait**) are considered to be infinitely fast. Several processes can be executing **pause** statements simultaneously.

The use of simulated time allows Turing to be used as a simulation language in which the **pause** statement simulates the passage of time in the simulated system.

PC

Description	This unit contains the predefined subprograms that deal with direct access to the hardware under the IBM PC architecture. All routines in the PC unit are exported qualified (and thus must be prefaced with " PC .").		
Entry	<u>ParallelGet</u>	Returns the value of the pins set on the parallel port.	
Points	<u>ParallelPut</u>	Sets the values of the pins on the parallel port.	

PC.ParallelGet

Syntax **PC.ParallelGet** (*port* : **int**) : **nat1**

DescriptionThe **PC.ParallelGet** function is used to read the value of certain
pins on a parallel port. The port is specified with the *port*
parameter which can have the value 1, 2 or 3 corresponding to
"LPT1", "LPT2" and "LPT3". This procedure can be used to
control robots and peripherals.

This program reads and prints the values of the five data pins of the PC's parallel port.

Example

% Read in the set of pin values from LPT1 const val : int := PC.ParallelGet (1) put "Pin 10 is: ", (val div 64) mod 2 put "Pin 11 is: ", (val div 128) mod 2 put "Pin 12 is: ", (val div 32) mod 2 put "Pin 13 is: ", (val div 36) mod 2 put "Pin 15 is: ", (val div 8) mod 2

The five pins that are used for parallel input are pins 10-15. The **PC.ParallelGet** procedure returns the sum of

	64	Pin 10 high
	128	Pin 11 high
Details	32	Pin 12 high
Detuns	16	Pin 13 high
	8	Pin 15 high

The **mod** and **div** operators can be used to determine which pins are high or low.

Exported qualified.

StatusThis means that you can only call the function by calling
PC.ParallelGet, not by calling ParallelGet.

<u>PC.ParallelPut</u> procedure for a diagram of the pins. That

See also procedure is used to set the values on the parallel port.

PC.ParallelPut

Part of **PC** module

Syntax **PC.ParallelPut** (*port* : **int**, *value* : **int**)

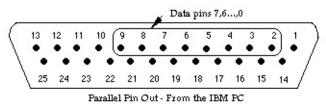
Description The **PC.ParallelPut** procedure is used on a PC to set the values on the data pins on the parallel port. The port is specified with the *port* parameter which can have the value 1, 2 or 3 corresponding tc "LPT1", "LPT2" and "LPT3". This procedure can be used to control robots and peripherals.

This program sets data bit 0, data bit 1 and so on to data bit 7.

Example

```
for i : 0 .. 7
    %Set data bit i on parallel port LPT2
    PC.ParallelPut (2, 2 ** i)
    put "Data bit ", i, " or Pin ", i + 2, "has
end for
```

The **PC.ParallelPut** procedure is used to set the eight data bits on the PC's parallel port. These data bits 0 - 7 correspond to pins 2 - 9 on the parallel port.



The value sent to **PC.ParallelPut** is the sum of the following:

1	Data bit 0	16	Data bit 4
2	Data bit 1	32	Data bit 5
4	Data bit 2	64	Data bit 6
8	Data bit 3	128	Data bit 7

For example, the command **PC.ParallelPut** (97) sets data bits 0, 5 and 6 high (97 = 1 + 32 + 64) and sets the other data pins low. Because there are only 8 data pins in the parallel port, the value passed to **PC.ParallelPut** must be from 0 to 255.

Details

	The PC.ParallelPut procedure is not meant for sending a stream of characters to the parallel port (for example, if you want to send the string "Hello" to the printer). If you want to do this, open the file "LPT1" using the open statement and put to the file.
	Exported qualified.
Status	This means that you can only call the function by calling PC.ParallelPut , not by calling ParallelPut .
See also	PC.ParallelGet function, which is used to <u>read</u> the values of pins on the parallel port.

pervasive

declaration modifier

Description	When a variable, constant, type or subprogram is declared, you car it is to be pervasive , which means that it does not need to be explic into modules, monitors or classes in the current scope. The keywor can be abbreviated as an asterisk (*).		
Example	<pre>var pervasive counter : int % Short form: var * const * maxCounter : int := 100 procedure * p (x : real) end p </pre>		
Details	The keyword pervasive is also used in export lists along with the lunqualified. See export list for details.		
See also	var <u>declaration</u> , <u>const</u> <u>declaration</u> , <u>procedure</u> <u>declaration</u> , <u>functio</u> subprogram header and <u>export</u> list for uses of <u>pervasive</u> .		

Pic

This unit contains the predefined subprograms that deal with taking pictures of part of the screen, displaying them and moving pictures from file to screen and back.

Description

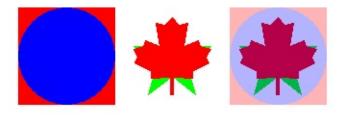
Entry Points All routines in the **Pic** unit are exported qualified (and thus must be prefaced with "**Pic**.").

New	Creates a picture from a specified portion of the screen.	
Draw	Draws a picture on the screen.	
DrawSpecial	Draws a picture on the screen using special effects.	
<u>DrawSpecialBack</u>	Draws a picture on the screen while continuing execution using special effects.	
<u>Free</u>	Frees up the picture created by using New or FileNew .	
FileNew	Creates a picture from a file in an allowed format.	
<u>Save</u>	Saves a picture as a file in an allowed format.	
ScreenLoad	Displays a file in an allowed format on the screen directly.	
ScreenSave	Saves a specified portion of the screen as a file in an allowed format.	
Rotate	Creates a new picture by rotating an existing picture.	
<u>Scale</u>	Creates a new picture by scaling an existing picture.	
<u>Flip</u>	Creates a new picture by flipping an existing picture upside-down.	
<u>Mirror</u>	Creates a new mirror-image of an existing picture.	
	Creates a new picture by blending	

<u>Blend</u>	two existing pictures together.	
Blur	Creates a new picture by blurring an existing pictures.	
<u>Width</u>	Returns the width of a picture.	
<u>Height</u>	Returns the height of a picture.	
Frames	Returns the the number of frames in a multi-frame GIF file.	
FileNewFrames	Creates an array of pictures from a multi-frame GIF file.	
DrawFrames	Draws a series of pictures on the screen in sequence while continuing execution.	
DrawFramesBack	Draws a series of pictures on the screen in sequence.	
<u>SetTransparentColor</u>	Sets the color to be ignored when using picMerge mode.	
<u>SetTransparentColour</u>	Sets the color to be ignored when using picMerge mode.	

Pic.Blend

Pic.Blend (*picID1*, *picID2*, *pct* : **int**) : **int** Syntax **Pic.Blend** is used to create a new picture by blending two identical pictures. The resulting picture is created by taking *pct* % of the firs **Description** adding it to (100 - *pct*) % of the second picture. Note that the blended picture is a newly created picture. When it is needed, its memory should be released by using **Pic.Free**. Details Note that if *pct* is 100, then the resulting picture will be identical to is 0, then the resulting picture will be identical to *picID2*. The program creates two identically sized pictures, blends them to: displays all three pictures. You can vary the blend percentage to ob results. View.Set ("graphics:340;140, nobuttonbar") % Create the original pictures var pic1, pic2, picBlended : int Draw.FillBox (0, 0, 100, 100, brightred) **Draw.FillOval** (50, 50, 50, 50, *brightblue*) pic1 := **Pic.New** (0, 0, 100, 100) cls Draw.FillStar (10, 10, 90, 90, brightgreen) **Draw.FillMapleLeaf**(10, 10, 90, 90, *brightred*) *pic2* := **Pic.New** (0, 0, 100, 100) cls % Create new picture by blending 30% from pic1, Example picBlended := Pic.Blend (pic1, pic2, 30) % Draw the three images Pic.Draw (pic1, 10, 10, picCopy) **Pic.Draw** (*pic2*, 120, 10, *picCopy*) **Pic.Draw** (*picBlended*, 230, 10, *picCopy*)



Output from the Program

Execute

Exported qualified.

Status This means that you can only call the function by calling **Pic.Blen**(**Blend**.

Pic.Blur

Syntax **Pic.Blur** (*picID*, *blurAmount* : **int**) : **int**

Description Pic.Blur is used to create a new picture by blurring an existing picture is created by mixing pixels in a picture with pixels As the *blurAmount* increases, the image grows more and more blur

Note that the blurred picture is a newly created picture. When it is needed, its memory should be released by using **Pic.Free**.

Details Note that this is a fairly CPU intensive routine. On slow machines, to a second or more when *blurAmount* is large. In such cases, it is t precompute the pictures before starting the program. The second ex this being done.

The program creates a picture and then progressively blurs it.

View.Set ("graphics:270;120, nobuttonbar") % Create the original picture var f := Font.New ("serif:60:bold,italic,noantia Font.Draw ("Turing", 10, 30, f, red) Draw.FillStar (70, 80, 90, 100, brightgreen) Draw.FillBox (240, 5, 270, 35, brightblue) var oldPic : int oldPic := **Pic.New** (0, 0, maxx, maxy) **loop var** newPic : **int** % Create the new picture by blurring the newPic := Pic.Blur (oldPic, 10) % Free the old picture so we don't run c **Pic.Free** (*oldPic*) Example **Pic.Draw** (*newPic*, 0, 0, *picCopy*) **delay** (300) oldPic := newPic end loop



Execute

By precalculating and saving the results of the blurred picture, you visual effect where an object seems to come into focus. The program below blurs an image, saving each step. It then draws reverse order, making it appear as if the image is becoming success

Execute

Exported qualified.

StatusThis means that you can only call the function by calling Pic.Blur,
Blur.

Pic.Draw

Syntax Pic.Draw (*picID*, *x*, *y*, *mode* : **int**)

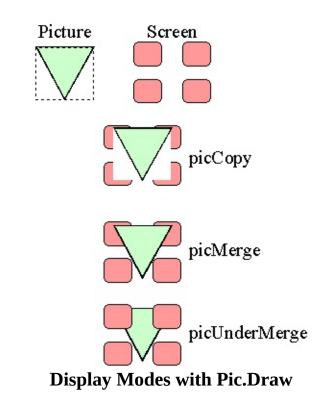
Pic.Draw is used to draw a picture on the screen. The picture is drawn with the lower left corner at (x, y).

The *mode* parameter has one of the following values:

Description	picCopy picXor	This draws the picture on top of what was underneath, obscuring it completely. This draws the picture XORing it with the background. In DOS, you can use this function to do animation. Drawing an object on top of itself with XOR erases it and restores the background.
	picMerge	This draws the picture like <i>picCopy</i> except that any occurrence of the background color in the picture is not drawn to the screen. This allows you to draw an irregularly-shaped object and draw it to the screen.
	picUnderMerge	This draws the picture, but only where the background color was displayed underneath it. The effect of this is to make the picture appear to be displayed behind the background.

Execute

If the **Pic.Draw** call fails, **Error.Last** will return a non-zero value indicating the reason for the failure. **Error.LastMsg** will return a string which contains the textual version of the error.





The program draws a graphic on the screen and then repeats it 50 times in random positions.

var picID: int var x, y : int Draw.FillBox (50, 50, 150, 150, red) Draw.FillStar (50, 50, 150, 150, green) Draw.FillOval (100, 100, 30, 30, blue) picID := Pic.New (50, 50, 150, 150) for i : 1 .. 50 x := Rand.Int (0, maxx) % Random x y := Rand.Int (0, maxy) % Random y Pic.Draw (picID, x, y, picCopy) end for Pic.Free (picID)

Example

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Pic.Draw, not by calling **Draw**.

<u>View.Update</u> for information on how to produce smooth animation.

Pic.DrawSpecial for information on how to make pictures appear**See also**using special effects such as wipes and slides.

<u>Pic.DrawFrames</u> for information on how to display multiple frame images such as are obtained from GIF files.

Pic.DrawFrames

Syntax	Pic.DrawFrames (picIds : array 1 * of int , x, y, mode : numFrames, delayBetweenFrames : int , eraseAj boolean)
Description	Pic.DrawFrames is used to draw a set of pictures stored in an arra pictures are displayed one at a time, and there is a delay of <i>delayBe</i> milliseconds between frames. The <i>x</i> , <i>y</i> , and <i>mode</i> parameters are th Pic.Draw . The <i>numFrames</i> parameter specifies the number of fran drawn (the <i>picIds</i> array must be at least this big). The <i>eraseAfter</i> pa specifies whether the last frame of the animation should be left on window when the call finishes. If <i>eraseAfter</i> is set to true , then the restored after the last picture has been drawn and <i>delayBetweenFrc</i> milliseconds has passed.
	GIF files can contain multiple frames (pictures). Animated GIFs fu frame in the GIF one after another with a delay between them.
Details	Turing allows users to load multiple frame GIF images into a series into an array using the Pic.FileNewFrames procedure. The user ca how many frames are found in the GIF file using Pic.Frames . The sequentially displayed using either Pic.DrawFrames or Pic.Draw which display the images one at a time. (Pic.DrawFrames returns images have been displayed, Pic.DrawFramesBack returns immed allowing the program to continue execution while the frames are be
	Pic.Frames is a function which returns the number of frames found it is used on a GIF that does not contain multiple images, or on a nu- file, it returns 1.
	The program loads a multiple frame GIF called "globe.gif" and dis
	% Determine the number of frames in "globe.gif" var numFrames := Pic.Frames ("globe.gif") % Load the picture var delayTime : int var pics : array 1 numFrames of int

Example	Pic.FileNewFrames ("globe.gif", <i>pics</i> , delayTime) Pic.DrawFrames (<i>pics</i> , 10, 10, <i>picCopy</i> , <i>numFrames</i> for <i>i</i> : 1 <i>numFrames</i>
	<pre>Pic.Free (pics (i)) end for</pre>

Execute

Exported qualified.

Status This means that you can only call the function by calling **Pic.Draw** by calling **DrawFrames**.

Pic.Frames for information on how to determine the number of fra image.

Pic.FileNewFramesfor information on how to load a GIF image vSee alsoframes into an array of ints.

<u>Pic.DrawFramesBack</u> for information on how to sequentially disp images stored in array of pictures while continuing to execute the p

Pic.DrawFramesBack

Suntar	Pic.DrawFramesBack (<i>picIds</i> : array 1 * of int , <i>x</i> , <i>y</i> , <i>m</i>		
Syntax	numFrames, delayBetweenFrames : int , eraseAj		
Description	Pic.DrawFramesBack is very similar to Pic.DrawFrames . The or Pic.DrawFramesBack returns immediately after being called and the frames are being drawn. This allows you to continue to draw ot being drawn.		
	For example, this procedure is necessary if you want to have two si simultaneously.		
	If you are wish to have several sets of images displayed at the same Pic.DrawFramesBack to display each set of images but the last ar display the last set so that Turing will wait until they are completed		
	If you want to draw the same set of frames continuously, do not us		
Details	<pre>for i : 1 20 Pic.DrawFrames (pics, 10, 10, picCopy, 10, 50, false % Do not use Pic.DrawFramesBack in the previous stat end for</pre>		
	If you do, Turing will attempt to run each of the calls at the same ti only run once.		
	GIF files can contain multiple frames (pictures). Animated GIFs fu after another with a delay between them.		
Details	Turing allows users to load multiple frame GIF images into a series Pic.FileNewFrames procedure. The user can determine how many using Pic.Frames . The frames can be sequentially displayed using Pic.DrawFramesBack which display the images one at a time. (Pi images have been displayed, Pic.DrawFramesBack returns immed continue execution while the frames are being displayed.		
	<u>Pic.Frames</u> is a function which returns the number of frames found		

that does not contain multiple images, or on a non-GIF image file,

The program loads a multiple frame GIF called "globe.gif" and dis

% Determine the number of frames in "globe.gif" var numFrames := Pic.Frames ("globe.gif") % Load the picture **var** delayTime : **int** var pics : array 1 .. numFrames of int Pic.FileNewFrames ("globe.gif", pics, delayTime) for i : 1 .. 50 **Pic.DrawFramesBack** (*pics*, 10, 10, *picMerge*, Pic.DrawFramesBack (pics, 20 + Pic.Width (pi Example % All 50 iterations will execute at once if % Pic.DrawFramesBack, and not Pic.DrawFrames **Pic.DrawFrames** (*pics*, 30 + 2 * Pic.Width (pi end for for *i* : 1 . . numFrames **Pic.Free** (pics (i)) end for

Execute

Exported qualified.

StatusThis means that you can only call the function by calling Pic.Draw
DrawFrames.DrawFrames.Pic.Frames for information on how to determine the number of fra
Pic.FileNewFrames for information on how to load a GIF image v
ints.See alsoPic.DrawFramesBack
pictures while continuing to execute the program.

Pic.DrawSpecial

Syntax Pic.DrawSpecial (*picID*, *x*, *y*, *mode*, *transition*, *duration* :

Pic.DrawSpecial is used to draw a picture on the screen with a spe wipe, a slide, or a fade-in. The picture is drawn with the lower left *duration* specifies how long the transition should take in millisecor fade-in could be specified to last 1/2 a second by using a duration c

The *mode* parameter is the same as in **Pic.New** and has one of the f

рісСору	This draws the picture on top of what was under completely.
picXor	This draws the picture XORing it with the backg can use this function to do animation. Drawing <i>z</i> itself with XOR erases it and restores the backgr
picMerge	This draws the picture like <i>picCopy</i> except that a background color in the picture is not drawn to t you to draw an irregularly-shaped object and drawn drawn drawn between the picture object and
picUnderMerge	This draws the picture, but only where the backg displayed underneath it. The effect of this is to n to be displayed behind the background.

The transition parameter indicates the special effect and has one of

	The picture appears as a from left to right, "reveal as the screen underneath replaced.
picWipeLeftToRight	In the picture below, "to ' the "Welcome" undernea from left to right.
	to TropicWipeLeftToRight

	transi
picWipeRightToLeft, picWipeTopToBottom, picWipeBottomToTop	As <i>picWipeLeftToRight</i> , « in different directions.
picWipeUpperLeftToLowerRight	The picture appears as a from the upper left corne right, "revealing" the new underneath the picture is In the picture below "to ". the "Welcome" undernea from upper-left to lower-
picWipeUpperRightToLowerLeft, picWipeLowerLeftToUpperRight,	picWipeUpperLeftToL through tr As picWipeLeftToRight, e in different directions
picWipeLowerRightToUpperLeft picWipeCentreToEdge, picWipeCenterToEdge	in different directions. The picture appears insid outlined box as it sweeps the edges, "revealing" the screen underneath the pic In the picture below, "to ' the "Welcome" undernea from the centre out. WTU picWipeCentreToEdge transi
picWipeLeftToRightNoBar,	

Description	picWipeRightToLeftNoBar, picWipeTopToBottomNoBar, picWipeBottomToTopNoBar picWipeUpperLeftToLowerRightNoBar, picWipeUpperRightToLowerLeftNoBar, picWipeLowerLeftToUpperRightNoBar, picWipeLowerRightToUpperLeftNoBar, picWipeCentreToEdgeNoBar, picWipeCenterToEdgeNoBar	As the constants above, ϵ to mark the sweep across looks better when display of a similar one. No bar i when the images are com two photographs, and so
	picSlideLeftToRight	The picture "slides in" from black bar sweeps from le the screen underneath off drawing area. In the picture below "to " the "Welcome" originally the "Welcome" to the rig slides in from the left. ing W picSlideLeftToRight transi
	picSlideRightToLeft, picSlideTopToBottom, picSlideBottomToTop	As <i>picSlideLeftToRight</i> , (in different directions.
	picSlideLeftToRightNoBar, picSlideRightToLeftNoBar, picSlideTopToBottomNoBar, picSlideBottomToTopNoBar	As the constants above, ϵ to mark the sweep across looks better when display of a similar one. No bar i when the images are com two photographs, and so
		The picture "grows" fron black bar sweeps from le "squeezing" the screen u edge of the drawing area

picGrowLeftToRight	In the picture below "to " the "Welcome" originally the "Welcome" to the rig "grows" from the left.
	to Turing We picGrowLeftToRight transi
picGrowRightToLeft, picGrowTopToBottom, picGrowBottomToTop	As <i>picGrowLeftToRight</i> , in different directions.
picGrowCentreToEdge, picGrowCenterToEdge	The picture "grows" fron outlined black box sweep replacing the screen unde In the picture below "to " the "Welcome" originally Turing" grows for the cei
	picGrowCentreToEdg transi
picGrowLowerLeftToUpperRight	The picture "grows" fron as a outlined black box s upward, replacing the sci In the picture below "to 7 the "Welcome" originally Turing" "grows" from the IIZALA to Turin picGrowLowerLeftToU through tr

P	17
picGrowUpperLeftToLowerRight, picGrowLowerRightToUpperLeft, picGrowUpperRightToLowerLeft	As <i>picGrowLowerLeftTo</i> the box sweeps in differe
picGrowLeftToRightNoBar, picGrowRightToLeftNoBar, picGrowTopToBottomNoBar, picGrowBottomToTopNoBar picGrowUpperLeftToLowerRightNoBar, picGrowLowerLeftToUpperRightNoBar, picGrowUpperRightToLowerLeftNoBar, picGrowLowerRightToUpperLeftNoBar, picGrowCentreToEdgeNoBar	of a similar one. No bar i when the images are cor
picFadeIn	The picture "fades in" ov underneath it. At the end new image completely re underneath it. IVATA picFadeIn - half way
picBlend	This transition is somewl doesn't take any time to ϵ <i>picBlend</i> causes the new with the background. The can be used by itself, in v image is 70% the new im image. You can also add 100 to <i>picBlend</i> , in whicl whatever was added fron image over top of the old transition of <i>picBlend</i> + 1 image that is composed c and 85% the old image.)



Details	The Pic.DrawSpecial requires a moderately fast machine to operat (Pentium III or higher). The <i>picFadeIn</i> and <i>picBlend</i> transitions do (256 color) displays. On Microsoft Windows machines, you can de of the display (the number of colors available) by selecting the <i>Dis</i> from the <i>Start</i> menu. You can also use the
	Config.Display (cdNumMaxColors)
	function to determine the bit-depth of the display (anything over 2) acceptable results).
Details	If the Pic.DrawSpecial call fails, Error.Last will return a non-zero reason for the failure. Error.LastMsg will return a string which co version of the error.
	The program draws a blue star on the screen, then a red circle, then have the two replace each other with a variety of transitions.
Evampla	<pre>var redID, blueID: int var x, y : int Draw.FillStar (50, 50, 150, 150, brightred) redID := Pic.New (50, 50, 150, 150) cls Draw.FillOval (100, 100, 50, 50, brightblue) hlueID = Pic.New (50, 50, 50, brightblue)</pre>
Example	blueID := Pic.New (50, 50, 150, 150) Pic.DrawSpecial (<i>redID</i> , 50, 50, picCopy, picWipε Pic.DrawSpecial (blueID, 50, 50, picCopy, picWip

Pic.DrawSpecial (redID, 50, 50, picCopy, picSlic Pic.DrawSpecial (blueID, 50, 50, picCopy, picSli Pic.DrawSpecial (redID, 50, 50, picCopy, picFade Pic.DrawSpecial (blueID, 50, 50, picCopy, picBle Pic.Free (redID) Pic.Free (blueID)

Execute

The following program demonstrates each of the different special ϵ pictures containing text, then pictures loaded from photographs, the

Execute

Exported qualified.

- StatusThis means that you can only call the function by calling Pic.Draw
DrawSpecial.Pic.Draw
for information on the meaning of the *mode* argument.
- See also <u>Pic.DrawSpecialBack</u> for information on how to continue execution special effect is occurring. This allows one to produce several special special effect is occurring.

Pic.DrawSpecialBack

Part of <u>Pic</u> module

Syntax	Pic.DrawSpeci	alBack (picID, x, y, mode, transition, dura
Description	Pic.DrawSpecial	Back is very similar to Pic.DrawSpecial . The on Back returns immediately after being called and j continues. This allows you to continue to draw ot g.
	For example, this effects simultaneo	procedure is necessary if you want to have two pously.
	Pic.DrawSpecial	have several images displayed at the same time u Back to display all the images but the last and the hat Turing will wait until the special effects are c
Details	If you want to draw the several special effects continuously over the Pic.DrawSpecialBack .	
		(pic1, 10, 10, picCopy, picWipeLeftToRig (pic2, 10, 10, picCopy, picFadeIn, 1000)
	If you do, Turing will not be seen.	will attempt to run each of the calls at the same ti
	slide, or a fade-in. The picture is dra	Back is used to draw a picture on the screen with Note that all the parameters are identical to the μ wn with the lower left corner at (<i>x</i> , <i>y</i>). The <i>durati</i> take in milliseconds. For example, a fade-in could n of 500.
	The <i>mode</i> parame	ter is the same as in Pic.New and has one of the f
	рісСору	This draws the picture on top of what was under This draws the picture XORing it with the backs
Description	picXor	function to do animation. Drawing an object on and restores the background.

	picMerge picUnderMerge	This draws the picture like <i>picCopy</i> except that a color in the picture is not drawn to the screen. This draws the picture, but only where the backg underneath it. The effect of this is to make the pi behind the background.
	See <u>Pic.DrawSp</u>	ecial for the list of possible values for the <i>transitic</i>
	or higher). The provide the matrix of the provident of the matrix of the	ecialBack requires a moderately fast machine to (<i>icFadeIn</i> and <i>picBlend</i> transitions do not work we ws machines, you can determine the bit-depth of 1 focting the <i>Display</i> control panel from the <i>Start</i> me
Details	Config.[Display (cdNumMaxColors)
	function to deterr results).	nine the bit-depth of the display (anything over 2
Details		pecialBack call fails, Error.Last will return a not ror.LastMsg will return a string which contains t
		ws a blue star on the screen, then a red circle, then altaneously, side-by-side.
Example	Draw.Fi redID cls Draw.Fi blueID cls Pic.Dra Pic.Dra Pic.Dra Pic.Dra Pic.Dra Pic.Dra Pic.Dra Pic.Dra Pic.Dra Pic.Dra	<pre>dID, blueID: int illStar (0, 0, 100, 100, brightred) = Pic.New (0, 0, 100, 100) illOval (50, 50, 50, 50, brightblue) := Pic.New (0, 0, 100, 100) awSpecialBack (redID, 10, 10, picCopy, pic awSpecial (blueID, 160, 10, picCopy, picWi awSpecialBack (blueID, 10, 10, picCopy, picSli awSpecialBack (redID, 10, 10, picCopy, picSli awSpecialBack (redID, 10, 10, picCopy, picSli awSpecialBack (redID, 10, 10, picCopy, picGr awSpecialBack (blueID, 10, 10, picCopy, picGr awSpecialBack (blueID, 10, 10, picCopy, picGr awSpecialBack (redID, 10, 10, picCopy, picBacSpecialBack (redID, 10, 10, picCopy, picBacSpecialBacSpecialBack (redID, 10, 10, picCopy, picBacSpecial</pre>

Pic.Free (blueID)

Execute

Exported qualified.

StatusThis means that you can only call the function by calling Pic.Draw
DrawSpecialBack.Pic.Draw
for information on the meaning of the *mode* argument.See alsoPic.DrawSpecial
effect.

Pic.FileNew

Syntax	Pic.FileNew (fileName : string) : int	
	Pic.FileNew is used to obtain a picture from a file. The Pic.FileNew procedure allocates the memory for the picture, which can be very large for pictures of large areas. The memory is freed up when the program calls Pic.Free with the picture ID. The picture can be used with the Pic.Draw and Pic.Save .	
Description	The <i>fileName</i> parameter must give the format of the file:	
	GIF files "GIF:filename" or "filename.GIF"JPG files "JPG:filename" or "filename.JPG"BMP files "BMP:filename" or "filename.BMP"	
Details	Various versions of Turing can convert different formats of files. Turing 4.1 for Windows can load BMP, GIF and JPG files.	
Details	For, multi-frame GIF files (GIF files that have several frames or pictures and are used for animation), Pic.FileNew will only load the first frame. See the Pic.FileNewFrames and Pic.Frame for information on loading and displaying a multi-frame GIF file.	
Details	If the Pic.FileNew call fails, then it returns 0. Also Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.	
	The program reads a graphic from the file <i>mypic.bmp</i> and then draws it 50 times.	
	<pre>var picID: int var x, y : int</pre>	
Example	<pre>picID := Pic.FileNew ("mypic.bmp") for i : 1 50 x := Rand.Int (0, maxx) % Random x y := Rand.Int (0, maxy) % Random y Pic.Draw (picID, x, y, picCopy)</pre>	

end for
Pic.Free (picID)

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Pic.FileNew, not by calling **FileNew**.

Pic.FileNewFrames

Syntax	Pic.FileNewFrames (<i>pathName</i> : string , var <i>picIDs</i> : arr int , var <i>delayTime</i> : int)
Description	Pic.FileNewFrames loads multiple pictures stored in a single mult image file into an array of integers. GIF files can contain multiple frames (pictures). Animated GIFs fu displaying each frame in the GIF one after another with a delay bet This delay can also be specified in the GIF file.
	Pic.FileNewFrames reads the series of frames from the multiframe turns each frame into a picture. The picture is then assigned to an e <i>picIDs</i> array. If the array is not large enough, then an error occurs a are loaded.
	Pic.FileNewFrames also reads the delay specified in the GIF file <i>a delayTime</i> to the delay in milliseconds. Note that many multiple fra do not specify a delay, in which case <i>delayTime</i> will be set to 0.
Details	In order to determine the number of frames in multiple frame GIF i use the <u>Pic.Frames</u> function. This returns a number that can be use the array that will be passed to Pic.FileNewFrames .
	<pre>var numFrames = Pic.Frames ("mypic.gif") var pics : array 1 numFrames of int var delayTime : int Pic.FileNewFrames ("mypic.gif", pics, delayTime)</pre>
	The frames can be sequentially displayed using either Pic.DrawFr Pic.DrawFramesBack which display the images one at a time. (Pic.DrawFrames returns once all the images have been displayed Pic.DrawFramesBack returns immediately allowing the program execution while the frames are being displayed.
Details	GIF files can have a transparent color. This color will be added to t palette, if not already present. Thus you may notice that maxcolor calling Pic.FileNew or Pic.FileNewFrames . The GIF image will b without the transparent color if <i>mode</i> parameter in any of the Pic.D

procedures is set to *picMerge*.

DetailsEach picture is in the array has been allocated by the system and sh
separately once the program is finished with the pictures. Failing to
up the system's memory.The program loads a multiple frame GIF called "globe.gif" and dis
% Determine the number of frames in "globe.gif"
% Create the original picture
var numFrames := Pic.Frames ("globe.gif")
% Load the picture
var delayTime : int
var pics : array 1 .. numFrames of intPic.FileNewFrames ("globe.gif", pics, delayTime)
Pic.DrawFrames (pics, 10, 10, picCopy, numFrames

Execute

Exported qualified.

StatusThis means that you can only call the function by calling **Pic.FileN**not by calling **FileNewFrames**.

<u>Pic.Frames</u> for information on how to determine the number of fra image.

Pic.DrawFramesfor information on how to sequentially display tlSee alsostored in array of pictures.

<u>Pic.DrawFramesBack</u> for information on how to sequentially disp images stored in array of pictures while continuing to execute the p

Pic.Flip

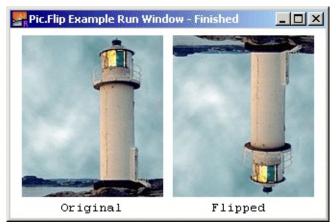
Syntax **Pic.Flip** (*picID* : int) : int

Description Pic.Flip is used to create a new picture by flipping the picture verti resulting picture is exactly the same size as the original, except it a "upside-down".

DetailsNote that the upside-down picture is a newly created picture. When
longer needed, its memory should be released by using **Pic.Free**.

The program loads a picture from a file, flips it and then draws the and the upside-down image side by side in a window after resizing window to fit the two pictures.

% Get the original picture var pic, newPic, width, height : int pic := Pic.FileNew ("lighthouse.jpg") newPic := Pic.Flip (pic) width := Pic.Width (pic) height := Pic.Height (pic) % Draw the two images: original and flipped View.Set ("graphics:" + intstr (2 * width + 30) intstr (height + 25) + ",nobuttonbar") Pic.Draw (pic, 10, 20, picCopy) Draw.Text ("Original", 50, 5, defFontID, black) Pic.Draw (newPic, 20 + Pic.Width (pic), 20, picC Draw.Text ("Flipped", 60 + Pic.Width (pic), 5, c



Output from the Program

Example

Exported qualified.

Status This means that you can only call the function by calling **Pic.Flip**, calling **Flip**.

Pic.Frames

Part of **Pic** module

Pic.Frames (*pathName* : **string**) : **int Syntax Pic.Frames** is used to determine the number of frames found in a r **Description** GIF file. GIF files can contain multiple frames (pictures). Animated GIFs fu frame in the GIF one after another with a delay between them. Turing allows users to load multiple frame GIF images into a serie: into an array using the **<u>Pic.FileNewFrames</u>** procedure. The user ca how many frames are found in the GIF file using **Pic.Frames**. The sequentially displayed using either **<u>Pic.DrawFrames</u>** or **<u>Pic.Draw</u>** Details which display the images one at a time. (**<u>Pic.DrawFrames</u>** returns images have been displayed, **Pic.DrawFramesBack** returns imme allowing the program to continue execution while the frames are by **Pic.Frames** is a function which returns the number of frames foun it is used on a GIF that does not contain multiple images, or on a ne file, it returns 1. The program loads a multiple frame GIF called "globe.gif" and dis % Determine the number of frames in "globe.gif" var numFrames := Pic.Frames ("globe.gif") % Load the picture Example **var** delayTime : **int** var pics : array 1 .. numFrames of int Pic.FileNewFrames ("globe.gif", pics, delayTime) **Pic.DrawFrames** (pics, 10, 10, picCopy, numFrames

Execute

Exported qualified.

Status This means that you can only call the function by calling **Pic.Fran** calling **Frames**.

<u>Pic.FileNewFrames</u> for information on how to load a GIF image v frames into an array of **int**s.

Pic.DrawFrames for information on how to sequentially display tlSee alsostored in array of pictures.

<u>Pic.DrawFramesBack</u> for information on how to sequentially disp images stored in array of pictures while continuing to execute the p

Pic.Free

Syntax **Pic.Free** (*picID* : int)

Pic.Free is used to release the memory allocated by **Pic.New**. It free allocated to the parameter *picID*. This means that *picID* can not be or **Draw.Save** procedure after **Pic.Free** is called.

Turing has a limited number of pictures that are available at any on (approximately 1,000). A program that continuously allocates pictures or **Pic.FileNew**) will eventually fail if it does not free the pictures or well, if a program allocates hundreds of pictures and does not free memory devoted to the pictures will not be freed and eventually the exhaust the memory on the machine.

If **Pic.Free** is passed an invalid picture ID, a fatal error occurs. If the for other (non-fatal) reasons, **Error.Last** will return a non-zero vale reason for the failure. **Error.LastMsg** will return a string which conversion of the error.

The program animates a picture moving across a background. Each about to move to a new location, a picture of the background at tha When the picture is to be moved to the next location, the backgroun over top of the picture, thus "erasing" it. The **Pic.Free** command fr background picture once it is no longer in use. Without it, the progthe picture identifiers and crash after one or two moves across the v

The animation in the example flickers slightly. See <u>View.Update</u> fo to produce flicker-free animation.

var picID, bgID : int var x, y, c, direction : int % Create the picture being moved Draw.FillBox (50, 50, 150, 150, brightred) Draw.FillStar (50, 50, 150, 150, brightgreen) Draw.FillOval (100, 100, 30, 30, brightblue) picID := Pic.New (50, 50, 150, 150)

% Create a background

```
for i : 1 .. 1000
Example
                        x := Rand.Int (0, maxx)
                        y := Rand.Int (0, maxy)
                        c := Rand.Int (9, 15) % Use bright colors
                        Draw.FillBox (x, y, x + 30, y + 30, c)
                    end for
                    x := 1
                    y := 100
                    direction := 1
                    % Main loop
                    loop
                        % Take a picture of the background
                        bgID := Pic.New (x, y, x + 100, y + 100)
                        Pic.Draw (picID, x, y, picCopy)
                                                             % Draw t
                        delay (20)
                                                             % Delay
                        Pic.Draw (bgID, x, y, picCopy)
                                                             % Draw t
                                                             % Free t
                        Pic.Free (bgID)
                        if x \le 0 or (x + 100) \ge maxx then
                            direction := -direction
                        end if
                        x += direction
                    end loop
```

Exported qualified.

Status This means that you can only call the function by calling **Pic.Free**,

Syntax **Pic.Height** (*picID* : int) : int

Pic.Height returns the width in pixels of the picture represented by *picID*.

Description This function is often used in conjunction with **Pic.Width** to obtain the dimensions of a picture loaded using **Pic.FileNew**.

The program draws loads a picture from the file *lighthouse.jpg* and proceeds to tile the entire run window with copies of the picture.

```
var pic : int := Pic.FileNew ("lighthouse.jpg")
var width : int := Pic.Width (pic)
var height : int := Pic.Height (pic)
var x, y : int := 0
loop
      exit when y > maxy
      loop
      exit when x > maxx
      Pic.Draw (pic, x, y, picCopy)
      x := x + width
    end loop
      x := 0
      y := y + height
end loop
```

Example



Output of Example Program

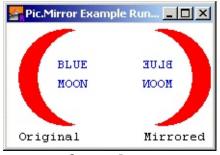
Exported qualified.

StatusThis means that you can only call the function by calling
Pic.Height, not by calling **Height**.

Pic.Width to obtain a picture's width and **Pic.FileNew** to load aSee alsopicture from a graphics file.

Pic.Mirror

Pic.Mirror (*picID* : **int**) : **int Syntax Pic.Mirror** is used to create a new picture by flipping the picture horizontally. The resulting picture is exactly the same size as the **Description** original, except it appears as a "mirror image". Note that the mirror-imaged picture is a newly created picture. When it is no longer needed, its memory should be released by Details using **Pic.Free**. The program draws a crescent on the screen, takes a picture of it and then creates a second picture by mirror-imaging the crescent. It then draws both the original picture and the mirror-imaged version. View.Set ("graphics:210;125,nobuttonbar") % Get the original picture var pic, newPic : int Draw.FillOval (50, 50, 50, 50, brightred) Draw.FillOval (70, 50, 50, 50, colorbg) Draw.Text ("BLÙE", 40, 60, defFontID, brightbluε Draw.Text ("MOON", 40, 40, defFontID, brightbluε *pic* := **Pic.New** (0, 0, 100, 100) newPic := Pic. Mirror (pic) cls % Draw the two images: original and flipped **Pic.Draw** (*pic*, 10, 20, *picCopy*) Draw.Text ("Original", 10, 5, defFontID, black) Pic.Draw (newPic, 110, 20, picCopy) Example Draw.Text ("Flipped", 140, 5, defFontID, black)



Output from the Program

Execute

Exported qualified.

StatusThis means that you can only call the function by calling
Pic.Mirror, not by calling **Mirror**.

Pic.New

Syntax	Pic.New (<i>x</i> 1, <i>y</i> 1, <i>x</i> 2, <i>y</i> 2 : int) : int
Description	Pic.New is used to obtain a picture of a portion of the screen. The Pic.New procedure allocates the memory for the picture, which can be very large for pictures of large areas. The memory is freed up when the program calls Pic.Free with the picture ID. The picture can be used with the Pic.Draw and Pic.Save .
	The picture is of the screen area defined by the rectangle $(x1, y1) - (x2, y2)$.
Details	If the Pic.New call fails, then it returns 0. Also Error.Last will return a non-zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.
	The program draws a graphic on the screen and then draws it 50 times.
Example	<pre>var picID: int var x, y : int Draw.FillBox (50, 50, 150, 150, red) Draw.FillStar (50, 50, 150, 150, green) Draw.FillOval (100, 100, 30, 30, blue) picID := Pic.New (50, 50, 150, 150) for i : 1 50 x := Rand.Int (0, maxx) % Random x y := Rand.Int (0, maxy) % Random y Pic.Draw (picID, x, y, picCopy) end for</pre>
	Pic.Free (<i>picID</i>)

Exported qualified.

StatusThis means that you can only call the function by calling
Pic.New, not by calling New.

Syntax	Pic.Rotate (<i>picID</i> , <i>angle</i> , <i>x</i> , <i>y</i> : int) : int
	Pic.Rotate is used to create a new picture by rotating an already existing picture. Rotation can either be around a specific point in the picture (often used for rotating a picure in place) or just a general rotation.
Description	The <i>angle</i> is specified in degrees. The rotation is done in a counter- clockwise direction. The original picture is not modified by the cal to Pic.Rotate and must still be freed when no longer used. The picture produced by Pic.Rotate may be a different size than the original picture.
	The (x, y) point is the point around which the rotation is to take place and is relative to the picture being rotated. If it is not important to rotate the picture in place, x and y should be set to -1, which make the new picture the minimum size required to fit the rotated image.
Details	The Pic.Rotate command can fail, in which case it returns 0. The Error.LastMsg function can then be used to obtain more information about the failure.
	If x and y are set to a point in the picture (rather than 1), it is possible for parts of the original picture to be rotated off the left an bottom edge of the new picture. This occurs because Pic.Rotate guarantees that the point specified by (x, y) in the original picture will be located at (x, y) in the rotated picture. You can avoid losing parts of the picture by making certain there is a margin of background color on the left and bottom sides of the picture.
Dataila	

Details



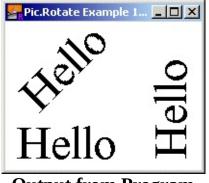
As well, any pixels in the rotated picture that were not part of the original picture are set to the background color.

Rotation can be slow on older machines. Programs that are using animation should create and store all the rotated images that may b needed. Often pictures of an object at various angles are stored in a array.

This program draws "Hello" on the screen rotated at 0, 45 and 90 degrees.

```
View.Set ("graphics:200;150,nobuttonbar")
var f : int := Font.New ("Serif:36")
Font.Draw ("Hello", 10, 10, f, black)
var pic : int := Pic.New (5, 5, 120, 45)
var pic45 : int := Pic.Rotate (pic, 45, -1, -1)
var pic90 : int := Pic.Rotate (pic, 90, -1, -1)
Pic.Draw (pic45, 5, 50, picCopy)
Pic.Draw (pic90, 150, 5, picCopy)
```

Example



Output from Program

This program moves a spinning "Hello" around the screen, bouncing it off the edges of the output window. Notice that the original picture contains adequate space on the left and bottom side to contain the rotation.

```
View.Set ("graphics:300;250,nobuttonbar")
                    var pic : array 0 .. 35 of int
                    var f : int := Font.New ("Serif:36")
                    const CTR : int := 57
                    Font.Draw ("Hello", 5, 45, f, black)
                    Draw.FillOval (CTR, CTR, 3, 3, brightred)
                    pic (0) := Pic.New (0, 0, 115, 115)
                    cls
                    for angle : 1 .. 35
                        pic (angle) := Pic.Rotate (pic (0), angle *
                    end for
Example
                    var x : int := CTR
                    var y : int := CTR
                    var dx : int := 1
                    var dy : int := 1
                    loop
                        for angle : 0 .. 35
                            Pic.Draw (pic (angle), x - CTR, y - CTR
                            if x + dx < CTR or x + dx > maxx - CTR t
                                dx := dx
                            end if
                            if y + dy < CTR or y + dy > maxy - CTR t
                                dy := dy
                            end if
                            x += dx
                            v += dv
                            delay (50)
                        end for
                    end loop
```

Details	Note that the rotated picture is a newly created picture. When it is no longer needed, its memory should be released by using Pic.Free
	Exported qualified.
Status	This means that you can only call the function by calling Pic.Rotate , not by calling Rotate .

Pic.Save

Syntax	Pic.Save (<i>picID</i> : int , <i>fileName</i> : string)
	Pic.Save is used to save a picture on the screen to a file.
Description	The <i>fileName</i> parameter must give the format of the file:
	BMP files "BMP:filename" or "filename.BMP"
Details	Various versions of Turing can save different formats of files. Turin Windows can save only BMP format files, as they are loss-less and 24-bit depth images.
Details	If Pic.Save is passed an invalid picture ID, a fatal error occurs. If the call fails for other (non-fatal) reasons, Error.Last will return a non indicating the reason for the failure. Error.LastMsg will return a s contains the textual version of the error.
	The program draws a graphic on the screen and then saves it as a E
Example	var picID: int var x, y : int Draw.FillBox (50, 50, 150, 150, red) Draw.FillStar (50, 50, 150, 150, green) Draw.FillOval (100, 100, 50, 50, blue)
	<i>picID</i> := Pic.New (50, 50, 150, 150) Pic.Save (<i>picID</i> , "BMP:mypic.dat") Pic.Free (<i>picID</i>)
	The following two programs save and load a file in BMP format.
	% Program to save a picture in mypic.bmp var picID: int var x, y : int Draw.FillBox (50, 50, 150, 150, red) Draw.FillStar (50, 50, 150, 150, green) Draw.FillOval (100, 100, 50, 50, blue) picID := Pic.New (50, 50, 150, 150) Pic.Save (picID, "mypic.bmp") Pic.Free (picID)

Example	% Program to load the picture back again and dra var picID: int var x, y : int
	<i>picID</i> := Pic.FileNew ("mypic.bmp") for i : 1 50
	x := Rand.Int (0, maxx) % Random x y := Rand.Int (0, maxy) % Random y Pic.Draw (picID, x, y, picCopy)
	end for Pic.Free (picID)

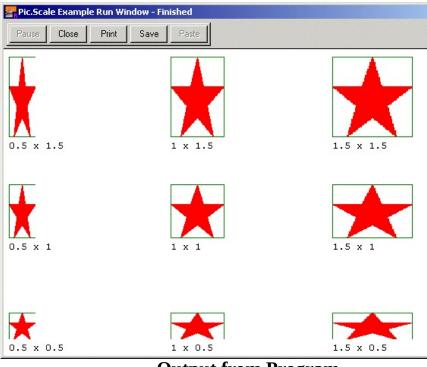
Exported qualified.

Status This means that you can only call the function by calling **Pic.Save**, calling **Save**.

Pic.Scale

Syntax	Pic.Scale (<i>picID</i> , <i>newWidth</i> , <i>newHeight</i> : int) : int	
Description	Pic.Scale is used to create a new picture by scaling (resizing) an al picture. Rotation can either be around a specific point in the picture rotating a picure in place) or just a general rotation.	
	The <i>newWidth</i> and <i>newHeight</i> parameters are the desired width and picture. The <i>newWidth</i> and <i>newHeight</i> parameters may be negative picture is mirror image (<i>newWidth</i> is negative) or upside-down (<i>new negative</i>) and the absolute values are used for the new width and he	
Details	The Pic.Scale command can fail, in which case it returns 0. The E ₁ function can then be used to obtain more information about the fail	
Details	Creating a larger picture by scaling a smaller picture will result in a with each pixel being scaled up into 2 or 3 pixels. In general image a large picture is scaled down. However, when a large picture is sca one-pixel wide lines can "disappear".	
	In order to scale a picture while retaining its original aspect ratio, s of the original width and height. The width and height of a picture using Pic.Width and Pic.Height .	
Details	Scaling can be slow on older machines. Programs that are using an create and store all the scaled images that may be needed. Often pie at various sizes are stored in an array.	
Details	Note that the scaled picture is a newly created picture. When it is n its memory should be released by using Pic.Free .	
	This program draws a set of stars scaled at 50%, 100% and 150% c Note how some single-pixel lines are removed when the picture is	
	% The "Pic.Scale Example" program. const <i>STAR_SIZE</i> : int := 70	
	var <i>pic</i> : int	

```
var newPic : int
                     var picWidth, picHeight : int
                     % Get the original picture
                     Draw.FillStar (0, 0, STAR_SIZE, STAR_SIZE, brigh
                     Draw.Box (0, 0, STAR_SIZE, STAR_SIZE, green)
                     pic := Pic.New (0, 0, STAR_SIZE, STAR_SIZE)
                     cls
Example
                     picWidth := Pic.Width (pic)
                     picHeight := Pic.Height (pic)
                     for x : 1 ... 3
                          for y : 1 .. 3
                              newPic := Pic.Scale (pic, x * picWidth d
                                  y * picHeight div 2)
                              Pic.Draw (newPic, (x - 1) * maxx div 3 +
                                   (y - 1) * \max y \operatorname{div} 3 + 20, \operatorname{picCopy})
                              Pic.Free (newPic)
                              Draw.Text (realstr (x / 2, 0) + "x" +
                                            (x - 1) * \max \operatorname{div} 3 + 5, (y)
                                   defFontID, black)
                          end for
                     end for
```



Output from Program

Exported qualified.

StatusThis means that you can only call the function by calling Pic.ScaleScale.

Pic.ScreenLoad

Syntax	Pic.ScreenLoad (fileName : string , x, y, mode : int)			
	Pic.ScreenLoad displays a picture from a file straight to the screen.			
	The <i>fileName</i> parameter must give the format of the file:			
Description	JPG files BMP files	"JPG:filename" or "filename.JPG" "BMP:filename" or "filename.BMP"		
	The <i>x</i> and <i>y</i> parameters set the lower left hand corner of the picture.			
	The <i>mode</i> parameter has one of the following values:			
	рісСору	This draws the picture on top of what was underneath, obscuring it completely.		
	picXOR	This draws the picture XORing it with the background. In DOS, you can use this function to do animation. Drawing an object on top of itself with XOR erases it and restores the background.		
	picMerge	This draws the picture like <i>picCopy</i> except that any occurrence of the background color in the picture is not drawn to the screen. This allows you to draw an irregularly-shaped object and draw it to the screen.		
	picUnderMerge	This draws the picture, but only where the background color was displayed underneath it. The effect of this is to make the picture appear to be displayed behind the background.		

Various versions of Turing can convert different formats of files. Turing 4.1 for Windows can load BMP files and JPG files. Unfortunately, due to strictly enforced patents on the GIF

Details	compression algorithm, Turing will not implement importation of GIF pictures until the patents expire in June 2004.
Details	At the time of writing, MacOOT supported only PICT files. Consult the release notes to find out which file formats are currently supported.
Details	If the Pic.ScreenLoad fails, then Error.Last will return a non- zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.
Example	The program displays a picture on the screen from the PCX file <i>mypic.BMP</i> . Pic.ScreenLoad ("mypic.bmp", 0, 0, <i>picCopy</i>)
	Exported qualified.
Status	This means that you can only call the function by calling Pic.ScreenLoad , not by calling ScreenLoad .

Pic.ScreenSave

Syntax	Pic.ScreenSave (<i>x</i> 1, <i>y</i> 1, <i>x</i> 2, <i>y</i> 2 : int , <i>fileName</i> : string)
Description	Pic.ScreenSave saves a portion of the screen into a file in a format specified by the file name.
	The picture saved to the file is the portion of the screen defined by the rectangle $(x1, y1) - (x2, y2)$.
	The <i>fileName</i> parameter must give the format of the file:
	BMP files "BMP:filename" or "filename.BMP"
Details	Various versions of Turing can save different formats of files. Turing 4.1 for Windows can save BMP. Unfortunately, due to strictly enforced patents on the GIF compression algorithm, Turing will not implement saving of GIF pictures until the patents expire in June 2004.
Details	If the Pic.ScreenSave fails, then Error.Last will return a non- zero value indicating the reason for the failure. Error.LastMsg will return a string which contains the textual version of the error.
Example	The program draws a graphic and saves it as a PICT file called <i>draw</i> .
	Draw.FillBox (50, 50, 150, 150, <i>red</i>) Draw.FillStar (50, 50, 150, 150, <i>green</i>) Draw.FillOval (100, 100, 50, 50, <i>blue</i>)
	<i>picID</i> := Pic.ScreenSave (50, 50, 150, 150, "PICI
	Exported qualified.
Status	This means that you can only call the function by calling Pic.ScreenSave , not by calling ScreenSave .

Pic.SetTransparentColor

Part of **Pic** module

Pic.SetTransparentColor (colorNumber : int)
The Pic.SetTransparentColor procedure sets the color in the pictus should be considered transparent when the picture is drawn using t picMerge or picUnderMerge modes. If no color is specified, then the default background color (colorbg , usually white) is used as the transparent color.
This call is often used when displaying images that were originally as GIFs and translated into another format. Many GIFs have a trans color that can be non-white. To use this call, you must know the co number that represents the color in the image that is to be transpare
This program displays two images loaded from the same file. In the second image, the transparent color has been set to bright red (that i parts of the image that are meant to be transparent are in bright red
<pre>var pic1 : int := Pic.FileNew ("airplane.bmp") var pic2 : int := Pic.FileNew ("airplane.bmp") Pic.SetTransparentColor (pic2, brightred) setscreen ("offscreenonly") for x : 100 maxx - 100 cls put "The lower image has the transparent col Pic.Draw (pic1, x, 150, picMerge) Pic.Draw (pic2, x, 50, picMerge) View.Update delay (5) end for</pre>

Execute

DetailsPic.SetTransparentColor.Exported qualified.StatusThis means that you can only call the function by calling
Pic.SetTransparentColor, not by calling SetTransparentColor.See alsoPic.Draw and Pic.FileNew.

Pic.Width (*picID* : **int**) : **int** Syntax **Pic.Width** returns the width in pixels of the picture represented by *picID*. Description This function is often used in conjunction with **Pic.Height** to obtain the dimensions of a picture loaded using **Pic.FileNew**. The program draws loads a picture from the file *lighthouse.jpg* and proceeds to tile the entire run window with copies of the picture.. var pic : int := Pic.FileNew ("lighthouse.jpg") var width : int := Pic.Width (pic) var height : int := Pic.Height (pic) **var** x, y : **int** := 0 **loop** Example exit when y > maxy**loop** exit when $x > \max$ **Pic.Draw** (*pic*, *x*, *y*, *picCopy*) x := x + widthend loop x := 0y := y + heightend loop Exported qualified. This means that you can only call the function by calling **Status** Pic.Width, not by calling Width. **<u>Pic.Height</u>** to obtain a picture's height and <u>**Pic.FileNew</u>** to load a</u> See also picture from a graphics file.

DescriptionThe **play** procedure is used to sound musical notes on the
computer.

This program sounds the first three notes of the C scale.

Example play ("cde")

This program plays from middle C to one octave above middle C and down again in 8th notes.

Example

play ("8cdefgab>c")
play ("<bagfedc")</pre>

Details	The play procedure takes strings containing characters that specify notes, rests, sharps, flats and duration. The notes are the letters a to g (or A to G). A rest is p (for pause). A sharp is + and a flat is The durations are 1 (whole note), 2 (half note), 4 (quarter note), 8 (eight note) and 6 (sixteenth note). The character > raises to the next octave and < lowers. For example, this is the way to play C and then C sharp one octave above middle C with a rest between them, all in sixteenth notes: play(">6cpc+"). Blanks can be used for readability and are ignored by play. Under some systems such as UNIX, the play procedure may have no effect. The current (1999) implementation does not support play.
See also	the playdone function, which is used to see if a note has finished sounding. See also the sound procedure, which makes a sound of a given frequency (Hertz) and duration (milliseconds).
	See also predefined unit Music

See also predefined unit **Music**.

play

Syntax

playdone

function

Syntax	playdone : boolean		
Description	The playdone function is used to determine when notes played by the play procedure have finished sounding.		
	This program sounds the first three notes of the C scale and outputs "All done" as soon as they are finished. Without the loop, the message would come out before the notes are finished.		
Example	<pre>play ("cde") loop exit when playdone end loop put "All done"</pre>		
Details	Under some systems such as UNIX, the playdone procedure may be meaningless.		
See also	the play procedure. See also the sound procedure which makes a sound of a given frequency (Hertz) and duration (milliseconds).		
	See also predefined unit Music.		

pointer

Syntax A *pointerType* is one of:

- (a) [**unchecked**] **pointer to** *collectionId* % Short form: ^ collectionId
- (b) [**unchecked**] **pointer to** *classId* % Short form: ^ classId
- (c) [unchecked] pointer to typeSpec % Short form: *^* typeSpec

A variable declared as a pointer type is used to locate an element o collection or class or a value of a type. The **new** statement creates a **Description** element (or value) and places the element's location in a pointer va The **free** statement destroys an element located by a pointer variab.

Using a collection, declare a list or records and allocate one record

var list : collection of record contents : string (10) Example next : pointer to list end record var first : pointer to list new list, first

Create a collection that will represent a binary tree.

var tree : collection of record name : string (10) left, right : pointer to tree end record var root : pointer to tree **new** tree, root tree (root).name := "Adam"

Using a class, create an object of that class. The object is located b

Example

Example

```
class node
    export var next, var name
    name : string ( 25 )
    next : pointer to node % Short form: next :
end node
var start : pointer to node % Short form: var st
new node, start % Short form: new start
node ( start ) . name := "Ed" % Short form: st
```

For collections and classes, a pointer is effectively a subscript (an i for that collection or class. Pointers can be assigned, compared for and passed as parameters.

The keywords **pointer to** can be replaced by the short form ^, as in

var first : ^ item

Given a pointer *p* that locates an object of class or collection *C*, the is referenced as C(p) or as the short form p . A field *f* of the object referenced as C(p).*f* or p .*f* or as the short form $p \rightarrow f$. For example, class given above, the *name* field of the object located by the *start* can be set to *Alice* by:

start -> name := "Alice"

Pointers to types use the same notation, except that pointers to type not allowed to use the form typeSpec(p). See **class** for an example use of a class with pointers.

The carat \land is called the *dereferencing operator* and has the highest precedence. For example, in $\land p.a$, the carat applies to p and not to apply \land to all of p.a, use parentheses: $\land(p.a)$. Several carats can approve, for example,

var r : ^ ^ int

declares a pointer to a pointer to an integer and $\wedge \wedge r$ is the notation referencing the integer.

A reference cannot begin with a left parenthesis, but can be surrour (\dots) , as in (q.b). If *f* is a parameterless function declared without

Details parentheses that returns a pointer, the form $\wedge f$ calls *f* before derefer the pointer.

By default, all pointers are checked. This means there is a run time make sure that references such as C(p) actually locate elements, i.e is initialized, is not **nil** and is not *dangling* (locating an object that l freed). This checking requires extra space (the implementation atta *time stamp* to each pointer and object) and time. In high-performan programs in which this extra space and time are not acceptable, the can be declared to be **unchecked**. When this is done, the program l *dangerous* and it is the programmer's responsibility to make sure th pointer usage is valid.

If this is not the case, the program becomes susceptible to uncontrc crashes.

Checked pointers cannot be assigned to unchecked pointers nor vic However, you may, at your peril, use an implementation-dependent *cheat*, to convert a checked pointer to a unchecked pointer, as in:

```
type checkedPtr : ^ R
type uncheckedPtr : unchecked ^ R
var c : checkedPtr % c is an checked po
var u : uncheckedPtr % u is an unchecked
...
u := cheat (uncheckedPtr, d) % This is a type
```

Unchecked pointers are equivalent to the pointers of the C languag are inherently error prone and cause difficult to locate bugs. An ent collection (but not a class) can be declared unchecked, in which can its pointers are implicitly unchecked. See **collection**.

See alsoinheritlists for a description of the assignability rules for pointers.
classes and collections for more details about the use of pointers. S
new and free statements. See also nil, objectclass and anyclass.

assertion

Pust	
Syntax	An postAssertion is: post trueFalseExpn
Description	A post assertion is a special form of an assert statement that is used in a procedure or function. It is used to give requirements that the body of the procedure or function is supposed to satisfy. These requirements are given by the <i>trueFalseExpn</i> . After the body has executed and just before the procedure or function returns, the <i>trueFalseExpn</i> is evaluated. If it is true, all is well and execution continues. If it is false, execution is terminated with an appropriate message. See assert statements and procedure and function declarations for more details. See also pre and invariant assertions.
Example	<pre>This function is supposed to produce an integer approximation of the square root of integer i. The post condition requires that this result, which is called answer, must be within a distance of 1 from the corresponding real number square root. function intSqrt (i : int) answer : int pre i >= 0 post abs (answer - sqrt (i)) <= 1 statements to approximate square root end intSqrt</pre>
Details	A post assertion can also be used in a module, monitor, class or process declaration to make sure that the initialization satisfies its requirements.
See also	module and process.

post

assertion

	An <i>preAssertion</i> is:				
Syntax	pre trueFalseExpn				
Description	A pre assertion is a special form of an assert statement that is used beginning of a procedure or function. It is used to give requirement caller of the procedure or functions is supposed to satisfy. These re given by the <i>trueFalseExpn</i> . The <i>trueFalseExpn</i> is evaluated. If it i well and execution continues. If it is false, execution is terminated appropriate message. See assert statements and procedure and fun declarations for more details. See also post and invariant assertion				
Example	<pre>This function computes the average of n values. Its pre condition r must be strictly positive, to avoid the possibility of dividing by zere computing the average. function average (a : array 1 * of real, n : pre n > 0 var sum : real := 0 for i : 1 n sum := sum + a (i) end for result sum / n end average</pre>				
Details	A pre assertion can also be used in a module , monitor , class or pr declaration to make sure that requirements for initialization are me				
See also	module and process.				

precedence

Turing's *precedence* rules determine the order of applying operators in an expression such as 3 + 4 * 5. These rules state, for example, that multiplication is done before addition, so this expression is equivalent to 3+(4 * 5).

Parenthesized parts of an expression are evaluated before being used. For example, in (1 + 2) * 3, the addition is done before the multiplication.

The precedence rules are defined by this table, in which operators appearing earlier in the table are applied first. For example, multiplication is applied before addition:

(1)	**, ^, #	
(2)	prefix + and -	
(3)	*, /, div, mod, rem, shr, shl	
• •	infix +, -, xor	
(5)	<, >, =, <=, >=, not=, in, not in	
	in	
(6)	not	
(7)	and	
(8)	or	
(9)	=>	(boolean
	-	implication)

Operators appearing on a single line in this table are applied from left to right. For example, *abc* is the same is (*ab*)*c*.

Here are some examples illustrating precedence, in which the left and right expressions are equivalent:

Description

1**2	(1**2)
a+b*c	a+(b*c)
a*b/c	(a*b)/c

b or c and d	b or (c and d)
x < y and $y < z$	(x < y) and $(y < z)$

The final example illustrates the fact that in Turing, parentheses are not required when combining comparisons using **and** and **or**. These would be required in the Pascal language.

The type cheat operator # is applied after subscripting, subprogram calling, dotting, and >. For example, in each of the following, # applies to the entire reference to the right.

```
#a(i)
#r.y
#p->x
```

The pointer following operator ^ is applied before subscripting, subprogram calling, dotting, and ->. For example, in the following, ^ applies to a, r and p.

```
^a(i)
^r.y
^p->x
```

Use parentheses to force \land to apply to more of the reference. For example, in \land (a(i)), the \land applies to a(i).

infix and *prefix* operators. See the *int*, *real*, *string*, *boolean*, *set*,*enum*, *char* and *char*(*n*) types.

pred	predecessor function		
Syntax	pred (<i>expn</i>)		
Description The pred function accepts an integer, character, or an enumeration value and returns the integer minus one, the previous character the previous value in the enumeration. For example, pred (6.			
	This part of a Turing program fills up array <i>a</i> with the enumerated values <i>red</i> , <i>yellow</i> , <i>green</i> , <i>red</i> , <i>yellow</i> , <i>green</i> , etc.		
Example	<pre>type colors : enum (green, yellow, red) var a : array 1 100 of colors var c : colors := colors . red for i : 1 100 a (i) := c if c = colors . green then c := colors . red else c := pred (c) end if end for</pre>		
Details	It is illegal to apply pred to the first value of an enumeration.		

See also <u>succ</u>, <u>lower</u> and <u>upper</u> functions.

prefix

operator

A *prefixOperator* is one of:

	(a)	+		% Integer and real identity
Syntax	(b)		% (does not change value) % Integer and real negation	
	(c)	not		% Not (Boolean negation)
	(d)	#		% Type cheat
	(e)	۸		% Pointer following
	_			

A *prefix operator* is placed before a value or *operand* to produce another value. For example, if the value of *x* is seven, then *-x* is negative seven. There are also infix operators such as multiplication (*) and addition (+), which are placed between two values to produce a third value. See *infix operator*.

Description The + and prefix operators can be applied only to numeric values (integer, real and natural numbers). The **not** prefix can be applied only to true/false (boolean) values. For example **not** (x > y) is equivalent to $x \le y$. The **not** operator produces **true** from **false** and **false** from **true**.

The # operators is a type cheat (see **cheat**), and the ^ operator is pointer following (see **pointer**).

See **int, real** and **boolean** types, as well as *precedence* (for the also order of applying operators) and *infix operators*.

procedure

declaration

A procedureDeclaration is:

Syntax

procedure id [(paramDeclaration {,
paramDeclaration })]
 statementsAndDeclarations
end id

DescriptionA procedure declaration creates (but does not run) a new
procedure. The name of the procedure (*id*) is given in two places,
just after **procedure** and just after **end**.

procedure greetings put "Hello world" end greetings greetings % This outputs Hello world procedure sayItAgain (msg : string, n : int) **for** *i* : 1 . . *n* put msq end for Example end savItAgain sayItAgain ("Toot", 2) % Toot is output twice procedure double (var x : real) x := 2 * xend double **var** *y* : **real** := 3.14 % This doubles the value of double (y)

The set of parameters declared with the procedure are called *formal* parameters. In the *double* procedure, for example, *x* is a formal parameter. A procedure is called (invoked) by a procedure *call statement* which consists of the procedure's name followed by the parenthesized list of *actual* parameters (if any). For example, *double*(*y*) is a call having *y* as an actual parameter. If there are no

parameters (see the *greet* procedure above), the call does not have parentheses. The keyword **procedure** can be abbreviated to **proc**.

Ordinarily, a procedure *returns* (finishes and goes back to the place where it was called) by reaching its **end**. However, the **return** statement in a procedure causes it to return immediately. Note that **return** can also be used in the main program to cause it to halt immediately.

Only parameters declared using **var** may be changed in the procedure, for example, *x* is changed in the *double* procedure. The upper bounds of arrays and strings that are parameters may be declared to be an asterisk (*). This means that the bound is that of the actual parameter. See *paramDeclaration* for details about parameters.

Details

Procedures and functions cannot be declared inside other procedures and functions.

The syntax of a *procedureDeclaration* presented above has been simplified by leaving out the optional **import** list, **pre** condition, **init** clause, **post** condition and exception handler. The full syntax is

```
procedure [ pervasive ] id
    [ ( [ paramDeclaration {, paramDeclaration }
    [ : deviceSpecification ]
    [ pre trueFalseExpn ]
    [ init id := expn {, id := expn } ]
    [ post trueFalseExpn ]
    [ exceptionHandler ]
    statementsAndDeclarations
end id
```

A procedure must be declared before being called. To allow for mutually recursive procedures, there are **forward** declarations of procedures with later declaration of each procedure **body**. See **forward** and **body** declarations for explanations.

import list, **pre** <u>condition</u>, **init** clause, **post** <u>condition</u> and *exceptionHandler* for explanations of these features. See **pervasive** for information on **pervasive** procedures. See

See also exceptionHandler. The optional *deviceSpecification* is used only in procedures declared in monitors and is used to create an *interrupt handling procedure*. See **monitor** for details.

procedureCall

statement

A procedureCall is:

Syntax

procedureId [([expn { , expn }])]

A procedure call is a statement that calls (invokes or activates) a **procedure**. If the procedure has parameters, a parenthesized list of expressions (*expns*) must follow the procedure's name (*procedureId*).

	<pre>procedure greet put "Hello" end greet</pre>
	greet % This is a call to the greet procec
Example	<pre>procedure times (var i : int, factor : int) i := factor * i end times</pre>
	var j : int times (j, 4) % Multiply j by 4

A parameter declared in the header of a procedure is a *formal* parameter. For example, *i* and *factor* above are formal parameters. Each expression in the call is an *actual* parameter. For example, *j* and 4 above are actual parameters.

If a formal parameter is declared using **var**, then the expression passed to that parameter must be a variable reference (so its value can potentially be changed by the procedure). This means, for example, that it would be illegal to pass j+3 as the first parameter to *times*. The variable reference and the formal parameter must have equivalent types (see *equivalence* for details).

Each actual parameter passed to a non-**var** formal parameter must be assignable to that parameter (see *assignability* for details). See also *procedureDeclaration*. **Details** In this explanation of *procedureCall*, we have up to this point ignored the possibility of procedures exported from modules, monitors and classes. If the procedure is being called from outside of a module or monitor *M* from which it has been exported, the syntax of the *procedureCall* is:

M . procedureId [([expn {, expn }])]

In other words, the module or monitor name and a dot must precede the procedure's name. If the procedure is being called from outside of a class from which it has been exported, the syntax of the *procedureCall* is one of:

(a) classId (p) . procedureId [([expn {, expr (b) p -> procedureId [([expn {, expn }])]

In these, *p* must the a pointer value that locates an object in the class. Form (b) is a short form for form (a).

See also

class.

process

declaration

A processDeclaration is:

Svi	ntax
Uyı	пал

process id [([paramDeclaration {,paramDeclaration }])] statementsAndDeclarations end id

A process declaration is much like a procedure declaration, but is activated by a **fork** statement rather than by a call. The **fork** Description statement starts concurrent (parallel) execution of the process while the statements following the **fork** continue to execute.

This program initiates (forks) two concurrent processes, one of which repeatedly outputs *Hi* and the other *Ho*. The resulting output is an unpredictable sequence of *Hi*'s and *Ho*'s as *greetings* executes twice concurrently, one instance with *word* set to *Hi* and the other with *word* set to *Ho*.

Example	<pre>process greetings (word : string) loop put word end loop end greetings</pre>
	fork greetings ("Hi") fork greetings ("Ho")

The **process** declaration creates a template for a process (a concurrent activity), which is activated by a **fork** statement.

A process declaration can appear wherever a module declaration is allowed except that a process declaration is not allowed in a class. The declarations and statements in a process declaration are the same as those in a procedure.

See *paramDeclaration* for details about parameters. There is an anomaly in parameters to processes, that can lead to errors. In

particular, non-**var** parameters that are non-scalars (such as strings and arrays) are passed by reference. The result is that the target of the reference may change value while the process is executing, which in turn means that the seemingly constant parameter is not really constant. For example, if the string variable *s* were passed to the *greetings* process and subsequently changed, the value of *greetings*' formal parameter would change.

Details The syntax of a *processDeclaration* presented above has been simplified by leaving out the optional stack size (*compileTimeExpn*), **import** list, **pre** condition, **init** clause, **post** condition and exception handler.

The full syntax is:

```
process [ pervasive ] id
      [ ( [ paramDeclaration {, paramDeclaratic
      [ : compileTimeExpn ]
      [ pre trueFalseExpn ]
      [ init id := expn {, id := expn } ]
      [ post trueFalseExpn ]
      [ exceptionHandler ]
      statementsAndDeclarations
end id
```

See **pervasive** for information on **pervasive** processes. The optional *compileTimeExpn* following the parameter list (if any) is used to specify the number of bytes for the process' stack.

import list, **pre** condition, **init** clause, **post** condition and**See also***exceptionHandler* for explanations of these additional features.

program

a (main) program

A program is:

Syntax	statementsAndDeclarations
Description	A Turing program consists of a list of statements and declarations.
Evample	This is a complete Turing program. It outputs <i>Alan M. Turing</i> .
Example	put "Alan M. Turing"
	This is a complete Turing program. It outputs a triangle of stars.
Example	<pre>var stars : string := "*" loop put stars stars := stars + "*" end loop</pre>
	This is a complete Turing program. It outputs <i>Hello</i> once and <i>Goodbye</i> twice.
Example	<pre>procedure sayItAgain (what : string, n : int) for i : 1 n put what end for end sayItAgain</pre>
	sayItAgain ("Hello", 1) sayItAgain ("Goodbye", 2)
Details	In a program there can be many units (see unit), one of which is the program (called the main program), the others of which are modules, monitors and classes. The main program is optionally preceded by an import list, which lists the units that it uses.
See also	<u>import</u> list.

statement

	A putStatement is:	
Syntax	<pre>put [: fileNumber ,] pu</pre>	utItem { , putItem } []
Description	is started in the output after the fi () is present, though, subsequen	er graphics, the omission of dot-dc
	· -	goes to the file specified by the ent for details). Also, output can b le, in which case all put statement
Example	% Note a	
	Statement	Output Notes
Example	<pre>put 24 24 put 1/10 put 100/10 put 5/3 1.66 put sqrt (2) put 4.86 * 10**9 put 121 : 5 put 1.37 : 6 : 3 put 1.37 : 11 : 3 : put "Say \"Hello\"" put "XX" : 4, "Y" put true and false put 1 < 2</pre>	Say "Hello" XXbbY % Blank shown as

	A single blank line is output this way:	
Example	<pre>put "" % Output null string then new line</pre>	
-	This put statement is sometimes used to close off a line that has been output piece by piece using put with dot-dot.	
	The general form of a <i>putItem</i> is one of:	
	 (a) expn [:widthExpn [:fractionWidth [:exponentWidth]]] (b) skip 	
Details	See the above examples for uses of <i>widthExpn</i> , <i>fractionWidth</i> and <i>exponentWidth</i> . For the exact meaning of these three widths, see th definitions of the functions <i>realstr</i> , <i>frealstr</i> and <i>erealstr</i> . The skip item is used to end the current output line and start a new line.	
Details	The put semantics allow put's of enum values. The values printed are the element names themselves, case sensitive. For example: type colors : enum (red, green, blue)	
	<pre>var c : colors := colors . red put c % outputs "red" (without the quotes)</pre>	
Details	The put semantics allow put's of boolean values. The values printed are either "true" or "false" (without the quotes). For example:	
	<pre>var c : boolean := true or false put c % outputs "true" (without the quotes</pre>	

fail statement

A quitStatement is:

Syntax quit [guiltyParty] [: quitReason]

The quit statement causes a program (or concurrent process) to fail. The failure (called an *exception*) either aborts the program (or process) or causes control to be passed to an exception handler.

In the *inputLines* procedure, halt the program if end of file is encountered before the string "stop" is read. Note that a **return** statement in the procedure would terminate the procedure but not the entire program.

inputLines

In the simple case, the optional *guiltyParty* and *quitReason* are omitted. The *guiltyParty* option is used to specify the position of failure. See *exceptionHandler* for an example of a **quit** statement used in conjunction with a handler. A handler, which is located at the beginning of a subprogram body, is given control when a **quit** is executed or a failure, such as division by zero, occurs in the subprogram.

quit

Example

The *guiltyParty* option is used to designate the location of the failure, for example, to tell the debugger what line is considered to be the location of the failure. A *guiltyParty* is one of:

```
(a) <
(b) >
```

If *guiltyParty* is omitted, the failure is considered to occur at the **quit** statement. If it is <, the failure is considered to occur at the call to the present subprogram. For example, if the present subprogram implements square root *sqrt* and is passed a negative argument, it can use < to specify that the caller provided a faulty argument. If *guiltyParty* is >, this means the failure has already occurred and is being passed on to the next handler or to the system. To summarize, the three possibilities for designating the location of the failure are:

(a)	<	Caller is cause of failure
(b)	>	The exception being handled is the cause.
(c)	(omitted <i>guiltyParty</i>) The present quit is the cause.	

The *quitReason* is an integer expression which is used to identify the kind of failure. If it is omitted, a default value is chosen in the following manner. If *guiltyParty* is omitted or is <, the default is 1. If *guiltyParty* is > and an exception handler is active, the default is the *quitReason* of the exception being handled. If no exception is being handled, the default is 1. In the case of program abortion, the implementation may pass the *quitReason* to the operating system or programming environment.

See also *exceptionHandler*, <u>return</u> and <u>result</u>.

Details

Rand

	This unit random 1	contains the predefined subprograms that deal with numbers.	
Description	All routines in the Rand unit are exported qualified (and thus must be prefaced with " Rand. ").		
	<u>Real</u>	Returns a random real number.	
Entry Points	Int	Returns a random integer.	
	<u>Reset</u>	Sets the seed in the default sequence to a default value.	
	<u>Set</u>	Sets the seed in the default sequence to a specified value.	
	<u>Next</u>	Returns a random real number from a sequence.	
	Seed	Sets a seed in a sequence.	

Rand.Int

Syntax	Rand.Int (low, high : int) : int
Description	The Rand.Int statement is used to create a pseudo-random integer in the range <i>low</i> to <i>high</i> , inclusive. For example, if <i>i</i> is an integer, after <i>i</i> := Rand.Int (<i>i</i> ,1, 10), <i>i</i> would have a value such as 7 or 2 or 10.
	This program simulates the repeated rolling of a six sided die.
Example	<pre>loop put "Rolled ", Rand.Int (1, 6) end loop</pre>
	The Rand.Int statement sets its parameter to the next value of a sequence of pseudo-random integers that approximates a uniform distribution over the range $low = i = high$. It is required that $low = high$.
Details	Each time a program runs, Rand.Int uses a different pseudo- random number sequence. To always get the same sequence (actually, to start the sequence at the same point), use the Rand.Set procedure.
	To use several sequences of repeatable pseudo-random number sequences, use the Rand.Seed and Rand.Next procedures.
	Exported qualified.
Status	This means that you can only call the function by calling Rand.Int , not by calling Int .
See also	Rand.Real, Rand.Set, Rand.Seed and Rand.Next.

Rand.Next

Syntax	Rand.Next (<i>seq</i> : 1 10) : real	
Description	The Rand.Next procedure is used when you need several sequences of pseudo-random numbers, and you need to be able to exactly repeat these sequences for a number of simulations. The Rand.Next procedure is the same as rand , except <i>seq</i> specifies one of ten independent and repeatable sequences of pseudo-random real numbers.	
	The Rand.Seed procedure is used to start one of these sequences at a particular point.	
	Exported qualified.	
Status	This means that you can only call the function by calling Rand.Next , not by calling Next .	
See also	Rand.Seed, Rand.Int, Rand.Real and Rand.Next.	

Rand.Real

Syntax	Rand.Real : real	
Description	The Rand.Real function returns a pseudo-random number in the range zero to one. For example, if x is a real number, after $x :=$ Rand.Real , x would have a value such as 0.729548 or 0.352879.	
	This program repeatedly and randomly prints out <i>Hi ho</i> , <i>hi ho</i> or <i>It's off to work we go</i> .	
Example	<pre>loop if Rand.Real > 0.5 then put "Hi ho, hi ho" else put "It's off to work we go" end if end loop</pre>	
	The Rand.Real function sets its parameter to the next value of a sequence of pseudo-random real numbers that approximates a uniform distribution over the range $0 < r < 1$.	
Details	Each time a program runs, Rand.Real uses a different pseudo- random number sequence. To always get the same sequence (actually, to start the sequence at the same point), use the Rand.Set procedure.	
	To use several sequences of repeatable pseudo-random number sequences, use the Rand.Seed and Rand.Next procedures.	
	Exported qualified.	
Status	This means that you can only call the function by calling Rand.Real , not by calling Real .	
See also	Rand.Int, Rand.Set, Rand.Seed and Rand.Next.	

Rand.Reset

Syntax	Rand.Reset
Description	This is a procedure with no parameters that resets the sequences of pseudo-random numbers produced by Rand.Real and Rand.Int . This allows identical executions of the same program to produce identical results.
	This program simulates the repeated rolling of a six sided die. Each time the program runs, the same sequence of rolls occurs.
Example	Rand.Reset loop put "Rolled ", Rand.Int (1, 6) end loop
	If Rand.Reset and Rand.Set are not used, each time a program runs Rand.Real and Rand.Int use a different pseudo-random number sequence. To get the same sequence each time (actually, to start the sequence at a different point), use Rand.Reset or Rand.Set .
Details	The Rand.Reset procedure can be called any time. However, to make it work, it should only be called once per program. Any call to Rand.Reset after the first one is ignored.
	To use several sequences of repeatable pseudo-random number sequences, use the Rand.Seed and Rand.Next procedures.
	Exported qualified.
Status	This means that you can only call the function by calling Rand.Reset , not by calling Reset .
See also	Rand.Set, Rand.Int, Rand.Real, Rand.Seed and Rand.Next.

Rand.Seed

Part of <u>Rand</u> module

Syntax	Rand.Seed (<i>seed</i> : nat4 , <i>seq</i> : 1 10)
Description	The Rand.Seed procedure restarts one of the sequences generated by Rand.Next . Each restart with the same seed causes Rand.Next to produce the same sequence for the given sequence.
	Exported qualified.
Status	This means that you can only call the function by calling Rand.Seed , not by calling Seed .
See also	Rand.Next, Rand.Int, Rand.Real, and Rand.Set.

Rand.Set

Syntax	Rand.Set (seed : nat4)
Description	This procedure sets the seed for sequences of pseudo-random numbers produced by Rand.Real and Rand.Int . This allows identical executions of the same program to produce identical results.
	This program simulates the repeated rolling of a six sided die. Each time the program runs, the same sequence of rolls occurs.
Example	Rand.Set (16#1234ABCD) loop put "Rolled ", Rand.Int (1, 6) end loop
Details	If Rand.Reset and Rand.Set are not used, each time a program runs Rand.Real and Rand.Int use a different pseudo-random number sequence. To get the same sequence each time (actually, to start the sequence at a different point), use Rand.Reset or Rand.Set .
	To use several sequences of repeatable pseudo-random number sequences, use the Rand.Seed and Rand.Next procedures.
Status	Exported qualified.
	This means that you can only call the function by calling Rand.Set , not by calling Set .
See also	Rand.Reset, Rand.Int, Rand.Real, Rand.Seed and Rand.Next.

random real number procedure

Syntax rand (var *r* : real)

Description The **rand** statement is used to create a pseudo-random number in the range zero to one. For example, if x is a real number, after **rand**(x), x would have a value such as 0.729548 or 0.352879.

This program repeatedly and randomly prints out *Hi ho, hi ho* or *It's off to work we go*.

	var r : real
	loop
	rand (r)
Example	if $r > 0.5$ then
	put "Hi ho, hi ho"
	else
	<pre>put "It's off to work we go"</pre>
	end if
	end loop

The **rand** statement sets its parameter to the next value of a sequence of pseudo-random real numbers that approximates a uniform distribution over the range 0 < r < 1.

Each time a program runs, **rand** uses a different pseudo-random number sequence. To get the same sequence (use **Rand.Set**).

DetailsTo use several sequences of repeatable pseudo-random number
sequences, use the randseed and randnext procedures.

In many languages, **rand** would be a function rather than a procedure. It has been designed as a procedure in Turing to respect the mathematical idea that every call to a function using the same arguments (or no arguments at all) should return the same value. If **rand** were a function, this would not be true.

randint, randomize, randseed and randnext.

See also N

See also predefined unit **Rand**.

randint	random integer procedure
Syntax	<pre>randint (var i : int, low, high : int)</pre>
Description	The randint statement is used to create a pseudo-random integer in the range <i>low</i> to <i>high</i> , inclusive. For example, if <i>i</i> is an integer, after randint (<i>i</i> ,1, 10), <i>i</i> would have a value such as 7 or 2 or 10.
	This program simulates the repeated rolling of a six sided die.
Example	<pre>var roll : int loop randint (i, 1, 6) put "Rolled ", i end loop</pre>
	The randint statement sets its parameter to the next value of a sequence of pseudo-random integers that approximates a uniform distribution over the range $low = i = high$. It is required that $low = high$.
Details	Each time a program runs, randint uses the same pseudo-random number sequence. To get a different sequence (actually, to start the sequence at a different point), use the randomize procedure.
	To use several sequences of repeatable pseudo-random number sequences, use the randseed and randnext procedures.
See also	rand, randomize, randseed and randnext.

randnext

procedure

Syntax	randnext (var <i>v</i> : real , <i>seq</i> : 1 10)
Description	The randnext procedure is used when you need several sequences of pseudo-random numbers, and you need to be able to exactly repeat these sequences for a number of simulations. The randnext procedure is the same as rand , except <i>seq</i> specifies one of ten independent and repeatable sequences of pseudo-random real numbers.
	The randseed procedure is used to start one of these sequences at a particular point.
See also	randseed, randint, rand and randnext.

randomize

Description

Syntax randomize

This procedure is obsolete. It was originally used to produce a different sequence of random numbers each time a program executed. With current versions of Turing, the random number sequence is "randomized" each time a Turing program is executed, eliminating the need for this procedure.

To reset the random number sequence and thus allow for a predetermined sequence of pseudo-random numbers, use **Rand.Set**.

randint, rand, randseed and randnext.

See also See also predefined unit **Rand**.

randseed

procedure

Syntax	randseed (<i>seed</i> : int , <i>seq</i> : 1 10)		
Description	The randseed procedure restarts one of the sequences generated by randnext . Each restart with the same seed causes randnext to produce the same sequence for the given sequence.		
See also	randnext, randint, rand, and randomize.		

read

file statement

Dangerous parts

A readStatement is:

Syntax

read : fileNumber [: status] , readItem { ,readItem }

The **read** statement inputs each of the *readItems* from the specified These items are input directly using the *binary* format that they hav the file. In other words, the items are not in source (ASCII or EBC) format. In the common case, these items have been output to the fil the **write** statement.

By contrast, the **get** and **put** statements use source format, which a can read using an ordinary text editor.

This example shows how to input a complete employee record usir **read** statement.

var employeeRecord :
 record
 name : string (30)
 pay : int
 dept : 0 .. 9
 end record
 var fileNo : int
 open : fileNo, "payroll", read
...
 read : fileNo, employeeRecord

The *fileNumber* must specify a file that is open with **read** capabilit program argument file that is implicitly opened).

The optional *status* is an **int** variable that is set to implementationdependent information about the **read**. If *status* is returned as zero, read was successful. Otherwise *status* gives information about the incomplete or failed **read** (which is not documented here). You commonly use *status* when you are reading a record or array from and you are not sure if the entire item exists on the file. If it does not

Example

exist, the **read** will fail part way through, but your program can col and diagnose the problem by inspecting *status*.

A *readItem* is:

Details	variableReference [: requestedSize [: actualSi			
	Each <i>readItem</i> specifies a variable to be read in internal form. The optional <i>requestedSize</i> is an integer value giving the number of byt data to be read. The <i>requestedSize</i> should be less than or equal to th of the item's internal form in memory (else a warning message is is If no <i>requestedSize</i> is given, the size of the item in memory is used optional <i>actualSize</i> is an int variable that is set to the number of by actually read.			
	An array, record or union may be read and written as a whole.			
	It is dangerous to read into pointer variables, as this allows the pose of creating incorrect addresses in the pointers. It is also dangerous more bytes than are in the <i>readItem</i> .			
See also	the write , open , close , seek , tell , get and put statements.			

real	the real number type		
Syntax	real		
Description	The real number type is used for numbers that have fractional parts for example, 3.14159. Real numbers can be combined by various operators such as addition (+) and multiplication (*). Real numbers can also be combined with integers (whole numbers, such as 23, 0 and -9), in which case the result is generally a real number. An integer can always be assigned to a real variable, with implicit conversion to real .		
Example	<pre>var weight, x : real var x : real := 9.83 var tax := 0.7 % The type is implicitly rea % 0.7 is a real number</pre>		
	See also <i>explicitRealConstant</i> . The int type is used instead of real , when values are whole numbers. See int for details.		
	Real numbers can be converted to integers using ceil (ceiling), floor , or round . Real numbers can be converted to strings using erealstr , frealstr , and realstr . These conversion functions correspond exactly to the formatting used for the put statement wit real numbers. Strings can be converted to real numbers using strreal . See descriptions of these conversion functions.		
Details	The predefined functions for real numbers include min , max , sqrt , sin , cons , arctan , sind , cosd , arcand , ln and exp . See the descriptions of these functions.		
	Pseudo-random sequences of real numbers can be generated using rand . See the description of this procedure.		
	The Turing Report gives a formal definition (not repeated here) of implemented real numbers in terms of their required accuracy relative to infinitely accurate (mathematical) real numbers.		

Turing implements real numbers using 8 byte floating point representation. This provides 14 to 16 decimal digits of precision and an exponent range of at least -38 .. 38. The PC and Macintosh versions of Turing have 16 decimal digits of accuracy because they use IEEE standard floating point representation.

See also <u>real</u>*n*.

n-byte real number type

(a) real4	% 4-byte real number
(b) real8	% 8-byte real number

The **realn** (*n*-byte real number) types are machine-dependent types that occupy a specified number of bytes. By contrast, the **real** type is, in principle, a machine-independent and mathematical type (however, it overflows when the exponent of the value is too large or small and it has only a limited amount of precision).

Example var width : real4 var height : real8

Turing implements the type **real** using 8 byte floating point representation. This provides 14 to 16 decimal digits of precision and an exponent range of at least -38 .. 38. The PC and Macintosh versions of Turing have 16 decimal digits of accuracy because they use IEEE standard floating point representation.

This implies that real8 and real are essentially the same type, so in practice there is no advantage to using real8 rather than real.
However, real4 has the advantage of occupying half as much space (with correspondingly reduced precision).

Arithmetic for all real types (**real**, **real4** and **real8**) is carried out with the accuracy and exponent range of 8-byte reals.

The type **real4** is sometimes called *single precision* (because it occupies a single 4-byte word) and **real8** is sometimes called *double precision*.

realn

Syntax

Details

realstr real-to-string function **realstr** (*r* : **real**, *width* : **int**) : **string** Syntax The **realstr** function is used to convert a **real** number to a string. For example, **realstr** (2.5e1, 4)="*bb25*" where *b* represents a blank. The string is an approximation to r, padded on the left with blanks as necessary to a length of *width*. The *width* parameter must be non-negative. If the *width* parameter is not large enough to represent the value of *r* it is implicitly increased as needed. The displayed value is rounded to the nearest decimal equivalent with this accuracy. In the case of a tie, the display value is rounded to the next larger value. The string **realstr** (*r*, *width*) is the same as the string **frealstr** (*r*, width, defaultfw) when r = 0 or when 1e-3 < abs (r) < 1e6, otherwise the same as **erealstr** (*r*, *width*, *defaultfw*, *defaultew*), with the following exceptions. With *realstr*, trailing fraction zeroes are omitted, and the decimal point is omitted if the entire Description fraction is zero. (These omissions take place even if the exponent part is printed.) If an exponent is printed, any plus sign and leading zeroes are omitted. Thus, whole number values are in general displayed as integers. *Defaultfw* is an implementation-defined number of fractional digits to be displayed. For most implementations, *defaultfw* will be 6. *Defaultew* is an implementation-defined number of exponent digits to be displayed. For most implementations, *defaultew* will be 2. The **realstr** function approximates the inverse of **strreal**, although round-off errors keep these from being exact inverses. the **erealstr**, **frealstr**, **strreal**, **intstr** and **strint** functions. See also

record

A recordType is:

record

Syntax

id {, id } : typeSpec
{ id {, id } : typeSpec }
record

end record

Description Each value of a record type consists of fields, one field for each name (*id*) declared inside the record. In the following example, the fields are *name*, *phoneNumber* and *address*.

	<pre>type phoneRecord : record name : string (20) phoneNumber : int address : string (50) end record</pre>		
Example	 var oneEntry : phoneRecord var phoneBook : array 1 100 of phoneRecord var i : int oneEntry .name := "Turing, Alan" phoneBook (i) := oneEntry % Assign whole recor		
Details	In a record, <i>id</i> 's of fields must be distinct. However, these need not be distinct from identifiers outside the record. Records can be assigned as a whole (to records of an equivalent type), but they cannot be compared. A semicolon can optionally follow each <i>typeSpec</i> .		
	Any array contained in a record must have bounds that are known at compile time.		
	The notation > can be used to access record fields. For example, if <i>p</i> is a pointer to <i>phoneRecord</i> , <i>p</i> > <i>name</i> locates the <i>name</i> field. See pointer .		

register

Dirty

Description	When a variable, constant or parameter is declared, you can request that the item be placed in a machine register. This should be done only for programs requiring considerable efficiency.		
Example	<pre>var register counter : int const register maxCounter : int := 100 procedure p (register x : real) end p end p</pre>		
Details	Items can be requested to be in registers only if they are local to a subprogram (not global variables, declared in the main program, a module, monitor or class). Items requested to be in registers cannot be bound to, passed to reference parameters, have their address taken by addr , or have certain type cheats applied to them (since a machine register has no address). The request to use a register may be ignored. For example, the current (1999) interpretive implementation uses pseudo-code, which has no machine registers, and so ignores the register		
	keyword. For the syntax of using this keyword, see var declaration, const declaration and paramDeclaration .		

rem	remainder operator		
Syntax	rem		
Description	The rem (<i>remainder</i>] operator produces the remainder of one number divided by another. For example, 7 rem 2 produces 1 and -12 rem 5 produces -2.		
Evomala	In this example, <i>eggCount</i> is the total number of eggs. The first put statement determines how many dozen eggs there are. The second put statement determines how many extra eggs there are beyond the last dozen.		
Example	var eggCount : int get eggCount put "You have ", eggCount div 12, " dozen eggs" put "You have ", eggCount rem 12, " left over"		
See also	<i>infix operators, <u>precedence</u></i> of operators and the mod and div operators.		

repeat

make copies of string

function

Syntax	repeat (<i>s</i> : string , <i>i</i> : int) : string		
Description	The repeat function returns <i>i</i> copies of string <i>s</i> catenated together. For example, repeat ("X", 4) is <i>XXXX</i> .		
	This program outputs <i>HoHoHo</i> .		
Example	<pre>var word : string := "Ho" put repeat (word, 3)</pre>		
Details	If <i>i</i> is less than or equal to zero, the null string "" is returned. The repeat function is often used for spacing of output. For example, this statement skips 20 blanks before outputting <i>x</i> . put repeat (" ", 20), <i>x</i>		

statement

result

A resultStatement is:

Syntax	result <i>expn</i>	
Description	A result statement, which must appear only in a function , is used t provide the value of the function.	
	This function doubles its parameter.	
Example	<pre>function double (x : real) : real result 2 * x end double put double (5.3) % This outputs 10.6</pre>	
	This function finds the position of a name in a list.	
Example	<pre>function find (a : array 1 100 of string) : for i : 1 100 if a (i) = name then</pre>	
	The execution of a result statement computes the value of the expression (<i>expn</i>) and terminates the function, returning the value a the value of the function.	
Details	The expression must be assignable to the result type of the function example, in <i>double</i> , $2*x$ is assignable to real . (See the <i>assignmentStatement</i> for the definition of assignable.)	
	A function must terminate by executing a result statement and not reaching the end of the function.	

return

statement

A returnStatement is:

Syntax	return		
Description	A return statement terminates the procedure (or main program) in which it appears. Ordinarily, a procedure (or main program) terminates by reaching its end; the return statement is used to cause early termination.		
	This procedure takes no action if the <i>errorHasOccurred</i> flag has been set to true.		
Example	procedure double if errorHasOccurred then return % Terminate this procedure end if handle usual case in this procedure end double		
Details	A return must not appear as a statement in (the outermost level of) a module, nor can it appear in a function .		

RGB

This unit contains the predefined constants for the basic colors and the subprograms to change the color palette.

Description All subprograms in the **RGB** unit are exported qualified (and thus must be prefaced with "**RGB**."). All the color constants are exported unqualified and thus do not need the **RGB** prefix.

For a program that displays the 256 predefined colors (and their associated color numbers) in Turing, run the "Colors.t" program.

Details

The following names can be used for colors. They represent color numbers and thus will not be accurate if **RGB.SetColor** has been used to change color numbers 0 through 15.

black, blue, green, cyan, red, magenta, purple, brown, white, gray, grey, brightblue, brightgreen, brightcyan, brightred, brightmagenta, brightpurple, yellow, brightwhite, darkgray, darkgrey

Details

The remaining four colors represent the foreground color (black on Windows or the Macintosh, white on a DOS system) and the background color (white on Windows or Macintosh, black on a DOS system). Using these four colors allows you to write programs that work on both Windows and DOS versions of Turing.

colorfg, colourfg, colorbg, colourbg

Turing uses a *palette* to access colors for display. This palette is a

list of color numbers and their associated actual color. The Turing palette starts with 256 entries (the color numbers from 0 to 255). With 16, 24, and 32-bit color displays, however, a Turing program can display thousands of colors at once. These colors, however, will all have color numbers associated with them. For example, if a Turing program loads a JPEG image, there may be thousands of colors on the screen, but the number of color numbers will stay at 256. Likewise, 16, 24, and 32-bit color displays, if you load a picture in GIF format that has a 256 colors that are different from the initial Turing color palette, they will appear in the run window. However, they will not be added to Turing's color palette.

If you attempt to determine the color number of a particular pixel that does not match any of the colors in Turing's color palette, then Turing will return the color number of the color in the Turing palette that most closely matches that color of the pixel.

	<u>GetColor</u> <u>GetColour</u>	Gets the current red, green and blue values of a specified color number.
Entry Points	<u>SetColor</u> <u>SetColour</u>	Sets the red, green and blue values of a specified color number.
	<u>AddColor</u> <u>AddColour</u>	Creates a new color number with a specified red, green and blue value.

Details

RGB.AddColor

Part of **<u>RGB</u>** module

Syntax	RGB.AddColor (redComp, greenComp, blueComp : real) : int		
Description	The RGB.AddColor function attempts to create a new color with the red, green and blue components specified. If successful, the function returns a new color number (usually one greater than maxcolor) and maxcolor is updated by adding 1 to it. If it is unsuccessful, the function returns 1 and Error.Last and Error.LastMsg can be used to determine the cause of the problem.		
	The red, green and blue values must normalized to be between 0 and 1. Thus to add the pure red to the color palette, you would call:		
	newColor := RGB.AddColor (1.0, 0.0, 0.0)		
	<i>newColor</i> would be set to the color added, or 1 if the attempt to add a color failed.		
	This program adds a palette of 16 blues to the end of the color palette.		
Example	<pre>var clr : int for blueShade : 0 15 clr = RGB.AddColor (0, 0, blueShade / 15) if clr = 1 then put "Color add failed on shade number ", exit else put "Added color number ", clr end if end for</pre>		
	Details RGB.AddColour is an alternate spelling for RGB.AddColor .		

Exported qualified.

Status This means that you can only call the function by calling

RGB.AddColor, not by calling **AddColor**.

See also **<u>RGB.GetColor</u>** and <u>**RGB.SetColor**</u>.

RGB.GetColor

Syntax	RGB.GetColor (colorNumber : int, var redComp, greenComp, blueComp : real)		
Description	The RGB.GetColor procedure returns the red, green and blue components to the color associated with the <i>colorNumber</i> parameter. The red, green and blue values are normalized to be between 0 and 1. Thus color white returns 1.0 for the <i>redComp</i> , <i>greenComp</i> and <i>blueComp</i> values and color black returns 0.0 for all three.		
Example	This program gets the components of all the available colors. put "Color Red Green Blue" for clr: 0 maxcolor var redComp, greenComp, blueComp : int RGB.GetColor (clr, redComp, greenComp, blueC put clr: 4, " ", redComp : 6 : 4, " ", G blueComp : 6 : 4 end for Details RGB.GetColor.		
	Exported qualified.		
Status	This means that you can only call the function by calling RGB.GetColor , not by calling GetColor .		
See also	RGB.SetColor and RGB.AddColor .		

RGB.maxcolor

Syntax	maxcolor : int		
Description	The maxcolor function is used to determine the maximum color number for the current mode of the screen. The alternate spelling is maxcolour .		
	This program outputs the maximum color number.		
Example	<pre>setscreen ("graphics")</pre>		
ľ	 put "The maximum color number is ", maxcolor		
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. If it is not, it will automatically be set to " <i>screen</i> " mode. See View.Set for details.		
	For IBM PC compatibles in " <i>screen</i> " mode, maxcolor = 15. For the default IBM PC compatible " <i>graphics</i> " mode (CGA), maxcolor = 3.		
Status	Exported unqualified.		
	This means that you can call the function by calling maxcolor or by calling RGB.maxcolor .		
See also	Draw.Dot for examples of the use of maxcolor . See the Text.Color procedure which is used for setting the currently-active <u>color</u> .		

RGB.SetColor

	RGB.SetColor (colorNumber: int,
Syntax	redComp, greenComp, blueComp : real)
Description	The RGB.SetColor function sets the red, green and blue components of the color associated with the <i>colorNumber</i> parameter. The red, green and blue values must normalized to be between 0 and 1. Thus to set the color associated with the <i>colorNumber</i> parameter to pure red, you would call: RGB.SetColor (<i>colorNumber</i> , 1.0, 0.0, 0.0) It is wise to use Error.Last and Error.LastMsg to check to see if the color change is successful.

This program sets all the available colors to shades of red

 for clr : 0 .. maxcolor

 if not RGB.SetColor (clr, clr / maxcolor, 6

 put "Color set failed on color number ",

 exit

 end if

 Details

 RGB.SetColour is an alternate spelling for

RGB.SetColor.

Example

Exported qualified.

StatusThis means that you can only call the function by calling
RGB.SetColor, not by calling SetColor.

See also **<u>RGB.GetColor</u>** and <u>**RGB.AddColor**</u>.

round

real-to-integer function

Syntax	round (r : real) : int
Description	The round function is used to convert a real number to an integer. The result is the nearest integer to <i>r</i> . In the case of a tie, the numerically larger value is returned. For example, round (3) is 3, round (2.85) is 3 and round (-8.43) is -8.
See also	the <u>floor</u> and <u>ceil</u> functions.

scalar

A *scalarType* is one of:

	(a)	standardType	% int, real, boolean or string	
	(b)	enumeratedType		
	(C)	subrangeType		
	(d)	pointerType		
Syntax	(e)	char		
	(f)	int n		
	(g)	natn		
	(h)	real n		
	(i)	namedType		% Must name one of the above types

Scalar types are sometimes called *simple* or *primitive* types. The non-scalar types are strings, sets, arrays, records, unions and in OOT char(n). They are defined using scalar types. Scalar types are passed by value to parameters, while non-scalars are passed by reference (by passing an implicit pointer to the non-scalar value).

Description In current Turing implementations scalar types are directly represented in 1, 2, 4 or 8 bytes in a computer's memory. This implies that they can be efficiently passed by value.

(file) statement

A *seekStatement* is one of:

Syntax

- (a) **seek** : fileNumber , filePosition
- (b) **seek** : *fileNumber* , *

Random access of both source (ASCII or EBCDIC) and internal form (binary) files is provided by the **seek** and **tell** statements. The **seek** statement repositions the specified file so that the next input/output operation will begin at the specified point (*filePosition*) in the file.

The *fileNumber* must specify a file that is open with seek capability. The *filePosition* is a non-negative integer offset in bytes from the beginning of the file. Usually, this is a number returned by the tell statement. (The first position in the file is position zero.)

Form (b) specifies that the next operation is to begin at the position immediately following the current end of the file. A *filePosition* of zero specifies that the next operation is to start at the beginning of the file. Seeking to a position beyond the current end of the file and then writing, automatically fills the intervening positions with the internal representation of zero.

This example shows how to use **seek** to append to the end of a file.

```
var employeeRecord :<br/>record<br/>name : string ( 30 )<br/>pay : int<br/>end record<br/>var fileNo : int<br/>open : fileNo, "payroll", write, seek, mod<br/>seek : fileNo, * % Seek to the end of the<br/>write : fileNo, employeeRecord<br/>% This record is added to the end
```

seek

See alsoread, write, open, close, tell, get and put statements. Another
example use of seek is given with the explanation of the tell
statement.

self	pointer to current object		
Syntax	self		
Description	The self function produces a pointer to the current object. This function can be used only inside a class declaration. See class .		
	Enter the current object onto a list of displayable objects. The module called <i>displayable</i> has exported a procedure called <i>enter</i> whose parameter type is pointer to anyclass . Since self is a pointer to <i>C</i> and <i>C</i> is a descendant of anyclass , it is legal to pass self to <i>displayable.enter</i> .		
Example	class C import displayable		
	displayable.enter (self) …		
	end C		
Details	It is illegal to call the exported entries of a class until the current object has been completely initialized, so, many calls to the current object using self will not be legal.		
	The notation to call exported subprogram p of an enclosing class C or of its ancestor D , is $C.p$ or $D.p$. Calls of this form, which can appear only within class C , call the subprogram in C (or in D in the case of $D.p$) regardless of the object type, or of any overriding, or of the status of initialization.		

separator

between tokens in a program

A Turing program is made up of a sequence of *tokens* (see *tokens*), such as var, x, :, and int. These tokens may have *separators* between them. A separator is a comment (see *comment*), blank, tab, form feed or an end of line.

	A <i>setType</i> is:	
Syntax	set of typeSpec	
Description	Each value of a set type consists of a set of elements. The <i>typeSpec</i> , which is restricted to being a subrange or an enumerated type, gives the type of these elements.	
	The <i>smallSet</i> type is declared so that it can contain any and all of the values 0, 1 and 2. Variable <i>s</i> is initialized to be the set containing 1 and 2.	
Example	type smallSet : set of 0 2 var s : smallSet := smallSet (0, 1)	
	 if 2 in <i>s</i> then	
	In classical mathematics, the set consisting of 0 and 1 is written as $\{0,1\}$. This is written in Turing using a <i>set constructor</i> consisting of the name of the set type followed by a parenthesized list of elements, which in this example is <i>smallInt</i> (0,1). The empty set is written, for example, as <i>smallInt</i> (). The full set is written as <i>smallInt</i> (all), so <i>smallInt</i> (all) = <i>smallInt</i> (0,1,2).	
	Sets can be assigned as a whole (to sets of an equivalent type). See also <i>equivalence</i> of types.	
Details	The operators to combine two sets are union (+), intersection (*), set subtraction (-), equality (=), inequality (not =), subset (<=), strict subset (<), superset (>=), strict superset (>), and xor ("exclusive or" also known as symmetric difference). Only sets with equivalent types (equal bounds on their index types) can be combined by these operators. The operators which determine if an element is, or is not, in a set are in and not in . For example, the test to see if 2 is in set <i>s</i> is written in the above example as: 2 in <i>s</i> .	
	The <i>indexType</i> of a set type must contain at least one element. For	

	example, the range 1 0 would not be allowed. See also <i>indexType</i> . In Turing, sets are limited to at most 31 elements. OOT allows a very large number of elements.		
	It is illegal to declare an "anonymous" set. The only legal declaration for an set is in a type declaration. For example, the following is now illegal:		
Details	var <i>a</i> : array 1 10 of set of 0 3		
	Given that there is no (easy) way of generating a set value without it being a named type, this should not impact any but the most bizarre code.		
See also	<i>precedence</i> of operators for the order of applying <u>set</u> operations.		

setConstructor

A setConstructor is:

Syntax setTypeId (membersOfSet) Each value of a **set** type consists of a set of elements. In classical mathematics, the set consisting of 0 and 1 is written as $\{0,1\}$. This is written in Turing using a *set constructor* consisting of the name Description of the set type (*setTypeId*) followed by a parenthesized list of elements. The *smallSet* type is declared so that it can contain any and all of the values 0, 1 and 2. Variable *s* is initialized to be the set containing 1 and 2. The set {0,1} is written in this Turing example as *smallInt* (0,1). Example type smallSet : set of 0 .. 2 var s : smallSet := smallSet (0, 1) if 2 in s then ... The form of *membersOfSet* is one of: (a) expn { , expn} % List of members of set (b) **all** % All member of index type of $s\epsilon$ % Nothing, meaning the empty set (C) The empty set is written, for example, as *smallInt* (). The full set is written as *smallInt* (**all**), so *smallInt* (**all**) = *smallInt* (0,1,2). See also the **set** type. Details The syntax of *setConstructor* as given above has been simplified by ignoring the fact that **set** types can be exported from modules. When a **set** type is exported and used outside of a module, you must write the module name, a dot and then the type name. For example, the set constructor above would be written as

m.smallSet(1,2), where *m* is the module name.

setpriority

procedure

Syntax	setpriority (<i>p</i> : nat)
Description	The setpriority procedure is used to set the priority of a process in a concurrent program. This priority cannot be counted on to guarantee critical access to shared variables. A smaller value of p means increased speed. The argument to setpriority may be limited to the range 0 to 2**15 - 1.
	getpriority, fork and monitor.
See also	See also predefined unit Concurrency .

setscreen

setscreen ("graphics:300;100")

This program outputs the square roots for the first 200 numbers. The user can inspect all the output and print the values after the program has finished execution

```
setscreen ("text")
for value : 1 .. 200
    put value : 3, " ", sqrt (value)
end for
```

This program creates a window without a button bar at the top that is sized to fit the screen. It then draws an "X" in red in the window.

```
setscreen ("graphics:max;max,nobuttonbar")
drawline (0, 0, maxx, maxy, red)
drawline (maxx, 0, 0, maxy, red)
```

Syntax setscreen (*s* : string)

Here are example uses of the **setscreen** procedure. In many cases, 1 appear as the first statement of the program. They can, however, applace in a program.

Example	setscreen (setscreen ("graphics") "graphics:400;300" "nocursor")	% Turn off c
	setscreen ("noecho")	% Do not echo ke

The setscreen statement is used to change the mode of the screen, the way in which Turing does input and output. The parameter to s a string, such as "graphics". The string contains one or more option separated by commas, such as "text, noecho".

Users should look at **View.Set** in order to find out the implementat: specified details of **setscreen** on their systems.

There are two screen modes, **text** and **graphics**. In **text** mode, outp window is only allowed using **put** and **get**. Attempting to use grap moving the cursor using **locate** will result in an error. However, a v **text** mode will keep all text output sent to it. Any output that scroll edge of the window can be read by adjusting the scroll bars of the v Printing a **text** output window prints all the output sent to the wind just the output currently visible in the window. Programs often use when they are displaying large amounts of text data that the user m view or print later. Saving the contents of a **text** output window cre file containing all the output sent to the window.

In **graphics** mode, a program can use both **put** and **get**, **locate** and commands to send output to the window. However, any output that the window is lost. Printing the window outputs the current content window. Saving a **graphics** output window saves the current conte window as a BMP file.

Where the options to **setscreen** are mutually exclusive, they are lis with the default underlined. Here are the options:

"**graphics**", "**text**", Sets the mode of the window to the given mode can for the window can be specified at the same time. If the windo[¬] size changes, then the window will be cleared.

To specify the size of a window in **graphics** mode, the **graphics** ca modifier in the form "graphics:<*width*>;<*height*>". This sets the wi be <*width*> pixels by <*height*> pixels in size. The maximum size o is the size of the screen.

Details To specify the size of a window in **text** mode, the **text** can have a n the form "text:<*rows*>;<*columns*>". This sets the window to be <*rc* lines in height by <*columns*> characters in width. The maximum si window is the size of the screen.

To set a window to the maximum size available on the screen, you "max" for the *<width>*, *<height>*, *<rows>* or *<columns>* parameter

"**cursor**", "**nocursor**" Causes the cursor to be shown (or hidden). T only appears when the program is waiting for keyboard input from

"**echo**", "**noecho**" Causes (or suppresses) echoing of characters tha Echoing is commonly turned off in interactive programs to keep ty characters from being echoed at inappropriate places on the screen.

"**nooffscreenonly**", "**offscreenonly**" Causes or (suppresses) output being sent to the visible window. When the **offscreenonly** option is any text and graphics output is drawn to the offscreen buffer that is maintained for every Run window but not drawn to the screen. **Vie** is then used to copy the entire contents of the offscreen buffer to th window. By allowing numerous drawing commands to be sent to th offscreen buffer and then updating the window at one time, it is pos get smoother animation.

"**msdos**", "**nomsdos**" Causes the window to use the MS-DOS chara (with line drawing characters) instead of the Windows ANSI chara The "**nomsdos**" option causes the window to use the Windows AN character set. Note that the "**msdos**" option only works if the Wind (usually Courier New) supports it.

"**buttonbar**", "**nobuttonbar**" Causes or (suppresses) the display of bar at the top of the output window which allows the user to easily program execution or save and print the output window.

ExampleThis program creates a graphics window that is 300 pixels by 100 µSee alsothe Draw module, the draw... subprograms and the View.Update
for doing smooth animation. See also View.Set for further setscree
including positioning the output window on the screen.

Syntax	A shl B		
Description	The shl (shift left) operator produces the value of <i>A</i> shifted <i>B</i> bits to the left. Both A and B must be non-negative integers (natural numbers).		
Example	Assign the base 2 value 11 to i and then shift it left by 2 places and assign the resulting base 2 value 1100 to j .		
	var i, j : int i := 2 # 11 % 2#11 = 3 (base 10) j := i shl 2 % j becomes 2#1100 = 12 (bas		
Details	The shl operator is defined mathematically (in a machine- independent way) as follows: A shl $B = A * (2^{**}B)$. Overflow occurs when the result exceeds the maximum value of the nat4 (4-byte natural number) type.		
	Value <i>A</i> can be of any integer type (as long as it is non-negative) or any natural number type.		
	The shl operator has the same precedence as the * operator.		
See also	shr (shift right), or, and and xor, which also are bit manipulation operators that act on non-negative values. See also explicitIntegerConstant which describes values such as 2#1100.		

shl

shift right operator

Syntax	A shr B
Description	The shr (shift right) operator produces the value of <i>A</i> shifted <i>B</i> bits to the right. Both A and B must be non-negative integers (natural numbers).
	Assign the base 2 value 1101 to i and then shift it right by 2 places and assign the resulting base 2 value 11 to j .
Example	var i, j : int i := 2 # 1101 % 2#1101 = 13 (base 10) j := i shr 2 % j becomes 2#11 = 3 (base 1
	The shr operator is defined mathematically (in a machine- independent way) as follows: A shr $B = A$ div $2^{**}B$.
Details	Value <i>A</i> can be of any integer type (as long as it is non-negative) or any natural number type.
	The shr operator has the same precedence as the * operator.
See also	shl (shift left), or, and and xor, which also are bit manipulation operators that act on non-negative values. See also explicitIntegerConstant which describes values such as 2#1101.

shr

Syntax	sign (<i>r</i> : real) : -1 1
Description	The sign function is used to determine whether a number is positive, zero or negative. It returns 1 if $r > 0$, 0 if $r = 0$, and -1 if $r < 0$. For example, sign (5) is 1 and sign (-23) is -1.
	This program reads in numbers and determines if they are positive, zero or negative:
Example	<pre>var x : real get x case sign (x) of label 1 : put "Positive" label 0 : put "Zero" label -1 : put "Negative" end case</pre>
	See also prodefined unit Math

See also See also predefined unit Math.

sign

wake up a process statement

A signalStatement is:

signal variableReference

A signal statement is used in a concurrent program to wake up a process that is blocked (waiting on a condition variable). The statement can only be used inside a monitor (a special kind of module that handles concurrency). A signal statement operates on a condition variable (the *variableReference*), which is essentially a queue of sleeping processes. See condition for an example of a signal statement.

A signal statement wakes up one process that is doing a wait on the specified condition queue, if such a process exists. If the condition is deferred (or timeout; see condition), the signaler continues in the monitor, and the awakened process is allowed to continue only when the monitor becomes inactive. A signal to an *immediate* (non-deferred) condition causes the signaled process to begin running in the monitor immediately. The signaling process waits to re-enter the monitor when the monitor becomes inactive.

conditionand wait. See also monitor and fork. See also empty.See alsoSee also pause.

signal

Syntax

simutime

simulated time function

Syntax	simutime : int
Description	The simutime function returns the number of simulated time units that have passed since program execution began.
Details	Simulated time only passes when all process are either paused or waiting. This simulates the fact that CPU time is effectively infinitely fast when compared to "pause" time.
Example	This prints out the simulated time passing between two processes. This will print out 3, 5, 6, 9, 10, 12, 15, 15, 18, 20, 21, process p (t : int) loop pause t put simutime end loop end p fork p (3) fork p (5)

Execute

See also See also predefined unit <u>Concurrency</u>.

sine function (radians)

Description The **sin** function is used to find the sine of an angle given in radians. For example, **sin** (0) is 0.

This program prints out the sine of p/6, 2p/6, 3p/6, up to 12p/6 radians.

Example	<pre>const pi := 3.14159 for i : 1 12</pre>
	const angle := i * pi / 6
	<pre>put "Sin of ", angle, " is ", sin (angle)</pre>
	end for

Execute

	the <u>cos</u> and <u>tan</u> functions for calculating cosine and tangent.
See also	the <u>sind</u> function which finds the sine of an angle given in degrees. (2p radians are the same as 360 degrees.)
	See also predefined unit <u>Math</u> .

sin

sine function (degrees)

Syntaxsind (r : real) : realDescriptionThe sind function is used to find the sine of an angle given in
degrees. For example, sind (0) is 0.This program prints out the sine of 30, 60, 90, up to 360 degrees.For i : 1 ... 12
const angle := i * 30
put "Sin of ", angle, " is ", sind (angle)
end for

Execute

the **<u>cosd</u>** and **<u>tand</u>** functions for calculating cosine and tangent.

the sin function which finds the sine of an angle given in radians.(2p radians are the same as 360 degrees.)

See also predefined unit **Math**.

sind

012001	51
Dirty	
Syntax	sizeof (typeNameOrVariableReference)
Description	The sizeof attribute is used to find the number of bytes used to represent the type or variable. This is implementation-dependent (dirty).
	The size of int2 and nat2 is 2.
Example	<pre>var i : int2 const nat2size := sizeof (i) % size is 2</pre>
	The <i>typeNameOrVariableReference</i> must be the name of a user- defined type, a variable reference, a basic type (such as real), or a constant.
Details	In principle, sizeof returns the number of <i>storage units</i> which would not necessarily be 8-bit bytes. For example, in some older machines, such as the CDC 6000 series, the storage units are 60

ne older are 60 Detalls bit words. However, almost all modern computers use 8-bit bytes so these are the units of **sizeof**.

> Beware that sizes may reflect alignment constraints in the underlying computer. For example, string sizes may be rounded up to even values (2-byte word alignments).

the *indirection* operator @, **cheat**, *explicitIntegerConstant* (how to write hexadecimal constants), and pointers (in particular See also <u>unchecked</u> pointers). See also <u>addr</u>, which returns the address of a variable.

size of a type

sizepic

graphics function

Syntax	sizepic (<i>x</i> 1, <i>y</i> 1, <i>x</i> 2, <i>y</i> 2 : int) : int
Description	The sizepic function is used to determine the size buffer needed to record a picture from the screen (see description of takepic). This gives the minimum number of elements of the int array used by takepic . The buffer is used by drawpic to make copies of the picture on the screen.
	This program outputs the size of array needed to hold a picture with left bottom corner at $x=10$, $y=20$ and right top corner at $x=50$, $y=60$.
Example	<pre>setscreen ("graphics")</pre>
	 put "The size of the array needs to be", sizepic (10, 20, 50, 60)
Details	See takepic for an example of the use of sizepic and for further information about buffers for drawing pictures.
	The screen should be in a " <i>graphics</i> " mode. See the setscreen procedure for details. If the screen is not in a " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode.
See also	drawpic. See also <u>setscreen, maxx, maxy, drawdot, drawline,</u> drawbox, and drawoval.
	See also predefined unit Pic .

skip	used in get statement
Syntax	skip
Description	Using skip as an input item in a get statement causes the current input to be skipped until a non-whitespace token is encountered. Whitespace includes all blanks, tabs, form feeds and newlines.
Example	The skip input item was originally intended to be used to see if more input exists in an input file. This use has been largely made redundant by a change in the Turing language. The new version of Turing reads a token, as in <i>get</i> s but not in <i>get</i> s:* or <i>get</i> s:3, and automatically skips any white space following the input value, but will not go beyond the beginning of the next input line. Originally this automatic skipping did not take place, so skip was required. The form of an input loop that used skip was as follows:
	loop get skip % This is line now redur exit when eof get
	end loop
Details	The skip bypasses all whitespace characters including any trailing newlines and blank lines. By skipping these characters, a true end-of-file condition was detected. Otherwise, the end-of-file could have been hidden by any whitespace following the last input item. With the change in Turing, the line get skip is no longer needed (although it still works correctly).
	The skip can also be used to correctly identify the start of a long string (usually to be read in <i>line</i> or <i>counted</i> mode). Here, it skips the whitespace and trailing newline as follows:
Example	<pre>var i : int var line : string loop get i, skip, line:*</pre>

The first item in the get statement reads an integer by skipping all whitespace and reading digits until whitespace is encountered. The input stream is then left with the whitespace as the next input character. The skip then skips past the whitespace, effectively beginning the next input at the next non-whitespace character. This truncates leading blanks and has another, potentially more important, effect. If the integer is the last data on a line and the string is on a following line, the skip is necessary to avoid setting *line* to a null string value.

See also get statement and loop statement.

skip	used in put statement
Syntax	skip
Description	Using skip as an output item in a put statement causes the current output line to be ended and a new line to be started.
Evample	This example, <i>To be</i> is output on one line and <i>Or not to be</i> on the next.
Example	put "To be", skip , "Or not to be"
Details	Using skip is equivalent to outputting the newline character "\n".

sound	statement
Syntax	sound (frequency, duration : int)
Description	The sound procedure is used to cause the computer to sound a note of a given frequency for a given time. The frequency is in cycles per second (Hertz). The time duration is in milliseconds. For example, middle A on a piano is 440 Hertz, so sound (440, 1000) plays middle A for one second.
	This program sounds the frequencies 100, 200 up to 1000 each for half a second.
Example	<pre>for i : 1 10 put i sound (100 * i, 500) % Sound note for 1/2 end for</pre>
Details	On IBM PC compatibles, the hardware resolution of duration is in units of 55 milliseconds. For example, sound (440, 500) will delay the program by about half a second, but may be off by as much as 55 milliseconds.
Details	The sound procedure does not currently work under MacOOT.
See also	play statement, which plays notes based on musical notation. For example, play ("8C") plays an eighth note of middle C. See also the delay , clock , sysclock , wallclock , time and date statements. See also predefined unit <u>Music</u> .

Sprite

Sprites are a way of doing animation in Turing bypassing the **Pic** module. A sprite is essentially a picture with a specific location and "depth". You create a sprite by calling **Sprite.New** with a picID received from **Pic.New**. You can then move the sprite around by calling **Sprite.SetPosition**. When you are finished with the sprite, you call **Sprite.Free**.

Description

Note that sprites work best when they are moderately small. If you have large sprites, you will continue to have flashing.

All subprograms in the **Sprite** unit are exported qualified (and thus must be prefaced with "**Sprite**.").

	<u>New</u>	Creates a new sprite from a picture.
	<u>Free</u>	Disposes of a sprite and free up its memory.
Entry Points	<u>SetHeight</u>	Sets the height of a sprite. Sprites with a greater height appear above sprites with a lesser height. The background is considered height 0. The height may be negative.
	SetPosition	Sets the location of the sprite. Can specify the center of the sprite or the lower-left corner.
	SetFrameRate	Sets the maximum number of times a second the sprites will be moved on screen.
	<u>ChangePic</u>	Changes the picture associated with a sprite.
	<u>Animate</u>	Changes the location and the picture associated with a sprite. Used for animating a moving changing image.
	<u>Show</u>	Shows a previously hidden sprite.
	<u>Hide</u>	Hides a visible sprite.

Sprite.Animate

Syntax

Sprite.Animate (*spriteID*, *picID*, *x*, *y* : **int**, *centered* : **boolean**)

Moves the sprite specified by *spriteID* to the location specified by **true**, then the sprite is centered on (x, y). Otherwise (x, y) specifies of the sprite. At the same time, it changes the picture associated wi

Description A simple example of the **Sprite.Animate** procedure would be of a picture associated with the sprite would constantly change as the fi the same time, the location of the figure would also change.

Here is a program that loads a series of images from an animated C "mypic.gif". It works by loading the images from the GIF file into **Pic.FileNewFrames** for more information) and then displays the it the screen, moving the image eight pixels each time.

var numFrames := Pic.Frames ("mypic.gif") % Load the picture **var** delayTime : **int** var pics : array 1 .. numFrames of int Pic.FileNewFrames ("mypic.gif", pics, delayTime) Example var sprite: int sprite:= Sprite.New (pics (1)) **Sprite.SetPosition** (*sprite*, 0, 100, **false**) **Sprite.Show** (*sprite*) for x : 8 .. maxx by 8 **Sprite.Animate** (*sprite*, *pics* ((*x* **div** 8) **mod** % Stop sprite from moving too **delay** (40) end for Sprite.Free (sprite)

Execute

Exported qualified.

- Status This means that you can only call the function by calling Sprite.AI Animate.
- See also **Sprite.New**, **Sprite.SetPosition** and **Sprite.ChangePic**.

Sprite.ChangePic

Part of **Sprite** module

Syntax Sprite.ChangePic (*spriteID*, *picID* : int)

Description Changes the picture associated with a sprite while maintaining the sprites height and visibility status. A typical use **Sprite.ChangePic** would be to animate a sprite that stays in position.

Here is a program that t that loads six images from files *Pic1.bmp* through *Pic6.bmp* and then displays them sequentially in the center of the screen.

Example	<pre>var pics : array 0 5 of int var sprite: int for i : 1 6 pics (i 1) := Pic.FileNew ("Pic" + intstr (if Error.Last not= 0 then put "Error loading image: ", Error.LastM return end if end for figure := Sprite.New (pics (0)) Sprite.SetPosition (sprite, maxx div 2, maxy div Sprite.Show (sprite) for i : 1 100 Sprite.ChangePic (sprite, pics (i mod 6)) end for Sprite.Free (sprite)</pre>
	Exported qualified.
Status	This means that you can only call the function by calling Sprite.ChangePic , not by calling ChangePic .

See also **<u>Sprite.New</u>**.

Sprite.Free

Syntax	Sprite.Free (<i>spriteID</i> : int)		
Description	Destroys the sprite and frees up the memory the sprite used. It is an error to use the <i>spriteID</i> after the sprite has been freed.		
Example	See Sprite.Animate for an example of Sprite.Free .		
	Exported qualified.		
Status	This means that you can only call the function by calling Sprite.Free , not by calling Free .		
See also	<u>Sprite.New</u> .		

Sprite.Hide

Syntax Sprite.Hide (*spriteID* : int)

Hides a previously visible sprite. **Sprite.Hide** has no effect if the s_j already invisible.

The following program animates four balls on the screen. When the close to each other or to a wall, they appear, otherwise they are hid

```
var pic, sprite, x, y, dx, dy, radius : array 1
var visible : array 1 .. 6 of boolean
setscreen ("nocursor")
% Create all the sprites.
for i : 1 .. 6
    radius (i) := Rand.Int (10, 25)
    Draw.FillOval (25, 25, radius (i), radius (i
    Font.Draw (intstr (i), 20, 20, 0, black)
    pic (i) := Pic.New (0, 0, 50, 50)
    Draw.FillBox (0, 0, 50, 50, 0)
   x (i) := Rand.Int (radius (i), maxx < radius
    y (i) := Rand.Int (radius (i), maxy < radius
    dx (i) := Rand.Int (3, 3)
    dy(i) := Rand.Int(3, 3)
    sprite (i) := Sprite.New (pic (i))
    Sprite.SetPosition (sprite (i), x (i), y (i)
    Sprite.SetHeight (sprite (i), i)
    visible (i) := false
end for
% Now move all the sprites around the screen.
100p
    for i : 1 .. 6
       if x(i) + dx(i) < radius(i) or
               x (i) + dx (i) > maxx radius (i)
            dx(i) := dx(i)
       end if
       x (i) := x (i) + dx (i)
       if y(i) + dy(i) < radius(i) or
               y(i) + dx(i) > maxy radius (i
            dy(i) := dy(i)
       end if
       y(i) := y(i) + dy(i)
    end for
```

Example

```
for i : 1 .. 6
        var near : boolean := false
        if (x (i) < 50) or (x (i) > maxx 50) or
                (y(i) < 50) or (y(i) > maxy 5)
            near := true
        end if
        if not near then
            for j : 1 .. 6
                if i not= j then
                    if sqrt ( (x (i) < x (j)) **
                        (y (i) < y (j)) ** 2) <
                        near := true
                        exit
                    end if
                end if
            end for
        end if
        if near and not visible (i) then
            Sprite.Show (sprite (i))
            visible (i) := true
        elsif not near and visible (i) then
            Sprite.Hide (sprite (i))
            visible (i) := false
        end if
        Sprite.SetPosition (sprite (i), x (i), y
    end for
    Time.Delay (40)
    exit when hasch
end loop
for i : 1 .. 6
    Sprite.Free (sprite (i))
end for
```

Exported qualified.

Status This means that you can only call the function by calling **Sprite.H**i calling **Hide**.

See also **Sprite.Show**.

Sprite.New

Syntax	Sprite.New (<i>picID</i> : int) : int		
	Creates a new sprite from a picture specified by <i>picID</i> . The sprite starts invisible and should be given a <i>depth</i> using Sprite.SetHeight and a position, given Sprite.SetPosition before being made visible using Sprite.Show . When you are finished using the sprite, the sprite should be freed using Sprite.Free .		
Description	Sprites work best when they are of moderate size. Large sprites will cause flashing when moved across the screen.		
	Anything that is is color 0 in the picture will not appear when the sprite is drawn. In other words, color 0 is transparent. (This is by default. You can set the transparent color for a sprite by setting the picture's transparent colour using Pic.SetTransparentColor .		
Example	See Sprite.Animate for an example of Sprite.New .		
	Exported qualified.		
Status	This means that you can only call the function by calling Sprite.New , not by calling New .		
See also	<u>Sprite.SetHeight, Sprite.SetPosition, Sprite.Show</u> and <u>Sprite.Free</u> .		

Sprite.SetFrameRate

Part of **Sprite** module

Sprite.SetFrameRate (*framesPerSec* : **int**) **Syntax** Specifies the maximum number of times per second that sprite positions will be changed on the screen. The higher the number, Description the smoother the animation will appear (and the more processor time will be used to perform the animation). When you change the position or appearance of a sprite, Turing does not update the window immediately. Instead, it waits a certain length of time and then updates all sprites that have moved since the last update. This allows for relatively smooth animation. This means that you have a simple loop that calls Sprite.SetPosition 200 times a second, the image of the sprite on the screen is still only being moved (by default) 33 times a second. You can specify how many times a second Turing checks to see if a sprite has moved by using the **Sprite.SetFrameRate** procedure. Rates of 5-10 will make the movement of the sprites seem very Details choppy. Rates of 100 are more or less completely smooth (the maximum is 120) and most CRT's will not notice any difference after 60. The default rate is 33 frames per second. You should note that if you set the rate higher than your computer can handle, every other part of your program will slow down as Turing will be checking for sprite movement rather than executing your program. A modern PC (2+ GHz with a decent graphics card) can probably handle a frame rate of 60+, depending on the number of sprites. A Pentium III should probably use the default rate. Slower machines should probably use a lower rate (around 20). Experimentation is the best way to determine the ideal frame rate. Exported qualified.

Status This means that you can only call the function by calling

Sprite.SetFrameRate, not by calling **SetFrameRate**.

Sprite.SetHeight

Part of <u>Sprite</u> module

Syntax	Sprite.SetHeight (<i>spriteID</i> , <i>newHeight</i> : int)		
Description	Sets the height of the sprite specified by <i>spriteID</i> to the value specified by <i>newHeight</i> .		
	The height of a sprite determines which sprite appears above another when they overlap. The "higher" sprite (the one with the greater height) will appear on top of the sprite with the lower height, even if the lower sprite is drawn second.		
	The background (i.e. any non-sprite) is considered to be in height 0. Sprites with a negative height will appear "behind" the background. Note that if two sprites have the same height, the one drawn last will appear above the first one.		
Example	See Sprite.Hide for an example of Sprite.SetHeight .		
Details	It should be noted that a lot more computation must be done to display sprites with a negative height. If you are on a slower machine and performance is not sufficient, consider trying to eliminate the use of sprites moving behind the background.		
	Exported qualified.		
Status	This means that you can only call the function by calling Sprite.SetPosition , not by calling SetPosition .		
See also	<u>Sprite.New</u> .		

Sprite.SetPosition

Syntax	Sprite.SetPosition (<i>spriteID</i> , <i>x</i> , <i>y</i> : int , <i>centered</i> : boolean)	
Description	Moves the sprite specified by <i>spriteID</i> to the location specified by (x, y) . If <i>centered</i> is true , then the sprite is centered on (x, y) . Otherwise (x, y) specifies the lower-left corner of the sprite.	
Example	See Sprite.Hide for an example of Sprite.SetPosition .	
	Exported qualified.	
Status	This means that you can only call the function by calling Sprite.SetPosition , not by calling SetPosition .	
See also	<u>Sprite.New</u> .	

Sprite.Show

Part of <u>Sprite</u> module

Syntax	Sprite.Show (<i>spriteID</i> : int)
Description	Displays a previously hidden sprite. Sprite.Show has no effect if the sprite is already visible.
Example	See Sprite.Hide for an example of Sprite.Show .
	Exported qualified.
Status	This means that you can only call the function by calling Sprite.Show , not by calling Show .
See also	<u>Sprite.Hide</u> .

square root function

Syntax	sqrt (<i>r</i> : real) : real		
Description	The sqrt function is used to find the square root of a number. For example, sqrt (4) is 2.		
	This program prints out the square roots of 1, 2, 3, up to 100.		
Example	<pre>for i : 1 100 put "Square root of ", i, " is ", sqrt (i) end for</pre>		
	It is illegal to try to take the square root of a negative number. The result of sqrt is always positive or zero.		
Details	The opposite of a square root is the square. For example, the square of x is written is x^{**2} .		
See also	See also predefined unit <u>Math</u> .		

sqrt

standardType

A *standardType* is one of:

	(a) (b)	int real string [(
	(c)	maximumLength)]		
	(d)	boolean		
	(e)	nat		% natural number
Syntax	(f)	int n		% n-byte integer (n=1, 2, 4)
	(g)	natn		% n-byte natural (n= 1, 2, 4)
	(h)	real n	% n-byte real (n=4, 8)	
	(i)	char		% single character
	(j)	char (<i>n</i>)	% n characters	

DescriptionThe standard types can be used throughout a program. They
should not be included in an **import** list.

int, real, string and boolean. See also nat, intn, natn, realn,See also char, char(n)

statement

Syntax

A statement is one of:

		%
(a)	assignmentStatement	variableReference
		:= expn
(b)	openStatement	% open
(C)	closeStatement	% close
(d)	putStatement	% put
(e)	getStatement	% get
(f)	readStatement	% read
(g)	writeStatement	% write
(h)	seekStatement	% seek
(i)	tellStatement	% tell
(j)	forStatement	% for end for
(k)	loopStatement	% loop end loop
(l)	exit [when trueFalseExpn]	
(m)	ifStatement	% if end if
(n)	caseStatement	% case end case
(0)	assert trueFalseExpn	
(p)	begin	
	statementsAndDeclarations	
	end	
(q)	procedureCall	% procedureId [(parameters)]
(r)	return	

(s) **result** *expn*

(t)	new [collectionId ,]
(-)	pointerVariableReference

- (u) **free** [collectionId ,]
- (u) pointerVariableReference
- (v) **tag** unionVariableReference,
 - ') expn
- (w) *forkStatement*

% *fork* ...

- (x) **signal** variableReference
- (w) wait variableReference [,
- (y) expn]
- (z) pause expn
- (aa) **quit** [*guiltyParty*] [:
- quitReason]
- (bb) **unchecked**
- (cc) **checked**

A *statement* (or *command*) causes a particular action, for example, the *putStatement*:

put "Hello"

Description

outputs *Hello*. See the descriptions of the individual statements for explanations of their actions. Each statement can optionally by followed by a semicolon (;).

	width := 24	% Assignment statement
Evampla	put "Hello world"	<i>% Put statement</i>
Example	exit when <i>i</i> = 100	% Exit statement
	assert width < 320	% Assert statement

You can use a **result** statement only in a function. You can use a **return** statement only to terminate a procedure or the main program (but not to terminate the initialization of a module). See also **result** and **return**.

Details

There are a number of predefined procedures, such as *drawline*, which are not listed as statements above. These are considered procedure calls, which is one form of statement.

statementsAndDeclarations

StatementsAndDeclarations are:

Syntax

{ statementOrDeclaration }

StatementsAndDeclarations are a list of statements and declarations. For example, a Turing program consists of a list of statements and declarations. The body of a procedure is a list of statements and declarations.

Description Each *statementOrDeclaration* is one of:

- (a) *statement*
- (b) declaration

See also statement and declaration.

This list of statements and declarations is a Turing program that outputs *Hello Frank*.

Example var name : string
name := "Frank"
put "Hello ", name

Str

D	This unit contains the predefined constants for manipulating strings.		
Description	Several routines in the Str module are exported unqualified.		
	<u>Lower</u>	Convert a string to lower case.	
	<u>Upper</u>	Convert a string to upper case.	
Entry Points	<u>Trim</u>	Remove leading and trailing blanks.	
	<u>index</u>	Finds a specified string in another string.	
	<u>length</u>	Returns the length of a string.	
	<u>repeat</u>	Creates a string by repeating a specified string a number of times.	

Str.Lower

Syntax Str.Lower (s : string) : string

Description The **Str.Lower** function takes the string *s* and returns a string in which all the upper case letters are converted to lower case. For example, Str.Lower ("ABC123def") returns "abc123def".

This program obtains lines of text from the user and outputs the lines converted to lower case.

```
var line : string
loop
put "Enter a line of text (empty to quit): "
get line : *
exit when line = ""
put "The lower case version: ", Str.Lower (1
end loop
```

Execute

Exported qualified.

- StatusThis means that you can only call the function by calling
Str.Lower, not by calling Lower.
- See also **Str.Upper** and **Str.Trim**.

Str.Trim

Str.Trim (*str* : **string**) : **string Syntax**

The **Str.Trim** function takes the string *str* and returns a string in all the leading and trailing spaces (the spaces at the beginning and the end) are deleted. For example, Str.Trim (" This is a test Description ") returns "This is a test". If *str* only has spaces in it, then Str.Trim will return an empty string.

> This program obtains input from the user until the user enters a line with non-whitespace characters in it.

```
var line : string
loop
    put "Enter a non-empty line of text: " ..
    get line : *
    exit when Str.Trim (line) not= ""
end loop
put "You entered ", line
```

Example

Execute

Exported qualified.

Status This means that you can only call the function by calling **Str.Trim**, not by calling **Trim**.

Str.Lower and Str.Upper. See also

Str.Upper

Syntax Str.Upper (s : string) : string

Description The **Str.Upper** function takes the string *s* and returns a string in which all the lower case letters are converted to upper case. For example, Str.Upper ("ABC123def") returns "ABC123DEF".

This program obtains lines of text from the user and outputs the lines converted to upper case.

```
var line : string
loop
put "Enter a line of text (empty to quit): "
get line : *
exit when line = ""
put "The upper case version: ", Str.Upper (1
end loop
```

Execute

Exported qualified.

- StatusThis means that you can only call the function by calling
Str.Upper, not by calling Upper.
- See also **Str.Lower** and **Str.Trim**.

Stream

This unit contains the predefined subprograms that deal with I/O streams. The basic I/O in Turing is done with I/O statements. However, extra functions are all part of the **Stream** unit.

Description All routines in the **Stream** unit are exported qualified (and thus must be prefaced with "**Stream**."), with the exception of **eof** which is part of the language but conceptually part of this unit and is considered to be exported unqualified.

	<u>eof</u> *	Determines if the end of file has been reached.
	<u>Flush</u>	Flushes a specified stream.
Entry Points	<u>FlushAll</u>	Flushes all open output streams.
Points	* Part of the	e language, conceptually part of the Stream unit.

eof (*streamNumber* : **int**) : **boolean Syntax** The **eof** (end of file) function is used to determine if there is any more input. It returns **true** when there are no more characters to be read. The parameter and its parentheses are omitted when referring to the standard input (usually this is the keyboard); **Description** otherwise the parameter specifies the number of a stream. The stream number has been determined (in most cases) by an **open** statement. This program reads and outputs all the lines in the file called "info". var line : string var fileNumber : int open : fileNumber, "info", get Example 1000 exit when eof (fileNumber) get : fileNumber, line : * put line end loop See also the description of the **get** statement, which gives more examples of the use of **eof**. See also the **open** and **read** statements. When the input is from the keyboard, the user can signal end-offile by typing control-Z on a PC (or control-D on UNIX). If a program tests for **eof** on the keyboard, and the user has not typed Details control-Z (or control-D) and the user has typed no characters beyond those that have been read, the program must wait until the next character is typed. Once this character is typed, the program knows whether it is at the end of the input, and returns the corresponding **true** or **false** value for **eof**.

Part of the language and only conceptually part of the **Stream** unit.

Status

This means that you can only call the function by calling **eof**, not by calling **Stream.eof**.

Stream.Flush

Part of <u>Stream</u> module

Syntax	Stream.Flush (<i>streamNumber</i> : int)
Description	The Stream.Flush procedure is used to flush any buffered output associated with the <i>streamNumber</i> parameter.
Details	Turing automatically flushes any buffered output when a stream is closed. Turing also automatically closes any open files when execution is terminated.
	Exported qualified.
Status	This means that you can only call the function by calling Stream.Flush , not by calling Flush .

Stream.FlushAll

Syntax	Stream.FlushAll
Description	The Stream.FlushAll procedure is used to flush any buffered output in any open file.
Details	Turing automatically flushes any buffered output when a stream is closed. Turing also automatically closes any open files when execution is terminated.
	Exported qualified.
Status	This means that you can only call the function by calling Stream.FlushAll , not by calling FlushAll .

string

A *stringType* is:

Syntax	<pre>string [(maximumLength)]</pre>
Description	Each variable whose type is a <i>stringType</i> can contain a sequence (a string) of characters. The length of this sequence must not exceed the <i>stringType</i> 's maximum length.
Example	var name : string name := "Nancy" var licenceNumber : string (6) licenceNumber := "175AJN"
Details	Strings can be assigned and they can be compared for both equality and for ordering. See also <i>string comparison</i> and <i>assignment statement</i> .
	Strings can be catenated (joined together) using the + operator and separated into substrings. See <i>catenation</i> and <i>substring</i> . String functions are provided to find the length of a string, to find where one string appears inside another, and to make repeated copies of a string all joined together. See <i>length</i> , <i>index</i> , and <i>repeat</i> .
	A string type written without a maximum length is limited to holding a maximum of 255 characters.
	The <i>maximumLength</i> of a string, if given as a part of the type, must be known at compile time, and must be at least 1 and at most 255. The maximum length of a string is given by <i>upper</i> , for example, <i>upper(licenceNumber)</i> is 6. See also <i>upper</i> .
	In the declaration of a string that is a var formal parameter of a procedure or function, the <i>maximumLength</i> can be written as an asterisk (*). Here, the maximum length is taken to be that of the corresponding actual parameter, as in:

procedure deblank (var s : string (*)).

The star can also be used when the parameter is an array of strings.

see alsoexplicitStringConstants for exact rules for writing string valuessuch as "Nancy". See also char(n) and char types.

comparison

string

A *stringComparison* is one of:

- (a) *stringExpn* = *stringExpn*
- (b) *stringExpn* **not**= *stringExpn*

Syntax

- (c) *stringExpn* > *stringExpn*
- (d) *stringExpn* < *stringExpn*
- (e) *stringExpn* >= *stringExpn*
- (f) stringExpn <= stringExpn

Description String (*stringExpns*) can be compared for equality (= and **not=**) and for ordering (>, <, >= and <=).

	<pre>var name : string := "Nancy"</pre>
Example	<pre>var licenceNumber : string (6)</pre>
•	<i>licenceNumber</i> := "175AJN"

Two strings are considered to be equal (=) if they have the same length and are made up, character by character, of the same characters. If they differ, they are considered to be unequal (**not=**).

Ordering among strings is essentially alphabetic order. String *S* is considered to come before string *T*, that is S < T, if the two are identical up to a certain position and after that position, either the next character of *S* comes before the next character of *T*, or else there are no more characters in *S* while *T* contains more characters.

Details S > T (S comes after T) means the same thing as T < S. S >= T means the same thing as S > T or S = T. S <= T means the same thing as S < T or S = T.

ASCII gives the ordering among individual characters. It specifies, among other things, that letter capital *L* comes

alphabetically before capital letter *M* and similarly for small (lower case) letters.

On IBM mainframe computers, the EBCDIC specification of characters may be used instead of ASCII.

strint

function

Syntax	<pre>strint (s : string [, base : int]) : int</pre>
Description	The strint function is used to convert a string to an integer. The integer is equivalent to string <i>s</i> . The number <i>base</i> parameter is optional, for example, strint ("47") = 47. In Turing proper, the <i>base</i> is not allowed and is assumed to be 10.
	String <i>s</i> must consist of a possibly null sequence of blanks, then an optional plus or minus sign, and finally a sequence of one or more digits. For number bases larger than 10, the digits can include a, b, c (alternately A, B, C) which represent the digit values 10, 11, 12 The <i>base</i> , if given, must be in the range 2 to 36 (36 because there are 10 base ten digits and 26 letters). For example, strint ("FF", 16) = 255.
	The intstr function is the inverse of strint , so for any integer <i>i</i> ,
	strint (intstr (i)) = i .
See also	chr, ord, intstr and strnat functions.

strintok

function

Syntax	<pre>strintok (s : string [, base : int]) : boolean</pre>
Description	The strintok function is used determine whether the strint function can be used to convert the string to an integer without causing an error. If the string can be successfully converted, then strintok returns true , otherwise it returns false .
	String <i>s</i> should consist of a possibly null sequence of blanks, then an optional plus or minus sign, and finally a sequence of one or more digits. For number bases larger than 10, the digits can include a, b, c (alternately A, B, C) which represent the digit values 10, 11, 12 If <i>s</i> is correctly constructed, then strnatok will return true , otherwise it returns false . The <i>base</i> , if given, must be in the range 2 to 36 (36 because there are 10 base ten digits and 26 letters). For example, strintok ("FF", 16) = true .
See also	strint function that does the actual conversion.

strnat	string to natural number function
Syntax	<pre>strnat (s : string [, base : int]) : nat</pre>
Description	The strnat function is used to convert a string to a natural number. The natural number is equivalent to string <i>s</i> . The number <i>base</i> parameter is optional, for example, strnat ("47") = 47.
	String <i>s</i> must consist of a possibly null sequence of blanks, then an optional plus sign, and finally a sequence of one or more digits. For number bases larger than 10, the digits can include a, b, c (alternately A, B, C) which represent the digit values 10, 11, 12 The <i>base</i> , if given, must be in the range 2 to 36 (36 because there are 10 base ten digits and 26 letters). For example, strnat ("FF", 16) = 255.
	The natstr function is the inverse of strnat , so for any natural number n , strnat (natstr (n)) = n .
	The strnat function is similar to strint , except that strnat handles values that are larger than int values and does not handle negative values.
See also	the <u>chr</u> , <u>ord</u> , <u>intstr</u> <u>and strint</u> functions.

strnatok

string to natural number function

Syntax	<pre>strnatok (s : string [, base : int]) : boolean</pre>
Description	The strnatok function is used determine whether the strnat function can be used to convert the string to a natural number without causing an error. If the string can be successfully converted, then strnatok returns true , otherwise it returns false .
	String <i>s</i> should consist of a possibly null sequence of blanks, then an optional plus sign, and finally a sequence of one or more digits. For number bases larger than 10, the digits can include a, b, c (alternately A, B, C) which represent the digit values 10, 11, 12 If <i>s</i> is correctly constructed, then strnatok will return true , otherwise it returns false . The <i>base</i> , if given, must be in the range 2 to 36 (36 because there are 10 base ten digits and 26 letters). For example, strnatok ("FF", 16) = true .
See also	strnat function that does the actual conversion.

strreal

Syntax	strreal (<i>s</i> : string) : real
Description	The strreal function is used to convert a string to a real number. For example, strreal ("2.5e1") will produce an approximation to the number 25.0.
	String <i>s</i> must consist of a possibly null sequence of blanks, then an optional plus or minus sign and finally an explicit unsigned real or integer constant.
	The realstr , erealstr and frealstr functions approximate the inverse of strreal , although round-off errors keep these from being exact inverses.
See also	realstr, erealstr, frealstr, intstr and strint functions.

strrealok

Syntax	<pre>strrealok (s : string) : boolean</pre>
Description	The strrealok function is used determine whether the strreal function can be used to convert the string to a real number without causing an error. If the string can be successfully converted, then strrealok returns true , otherwise it returns false .
	String <i>s</i> should consist of a possibly null sequence of blanks, then an optional plus or minus sign and finally an explicit unsigned real or integer constant. If it does so, then strrealok will return true , otherwise it returns false .
See also	strreal function that does the actual conversion.

subprogramHeader

A subprogramHeader is one of:

(a) **procedure** [**pervasive**] *id*

[([paramDeclaration {, paramDeclaration }])]

Syntax

(b) **function** [**pervasive**] *id*

[([paramDeclaration {, paramDeclaration}])] [id] : typeSpec

A subprogram header is used to describe the interface to a subprogram. Subprogram headers are used within other language features such as subprogram types and external declarations.

Parameterless subprograms may use parentheses (with nothing between them), as is required in the C programming language. These parentheses can be used to disambiguate between the call to the subprogram (parentheses present) and a reference the subprogram (parentheses missing).

Description Suppose f is a parameterless subprogram declared without parentheses and g is a parameterless subprogram declared with parentheses. Their headers are:

```
procedure f procedure g ()
```

In a program, f and g() are calls to these functions, while g is a reference to (not a call to) the procedure. There is no way to write a reference to f. When in doubt, use parentheses in the declaration, as in the case for g, so that calls always have parentheses and references always do not. A reference to a subprogram can be assigned to a subprogram variable. See subprogram type.

Example	Specify that <i>t</i> is the type of procedure with a var integer parameter and a real parameter. See also <i>subprogramType</i> .
	type t : procedure q (var j : int , y : real)
Details	The keyword pervasive can be inserted just after procedure or function . When this is done, the subprogram is visible inside all subconstructs of the subprogram's scope. Without pervasive , the subprogram is not visible inside modules unless explicitly imported. Pervasive subprograms need not be imported. You can abbreviate pervasive as an asterisk (*).
See also	pervasive.

subprogramType

A subprogramType is:

Syntax

subprogramHeader

DescriptionA variable or constant can contain a reference to a subprogram.**Description**The type of the variable or constant is a *subprogramType*. See also
subprogramHeader.

In the following *t* is a subprogram type, and *u* is a variable of type *t* initialized to refer to procedure *rnd*.

The name of the subprogram, for example q, and the parameters, for example i and x, have no meaning in a subprogram type. They are present only because of the form of subprogram headers.

If v is a variable or constant that refers to a subprogram, v can be called, compared for equality to other subprogram variables, assigned and passed as a parameter. Variable v is not an integer, string or pointer and cannot participate in their corresponding operations.

A reference to a subprogram, rather that the code of the subprogram, is contained in a variable *v* whose type is a subprogram type. This implies that **addr** (*v*) is the address of the reference to subprogram, rather than the address of the subprogram. The address of the code is given by *#v*. See **cheat** for an explanation of the *#* operator.

You cannot assign a reference to a subprogram exported from a class. This restriction exists because these subprograms are meaningless without an accompanying reference to an object.

Many potential uses of subprogram variables are better programmed using classes and overriding exported subprograms. See **class**.

subrangeType

A subrangeType is:

Syntax

Details

expn .. expn

Description A subrange type defines a set of values, for example, the subrange ... 4 consists of 1, 2, 3 and 4.

var i : 1 .. 10% i can be 1, 2 ... up to
type xRange : 0 .. 319% Define integer suk
var pixels : array xRange of int
% Array elements are
% numbered 0, 1, ... 319Example% Array elements are
% numbered 0, 1, ... 319for k : xRange% k ranges from 0 to 319
pixels (k) := 0
end for

A subrange must contain at least one element. In other words, the second expression (*expn*) must be at least as large as the first expression.

The lower bound of a subrange must be known at compile time. The upper bound is allowed to be a run time value only in one situation and that is when it gives the upper bound of an array being declared in a variable declaration, in other words when declaring a *dynamic* array.

Subranges are usually a subset of the integers, as in 1 .. 10. You cai also have subranges of enumerated types and characters (the **char** type).

You can apply **lower** and **upper** to subrange types.

of another string

substring

A *substring* is one of:

Syntax

(a) stringReference (leftPosition .. rightPosition)(b) stringReference (charPosition)

A substring selects a part of another string. In form (a) the substring starts at the left position and runs to the right position. In form (b), the substring is only a single character. Turing support substrings o char(n) values.

var word : string := "bring"put word (2 .. 4)% Outputs rinput word (3)% Outputs iput word (2 .. *)% Outputs ring; the star% the end of the string.put word (* - 2 .. * - 1)% Outputs in

The leftmost possible position in a string is numbered 1. The last position in a string can be written as an asterisk (*). For example, *word* (2 .. *) is equivalent to *word* (2 .. *length(word)*).

Each of *leftPosition*, *rightPosition*, and *charPosition* must have on of these forms:

(a) expn
(b) *
(c) * - expn

The exact rules for the allowed values of *leftPosition* and *rightPosition* are:

(1) *leftPosition* must be at least 1,
(2) *rightPosition* must be at most *length* (*stringReference*), at
(3) the length of the selected substring must zero or more.

This specifically allows null substrings such as *word* (1, 0) in whic *rightPosition* is 0 and *word* (6, 5) in which *leftPosition* is one more that **length** (*stringReference*).

Note that substrings are not assignable. For example, if *s* is a string the statement s(3) := "a" is illegal in Turing.

Turing supports substrings of char(n) values. See char(n). If a substring of char(n) value *t* has two operands, as in t(2..77), the result type of this operation is a **string**. If the substring has one operand, as in t(7), this becomes, in effect, a subscript into an array of characters. The result is a reference to a **char**, which can be assigned to or passed to a **var** parameter.

string, char, char(n), explicitStringConstant, explicitCharConstantSee also catenation and length.

SUCC	successor function
Syntax	succ (<i>expn</i>)
Description	The succ function accepts an integer, character or an enumerated value and returns the integer plus one, the next character, or the next value in the enumeration. For example, succ (7) is 8.
Example	This part of a Turing program fills up array <i>a</i> with the enumerated values green, yellow, red, green, yellow, red, etc. type colors : enum (green, yellow, red) var a : array 1 100 of colors var c : colors := colors .green for <i>i</i> : 1 100 a (<i>i</i>) := c if c = colors . red then c := colors . green else c := succ (c) end if end for
Details	You cannot apply succ to the last value of an enumeration.
See also	the <u>pred</u> , <u>lower</u> and <u>upper</u> functions.

This unit contains the predefined subprograms that deal with the operating system directly (getting the process id, getting run time arguments and executing commands in the operating system, etc.).

Description etc

All routines in the **Sys** unit are exported qualified (and thus must be prefaced with "**Sys**.").

Entry Points	<u>GetComputerName</u>	Gets the name of the computer.
	<u>GetEnv</u>	Gets a string associated with an environment variable.
	<u>GetPid</u>	Gets the current process ID for Turing.
	<u>GetUserName</u>	Gets the name of the user currently logged on.
	<u>Exec</u>	Executes a program or opens a data file using the operating system.
	<u>Nargs</u>	Gets the number of run time arguments (exported unqualified).
	FetchArg	Gets a specified run time argument (exported unqualified).

Part of Sys module

Sys.Exec (*command* : **string**) : **boolean Syntax** The **Sys.Exec** function is used to execute an application or more of file with its associated application. **Sys.Exec** can be used to launch the Internet Browser by specifying a URL. **Sys.Exec** launches the a **Description** associated with file's suffix. (In essence, it performs the same opera double clicked on the file.) This program launches an internet browser and points it to Holt So page. It then launches a movie using the default video player. if not Sys.Exec ("http://www.holtsoft.com/turing put "The Sys.Exec call failed" put "Error: ", Error.LastMsg Example end if if not Sys.Exec ("skate.avi") then put "The Sys.Exec call failed" put "Error: ", Error.LastMsg end if When the **Sys.Exec** procedure is used, the executing program conti Details immediately while the launched application is running. Exported qualified. Status This means that you can only call the function by calling **Sys.Exec**

See also <u>Sys.Nargs</u>, <u>Sys.FetchArg</u> and <u>Sys.GetEnv</u> functions.

Exec.

Sys.Exec

Sys.FetchArg

Syntax System.FetchArg (*i* : int) : string

The **Sys.FetchArg** function is used to access the *i*-th argument that passed to a program from the command line. For example, if the pr run from the Turing environment using

:r file1 file2

then **Sys.FetchArg** (2) will return "file2". If a program called *prog* under UNIX using this command:

Description prog.x file1 file2

the value of **Sys.FetchArg**(2) will similarly be "file2".

The **Sys.Nargs** function, which gives the number of arguments pas program, is usually used together with the **Sys.FetchArg** function. Parameter *i* passed to **Sys.FetchArg** must be in the range 0 .. **Sys.N**

The 0-th argument is the name of the running program.

This program lists its own name and its arguments.

Exampleput "The name of this program is : ", Sys.FetchAExamplefor i : 1 .. Sys.Nargs
put "Argument ", i, " is ", Sys.FetchArg (iend for

Exported qualified.

StatusThis means that you can only call the function by calling Sys.Fetcl
by calling FetchArg.

See also Sys.Nargs

Sys.GetComputerName

Part of <u>Sys</u> module

Syntax	Sys.GetComputerName : string	
Description	The Sys.GetComputerName function is used to determine the name of the computer. On the PC, this is the NetBIOS name. It returns "No Name" if a name could not be determined.	
	This program outputs a greeting to the user .	
Example	<pre>var computerName, userName : string computerName := Sys.GetComputerName userName := Sys.GetUserName put "Hello ", userName, " on ", computerName</pre>	
	Exported qualified.	
Status	This means that you can only call the function by calling Sys. GetComputerName , not by calling GetComputerName .	
See also	Sys.GetUserName to obtain the user name of the user currently logged in.	

Sys.GetEnv

Part of <u>Sys</u> module

Syntax	Sys.GetEnv (<i>symbol</i> : string) : string
Description	The Sys.GetEnv function is used to access the environment string whose name is <i>symbol</i> . These strings are determined by the shell (command processor) or the program that caused your program to run. See also the Sys.Nargs and Sys.FetchArg functions.
Example	<pre>On a PC, this retrieves the environment variable USERLEVEL and prints extra instructions if USERLEVEL had been set to NOVICE. USERLEVEL can be set to NOVICE with the command SET USERLEVEL = NOVICE in the autoexec.bat file or in any batch file.</pre>
	Exported qualified.
Status	This means that you can only call the function by calling Sys.GetEnv , not by calling GetEnv .

Sys.GetPid

Syntax	Sys.GetPid : int
Description	The Sys.GetPid function is used to determine the I.D. (number) that identifies the current operating system task (process). Beware that there are processes, activated by the fork statement, that are independent of the operating systems tasks.
	Under UNIX, the number is used, for example, for creating a unique name of a file.
Status	Exported qualified.
	This means that you can only call the function by calling Sys.GetPid , not by calling GetPid .
See also	Sys.Nargs, Sys.FetchArg and Sys.GetEnv.

Sys.GetUserName

Part of <u>Sys</u> module

Syntax	Sys.GetUserName : string
Description	The Sys.GetUserName function is used to determine the name of the current user. It returns "Unknown" if a name could not be determined.
	This program outputs a greeting to the user .
Example	<pre>var computerName, userName : string computerName := Sys.GetComputerName userName := Sys.GetUserName put "Hello ", userName, " on ", computerName</pre>
	Exported qualified.
Status	This means that you can only call the function by calling Sys.GetUserName , not by calling GetUserName .
See also	Sys.GetComputerName to obtain the name of the computer.

Sys.Nargs

Syntax	Sys.Nargs : int	
Description	The Sys.Nargs function is used to determine the number of arguments that have been passed to a program from the command line. For example, if the program is run from the Turing environment using	
	:r file1 file2	
	then Sys.Nargs will return 2. If a program called <i>prog.x</i> is run under UNIX using this command:	
	prog.x file1 file2	
	the value of Sys.Nargs will similarly be 2.	
	The Sys.Nargs function is usually used together with the Sys.FetchArg function to access the arguments that have been passed to the program.	
	Exported qualified.	
Status	This means that you can only call the function by calling Sys.Nargs , not by calling Nargs .	
See also	Sys.FetchArg for an example of the use of Sys.Nargs .	

sysclock

millisecs used procedure

sysclock (var c : int) **Syntax** The **sysclock** statement is used on a multitasking system such as UNIX to determine the amount of time that has been used by this program (process). Variable *c* is assigned the number of central Description processor milliseconds assigned to this program. This is of little use on a personal computer, where **sysclock** returns the same value as **clock**. On a UNIX system, this program tells you how much time it has used. var timeUsed : int Example sysclock (timeUsed) put "This program has used ", timeUsed, " milliseconds of CPU time" delay, time, clock, wallclock and date statements. See also See also predefined unit **<u>Time</u>**.

Syntax system (*command* : string, var *ret* : int)

Description The **system** statement is used to execute the shell (operating system if it were typed at the terminal. The return code is in *ret*. A return c means no detected errors. A return code of 127 means the comman could not be accessed. A return code of 126 means the command p have room to run on the PC.

This program creates a directory listing when run under DOS on ar compatible computer. The same program will run under UNIX by (to "*ls*".

Example

```
var success : int
system ( "dir", success )
if success not= 0 then
    if success= 127 then
        put "Sorry, can't find 'command.com'"
    elsif success = 126 then
        put "Sorry, no room to run 'dir'"
    else
        put "Sorry, 'dir' did not work"
    end if
end if
```

When the **system** procedure is used, the executing program usually memory while the system command is executing, and once executi system command is finished, control returns to the original prograr the PC, there is variant of the **system** procedure that allows "chaini means that when the system command is executed, the originally ri is "thrown away" (i.e. removed from memory). When the executed terminates, one is returned to DOS.

To chain another program, one prepends "chain:" to the start comm

i.e. **system** ("chain:myprog.exe", *retCode*)

Details Note that this command is "hazardous". Specifically, if you call it f opposed to a program compiled with TComp) and you have not say file, **you will lose it!** Turing will be removed from memory withou

when the **system** procedure is executed. Likewise any open files w instantly. This means there is a danger if all files were not properly the **system** procedure was called.

The "chain:" command is often used for starting menu programs, w selects a program to run and doesn't want Turing to remain in mem be used with extraordinarily large Turing programs that can be spli parts. By using TComp and compiling each part separately, one can program call the other and never have all parts in memory at once.

This program uses chaining to launch one of several possible progruser choice. It gives an error if for some reason the **system** comma work. It assumes that c:\chemistry.exe, c:\math.exe, c:\english.exe c:\history.exe already exist.

Example	<pre>var choice, success : int put "Enter the subject (1-4): " get choice var command : string case choice of % Note the use of the double backslash in th % is because the backslash is a special char % in \t for tab and \n for a newline). To ge % one uses \\. label 1 : command := "c:\\chemistry.exe" label 2 : command := "c:\\math.exe" label 3 : command := "c:\\history.exe" label 4 : command := "c:\\history.exe" label 5 : put "Choice must be from 1-4." assert false % Wasn't a 1-4. Tern end case system ("chain:" + command, success)</pre>
	<pre>% If I reach this line, the system command faile % an error message. put "System called failed." put "Program \"", command, "\" couldn't be run. assert false % Terminate the program</pre>
	Here are the possible errors under PC-Turing
Details	 -1 Not enough memory to load COMMAND.COM -2 Not enough memory to run command -3 Argument list greater than 128 bytes or environment info

- -4 Couldn't find COMMAND.COM
- -5 COMMAND.COM corrupt
- -6 -noshell option is selected, the system procedure is disallowed

See also <u>nargs</u>, <u>fetcharg</u> and <u>getenv</u> functions.

See also predefined unit **Sys**.

statement

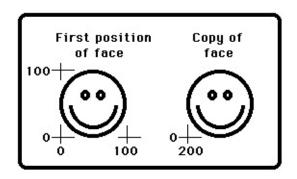
	A tagStatement is:
Syntax	tag unionVariableReference , expn
Description	A tag statement is a special-purpose assignment that is used for changing the tag of a union variable.
	In this example, the tag field of union variable <i>v</i> is set to be <i>passenger</i> , thereby activating the <i>passenger</i> field of <i>v</i> .
Example	<pre>type vehicleInfo : union kind : passenger recreational label passenger : cylinders : 116 label farm : farmClass : string (10) label : % No fields for "otherwise" clau end union var v : vehicleInfo</pre>
	 tag v, passenger % Activate passenger par
Details	A tag statement is the only way to modify the tag field of a union variable (other than by assigning an entire union value to the union variable).
Details	You cannot access a particular set of fields of a union unless the tag is set to match the corresponding label value.
See also	union types.

takepic

setscreen ("graphics") ... draw happy face in the box (0,0) to (100,100) ... % Create buffer big enough to hold happy face var face : array 1 .. sizepic (0, 0, 100, 100) of int % Copy picture into the buffer, which is the face array takepic (0, 0, 100, 100, face) % Redraw the picture with its left bottom at (200,0) drawpic (200, 0, face, 0) **takepic** (*x*1, *y*1, *x*2, *y*2 : **int**, **var** *buffer* : **array** 1 .. * **of int**) **Syntax** The **takepic** procedure is used to record the pixel values in a rectangle, with left bottom and right corners of (x1, y1) and (x2, y2)*y2*), in the buffer array. This requires a sufficiently-large buffer Description (see **sizepic**). The **drawpic** procedure is used to make copies of the recorded rectangle on the screen.

After drawing a happy face, this program copies the face to a new location.

Example



Execute

The integer values that **takepic** places in the buffer can be read or written (using the **read** and **write** statements). Unfortunately, if a

Details	value happens to be the pattern used to represent the uninitialized value (the largest negative number the hardware can represent) assignment (by:=) and put of the individual integer values in the buffer will fail.
	The screen should be in a " <i>graphics</i> " mode. See the setscreen procedure for details. If the screen is not in a " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode.
Details	The drawpic , takepic , and sizepic subprograms have been made obsolete by the subprograms <u>Pic.Draw</u> and <u>Pic.New</u> of the <u>Pic</u> module. Users are strongly suggested to use those routines instead. The <u>Pic</u> module also contains subprograms to load images from files.
See also	<pre>sizepic and drawpic. See also setscreen, maxx, maxy, drawdot, drawline, drawbox, and drawoval.</pre> See also predefined unit Pic.

tangent function	(radians)
------------------	-----------

Syntax	tan ([r : real]): real
--------	-------	---------------------	---------

Description The **tan** function is used to find the tangent of an angle given in radians. For example, **tan** (p/4) is 0.5.

This program prints out the tangent of 0, p/6, 2p/6, 3p/6, up to 12p/6 radians.

Execute

Details	An argument to tan of p/2, 3*p/2, etc. may result in an overflow (and halt of the program) as the result is technically ¥. (It may not cause an overflow due to rounding, in which case the result will simply be a very large number.) the <u>sin</u> and <u>cos</u> functions for calculating sine and cosine.
See also	the <u>tand</u> function which finds the tangent of an angle given in degrees. (2p radians are the same as 360 degrees.)
	See also predefined unit <u>Math</u> .

tan

tand	tangent function (degrees)
Syntax	tand (<i>r</i> : real) : real
Description	The tand function is used to find the tangent of an angle given in degrees. For example, tand (45) is 1.
	This program prints out the tangent of 0, 30, 60, 90, up to 360 degrees.
Example	<pre>for i : 0 12 const angle := i * 30 put "Tan of ", angle, " is ", tand (angle) end for</pre>

Execute

Details	An argument to tan of 90, 270, etc. may result in an overflow (and halt of the program) as the result is technically ¥. (It may not cause an overflow due to rounding, in which case the result will simply be a very large number.) the <u>sind</u> and <u>cosd</u> functions for calculating sine and cosine.
See also	the <u>tan</u> function which finds the tangent of an angle given in radians. (2p radians are the same as 360 degrees.)
	See also predefined unit Math.

file statement

An tellStatement is:

Syntax tell : *fileNumber* , *filePositionVar*

The tell statement sets *filePositionVar*, whose type must be int, to t current offset in bytes from the beginning of the specified file. The *fileNumber* must specify a file that is open with seek capability (or program argument file that is implicitly opened). The tell statemen useful for recording the file position of a certain piece of data for la access using seek.

This example shows how to use **tell** to record the location of a recc file. This location is later used by **seek** to allow the record to be rea

var *employeeRecord* : record name : string (30) pay : int dept : 0 .. 9 end record **var** fileNo : **int** Example var location : int open : fileNo, "payroll", write, seek **tell** : fileNo, location % Make note of t write : fileNo, employeeRecord % Write record a **seek** : fileNo, location % Go back to locatic **read** : fileNo, employeeRecord % Read the recor % that was previously writte

See also the <u>read</u>, <u>write</u>, <u>open</u>, <u>close</u>, <u>seek</u>, <u>get</u> and <u>put</u> statements.

Text

Entry Points This unit contains the predefined subprograms that handle character (text) output on the screen (i.e. output using **put**).

Description All routines in the **Text** unit are exported qualified (and thus must be prefaced with "**Text**.") with the exception of **maxrow** and **maxcol** which are exported unqualified.

<u>Cls</u>	Clears the screen to the text background color.
<u>Color</u>	Sets the text color used by put .
<u>Colour</u>	Sets the text color used by put .
ColorBack	Sets the text background color used by put .
ColourBack	Sets the text background color used by put .
<u>Locate</u>	Moves the cursor to the specified row and column.
LocateXY	Moves the cursor to the cursor location closest to a specified pixel position.
<u>maxcol</u>	The number of columns on the screen (exported unqualified).
<u>maxrow</u>	The number of rows on the screen (exported unqualified).
WhatRow	Returns the current cursor row.
<u>WhatCol</u>	Returns the current cursor column.
<u>WhatColor</u>	Returns the current text color.
WhatColour	Returns the current text color.
WhatColorBack	Returns the current text background color.
<u>WhatColourBack</u>	Returns the current text background color.

Text.Cls

Syntax	Text.Cls
Description	The Text.Cls (clear screen) procedure is used to blank the screen to the text background color. The cursor is set to the top left (to row 1, column 1).
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. If the screen mode has not been set, it will automatically be set to " <i>screen</i> " mode. See View.Set for details.
	Exported qualified.
Status	This means that you can only call the function by calling Text.Cls , not by calling Cls .

Text.Color

Syntax	Text.Color (<i>Color</i> : int)	
Description	The Text.Color procedure is used to change the currently-active color. This is the color of characters that are to be put on the screen. The alternate spelling is Text.Colour .	
	This program prints out the message "Bravo" three times, each in a different color.	
Example	<pre>View.Set ("graphics") for i : 1 3 Text.Color (i) put "Bravo" end for</pre>	
	This program prints out a message. The color of each letter is different from the preceding letter. For letter number i the color number is i mod maxcolor + 1. This cycles repeatedly through all the available colors.	
Example	<pre>View.Set ("screen") const message := "Happy New Year!!" for i : 1 length (message) Text.Color (i mod maxcolor + 1) put message (i) end for</pre>	
	In " <i>screen</i> " mode on the IBM PC, the color specified can actually range from 0 - 31. The upper 16 colors (16-31) are the same as the lower 16, except that they blink.	
Details	See View.Set for the number of colors available in the various " <i>graphics</i> " modes.	
	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. If the screen mode has not been set, it will automatically be set to " <i>screen</i> " mode. See View.Set for details.	

Exported qualified.

Status This means that you can only call the function by calling **Text.Color**, not by calling **Color**.

Text.ColorBack, Text.WhatColor, Text.WhatChar andSee alsoView.maxcolor.

Text.ColorBack

Syntax	Text.ColorBack (<i>Color</i> : int)
Description	The Text.ColorBack procedure is used to change the current text background color. The alternate spelling is Text.ColourBack .
	The Text.ColorBack procedure sets the text background color to the specified color. This is the color that surrounds characters when they are put onto the screen. On an IBM PC in " <i>screen</i> " mode, the color can be from 0 - 7. (You can not have the upper 8 colors as text background colors. On UNIX dumb terminals, Text.ColorBack (1) turns on highlighting and Text.ColorBack (0) turns it off. On other systems, this procedure may have no effect.
	Since this program is in " <i>screen</i> " mode, changing the background color has no immediately observable effect. When the message "Greetings" is output, the background surrounding each letter will be red.
Example	View.Set ("screen")
	 Text.ColorBack (<i>red</i>) put "Greetings"
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. If the screen mode has not been set, it will automatically be set to " <i>screen</i> " mode. See View.Set for details
	Exported qualified.
Status	This means that you can only call the function by calling Text.ColorBack , not by calling ColorBack .
See also	Text.Color and Text.WhatColorBack.

Text.Locate

Syntax	Text.Locate (<i>row</i> , <i>column</i> : int)
Description	The Text.Locate procedure is used to move the cursor so that the next output from put will be at the given row and column. Row 1 is the top of the screen and column 1 is the left side of the screen.
	This program outputs stars of random colors to random locations on the screen. The variable <i>colr</i> is purposely spelled differently from the word <i>color</i> to avoid the procedure of that name (used to set the color of output). The row number is purposely chosen so that it is one less than maxrow . This avoids the scrolling of the screen which occurs when a character is placed in the last column of the last row.
Example	<pre>View.Set ("screen") var row, column, colr : int loop row := Rand.Int (1, maxrow) column := Rand.Int (1, maxcol) colr := Rand.Int (0, maxcolor) Text.Color (colr) Text.Locate (row, column) put "*" % Use dot-dot to avoid clearing end loop</pre>
Details	The Text.Locate procedure is used to locate the next output based on row and column positions. See also the Text.LocateXY procedure which is used to locate the output based x and y positions, where x=0, y=0 is the left bottom of the screen.
	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. See the View.Set procedure for details. If the screen is not in one of these modes, it will automatically be set to" <i>screen</i> " mode.
	Exported qualified.
Status	This means that you can only call the function by calling Text.Locate , not by calling Locate .

See also <u>View.Set</u> and <u>Draw.Dot</u>.

Text.LocateXY

Syntax	Text.LocateXY (x , y : int)	
Description	The Text.LocateXY procedure is used to move the cursor so that the next output from put will be at approximately (x, y) . The exact location may be somewhat to the left of x and below y to force alignment to a character boundary.	
	This program outputs <i>Hello</i> starting at approximately (100, 50) on the screen.	
Example	View.Set ("graphics") Text.LocateXY (100, 50) put "Hello"	
Details	The Text.LocateXY procedure is used to locate the next output based on x and y positions, where the position x=0, y=0 is the left bottom of the screen. See also the Text.Locate procedure which is used to locate the output-based row and column positions, where row 1 is the top row and column 1 is the left column.	
	The screen should be in a " <i>graphics</i> " mode. See the View.Set procedure for details. If the screen is not in a " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode.	
Status	Exported qualified.	
	This means that you can only call the function by calling Text.LocateXY , not by calling LocateXY .	
See also	<u>View.Set</u> and <u>Draw.Dot</u> .	

Text.maxcol

Syntax	maxcol : int	
Description	The maxcol function is used to determine the number of columns on the screen.	
	This program outputs the maximum column number.	
Example	put "Number of columns on the screen is ", maxrc	
Details	For IBM PC compatibles as well as most UNIX dumb terminals, ir " <i>text</i> " or " <i>screen</i> " mode, maxcol = 80. For the default IBM PC compatible " <i>graphics</i> " mode (CGA), maxcol = 40.	
	Exported unqualified.	
Status	This means that you can call the function by calling maxcol or by calling Text.maxcol .	
See also	<u>Text.Locate</u> procedure for an example of the use of maxcol .	

Text.maxrow

Syntax	maxrow : int	
Description	The maxrow function is used to determine the number of rows on the screen.	
	This program outputs the maximum row number.	
Example	put "Number of rows on the screen is ", maxrow	
Details	For IBM PC compatibles, maxrow = 25. For many UNIX dumb terminals, maxrow = 24.	
	Exported unqualified.	
Status	This means that you can call the function by calling maxrow or by calling Text.maxrow .	
See also	<u>Text.Locate</u> procedure for an example of the use of <u>maxrow</u> .	

Text.WhatCol

Part of <u>Text</u> module

Syntax	Text.WhatCol : int	
Description	The Text.WhatCol function is used to determine the cursor position's column.	
	This program outputs <i>The current row</i> is 5, <i>the current column is</i> 15.	
Example	Text.Locate (5, 10) put "12345" put "The current row is", Text.WhatRow put "The current column is", Text.WhatCol	
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. Text.WhatCol functions properly even if the cursor is invisible.	
	Exported qualified.	
Status	This means that you can only call the function by calling Text.WhatCol , not by calling WhatCol .	
See also	the <u>Text.WhatRow</u> function, which is used to determine the cursor row. See also the <u>Text.Locate</u> , <u>Text.maxrow</u> and <u>Text.maxcol procedure.</u>	

Text.WhatColor

Syntax	Text.WhatColor : int	
Description	The Text.WhatColor function is used to determine the current text (foreground) color, ie., the color used for characters that are output using put . The alternate spelling is Text.WhatColour .	
	This program outputs the currently-active color number. The message is also given in the currently-active color.	
Example	View.Set ("graphics")	
	 put "This writing is in color number ", Text.Wha	
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. See View.Set for details.	
	Exported qualified.	
Status	This means that you can only call the function by calling Text.WhatColor , not by calling WhatColor .	
See also	the <u>Text.Color</u> <u>procedure</u> , which is used to <u>set</u> the <u>color</u> . See also <u>Text.ColorBack</u> and <u>Text.WhatColorBack</u> .	

Text.WhatColorBack

Part of <u>Text</u> module

Syntax	Text.WhatColorBack : int	
Description	The Text.WhatColorBack function is used to determine the current text background color. The alternate spelling is whatcolourback .	
	This program outputs the currently-active background color number. The background color of the message is determined by this number.	
Example	View.Set ("screen")	
	… put "The background of this writing" put "is in color number ", Text.WhatColorBack	
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. Beware that the meaning of background color is different in these two modes. See Text.ColorBack for details.	
	Exported qualified.	
Status	This means that you can only call the function by calling Text.WhatColorBack , not by calling WhatColorBack .	
See also	Text.Color and Text.WhatColor.	

Text.WhatRow

Part of <u>Text</u> module

Syntax	Text.WhatRow : int	
Description	The Text.WhatRow function is used to determine the cursor position's row.	
	This program outputs <i>The current row</i> is 5, <i>the current column is</i> 15.	
Example	Text.Locate (5, 10) put "12345" put "The current row is", Text.WhatRow put "The current column is", Text.WhatCol	
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. Text.WhatRow functions properly even if the cursor is invisible.	
	Exported qualified.	
Status	This means that you can only call the function by calling Text.WhatRow , not by calling WhatRow .	
See also	the Text.WhatCol function, which is used to determine the cursor column. See also the Text.Locate , Text.maxrow and Text.maxcol procedure.	

Time

This unit contains the predefined subprograms that handle anything to do with time, either as a date or as a timer.

Description

All routines in the **Time** unit are exported qualified (and thus must be prefaced with "**Time**.").

	<u>Sec</u>	Returns the number of seconds since 1/1/1970 00:00:00 GMT.
	Date	Returns the current date and time as a string.
Entry Points	<u>SecDate</u>	Converts a number of seconds into a date / time string.
	DateSec	Converts a date / time string to a number of seconds.
	<u>SecParts</u>	Converts the number of seconds since 1/1/1970 00:00:00 GMT into a day of month, month, year, day of week, hour, minute and second integers.
	PartsSec	Converts a day of month, month, year, hour, minute and second integers into the number of seconds since 1/1/1970 00:00:00 GMT.
	<u>SecStr</u>	Converts the number of seconds since 1/1/1970 00:00:00 GMT into a user specified formatted string representing the date.
	Elapsed	Returns the number of milliseconds elapsed since the program started to run.
	<u>ElapsedCPU</u>	Returns the number of milliseconds of CPU time elapsed since the program started to run.
	<u>Delay</u>	Sleeps for a specified number of milliseconds.
	DelaySinceLast	Sleeps until a specified number of milliseconds since the last call to Time.DelaySinceLast.

Time.Date

Syntax Time.Date : string

The Time.Date function returns the current date and time as a string. The returned string in the format "*dd mmm yy hh:mm:ss*", where *mmm* is the first 3 characters of the month, e.g., "*Apr*". For example, if the date is Christmas 1989 at 9:02:37 in the morning, Time.Date will return "25 Dec 89 09:02:37". Twenty-four hour time is used, so eleven thirty at night the same day would return "2 Dec 89 23:30:00"

This program greets you and tells you the date and time.

Example	<pre>var theDateTime, theDate, theTime : string theDateTime := Time.Date theDate := theDateTime (1 9) theTime := theDateTime (11 *) put "Greetings!! The date and time today is ",</pre>
	Be warned that on some computers, such as IBM PC compatibles c Apple Macintoshes, the date may not be set correctly in the operating system; in that case, the Time.Date procedure will give incorrect results.
Details	The string form of the date can be converted to a numeric form for comparison purposes using the Time.DateSec function. The numeric form can be converted to a string using the Time.SecDate function. The numeric form of the time can be obtained using the Time.Sec function.
	Exported qualified.
Status	This means that you can only call the function by calling Time.Date , not by calling Date .
See also	<u>Time.Sec</u> , <u>Time.DateSec</u> and <u>Time.SecDate</u> functions.

Syntax	Time.DateSec (<i>dateString</i> : string) : int
	The Time.DateSec function is used to convert a date and time string into a number, specifically, the number of seconds since 00:00:00 GMT Jan 1, 1970.
Description	The function can also convert just the date (" <i>dd mmm yy</i> "), in which case it returns the number of seconds since 00:00:00 GMT Jan 1, 1970 from midnight of the entered day. It will also convert a time without the date (" <i>hh:mm:ss</i> "), in which case it returns the number of seconds that have passed since midnight of that day.
	If the format is incorrect or can't be interpreted, then Time.DateSec will return 1 and Error.Last and Error.LastMsg will be set to the appropriate error.
	This program gives the number of seconds since 00:00:00 GMT Jan 1, 1970.
Example	<pre>var theDateTime, theDate, theTime : string theDateTime := Time.Date theDate := theDateTime (1 9) theTime := theDateTime (11 *) put "The number of seconds from 00:00:00 GMT Jar "from midnight ", theDate, "is ", Time.DateS put "The number of seconds from midnight to ", t Time.DateSec (theTime) put "The number of seconds from 00:00:00 GMT Jar "from ", theDateTime, "is ", Time.DateSec (t</pre>
	Exported qualified.
Status	This means that you can only call the function by calling Time.DateSec , not by calling DateSec .
See also	<u>Time.Sec</u> , <u>Time.Date</u> and <u>Time.SecDate</u> functions.

Time.Delay

Part of <u>Time</u> module

Syntax	Time.Delay (<i>duration</i> : int)	
Description	The Time.Delay procedure is used to cause the program to pause for a given time. The time duration is in milliseconds.	
	This program prints the integers 1 to 10 with a second delay between each.	
Example	for i : 1 10 put i Time.Delay (1000) % Pause for 1 second end for	
	Exported qualified.	
Status	This means that you can only call the function by calling Time.Delay , not by calling Delay .	
See also	Time.Elapsed and Time.ElapsedCPU.	

Time.DelaySinceLast

Part of <u>Time</u> module

Syntax	Time.DelaySinceLast (<i>duration</i> : int)	
Description	The Time.DelaySinceLast procedure is used to cause the program to pause for a given time since the last call to Time.DelaySinceLast . The time duration is in milliseconds.	
	This program outputs from 1 to 100,000, then outputs how long it took to do this (in milliseconds). It then calls Time.DelaySinceLast to wait until 10 seconds has passed since the beginning of the program (regardless of how long it took to execute the loop). Finally it outputs how many milliseconds has passed since the program started. This should be close to 10,000 milliseconds.	
Example	<pre>var t : int := Time.Elapsed put "Starting now, count to 100,000" for i : 1 100000 put i locate (whatrow, 1) end for put "Finished counting to 100,000" put "Elapsed time: ", Time.Elapsed - t, " millis Time.DelaySinceLast (10000) put "Elapsed time: ", Time.Elapsed - t, " millis</pre>	

Execute

The **Time.DelaySinceLast** is usually used to time a process so that it operates on the same speed no matter what the machine. For example, in the above example, the program will alwyas take 10 seconds to execute, regardless of the speed of the machine (up

Details	to a point). Fast machines will spend little time in the for loop and longer waiting to return from Time.DelaySinceLast . Slower machines will take longer to execute the loop and will consequently wait less time before returning from Time.DelaySinceLast .	
	The Time.DelaySinceLast procedure is often used to time the speed of animation in games.	
	Note that this only works up to a point. On a very slow machine, the for loop may take longer than 10 seconds to execute, in which case the call to Time.DelaySinceLast will return instantly.	
	Exported qualified.	
Status	This means that you can only call the function by calling Time.DelaySinceLast , not by calling DelaySinceLast .	
See also	Time.Elapsed and Time.ElapsedCPU.	

Time.Elapsed

Syntax	Time.Elapsed : int		
Description	The Time.Elapsed function returns the amount of time since a program (process) started running. The number of milliseconds since the program started running is returned.		
	This program tells you how much time it has used.		
Example	var timeRunning : int timeRunning := Time.Elapsed put "This program has run ", timeRunning, " mill		
Details	On IBM PC compatibles, this is the total time since the Turing system was started up. The hardware resolution of duration is in units of 55 milliseconds. For example, Time.Elapsed may be off b as much as 55 milliseconds.		
	On Apple Macintoshes, this is the total time since the machine was turned on. The hardware resolution of duration is in units of 17 milliseconds (1/60-th of a second).		
	Exported qualified.		
Status	This means that you can only call the function by calling Time.Elapsed , not by calling Elapsed .		
See also	<u>Time.ElapsedCPU</u> and <u>Time.Delay</u> subprograms.		

Time.ElapsedCPU

Part of <u>Time</u> module

Syntax	Time.ElapsedCPU : int		
Description	The Time.ElapsedCPU function is used on a multitasking system such as UNIX to determine the amount of time that has been used by this program (process). The number of central processor milliseconds assigned to this program is returned. This is of little use on a personal computer, where Time.ElapsedCPU returns the same value as Time.Elapsed .		
Example	<pre>On a UNIX system, this program tells you how much time it has used. var timeUsed : int timeUsed := Time.ElapsedCPU put "This program has used ", timeUsed,</pre>		
	" milliseconds of CPU time"		
	Exported qualified.		
Status	This means that you can only call the function by calling Time.ElapsedCPU , not by calling ElapsedCPU .		
See also	Time.Elapsed and Time.Delay subprograms.		

Time.PartsSec

Syntax	Time.PartsSec (year, month, day, hour, minute, second : int) : int	
Description	The Time.PartsSec function is used to convert the numeric parts of a date (specifically the year, month, day, hour, minute and second) into the number of seconds since 00:00:00 GMT Jan 1, 1970 and the date specified by the parts.	
	The function can also convert a time without a date (year, month and day are all 0), in which case it returns the number of seconds that have passed since midnight of the current day.	
	If the numbers don't make any sense or can't be interpreted, then Time.PartsSec will return 1 and Error.Last and Error.LastMsg will be set to the appropriate error.	
Example	This program gives the number of seconds between 00:00:00 GMT Jan 1, 1970 and 9:27 in the morning, Christmas Day, 1989).	
	<pre>put "The number of seconds from 00:00:00 GMT Jar "is ", Time.PartsSec (1989, 12, 25, 9, 27, 0)</pre>	
Status	Exported qualified.	
	This means that you can only call the function by calling Time.PartsSec , not by calling PartsSec .	
See also	Time.SecParts, Time.Date and Time.Sec functions.	

Time.Sec

Syntax	Time.Sec : int		
Description	The Time.Sec function returns the current date and time as a numb The returned integer is the time in seconds since 00:00:00 GMT (Greenwich Mean Time) January 1, 1970.		
	This program tells you how many seconds since 1970.		
Example	put "The number of seconds since 1970 is ", Time		
Details	Be warned that on some computers, such as IBM PC compatibles c Apple Macintoshes, the date may not be set correctly in the operati system; in that case, the Time.Date procedure will give incorrect results.		
	The string form of the date can be converted to a numeric form for comparison purposes using the Time.DateSec function. The numeric form can be converted to a string using the Time.SecDate function. The numeric form of the time can be obtained using the Time.Sec function.		
Status	Exported qualified.		
	This means that you can only call the function by calling Time.Sec by calling Sec .		
See also	<u>Time.Date</u> , <u>Time.DateSec</u> and <u>Time.SecDate</u> functions.		

Syntax	Time.SecDate (timeInSecs : int) : string	
	The Time.SecDate function is used to convert the number of secor since 00:00:00 GMT Jan 1, 1970 into a date and time string.	
Description	If <i>timeInSecs</i> is incorrect or can't be interpreted, then Time.SecDat will return the empty string and Error.Last and Error.LastMsg w be set to the appropriate error.	
	This program gives the number of seconds since 00:00:00 GMT Ja 1970 and the date in string form.	
Example	<pre>var timeInSecs : int := Time.Sec var theDateTime: string theDateTime := Time.SecDate (timeInSecs) put "The number of seconds since 1970 is ", time put "Greetings!! The date and time today is ",</pre>	
	Exported qualified.	
Status	This means that you can only call the function by calling Time.SecDate , not by calling SecDate .	
See also	Time.Sec, Time.Date and Time.DateSec functions.	

Time.SecParts

Syntax	Time.SecParts (sec : int , var year, month, day, dayOfWeek, hour, minute, second : int)	
Description	The Time.SecParts function is used to convert a single number for (the number of seconds since 00:00:00 GMT Jan 1, 1970) into a da numeric component parts.	
	The <i>dayOfWeek</i> parameter is 1 for Monday, 2 for Tuesday through	
	If the <i>sec</i> parameter doesn't make any sense or can't be interpreted, Time.PartsSec will set all the var parameters to 1 and Error.Last Error.LastMsg will be set to the appropriate error.	
	This program returns the current day of the week.	
Example	<pre>var year, month, day, dayOfWeek, hour, minute, s Time.SecParts (Time.Sec, year, month, day, dayOf</pre>	
	Exported qualified.	
Status	This means that you can only call the function by calling Time.Sec calling SecParts .	
	See also Time.PartsSec, Time.Date and Time.Sec functions.	

Syntax Time.SecStr (*timeInSecs* : **int**, *formatString* : **string**) : **str**

The **Time.SecStr** function is used to convert the number of second 00:00:00 GMT Jan 1, 1970 into a date and time string.

Description If *timeInSecs* is incorrect or can't be interpreted, then **Time.SecStr** the empty string and **Error.Last** and **Error.LastMsg** will be set to appropriate error.

The *formatString* parameter specifies how the output string will loc formatString consists of different letters specifying the different for spaces or commas between these letters. The different letters can b to provide different date formats.

Here are the letters:

	a	Abbreviated weekday name
	Α	Full weekday name
	b	Abbreviated month name
	B	Full month name
	С	Date and time representation appropriate for locale
	d	Day of month as decimal number (01 - 31)
	Η	Hour in 24-hour format (00 - 23)
	Ι	Hour in 12-hour format (01 - 12)
Detailsn	j	Day of year as decimal number (001 - 366)
Detailsh	m	Month as decimal number (01 - 12)
	Μ	Minute as decimal number (00 - 59)
	р	Current locale's A.M./P.M. indicator for 12-hour clock
	S	Second as decimal number (00 - 59)
	U	Week of year as decimal number, with Sunday as first day o - 53)
	w	Weekday as decimal number (0 - 6; Sunday is 0)

W	Week of year as decimal number, with Monday as first day (
	- 53)
x	Date representation for current locale
X	Time representation for current locale
у	Year without century, as decimal number (00 - 99)
Y	Year with century, as decimal number
z,Z	Time-zone name

This program outputs the current date and the day number in the ye Example output: Thursday November 13, 2003 is day number : 2003

Example

put Time.SecStr (Time.Sec, "A B d, Y"), " is day Time.SecStr (Time.Sec, "j"), " of ", Time.Sec

Execute

This example demonstrates each of the formatting letters.

Execute

Exported qualified.

Status This means that you can only call the function by calling **Time.Sec** calling **SecStr**.

<u>Time.Sec</u> and <u>**Time.SecParts**</u> functions.

See also

time time of day as a string procedure time (var *t* : string) **Syntax** The **time** statement is used to determine the current time of day. Variable *t* is assigned a string in the format "*hh:mm:ss*". For example, if the time is two minutes and 47 seconds after nine Description A.M., *t* will be set to "09:02:47". Twenty-four hour time is used. For example, eleven thirty P.M. gives the string "23:30:00". This program greets you and tells you the time of day. var timeOfDay : string Example time (timeOfDay) put "Greetings!! The time is ", timeOfDay Be warned that on some computers such as IBM PC compatibles or Apple Macintoshes, the time may not be set correctly in the Details operating system. In this case, the **time** procedure will give incorrect results. delay, clock, sysclock, wallclock and date statements. See also See also predefined unit **<u>Time</u>**.

token

keywords such as **get**, identifiers such as *incomeTax*, operators and special symbols, such as + and :=, and explicit constants, such as 1.5 and "Hello". Some keywords, such as **index**, are reserved and cannot be used in programs to name variables, procedures, etc.

Description A get statement, such as

get incomeTax

uses *token-oriented* input. This means that white space (blanks, tabs, etc.) is skipped before reading the input item and after the item (up to the beginning of the next line). See the **get** statement for details.

In this example, the tokens are **var**, x, :, **real**, x, := and 9.84.

Example

var x : **real** x := 9.84

Syntax true

A **boolean** (true/false) variable can be either **true** or **false** (see **boolean** type).

	<pre>var passed : boolean := true var mark : int for i : 1 10</pre>
	get mark
Example	passed := passed and mark >= 60
-	end for
	<pre>if passed = true then</pre>
	put "You passed all ten subjects"
	end if

DetailsThe line if *passed=*true then can be simplified to if *passed* thenwith no change to the meaning of the program.

true

A typeDeclaration is one of:

Syntax(a)type id : typeSpec(b)type id : forward

Description A type declaration gives a name to a type. This name can be used in place of the type.

```
type nameType : string ( 30 )
type range : 0 .. 150
type entry :
    record
    name : nameType
    age : int
end record
```

The keyword **pervasive** can be inserted just after **type**. When this is done, the type is visible inside all subconstructs of the type scope. Without **pervasive**, the type is not visible inside modules, monitors and classes unless explicitly imported. Pervasive types need not be imported. You can abbreviate **pervasive** as an asterisk (*).

A **forward** type allows pointers to be declared to the **type** before the type is *resolved*. To resolve a **type**, you must follow a **forward** with a declaration of the same name and in the same scope. This type declaration must include a *typeSpec*.

Example

Details

TypeConv

This unit contains the predefined subprograms that convert between different Turing standard types. There are also six routines that are part of the language, rather than part of the unit, but are conceptually part of this unit.

Description All routines in the **TypeConv** unit are exported unqualified.

Description of the routines in the **TypeConv** module can be found in this chapter.

	<u>intreal</u>	Converts an integer to a real.		
	intstr*	Converts an integer to a string.		
	<u>natreal</u>	Converts a natural number to a real.		
	<u>natstr</u> *	Converts a natural number to a string.		
	<u>round</u>	Converts a real to an integer (rounding).		
	<u>floor</u>	Converts a real to an integer (round down).		
	<u>ceil</u>	Converts a real to an integer (round up).		
	<u>realstr</u>	Converts a real to a string.		
	<u>erealstr</u>	Converts a real to a string (exponential notation).		
	<u>frealstr</u>	Converts a real to a string (no exponent).		
	<u>strint</u> *	Converts a string to an integer.		
Entry Points	<u>strintok</u> *	Returns whether a string can legally be converted to an integer.		
_ ••	<u>strnat</u> *	Converts a string to a natural number.		
	<u>strnatok</u> *	Returns whether a string can legally be converted to a natural number.		
	<u>strreal</u>	Converts a string to a real.		
	<u>strrealok</u>	Returns whether a string can legally be converted to a real.		
	<u>chr</u> *	Returns the ASCII value of a specified string of length one.		
	ord*	Returns a string of length one with the ASCII value specified.		

* Part of the language, conceptually part of the **TypeConv** unit.

typeSpec

Syntax

type specification

A *typeSpec* (type specification) is one of:

(a)	int	
(b)	real	
(C)	boolean	
(d)	stringType	% Example: string (20)
(e)	subrangeType	% Example: 1 150
(f)	enumeratedType	% Example: enum (red, green, blue)
(g)	arrayType	% Example: array 1 150 of real
(h)	setType	% Example: set of 1 10
(i)	recordType	% Example: record end record
(j)	unionType	% Example: union end union
(k)	pointerType	% Example: pointer to collectionVar
(l)	namedType	% Example: colorRange
(m)	nat	% natural number
(n)	intn	% n-byte integer (n=1, 2, 4)
(0)	natn	% n-byte natural (n= 1, 2, 4)
(p)	real n	% n-byte real (n=4, 8)
(q)	char	% single character
(r)	char (<i>n</i>)	% n characters
(a)	aubara aram Tura	

(s) *subprogramType*

	A type specification determines the allowed values for a variable
Description	or constant. For example, if variable <i>x</i> is an integer (its <i>typeSpec</i>
	is int), the possible values for <i>x</i> are numbers such as -15, 0, 3 and
	348207. If <i>x</i> is a real number (its <i>typeSpec</i> is real), then its
	possible values include 7.8, -35.0, and 15e12. If <i>x</i> is a boolean , its
	possible values are true and false . If <i>x</i> is a string , its possible
	values include <i>Hello</i> and <i>Good-bye</i> .

	<pre>var numberOfSides : int var x, y : real</pre>	
	type <i>range</i> : 0 150	% The typeSpec here is 6
Example	type entry : record	% Here is a record typeS
	name : string (25)
	age : range	
	end record	

See alsoint, real, boolean, string, subrangeType, enum, array, set,
record, union, pointer, named, nat, intn, natn, realn, char, and
char(n) types.

unchecked

compiler directive

Dangerous

Description	OOT adds the concept of "unchecked" to Turing. Here, you can request that certain run time tests, which take place by default, can be eliminated. This makes the program more efficient at the risk of unreliability.
Example	Declare <i>p</i> to be an unchecked pointer to an integer (see pointers for details). Pointer <i>p</i> will be dangerous to use, because the run time system will not check to see if it actually locates an integer, as opposed to arbitrary computer memory. In other words, unchecked pointers are like C language pointers.
	var <i>p</i> : unchecked ^ int
Example	Declare C to be an unchecked collection of records of type R (see collections for details). Pointers to C will be unchecked.
pre	var <i>C</i> : unchecked collection of <i>R</i>
	Remove checking from the body of a loop.
Example	<pre>for i : 1 500 unchecked if a (i) = key then exit end if end for</pre>
Details	In the above example, the unchecked keyword requests that all checking, in particular, array bounds checking for array <i>a</i> , are to be omitted. The disabling lasts from the occurrence of the keyword unchecked to the end of the surrounding construct, in this case, until end for . In a similar way, the checked keyword will request that checking be re-enabled from the occurrence of checked to the end of the surrounding construct.
	In the current (1999) implementation, the use of unchecked to

turn off checking in a block of statements is ignored. In general, an implementation may choose to ignore requests to disable checking.

union

A unionType is:

Syntax

A union type (also called a variant record) is like a record in which is a run time choice among sets of accessible fields. This choice is by the tag statement, which deletes the current set of fields and act new set.

This union type keeps track of various information about a vehicle, depending on the kind of vehicle.

```
const passenger := 0
const farm := 1
const recreational := 2

type vehicleInfo :
    union kind : passenger .. recreational of
    label passenger :
        cylinders : 1..16
    label farm :
        farmClass :string ( 10 )
        label : % No fields for "otherwise"
    end union
var v : vehicleInfo
...
tag v, passenger % Activate passenger part v.
```

The optional identifier following the keyword **union** is the name of *tag* of the union type. If the identifier is omitted, the tag is still conto exist, although its value cannot be accessed. The tag must be of index type, for example 1..7. You should limit the range of this ind type, as the compiler may have a limit (at least 255) on the maximurange it can handle.

Example

	Each <i>labelExpn</i> must be known at compile time and must lie within
	range of the tag's type. The fields, including the tag, of a union valu
	referenced using the dot operator, as in <i>v.cylinders</i> and these can be
	as variables or constants. A field can be accessed only when the tag
	matches one of the label expressions corresponding to the field. Th
Details	can be changed by the tag statement but it cannot be assigned to, p
	to a var parameter, or bound to using var .

In a union, *id*'s of fields, including the tag, must be distinct. Howev these need not be distinct from identifiers outside the union. Union be assigned as a whole (to unions of an equivalent type), but they c be compared. A semicolon can optionally follow each *typeSpec*.

Any array contained in a union must have bounds that are known a compile time.

The notation > can be used to access union fields. For example, if *µ* pointer to *vehicleRecord*, *p*>*farmClass* locates the *farmClass* field.

See also **pointer**.

file containing module, monitor, or class

A *compilationUnit* is one of:

(a)	[importList]	mainProgram
	L J	

Syntax

- (b) **unit** *moduleDeclaration*
- (c) **unit** monitorDeclaration
- (d) **unit** *classDeclaration*

DescriptionA program can be divided up into units, each in a separate file. All
files except the main program begin with the keyword **unit**. The ur
contains the main program, a module, a monitor or a class.

Here is stack module that is separated out into a file whose name is

unit % The keyword unit begins each separ module stack export push, pop var top : int := 0 var contents : array 1 .. 100 of int procedure push (i : int) top += 1 contents (top) := i end push procedure pop (i : int) i := contents (top) top -= 1 end pop end stack

Example

The main program, which is in another file, gains access to the stac importing it. Here is the main program:

import var stack % Use the stack
var n : int
...
stack . push (n)
...
stack . pop (n)

unit

	In this example, the keyword var in the import list is required beca main program causes a change in the stack, by calling <i>push</i> and <i>po</i> _j import lists of units that are modules, monitors and classes are used access to further units.
Details	If the stack were in a file with a different name, say <i>stk.t</i> , the impor would be rewritten to use an in clause, as follows:
	<pre>import var stack in "stk.t"</pre>
	A mainProgram is simply a program. See program .
See also	<u>module, monitor</u> and <u>class</u> . See also <u>export</u> list, <u>import</u> list, <u>inhe</u> <u>implement</u> list and <u>implement by</u> list.

unqualified

When an identifier x is exported from a module, monitor or class M using the keyword unqualified, it can be used outside of M without the qualification "M.". In other words, outside of M, it can be referred to as simply x. The keyword unqualified can be written in its short form as ~. which is pronounced "not dot", as in:

export ~. x

See als

o **export** list.

upper

Syntax	upper (reference [, dimension]) : int
Description	The upper attribute is used to find the upper bound of an array, string, char (n) or non-opaque subrange type. (See lower for finding the lower bound.)
	In a procedure, see if the bound of array parameter a is large enough that it can be subscripted by i . If it is large enough, it is set $a(i)$ to zero.
Example	<pre>procedure test (var a : array 1 * of real)</pre>

A *variableDeclaration* is one of:

Syntax(a) var id { ,id } [:typeSpec] [:=initializingValue](b) collectionDeclaration

Description A variable declaration creates a new variable (or variables). Only f collectionDeclaration for explanation of form (b). The *typeSpec* of initializing value is present.

Examplevar j, k : int := 1% j and k are assigned vvar t := "Sample"% The type of t is strinvar v : array 1 ... 3 of string (6) :=init ("George", "Fred", "Alice")

The initializing value, if present, must be an expression or else a lis inside **init** (...). The syntax of *initializingValue* is one of:

(a) expn(b) init (initializingValue {, initializingValue })

Each **init** (...) corresponds to an array, record or union value that nested for initialization of nested types.

If the *typeSpec* is omitted, the variable's type is taken to be the (roc expression, for example, **int** or **string**. The *typeSpec* cannot be omi initializing value is of the form **init** (...). The values inside **init** (...

Details The keyword **pervasive** can be inserted just after **var**. When this is all subconstructs of the variable's scope. Without **pervasive**, the va unless explicitly imported. Pervasive variables need not be importe an asterisk (*).

OOT extends Turing in the following way. OOT changes form (a) t **register** keyword to request that the variable be placed in a machin (a) is actually:

var [pervasive] [register] id { , id } [: typeS

In the current (1994) OOT implementation, programs are run interphas no machine registers, and the **register** keyword is ignored. See of register variables.

collection, **bind**, **procedure** and **function** declarations, parameter**See also** import lists for other uses of the keyword <u>var</u>.

variableReference

use of a variable

A variableReference is:

Syntax

variableId { componentSelector }

In a Turing program, a variable is declared and given a name (*variableId*) and then used. Each use is called a *variable reference*.

If the variable is an array, collection, record or union, its parts (*components*) can be selected using subscripts and field names (using *componentSelectors*). The form of a *componentSelector* is one of:\

Description

- (a) (*expn* {, *expn*})
- (b) . *fieldId*

Form (a) is used for subscripting (indexing) arrays and collections. The number of array subscripts must be the same as in the array's declaration. A collection has exactly one subscript, which must be a pointer to the collection. Form (b) is used for selecting a field of a record or union.

Following the declarations of *k*, *a* and *r*, each of *k*, *a* (*k*) and *r*.*name* are variable references.

A variable reference can contain more than one component

Example

Details selector, for example, when the variable is an array of records. For an example, see the **record** type. See also *constantReference* and **var** declaration.

View

This unit contains the predefined subprograms that deal with the current output surface, which is a window.

Description All routines in the **View** unit are exported qualified (and thus must be prefaced with "**View**.") with the exception of **maxx**, **maxy**, **maxcolor** and **maxcolour** which are exported unqualified.

<u>maxx</u>	Returns the maximum x coordinate (width 1) (exported unqualified).
<u>maxy</u>	Returns the maximum y coordinate (height 1) (exported unqualified).
<u>maxcolor</u>	Returns the maximum color number (# colors 1) (exported unqualified).
<u>maxcolour</u>	Returns the maximum color number (# colors 1) (exported unqualified).
<u>Set</u>	Changes the configuration of the output surface.
<u>ClipSet</u>	Clips output to a specified rectangle. Adds another
<u>ClipAdd</u>	specified rectangle to the clipping

	region.	
<u>ClipAddOval</u> <u>ClipOff</u>	Adds another specified oval to the clipping region. Stops all clipping.	
<u>WhatDotColor</u>		Gets the color of the pixel at a specified location.
<u>WhatDotColour</u>	Gets the color of the pixel at a specified location.	
<u>Update</u>	Updates the onscreen window from the offscreen bitmap.	
<u>UpdateArea</u>	Updates part of the onscreen window from the offscreen bitmap.	
<u>SetTransparentColor</u>	Sets the transparent color to be ignored when using picUnderMerge mode.	
<u>SetTransparentColour</u>	Sets the transparent colour to be ignored when using picUnderMerge mode.	
	ClipOff WhatDotColor WhatDotColour Update SetTransparentColor	ClipAddOvalAdds another specified oval to the clipping region. Stops all clipping.ClipOffStops all clipping.WhatDotColorSets the color of the pixel at a specified location. Updates the onscreen window from the offscreen bitmap.UpdateAreaUpdates part of the onscreen window from the offscreen bitmap.SetTransparentColorSets the transparent color to be ignored when using picUnderMerge mode.

View.ClipAdd

Syntax	View.ClipAdd (<i>x</i> 1, <i>y</i> 1, <i>x</i> 2, <i>y</i> 2 : int)
	The View.ClipAdd procedure adds another rectangle specified by $(x1, y1) - (x2, y2)$ to the clipping region. This only works on systems that support complex clipping regions. If no clipping region has been specified, then the rectangle becomes the complete clipping region.
Description	A clipping region is the region that the output will appear in. If the rectangle is specified as the clipping region, any drawing done outside the rectangle will not appear.
	To set the initial clipping, or remove the old region and replace it with a new one, use View.ClipSet . To set the clipping region back to the entire screen or window, use View.ClipOff .
	These commands only work in "graphics" mode.
	This program sets the clipping region to five rectangles and then draws random circles. The circles will only appear (or partially appear) in the rectangles.
Example	<pre>const maxx13 : int := maxx div 3 const maxx23 : int := 2 * maxx div 3 const maxy13 : int := maxy div 3 const maxy23 : int := 2 * maxy div 3 View.ClipSet (0, 0, maxx13, maxy13) View.ClipAdd (maxx23, 0, maxx, maxx13) View.ClipAdd (maxx13, maxy13, maxx23, maxy23) View.ClipAdd (0, maxy23, maxx13, maxy) View.ClipAdd (maxx23, maxy23, maxx13, maxy)</pre>
	<pre>% Draw the random ovals in the box var x, y, clr : int loop</pre>

Exported qualified.

StatusThis means that you can only call the function by callingView.ClipAdd, not by calling ClipAdd.

See also View.ClipSet and View.ClipOff functions.

View.ClipAddOval

Syntax	View.ClipAddOval (<i>x</i> , <i>y</i> , <i>xradius</i> , <i>yradius</i> : int)	
Description	The View.ClipAddOval procedure adds another oval specified by (x, y) and <i>xradius</i> and <i>yradius</i>) to the clipping region. If no clipping region has been specified, then the oval becomes the complete clipping region.	
	A clipping region is the region that the output will appear in. If the rectangle is specified as the clipping region, any drawing done outside the oval will not appear.	
	To set the initial clipping, or remove the old region and replace it with a new rectangle, use View.ClipSet . To set the clipping region back to the entire screen or window, use View.ClipOff .	
	These commands only work in " <i>graphics</i> " mode.	
	This program sets the clipping region to five circles and then draws random squares. The squares will only appear (or partially appear) in the ovals.	
Example	<pre>const c1 : int := maxy div 4 const c2 : int := 3* maxy div 4 const radius : int := maxy div 4 View.ClipAddOval (c1, c1, radius, radius) View.ClipAddOval (c1, c2, radius, radius) View.ClipAddOval (c2, c1, radius, radius) View.ClipAddOval (c2, c2, radius, radius)</pre>	
	<pre>% Draw the random squares in the box var x, y, clr : int loop x := Rand.Int (0, maxx - 30) % Random x y := Rand.Int (0, maxy - 30) % Random y clr := Rand.Int (0, maxcolor) % Random cc Draw.FillBox (x, y, x + 30, y + 30, clr) end loop</pre>	

Execute

By combining animation using **View.Update** and **View.ClipAddOval**, you can achieve a moving spotlight effect.

Execute

Exported qualified.

- StatusThis means that you can only call the function by calling
View.ClipAddOval, not by calling ClipAddOval.
- See also <u>View.ClipSet</u>, <u>View.ClipAdd</u> and <u>View.ClipOff</u> functions.

View.ClipOff

Syntax	View.ClipOff	
Description	The View.ClipOff procedure turns off clipping. This means that any drawing commands can appear on the entire drawing surface (the screen or the window, depending on the system).	
	These commands only work in " <i>graphics</i> " mode.	
	Exported qualified.	
Status	This means that you can only call the function by calling View.ClipOff , not by calling ClipOff .	
	See also View.ClipAdd and View.ClipSet functions.	

View.ClipSet

Syntax	View.ClipSet (<i>x</i> 1, <i>y</i> 1, <i>x</i> 2, <i>y</i> 2 : int)	
Description	The View.ClipSet procedure sets the clipping region to the rectangle specified by $(x1, y1) - (x2, y2)$. If a clipping region already exist, it is replaced by the specified rectangle.	
	A clipping region is the region in which the output will appear. If the rectangle is specified as the clipping region, any drawing done outside the rectangle will not appear.	
	To set the initial clipping, or remove the old region and replace it with a new one, use View.ClipSet . To set the clipping region back to the entire screen or window, use View.ClipOff .	
	These commands only work in " <i>graphics</i> " mode.	
	This program sets the clipping region to five rectangles and then draws random circles. The circles will only appear (or partially appear) in the rectangles.	
Example	<pre>const maxx13 : int := maxx div 3 const maxx23 : int := 2 * maxx div 3 const maxy13 : int := maxy div 3 const maxy23 : int := 2 * maxy div 3 View.ClipSet (0, 0, maxx13, maxy13) View.ClipAdd (maxx23, 0, maxx, maxy13) View.ClipAdd (maxx13, maxy13, maxx23, maxy23) View.ClipAdd (0, maxy23, maxx13, maxy) View.ClipAdd (maxx23, maxy23, maxx13, maxy)</pre>	
	<pre>% Draw the random ovals in the box var x, y, clr : int loop</pre>	

Exported qualified.

- StatusThis means that you can only call the function by calling
View.ClipSet, not by calling ClipSet.
- **See also** <u>View.ClipAdd</u> and <u>View.ClipOff</u> functions.

View.maxcolor

Syntax	View.maxcolor : int	
Description	The maxcolor function is used to determine the maximum color number for the current mode of the screen. The alternate spelling is maxcolour .	
	This program outputs the maximum color number.	
Example	<pre>setscreen ("graphics")</pre>	
ľ	 put "The maximum color number is ", View.maxcolc	
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. If it is not, it will automatically be set to " <i>screen</i> " mode. See View.Set for details.	
	For IBM PC compatibles in " <i>screen</i> " mode, maxcolor = 15. For the default IBM PC compatible " <i>graphics</i> " mode (VGA), maxcolor = 15.	
Details	View.maxcolor is identical to RGB.maxcolor . It is placed here for consistency with other screen information routines.	
Status	Exported qualified.	
	This means that you can only call the function by calling View.maxcolor . Note that RGB.maxcolor is exported unqualified, so that one can call maxcolor .	
See also	Draw.Dot for examples of the use of maxcolor . See the Text.Color procedure which is used for setting the currently-active <u>color</u> .	

View.maxx

Syntax	maxx : int			
Description	The maxx function is used to determine the maximum value of x for the current graphics mode.			
	This program outputs the maximum x value.			
Example	<pre>setscreen ("graphics")</pre>			
	 put "The maximum x value is ", maxx			
Details	The screen should be in a " <i>graphics</i> " mode. If it is not, it will automatically be set to " <i>graphics</i> " mode. See setscreen for details.			
	For the default IBM PC compatible graphics mode (CGA), maxx = 319.			
Status	Exported unqualified.			
	This means that you can call the function by calling maxx or by calling View.maxx .			
See also	Draw.Dot for an example of the use of maxx and for a diagram illustrating x and y positions.			

View.maxy

Syntax	maxy : int		
Description	The maxy function is used to determine the maximum value of y for the current graphics mode.		
	This program outputs the maximum y value.		
Example	<pre>setscreen ("graphics")</pre>		
	 put "The maximum y value is ", maxy		
Details	The screen should be in a " <i>graphics</i> " mode. If it is not, it will automatically be set to " <i>graphics</i> " mode. See setscreen for details.		
	For the default IBM PC compatible graphics mode (CGA), maxy = 199.		
Status	Exported unqualified.		
	This means that you can call the function by calling maxy or by calling View.maxy .		
See also	Draw.Dot for an example of the use of maxy and for a diagram illustrating x and y positions.		

View.Set

Syntax View.Set (*s* : string)

Here are example uses of the **View.Set** procedure. In many cases, t will appear as the first statement of the program. However, they ca appear any place in a program.

Example	View.Set ("graphics")	% Switch to graphics
	<pre>View.Set ("screen")</pre>	% Switch to screen mode
	<pre>View.Set ("nocursor")</pre>	% Turn off cursor
	View.Set ("noecho")	% Do not echo keystrokes

The View.Set statement is used to change the mode of the screen, a as the way in which Turing does input and output. The parameter to View.Set is a string, such as "graphics". The string contains one or options separated by commas, such as "text, noecho". View.Set aff the active window.

There are two window modes, **text** and **graphics**.

text mode does not allow any graphics whatsoever (including curst positioning, etc.). Only **put** and **get** are allowed. Any output that sc off the top of the window is preserved and can viewed or printed la

graphics mode allows character graphics and pixel graphics comm such and **Text.Locate** and **Draw.Box**.

Details The default graphics mode is defined in the Turing preferences. It i good practice to set the desired mode so that the program will func properly regardless of thedefault graphics mode. Note that if the us prints the output window, in **text** mode, all output sent to the window printed. In **graphics** mode, only the current output of the window i printed. If the user saves the output window, a **text** mode window v produce a text file containing all the output sent to the window. A **graphics** window will produce a BMP graphics file containing the current contents of the window.

Where the options to **View.Set** are mutually exclusive, they are list

here with the default underlined. Here are the options:

"<u>cursor</u>", "**nocursor**" - Causes the cursor to be shown (or hidden). cursor is only displayed when the program is awaiting input.

"<u>echo</u>", "**noecho**" - Causes (or suppresses) echoing of characters th typed. Echoing is commonly turned off in interactive programs to l typed characters from being echoed at inappropriate places on the screen.

"**noxor**", "**xor**" - **noxor** mode means that all drawing is done norm In **xor** mode, all pixel graphics are drawn XOR'ed on the backgrou (with the exception of the **Pic** routines, where the drawing mode is specified). The most important property of an XOR'ed object is tha can be erased and the background restored by XOR'ing the object c of itself.

"**msdos**", "**nomsdos**" - Causes the window to use the MS-DOS cha set (with line drawing characters) instead of the Windows ANSI character set. The "**nomsdos**" option causes the window to use the Windows ANSI character set. Note that the "**msdos**" option only w if the Windows font (usually Courier New) supports it.

"**visible**", "**invisible**", "**popup**" - Causes the active window to becc visible (invisible or, for popup, invisible until input or output occur the window).

"**title**:<*text*>" - Causes the title of the active window to be set to <*t*

"**position:** $\langle x \rangle$; $\langle y \rangle$ " - Causes the position of the upper left corner c active window to be set to (x, y). The $\langle x \rangle$ parameter can also be or "left", "center", "center" or "right", in which case the window will placed on the left, center or right side of the screen. The $\langle y \rangle$ param can also be one of: "top", "middle", "truemiddle", or "bottom" in v case the window will be placed at the top, middle about 1/3 from th middle or bottom of the screen. Note that when a window is support be centered in the middle of the screen, "middle" usually looks bet than "truemiddle".

Details

"nooffscreenonly", "offscreenonly" Causes or (suppresses) output

being sent to the visible window. When the **offscreenonly** option is active, any text and graphics output is drawn to the offscreen buffe is maintained for every Run window but not drawn to the screen. **View.Update** is then used to copy the entire contents of the offscree buffer to the Run window. By allowing numerous drawing comman be sent to the offscreen buffer and then updating the window at one it is possible to get smoother animation.

"**buttonbar**", "**nobuttonbar**" Causes or (suppresses) the display of button bar at the top of the output window which allows the user to easily stop program execution or save and print the output window.

"**text**", "**screen**", "**graphics**" - Sets window to the given mode and always erases the screen, even when already in the requested mode

The **text** mode can have a modifier in the form "text:<*rows*>;<*cols*. This sets the window to be <*rows*> by <*cols*> of text in size.

The **screen** mode actually sets the window to **graphics** mode. It ca have a modifier in the form "screen:<*rows*>;<*cols*>". This sets the window to be <*rows*> by <*cols*> of text in size.

The **graphics** mode can have a modifier in the form "graphics:<*wi*(<*height*>". This sets the window to be <*width*> by <*height*> pixels size.

To set a window to the maximum size available on the screen, you use "max" for the *<width>*, *<height>*, *<rows>* or *<columns>* param If the window requested is larger than will fit on the screen, the winwill fill the entire screen and scroll bars will be added to the output window to allow the window user to see the rest of the window.

This program creates a graphics window that is 300 pixels by 100 I

```
View.Set ("graphics:300;100")
```

This program outputs the square roots for the first 200 numbers. The user can inspect all the output and print the values after the program finished execution

```
View.Set ("text")
for value : 1 .. 200
    put value : 3, " ", sqrt (value)
end for
```

Example This program creates a window without a button bar at the top that sized to fit the screen. It then draws an "X" in red in the window.

View.Set ("graphics:max;max,nobuttonbar")
Draw.Line (0, 0, maxx, maxy, red)
Draw.Line (maxx, 0, 0, maxy, red)

This program resizes the window to 200x200, moves the output wi to the bottom-left of the screen and hides the button bar. It then set window title to "Bottom Left Window" and outputs the word "Hell

View.Set ("graphics:200;200,position:bottom;left
View.Set ("title:Bottom Left Window")
put "Hello"

See also setscreen for further information.

Exported qualified.

Status This means that you can only call the function by calling **View.Set**. by calling **Set**.

View.SetTransparentColor

Part of <u>View</u> module

Syntax	View.SetTransparentColor (colorNumber : int)	
Description	The View.SetTransparentColor procedure sets the color in the Run window that should be considered transparent when a picture is drawn on the window using the picUnderMerge mode. If no color is specified, then the default background color (colorbg , usually white) is used as the transparent color. You can change this background color using RGB.SetColor on color 0.	
Details	This call is often used when you want the Run window to have a background color other than color 0.	
Example	<pre>This program displays two images loaded from the same file. In the second image,the transparent color has been set to bright red (that is, the parts of the image that are meant to be transparent are in bright red).</pre> <pre>const SIZE : int := 20 procedure DrawCheckerBoard cls for x : 0 maxx by SIZE for y : 0 maxy by SIZE if ((x + y) div SIZE) mod 2 = 0 ther Draw.FillBox (x, y, x + SIZE, y end if end for end for end for end for end for if := Pic.FileNew ("data files/airpla Pic.SetTransparentColor (pic, brightred) RGB.SetColor (0, 1., 0., 0.) DrawCheckerBoard Pic.Draw (pic, 100, 100, picUnderMerge) View.SetTransparentColor (brightgreen) Pic.Draw (pic, 400, 100, picUnderMerge)</pre>	

Execute

DetailsView.SetTransparentColour is an alternate spelling for
View.SetTransparentColor.Exported qualified.StatusThis means that you can only call the function by calling
View.SetTransparentColor, not by calling
SetTransparentColor.See alsoPic.SetTransparentColor.

View.Update

Syntax View.Update

The **View.Update** procedure updates a Run window from an offscreen bitmap. It is used with the command

Description View.Set ("offscreenonly") which prevents the Run window from being updated until the View.Update command is given.

This program displays 30 circles bouncing around the screen. To se the effect without **View.Update**, comment out the line **View.Set** ("offscreenonly")

```
% Place some circles around the screen
                    const RADIUS : int := 30
                    const NUM_BALLS : int := 20
                    var x, y, dx, dy, clr : array 1 .. NUM_BALLS of
                    for i : 1 .. NUM BALLS
                        x (i) := Rand.Int (RADIUS, maxx - RADIUS)
                        y (i) := Rand.Int (RADIUS, maxy - RADIUS)
                        dx (i) := Rand.Int (-3, 3)
                        dy (i) := Rand.Int (-3, 3)
                        clr (i) := Rand.Int (1, 15)
                    end for
                    % Now, any drawing to the screen won't appear un
                    % View.Update is given.
                    View.Set ("offscreenonly")
Example
                    loop
                        cls % Clear the offscreen window
                        for i : 1 .. NUM_BALLS
                            if x(i) + dx(i) < RADIUS or
                                    x (i) + dx (i) > maxx - RADIUS t
                                dx (i) := -dx (i)
                            end if
                            if y(i) + dy(i) < RADIUS or
                                   y(i) + dy(i) > maxy - RADIUS t
                                dy(i) := -dy(i)
                            end if
                            x (i) := x (i) + dx (i)
                            y(i) := y(i) + dy(i)
                            Draw.FillOval (x (i), y (i), RADIUS, RAL
                        end for
                        % All the circles have been drawn. Now upda
                        View.Update
```

Execute

	All Turing Run windows have both an onscreen visible window an an offscreen window. Whenever any output is sent to the screen, bo the onscreen window and the offscreen window are updated. When the Run window needs to be updated (for example when another window is moved over top of it and then removed), Turing copies t offscreen window onto the onscreen window.
Details	When the View.Set ("offscreenonly") command is given, Turing no longer draws to the onscreen window when any drawing command given. However, it does update the offscreen window. When the View.Update command is given, the entire offscreen window is copied to the onscreen window.
	This can be used to provide smooth, flicker-free animation. Animated objects flicker when the object being animated disappear from the onscreen window for a period of time. By using View.Set ("offscreenonly") / View.Update , the onscreen window is never blank. Instead, the offscreen window drawn over top off the screen window, replacing it. This means that the on-screen window never blanked out, eliminating the flickering found in the animation
Details	It's very easy to forget that no output will appear in the Run window when using View.Update . Remember to use View.Set ("nooffscreenonly") to turn off this feature sending output for debugging purposes.
Details	View.Update should not be used in conjunction with the Sprite module. Sprites can be considered a limited version of this techniq View.Update also works well when the entire background is

changing.

Exported qualified.

StatusThis means that you can only call the function by calling
View.Update, not by calling Update.

<u>View.Set</u> for the "offscreenonly" and "nooffscreenonly" options.

See also <u>**View.UpdateArea**</u> for updating part of the window at a time.

View.UpdateArea

Syntax	View.UpdateArea (<i>x</i> 1, <i>y</i> 1, <i>x</i> 2, <i>y</i> 2)
	The View.UpdateArea procedure updates a rectanglular area Run window, specified by $(x1, y1) - (x2, y2)$ from the offscreen bitmap. It is used with the command View.Set ("offscreenonly") which prevents the Run window from being updated until the View.UpdateArea or View.Update command is given.
Description	Because the entire screen is not updated each time, the animation can be much faster, especially on slow machines. This procedure does take more effort to use than View.Update as it requires the programmer to calculate which portion of the screen must be updated.
	This program moves a star back and forth across the screen. To compare the speed up View.UpdateArea with View.Update , replace the calls to View.UpdateArea with View.Update .
Example	<pre>View.Set ("offscreenonly") var SIZE : int := 50 loop for x : 0 maxx - SIZE Draw.FillStar (x, 100, x + SIZE, 100 + 5 View.UpdateArea (x - 1, 100, x + SIZE, 1 Draw.FillStar (x, 100, x + SIZE, 100 + 5 end for for decreasing x : maxx - SIZE 0 Draw.FillStar (x, 100, x + SIZE, 100 + 5 View.UpdateArea (x, 100, x + SIZE, 100 + 5 View.UpdateArea (x, 100, x + SIZE, 100 + 5 end for end for end for end for end for for decreasing x : maxx - SIZE 0 Draw.FillStar (x, 100, x + SIZE, 100 + 5 View.UpdateArea (x, 100, x + SIZE, 100 + 5 View.UpdateArea (x, 100, x + SIZE, 100 + 5 end for end for end for for decreasing x : maxx (x, 100, x + SIZE, 100 + 5</pre>

Execute

If the entire screen is being updated each time through the animation loop (for example if the background image is changing), then use **View.Update** instead.

Details When using **View.UpdateArea**, it is important to correctly calculate the region to be updated. This region must not only contain the iter being drawn, but also any area where the background should be replaced. For example, in the program above, the area updated included the one pixel to the left of the star when the star was moving right-ward and one pixel to the right of the star when the star was moving left-ward.

Here is an example program that allows you to compare the speed of animation using **View.Update** with **View.UpdateArea**. The program animates an object on a background of stars. Pressing any key switches between the two calls.

Execute

All Turing Run windows have both an onscreen visible window an an offscreen window. Whenever any output is sent to the screen, both the onscreen window and the offscreen window are updated. When the Run window needs to be updated (for example when another window is moved over top of it and then removed), Turing copies the offscreen window onto the onscreen window.

Details	When the View.Set ("offscreenonly") command is given, Turing no longer draws to the onscreen window when any drawing command is given. However, it does update the offscreen window. When the View.Update command is given, the entire offscreen window is copied to the onscreen window.
	This can be used to provide smooth, flicker-free animation. Animated objects flicker when the object being animated disappear from the onscreen window for a period of time. By using View.Set ("offscreenonly") / View.Update , the onscreen window is never blank. Instead, the offscreen window drawn over top off the on screen window, replacing it. This means that the on-screen window is never blanked out, eliminating the flickering found in th animation.
Details	It's very easy to forget that no output will appear in the Run window when using View.UpdateArea . Remember to use View.Set ("nooffscreenonly") to turn off this feature sending output for debugging purposes.
Details	View.UpdateArea should not be used in conjunction with the Sprite module. Sprites can be considered a limited version of this technique. View.UpdateArea also works well when the entire background is changing.
	Exported qualified.
Status	This means that you can only call the function by calling View.UpdateArea , not by calling UpdateArea .
	View.Set for the "offscreenonly" and "nooffscreenonly" options.
See also	<u>View.Update</u> for procedure to update the entire screen at one time.

View.WhatDotColor

Part of <u>View</u> module

Syntax View.WhatDotColor (*x*, *y* : int) : int

DescriptionThe View.WhatDotColor function is used to determine the color
number of the specified pixel. The alternate spelling is
View.WhatDotColour.

This program draws a line which bounces off the edges of the screen and makes a beep when it finds a pixel that has already been colored.

```
View.Set ( "graphics" )
                    var x, y : int := 0
                    var dx, dx : int := 1
                    loop
                        if View.WhatDotColor ( x, y ) not= 0 and
                                View.WhatDotColor (x, y) not= brig
                            Draw.FillOval (x, y, 10, 10, brightred)
Example
                        end if
                        Draw.Dot (x, y, 1)
                        x := x + dx
                        y := y + dy
                        if x = 0 or x = maxx then
                            dx := -dx
                        end if
                        if y = 0 or y = maxy then
                            dy := -dy
                        end if
                    end loop
```

Execute

Another example illustrates the use of **View.WhatDotColor** to determine the color of pixels in a window painted with multi-color blocks. The actual color number appears in the block. The

color of the dot under the mouse cursor appears in the upper-left corner of the window. Note that you can also place the mouse cursor over the black text indicating the color number.

Example

Details	The screen should be in a" <i>graphics</i> " mode. If is not set to " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode. See View.Set for details.
	Exported qualified.
Status	This means that you can only call the function by calling View.WhatDotColor , not by calling WhatDotColor .
See also	Draw.Dot , which is used for setting the color of a pixel. See also maxx and maxy , which are used to determine the number of pixels on the screen.

wait	block a process statement
Syntax	A waitStatement is: wait variableReference [, expn]
Description	The wait statement is used in a concurrent program to cause the executing process to be blocked (to go to sleep) until it is awakened by a signal statement. The statement can only be used inside a monitor (a special kind of module that handles concurrency). A wait statement operates on a condition variable (the <i>variableReference</i>), which is essentially a queue of sleeping processes. See condition for an example of a wait statement.
Details	A wait statement for a priority condition must include the optional <i>expn</i> ,. This expression must be a non-negative int value which is used to order processes waiting for the condition, low numbers first. A wait statement for a timeout condition must include the
Details	optional <i>expn</i> , which must be a non-negative int value which gives the <i>timeout interval</i> . A process waiting for a timeout condition is implicitly awakened if it waits longer than its timeout interval.
See also	<u>condition</u> and <u>signal</u> . See also <u>monitor</u> and <u>fork</u> . See also <u>empty</u> . See also <u>pause</u> .

wallclock

seconds since 1/1/1970 procedure

Syntax	wallclock (var c : int)
Description	The wallclock statement is used to determine the time in seconds since 00:00:00 GMT (Greenwich Mean Time) January 1, 1970.
	This program tells you how many seconds since 1970.
Example	var seconds : string w allclock (<i>seconds</i>) put "The number of seconds since 1970 is ", <i>secc</i>
Details	Be warned that on some computers such as IBM PC compatibles or Apple Macintoshes, the time may not be set correctly in the operating system; in that case, the wallclock procedure will give incorrect results. Also, on IBM PC compatibles, the call is dependent on having the time zone TZ variable correctly set. On an IBM PC, the default time zone is set to PST (6 hours from GMT).
	On the Apple Macintosh, the wallclock procedure returns the number of seconds since 00:00:00 local time Jan. 1, 1970.
	delay, time, clock, sysclock and date statements.
See also	See also predefined unit <u>Time</u> .

cursor position function

Syntax	whatcol : int
Description	The whatcol function is used to determine the cursor position's column.
	This program outputs <i>The current row is</i> 5, <i>the current column is</i> 15.
Example	locate (5, 10) put "12345" put "The current row is", whatrow put "The current column is", whatcol
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. whatcol functions properly even if the cursor is invisible.
See also	the whatrow function, which is used to <u>set</u> the determine the cursor row. See also the locate , maxrow and maxcol procedure.
	See also predefined unit Text .

whatcol

whatcolor

text color graphics function

Syntax	whatcolor : int
Description	The whatcolor function is used to determine the current (foreground) color, ie., the color used for characters that are output using put . The alternate spelling is whatcolour .
	This program outputs the currently-active color number. The message is also given in the currently-active color.
Example	<pre>setscreen ("graphics")</pre>
	 put "This writing is in color number ", whatcolc
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. See setscreen for details.
See also	the <u>color procedure</u> , which is used to <u>set</u> the <u>color</u> . See also <u>colorback</u> and <u>whatcolorback</u> .
	See also predefined unit Text .

whatcolorback

color of background function

Syntax	whatcolorback : int
Description	The whatcolorback function is used to determine the current background color. The alternate spelling is whatcolourback .
	This program outputs the currently-active background color number. The background color of the message is determined by this number.
Example	<pre>setscreen ("screen")</pre>
	… put "The background of this writing" put "is in color number ", whatcolorback
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. Beware that the meaning of background color is different in these two modes. See colorback for details.
	color and whatcolor.
See also	See also predefined unit <u>Text</u> .

whatdotcolor

Syntax whatdotcolor (*x*, *y* : int) : int

DescriptionThe whatdotcolor function is used to determine the color numberof the specified pixel. The alternate spelling is whatdotcolour.

This program draws a line which bounces off the edges of the screen and makes a beep when it finds a pixel that has already been colored.

Example	<pre>setscreen ("graphics") var x, y : int := 0 var dx, dx : int := 1 loop if whatdotcolor (x, y) not= 0 then sound (400, 50) end if drawdot (x, y, 1) x := x + dx y := y + dy if x = 0 or x = maxx then dx := -dx end if if y = 0 or y = maxy then dy := -dy end if end loop</pre>
Details	The screen should be in a" <i>graphics</i> " mode. If is not set to " <i>graphics</i> " mode, it will automatically be set to " <i>graphics</i> " mode. See setscreen for details.
See also	drawdot , which is used for setting the <u>color</u> of a pixel. See also maxx and maxy , which are used to determine the number of pixels on the screen. See also sound , which causes the computer to make a <u>sound</u> .

See also predefined unit <u>View</u>.

cursor position function

Syntax	whatrow : int
Description	The whatrow function is used to determine the cursor position's row.
	This program outputs <i>The current row is 5, the current column is</i> 15.
Example	locate (5, 10) put "12345" put "The current row is", whatrow put "The current column is", whatcol
Details	The screen should be in a " <i>screen</i> " or " <i>graphics</i> " mode. whatrow functions properly even if the cursor is invisible.
See also	the <u>whatcol</u> function, which is used to <u>set</u> the determine the cursor column. See also the <u>locate</u> , <u>maxrow</u> and <u>maxcol</u> <u>procedure</u> .
	See also predefined unit <u>Text</u> .

whatrow

Window		
Description		ins the predefined subprograms that handle window nes to open, close, hide, show and select windows.
		the Window unit are exported qualified (and thus ed with " Window. ").
	window identif default Run win example, the fo	subprograms of the Window module all use fiers to indicate which window to act upon. The ndow can be specified by using defWindID . For ollowing code causes the main Run window to appea on and off three times.
Details	put" for <i>i</i> w d b end f	put some data to make the Run window appear Hello, World!" : 1 3 Helay (1000) Mindow.Hide (defWinID) Helay (1000) Mindow.Show (defWinID) For How are you?"
	<u>Open</u>	Opens a new execution window.
	<u>Close</u>	Closes an execution window.
	Select	Selects an execution window for output.
	GetSelect	Returns the currently-selected execution window.
	SetActive	Selects and activate (make front-most) an execution window.
Entry	GetActive	Gets the current active window.
Points	<u>GetPosition</u>	Get the screen position of an execution window.
	SetPosition	Set the screen position of an execution window.
	<u>Hide</u>	Hides an execution window.
	<u>Show</u>	Shows the current execution window.
	<u>Set</u>	Sets the configuration of the execution window.
	<u>Update</u>	Updates the onscreen window from the offscreen bitmap.

Window.Close

Part of <u>Window</u> module

Syntax	Window.Close (windowID : int)	
Description	The Window.Close procedure closes the window specified by the <i>windowID</i> parameter.	
	The following program opens a window, makes it active and then closes the window after getting a keystroke from the user.	
	% Open the window var winID : int winID := Window.Open ("position:300;300,graphics	
Example	<pre>% Draw the random ovals in the box var x, y, clr : int for : 1 20 x := Rand.Int (0, maxx) % Random x y := Rand.Int (0, maxy) % Random y clr := Rand.Int (0, maxcolor) % Random col Draw.FillOval (x, y, 30, 30, clr) end for</pre>	
	<pre>var ch : char := getchar % Wait for i</pre>	
	Window.Close (winID) % Close the winc	
Details	If a window is selected (i.e. output is going to that window) when it is closed, the main Run window becomes the selected window.	
	Exported qualified.	
Status	This means that you can only call the function by calling Window.Close , not by calling Close .	
See also	Window.Open and Window.Select.	

Window.GetActive

Syntax	Window.GetActive : int
Description	The Window.GetActive function returns the window ID of the active window. If the active window is a Turing run window, then Window.GetActive returns defWinID (which is -1) if the window is the default run window, or whatever number was returned from Window.Open for any other run window. If the active window is not a run window, then it returns -5 and sets Error.Last and Error.LastMsg to indicate the fact.
	An active window is defined as the window that has the input focus. This means that any typing will be sent to the active window. Under most systems an active window is indicated by a change in the appearance of the window.
	Exported qualified.
Status	This means that you can only call the function by calling Window.GetActive , not by calling GetActive .
See also	Window.SetActive.

Window.GetPosition

Syntax	Window.GetPosition (<i>windowID</i> : int , var <i>x</i> , <i>y</i> : int)	
Description	The Window.GetPosition procedure returns the location of the spectrum <i>y</i> parameters. The <i>x</i> and <i>y</i> parameters specify the lower left corner of lower left corner of the screen.	
	The following program outputs the current position of the run winc	
	% Constants for windows const titleBarHeight : int := 21 const windowEdgeSize : int := 13	
	% Calculate the actual size of a window var windowWidth : int := maxx + windowEdgeSize var windowHeight : int := maxy + windowEdgeSize	
Example	% Get the screen size var screenWidth : int := Config.Display (cdScree var screenHeight : int := Config.Display (cdScre	
	% Open the window var winID : int := Window.Open ("title:Upper Ric Window.SetPosition (winID, screenWidth windowWi	
	% Return the current position var windowXPosition, windowYPosition : int Window.GetPosition (winID, windowXPosition, winc put "Window located at ", windowXPosition, ",",	
Status	Exported qualified.	
	This means that you can only call the function by calling Window .	
See also	<u>Window.SetPosition</u> to <u>set</u> the current window position and <u>Confi</u>	

Window.GetSelect

Syntax	Window.GetSelect : int
Description	The Window.GetSelect function returns the window ID of the selected window. If the select window is the main (default) run window, then it returns <i>defWinID</i> (which is -1).
	A selected window is defined as the window that output will be sent to. It can be invisible. When a program starts execution, the selected window is the main Run window.
	Exported qualified.
Status	This means that you can only call the function by calling Window.GetSelect , not by calling GetSelect .
See also	Window.Select.

Window.Hide

Syntax	Window.Hide (windowID : int)
Description	The Window.Hide procedure hides the specified window. This means it disappears from the user's screen. However, it is still possible to select and draw the window while it remains hidden. If the user activates it (using Window.GetActive) it will automatically appear.
	To make a window appear after it's hidden, you use Window.Show.
Details	When a window is hidden, output to it is faster. It is quite possible for the you to hide a window, do complicated drawing to it and then make it appear in order to have the program execute faster.
	Exported qualified.
Status	This means that you can only call the function by calling Window.Hide , not by calling Hide .
See also	Window.Select and Window.SetActive.

Window.Open

Syntax	Window.Open (setUpString : string) : int
Description	The Window.Open function is used to create a window. A window ID is returned if the window is successfully created. If the window is not created then it returns 0. Error.Last and Error.LastMsg can then be used to determine the cause of the failure.
	The <i>setUpString</i> parameter is identical to that of View.Set . See View.Set for the list of options available.
	When the window is created, it is automatically selected (i.e. all output will be sent to that window unless redirected by a Window.Select command).
Example	The following program opens a window, makes it active and then close the window after getting a keystroke from the user.
	% Open the window var winID : int winID := Window.Open ("position:top;center,graph
	<pre>% Draw the random ovals in the box var x, y, clr : int for : 1 20 x := Rand.Int (0, maxx) % Random x y := Rand.Int (0, maxy) % Random y clr := Rand.Int (0, maxcolor) % Random col Draw.FillOval (x, y, 30, 30, clr) end for</pre>
	<pre>var ch : char := getchar % Wait for i</pre>
	Window.Close (winID) % Close the
	Exported qualified.
Status	This means that you can only call the function by calling Window.Open , not by calling Open .

View.Setfor the syntax of startUpString. See alsoSee alsoWindow.SelectMindow.SelectandWindow.SelectWindow.SetActive.

Window.Select

Syntax	Window.Select (windowID : int)
	The Window.Select selects the window that output is to be sent to.
Description	A selected window is defined as the window that output will be sent to. It can be invisible. When a program starts execution, the selected window is the main Run window.
	Exported qualified.
Status	This means that you can only call the function by calling Window.Select , not by calling Select .
See also	Window.Select and Window.SetActive.

Syntax Window.Set (*windowID* : **int**, *setUpString* : **string**)

The **Window.Set** procedure sets the configuration of the window specified by the *windowID* parameter. See **View.Set** for a complete list of available options. The *setUpString* parameter can be any combination options, separated by commas. Here is a selection of the available options.

text:<*rows*>;<*cols*> - Sets the output window to **text** mode and changes the window size to be <*rows*> rows by <*cols*> columns in size.

graphics:<*xsize*>;<*ysize*> - Sets the output window to **graphics** mode and changes the window size to be <*xsize*> pixels across and <*ysize*> pixels in height.

visible | invisible | popup - Sets the screen to be visible, invisible or popup. A popup window is hidden until output is sent to that window. The main Run window is a popup window. If you never send any output to it, it never appears.

noxor | **xor** - Sets whether all drawing operations draw using XOR.

nocursor | **cursor** - Sets whether the cursor is visible or not.

noecho | **echo** - Sets whether the input from the keyboard is echoed to the screen.

title:<*text*> - Sets the window title bar to <*text*>.

position: $\langle x \rangle$; $\langle y \rangle$ - Sets the position of the top left corner of the window to be ($\langle x \rangle$, $\langle y \rangle$).

offscreenonly | **nooffscreenonly** - Sets whether output to the window goes to offscreen window alone, or both the onscreen and offscreen window.

Exported qualified.

StatusThis means that you can only call the function by calling
Window.Set, not by calling Set.

See alsoView.Set
for a complete list of the options available with
Window.Set. See Window.Open
for how to create a new
window.

Window.SetActive

Syntax	Window.SetActive (windowID : int)
Description	The Window.SetActive procedure activates the window specified by the <i>windowID</i> parameter.
	An active window is defined as the window that has the input focus. This means that any typing will be sent to the active window. Under most systems an active window is indicated by a change in the appearance of the window.
Details	In general, it is unwise to change the active window. If the user is working on another program at the same time the program is running and the program executes the Window.SetActive procedure, she or he will suddenly be returned to OOT without warning.
	Exported qualified.
Status	This means that you can only call the function by calling Window.SetActive , not by calling SetActive .
See also	Window.GetActive and Window.Select.

Window.SetPosition

SyntaxWindow.SetPosition (windowID : int, x, y : int)DescriptionThe Window.SetPosition procedure moves the location of the specify the lower left corner of the window in screen coordinates. (
The following program opens four windows, one at each corner of % Constants for windows

const titleBarHeight : int := 21 const windowEdgeSize : int := 13 % Calculate the actual size of a window var windowWidth : int := maxx + windowEdgeSize var windowHeight : int := maxy + windowEdgeSize % Get the screen size var screenWidth : int := Config.Display (cdScree **var** screenHeight : **int** := **Config.Display** (cdScre Example % Open the window var winID1 : int := Window.Open ("title:Upper Ri Window.SetPosition (winID1, screenWidth windowh **var** winID2 : **int** := Window.Open ("title:Upper Le Window.SetPosition (winID2, 0, screenHeight wir var winID3 : int := Window.Open ("title:Lower Le Window.SetPosition (winID3, 0, 0) var winID4 : int := Window.Open ("title:Lower Ri Window.SetPosition (winID4, screenWidth windowh

Exported qualified.

Status This means that you can only call the function by calling **Window**.

See also <u>Window.GetPosition</u> to get the current window position and <u>Conf</u>

Window.Show

Syntax	Window.Show (windowID : int)
	The Window.Show procedure makes the specified window appear if it was invisible.
Description	To make a window disappear after it's visible, you use Window.Hide.
Details	When a window is hidden, output to it is faster. It is quite possible for the you to hide a window, do complicated drawing to it and then make it appear in order to have the program execute faster.
	Exported qualified.
Status	This means that you can only call the function by calling Window.Show , not by calling Show .
See also	Window.Select and Window.SetActive.

Window.Update

Syntax	Window.Update (windowID : int)
Description	The Window.Update procedure updates a specified Run window from an offscreen bitmap. It is used with the command View.Set ("offscreenonly") which prevents the Run window from being updated until the Window.Update command is given.
Description	This command is identical to the View.Update command except that it updates the specified window. It should be used when doing animation with more than one window.
See also	<u>View.Update</u> for information on flicker-free animation.

file statement

A writeStatement is:

Syntax write : fileNumber [:status], writeItem {, writeItem}

The write statement outputs each of the *writeItems* to the specified file. These items are output directly using the *binary* format that they have in the computer. In other words, the items are not in source (ASCII or EBCDIC) format. In the common case, these items will later be input from the file using the read statement. By contrast, the get and put statements use source format, which a person can read using a text editor.

This example shows how to output a complete employee record using a **write** statement.

```
var employeeRecord :
    record
        name : string ( 30 )
        pay : int
        dept : 0 .. 9
        end record
var fileNo : int
        open : fileNo, "payroll", write
...
write : fileNo, employeeRecord
```

An array, record or union may be read and written as a whole. The *fileNumber* must specify a file that is open with **write** capability (or else a program argument file that is implicitly opened).

The optional *status* is an **int** variable that is set to implementation-dependent information about the **write**. If *status* is returned as zero, the **write** was successful. If *status* is not returned as zero, *status* gives information about the incomplete or failed **write** (which is not documented here). Programmers often use *status* when they are writing a record or array to a file and are

write

Example

not sure if there is enough room on the disk to hold the item. If there is not enough room, the **write** will fail part way through, but the program can continue and diagnose the problem by inspecting *status*.

A writeItem is:

Details

```
reference [ : requestedSize [ : actualSize ] ]
```

Each *writeItem* is a variable or constant, to be written in internal form. The optional *requestedSize* is an integer expression giving the number of bytes of data to be written. The *requestedSize* should be less than or equal to the size of the item's internal form in memory (if it is not, a warning message is issued). If no *requestedSize* is given, the size of the item in memory is used. The optional *actualSize* is set to the number of bytes actually written.

See also write, open, close, seek, tell, get and put statements.

Description	When applied to set values, xor (symmetric difference) yields a set which includes element <i>e</i> if and only if <i>e</i> is contained in exactly one of the operands. When applied to non-negative integer values, xor yields a natural number whose bits are the xor of the corresponding bits of the operands. Both operands <i>A</i> and <i>B</i> are evaluated.
	Status <i>s</i> 3 will contain elements that are in <i>s</i> 1 or <i>s</i> 2 but not both. Here xor is a set operator. See enum and set types for explanations of these types.
Example	<pre>type status : enum (ready, sending, repeating) type statusSet : set of status var s1, s2, s3 : statusSet s1 := statusSet (status.read, status.sending) s2 := statusSet (status.read, status.repeating</pre>
	 s3 := s1 xor s2 % Same as (s1 + s2) - (s1 *
Example	Each bit of natural number $n3$ will be 1 if exactly one of the corresponding bits of $n1$ and $n2$ are 1. For example, if $n1 = 2#110$ (6) and $n2 = 2#010$ (2), $n3$ will be set to $2#100$ (4). Here xor is an integer operator.
-	var n1, n2, n3 : nat … n3 := n1 xor n2
Details	The xor operator is not a short circuit operator; in other words, both of its operands are always evaluated. The precedence of xor is the same as that of plus (+).
See also	set . See also <i>explicitIntegerConstant</i> which describes values such

See also as 2#110.

Syntax

A **xor** B

Turing Language Elements

- <u>#if used for conditional compilation</u>
- <u>addressint type</u>
- <u>all all members of a set</u>
- <u>and operator</u>
- anyclass the ancestor of all classes
- <u>array type</u>
- <u>assert statement</u>
- assignability of expression to variable
- assignment statement
- <u>begin statement</u>
- <u>bind declaration</u>
- <u>body declaration</u>
- <u>boolean true-false type</u>
- break debugger pause statement
- <u>case selection statement</u>
- <u>catenation (+) joining together strings</u>
- <u>char type</u>
- char(n) type
- <u>checked compiler directive</u>
- <u>class declaration</u>
- <u>close file statement</u>
- <u>collection declaration</u>
- <u>comment remark statement</u>
- <u>comparisonOperator</u>
- condition declaration
- const constant declaration
- constantReference use of a constant
- <u>declaration create a variable</u>
- deferred subprogram declaration
- <u>div integer truncating division operator</u>
- enum enumerated type
- enumeratedValue enumerated value
- <u>equivalence of types</u>
- <u>exit statement</u>

- explicitCharConstant character literal
- explicitConstant literal
- explicitIntegerConstant integer literal
- explicitRealConstant real literal
- <a>explicitStringConstant string literal
- explicitTrueFalseConstant boolean literal
- <u>expn expression</u>
- <u>export list</u>
- external declaration
- <u>false boolean value (not true)</u>
- <u>flexible array initialization</u>
- <u>for statement</u>
- <u>fork statement</u>
- forward subprogram declaration
- <u>free statement</u>
- <u>function declaration</u>
- <u>functionCall</u>
- get file statement
- handler exception handler
- id (identifier) name of an item in a program
- <u>if statement</u>
- implement clause
- <u>implement by clause</u>
- import list
- <u>in member of a set</u>
- <u>include source files</u>
- <u>indexType</u>
- indirection operator (@)
- <u>infix operator</u>
- inherit inheritance clause
- init array initialization
- int integer type
- <u>intn n-byte integer type</u>
- <u>invariant assertion</u>
- <u>loop statement</u>
- mod modulo operator
- module declaration
- monitor declaration
- <u>named type</u>

- <u>nat natural number type</u>
- <u>natn n-byte natural number type</u>
- <u>new statement</u>
- <u>nil pointer to a collection</u>
- not true/false (boolean) operator
- <u>opaque type</u>
- <u>open file statement</u>
- <u>or operator</u>
- paramDeclaration parameter declaration
- <u>pause statement</u>
- pervasive declaration modifier
- pointer type
- post assertion
- <u>pre assertion</u>
- precedence of operators
- prefix operator
- procedure declaration
- <u>procedureCall statement</u>
- process declaration
- program a (main) program
- <u>put statement</u>
- <u>quit fail statement</u>
- <u>read file statement</u>
- <u>real the real number type</u>
- <u>realn n-byte real number type</u>
- <u>record type</u>
- register use machine register
- <u>rem remainder operator</u>
- <u>result statement</u>
- <u>return statement</u>
- <u>scalar type</u>
- seek (file) statement
- separator between tokens in a program
- <u>set type</u>
- <u>setConstructor</u>
- <u>shl shift left operator</u>
- <u>shr shift right operator</u>
- signal wake up a process statement
- <u>skip used in get statement</u>

- <u>skip used in put statement</u>
- <u>standardType</u>
- <u>statement</u>
- <u>statementsAndDeclarations</u>
- <u>string type</u>
- <u>string comparison</u>
- <u>subprogramHeader</u>
- <u>subprogramType</u>
- <u>subrangeType</u>
- <u>substring of another string</u>
- <u>tag statement</u>
- <u>tell file statement</u>
- token in input
- true boolean value (not false)
- <u>type declaration</u>
- typeSpec type specification
- <u>unchecked compiler directive</u>
- <u>union type</u>
- <u>unit file containing module, monitor, or class</u>
- <u>unqualified export</u>
- <u>var declaration</u>
- <u>variableReference use of a variable</u>
- wait block a process statement
- <u>write file statement</u>
- <u>xor exclusive "or" operator</u>

Basic Predefined Subprograms

- Type Conversion
 - From Integer
 - intreal integer-to-real function
 - intstr integer-to-string function
 - From Real
 - <u>ceil real-to-integer function</u>
 - erealstr real-to-string function
 - <u>floor real-to-integer function</u>
 - frealstr real-to-string function
 - realstr real-to-string function
 - round real-to-integer function
 - From Nat
 - natreal natural number to real function
 - <u>natstr natural-number-to-string function</u>
 - From String
 - strint string-to-integer:function
 - strintok string-to-integer:function
 - strnat string to natural number function
 - strnatok string to natural number function
 - strreal string-to-real function
 - strrealok string-to-real function
 - To/From ASCII
 - <u>chr integer-to-character function</u>
 - ord character-to-integer function
- Maximum Numbers
 - maxint maximum integer function
 - maxnat maximum natural number function
 - minint minimum integer function
 - minnat minimum natural number function
- Math
 - <u>abs absolute value function</u>
 - arctan arctangent function (radians)
 - arctand arctangent function (degrees)
 - cos cosine function (radians)

- cosd cosine function (degrees)
- exp exponentiation function
- <u>In natural logarithm function</u>
- max maximum function
- min minimum function
- <u>sign function</u>
- <u>sin sine function (radians)</u>
- <u>sind sine function (degrees)</u>
- <u>sqrt square root function</u>
- Strings
 - index find pattern in string function
 - length of a string function
 - repeat make copies of string:function
- Enumerated Types
 - pred predecessor function
 - <u>succ successor function</u>
- Files
 - <u>eof end-of-file function</u>
- Arrays
 - <u>lower bound</u>
 - <u>upper bound</u>
- Random Numbers
 - rand random real number procedure
 - randint random integer procedure
 - <u>randnext procedure</u>
 - randomize procedure
 - <u>randseed procedure</u>
- Time
 - clock millisecs used procedure
 - <u>date procedure</u>
 - sysclock millisecs used procedure
 - time time of day as a string procedure
 - wallclock seconds since 1/1/1970 procedure
- Sound
 - play procedure
 - playdone function
 - <u>sound statement</u>
- System
 - <u>delay procedure</u>

- fetcharg fetch argument function
- getenv get environment function
- getpid get process id function
- <u>nargs number of arguments</u>
- <u>system statement</u>
- Parallel Port
 - parallelget parallel port function
 - parallelput parallel port procedure
- Mouse
 - buttonchoose switch mouse modes
 - <u>buttonmoved has a mouse event occurred</u>
 - buttonwait get a mouse event procedure
 - <u>mousewhere</u>
- Character Graphics
 - <u>cls clear screen graphics procedure</u>
 - color text color graphics procedure
 - <u>colorback background color procedure</u>
 - colour text color graphics procedure
 - <u>colourback background color procedure</u>
 - <u>locate procedure</u>
 - maxcol maximum column function
 - maxcolor graphics function
 - maxcolour graphics function
 - <u>maxrow maximum row function</u>
 - setscreen graphics procedure
 - whatcol cursor position function
 - whatcolor text color graphics function
 - whatcolorback color of background function
 - whatcolour text color graphics function
 - whatcolourback color of background function
 - <u>whatrow cursor position function</u>
- Pixel Graphics
 - <u>cls clear screen graphics procedure</u>
 - color text color graphics procedure
 - <u>colorback background color procedure</u>
 - colour text color graphics procedure
 - colourback background color procedure
 - drawarc graphics procedure
 - drawbox graphics procedure

- drawdot graphics procedure
- drawfill graphics procedure
- drawfillarc graphics procedure
- drawfillbox graphics procedure
- drawfillmapleleaf graphics procedure
- drawfilloval graphics procedure
- drawfillpolygon graphics procedure
- drawfillstar graphics procedure
- drawline graphics procedure
- drawmapleleaf graphics procedure
- drawoval graphics procedure
- drawpic graphics procedure
- drawpolygon graphics procedure
- drawstar graphics procedure
- locate procedure
- locatexy graphics procedure
- maxcol maximum column function
- maxcolor graphics function
- maxcolour graphics function
- maxrow maximum row function
- maxx graphics function
- maxy graphics function
- <u>setscreen graphics procedure</u>
- sizepic graphics function
- takepic graphics procedure
- whatcol cursor position function
- whatcolor text color graphics function
- whatcolorback color of background function
- whatcolour text color graphics function
- whatcolourback color of background function
- <u>whatdotcolor graphics function</u>
- whatdotcolour graphics function
- <u>whatrow cursor position function</u>
- Character Input
 - getch get character procedure
 - getchar get character function
 - hasch has character function
- Concurrency
 - empty condition function

- <u>getpriority function</u>
- <u>setpriority procedure</u>
- <u>simutime simulated time function</u>
- Addresses and Sizes
 - addr address of a variable
 - <u>sizeof size of a type</u>
- Bit Manipulation
 - <u>bits extraction</u>
- Classes
 - <u>objectclass of a pointer</u>
 - <u>self pointer to current object</u>
- Type Cheats
 - <u>cheat type cheating</u>

Predefined Modules

- <u>Concurrency</u>
 - <u>Concurrency.empty</u>
 - <u>Concurrency.getpriority</u>
 - <u>Concurrency.setpriority</u>
 - <u>Concurrency.simutime</u>
- <u>Config</u>
 - <u>Config.Display</u>
 - <u>Config.Lang</u>
 - Config.Machine
- <u>Dir</u>
 - <u>Dir.Change</u>
 - <u>Dir.Close</u>
 - <u>Dir.Create</u>
 - <u>Dir.Current</u>
 - <u>Dir.Delete</u>
 - <u>Dir.Get</u>
 - <u>Dir.GetLong</u>
 - <u>Dir.Open</u>
- <u>Draw</u>
 - Draw.Arc
 - Draw.Box
 - Draw.Cls
 - <u>Draw.Dot</u>
 - Draw.Fill
 - <u>Draw.FillArc</u>
 - <u>Draw.FillBox</u>
 - Draw.FillMapleLeaf
 - Draw.FillOval
 - <u>Draw.FillPolygon</u>
 - <u>Draw.FillStar</u>
 - Draw.Line
 - <u>Draw.MapleLeaf</u>
 - Draw.Oval
 - Draw.Polygon

- <u>Draw.Star</u>
- Draw.Text
- <u>Error</u>
 - Error.Last
 - <u>Error.LastMsg</u>
 - <u>Error.LastStr</u>
 - Error.Msg
 - Error.Str
 - Error.Trip
- ErrorNum
- Exceptions
- <u>File</u>
 - <u>File.Copy</u>
 - <u>File.Delete</u>
 - <u>File.DiskFree</u>
 - <u>File.Exists</u>
 - <u>File.Rename</u>
 - <u>File.Status</u>
- <u>Font</u>
 - <u>Font.Draw</u>
 - <u>Font.Free</u>
 - <u>Font.GetName</u>
 - <u>Font.GetSize</u>
 - Font.GetStyle
 - <u>Font.Name</u>
 - <u>Font.New</u>
 - Font.Sizes
 - <u>Font.StartName</u>
 - <u>Font.StartSize</u>
 - <u>Font.Width</u>
- <u>GUI</u>
 - GUI.AddLine
 - <u>GUI.AddText</u>
 - <u>GUI.Alert</u>
 - <u>GUI.Alert2</u>
 - <u>GUI.Alert3</u>
 - <u>GUI.AlertFull</u>
 - <u>GUI.Choose</u>
 - <u>GUI.ChooseFull</u>

- <u>GUI.ClearText</u>
- <u>GUI.CloseWindow</u>
- <u>GUI.CreateButton</u>
- <u>GUI.CreateButtonFull</u>
- <u>GUI.CreateCanvas</u>
- <u>GUI.CreateCanvasFull</u>
- <u>GUI.CreateCheckBox</u>
- <u>GUI.CreateCheckBoxFull</u>
- <u>GUI.CreateFrame</u>
- <u>GUI.CreateHorizontalScrollBar</u>
- <u>GUI.CreateHorizontalScrollBarFull</u>
- <u>GUI.CreateHorizontalSlider</u>
- <u>GUI.CreateLabel</u>
- <u>GUI.CreateLabelFull</u>
- <u>GUI.CreateLabelledFrame</u>
- <u>GUI.CreateLine</u>
- <u>GUI.CreateMenu</u>
- <u>GUI.CreateMenuItem</u>
- <u>GUI.CreateMenuItemFull</u>
- <u>GUI.CreatePicture</u>
- <u>GUI.CreatePictureButton</u>
- <u>GUI.CreatePictureButtonFull</u>
- <u>GUI.CreatePictureRadioButton</u>
- <u>GUI.CreatePictureRadioButtonFull</u>
- <u>GUI.CreateRadioButton</u>
- <u>GUI.CreateRadioButtonFull</u>
- <u>GUI.CreateTextBox</u>
- <u>GUI.CreateTextBoxFull</u>
- <u>GUI.CreateTextField</u>
- <u>GUI.CreateTextFieldFull</u>
- <u>GUI.CreateVerticalScrollBar</u>
- <u>GUI.CreateVerticalScrollBarFull</u>
- <u>GUI.CreateVerticalSlider</u>
- <u>GUI.Disable</u>
- <u>GUI.Dispose</u>
- <u>GUI.Draw...</u>
- <u>GUI.Enable</u>
- <u>GUI.FontDraw</u>
- <u>GUI.GetCheckBox</u>

- <u>GUI.GetEventTime</u>
- <u>GUI.GetEventWidgetID</u>
- <u>GUI.GetEventWindow</u>
- <u>GUI.GetHeight</u>
- <u>GUI.GetMenuBarHeight</u>
- <u>GUI.GetScrollBarWidth</u>
- <u>GUI.GetSliderValue</u>
- <u>GUI.GetText</u>
- <u>GUI.GetVersion</u>
- <u>GUI.GetWidth</u>
- <u>GUI.GetX</u>
- <u>GUI.GetY</u>
- <u>GUI.Hide</u>
- <u>GUI.HideMenuBar</u>
- <u>GUI.OpenFile</u>
- <u>GUI.OpenFileFull</u>
- <u>GUI.Pic...</u>
- <u>GUI.ProcessEvent</u>
- <u>GUI.Quit</u>
- <u>GUI.Refresh</u>
- <u>GUI.SaveFile</u>
- <u>GUI.SaveFileFull</u>
- <u>GUI.SelectRadio</u>
- <u>GUI.SetActive</u>
- <u>GUI.SetBackgroundColor</u>
- <u>GUI.SetBackgroundColour</u>
- <u>GUI.SetCheckBox</u>
- <u>GUI.SetDefault</u>
- <u>GUI.SetDisplayWhenCreated</u>
- <u>GUI.SetKeyEventHandler</u>
- <u>GUI.SetLabel</u>
- <u>GUI.SetMouseEventHandler</u>
- <u>GUI.SetNullEventHandler</u>
- <u>GUI.SetPosition</u>
- <u>GUI.SetPositionAndSize</u>
- <u>GUI.SetScrollAmount</u>
- <u>GUI.SetSelection</u>
- <u>GUI.SetSize</u>
- <u>GUI.SetSliderMinMax</u>

- <u>GUI.SetSliderReverse</u>
- <u>GUI.SetSliderSize</u>
- <u>GUI.SetSliderValue</u>
- <u>GUI.SetText</u>
- <u>GUI.SetXOR</u>
- <u>GUI.Show</u>
- <u>GUI.ShowMenuBar</u>
- <u>Input</u>
 - Input.KeyDown get keyboard state
 - Input.Pause pause for keystroke
 - <u>Input.getch</u>
 - Input.getchar
 - Input.hasch
- Joystick
 - Joystick.GetInfo
- <u>Keyboard</u>
- <u>Limits</u>
 - <u>maxint maximum integer function</u>
 - maxnat maximum natural number function
 - minint minimum integer function
 - minnat minimum natural number function
- <u>Math</u>
 - <u>abs absolute value function</u>
 - arctan arctangent function (radians)
 - arctand arctangent function (degrees)
 - cos cosine function (radians)
 - <u>cosd cosine function (degrees)</u>
 - exp exponentiation function
 - <u>In natural logarithm function</u>
 - max maximum function
 - min minimum function
 - <u>sign function</u>
 - <u>sin sine function (radians)</u>
 - <u>sind sine function (degrees)</u>
 - <u>sqrt square root function</u>
- <u>Mouse</u>
 - Mouse.ButtonChoose
 - <u>Mouse.ButtonMoved</u>
 - <u>Mouse.ButtonWait</u>

- <u>Mouse.Where</u>
- <u>Music</u>
 - <u>Music.Play</u>
 - <u>Music.PlayFile</u>
 - <u>Music.PlayFileStop</u>
 - <u>Music.Sound</u>
 - <u>Music.SoundOff</u>
- <u>Net</u>
 - <u>Net.BytesAvailable</u>
 - <u>Net.CharAvailable</u>
 - <u>Net.CloseConnection</u>
 - <u>Net.HostAddressFromName</u>
 - <u>Net.HostNameFromAddress</u>
 - <u>Net.LineAvailable</u>
 - <u>Net.LocalAddress</u>
 - <u>Net.LocalName</u>
 - <u>Net.OpenConnection</u>
 - <u>Net.OpenURLConnection</u>
 - <u>Net.TokenAvailable</u>
 - <u>Net.WaitForConnection</u>
- <u>PC</u>
 - <u>PC.ParallelGet</u>
 - <u>PC.ParallelPut</u>
 - parallelget parallel port function
 - parallelput parallel port procedure
- <u>Pic</u>
 - <u>Pic.Blend</u>
 - <u>Pic.Blur</u>
 - <u>Pic.Draw</u>
 - <u>Pic.DrawFrames</u>
 - <u>Pic.DrawFramesBack</u>
 - <u>Pic.DrawSpecial</u>
 - <u>Pic.DrawSpecialBack</u>
 - <u>Pic.FileNew</u>
 - <u>Pic.FileNewFrames</u>
 - <u>Pic.Flip</u>
 - <u>Pic.Frames</u>
 - <u>Pic.Free</u>
 - <u>Pic.Height</u>

- <u>Pic.Mirror</u>
- <u>Pic.New</u>
- <u>Pic.Rotate</u>
- <u>Pic.Save</u>
- <u>Pic.Scale</u>
- <u>Pic.ScreenLoad</u>
- <u>Pic.ScreenSave</u>
- <u>Pic.SetTransparentColor</u>
- <u>Pic.SetTransparentColour</u>
- <u>Pic.Width</u>
- <u>RGB</u>
 - <u>RGB.AddColor</u>
 - <u>RGB.AddColour</u>
 - <u>RGB.GetColor</u>
 - <u>RGB.GetColour</u>
 - <u>RGB.SetColor</u>
 - <u>RGB.SetColour</u>
 - <u>RGB.maxcolor</u>
 - <u>RGB.maxcolour</u>
- <u>Rand</u>
 - <u>Rand.Int</u>
 - <u>Rand.Next</u>
 - <u>Rand.Real</u>
 - <u>Rand.Reset</u>
 - <u>Rand.Seed</u>
 - <u>Rand.Set</u>
- <u>Sprite</u>
 - Sprite.Animate
 - Sprite.ChangePic
 - <u>Sprite.Free</u>
 - <u>Sprite.Hide</u>
 - Sprite.New
 - <u>Sprite.SetFrameRate</u>
 - Sprite.SetHeight
 - Sprite.SetPosition
 - <u>Sprite.Show</u>
- <u>Str</u>
 - index find pattern in string function
 - length of a string function

- repeat make copies of string:function
- <u>Stream</u>
 - <u>Stream.Flush</u>
 - <u>Stream.FlushAll</u>
 - <u>Stream.eof</u>
 - <u>eof end-of-file function</u>
- <u>Sys</u>
 - <u>Sys.Exec</u>
 - <u>Sys.FetchArg</u>
 - <u>Sys.GetComputerName</u>
 - <u>Sys.GetEnv</u>
 - <u>Sys.GetPid</u>
 - <u>Sys.GetUserName</u>
 - <u>Sys.Nargs</u>
 - getenv get environment function
 - getpid get process id function
- <u>Text</u>
 - <u>Text.Cls</u>
 - <u>Text.Color</u>
 - <u>Text.ColorBack</u>
 - <u>Text.Colour</u>
 - <u>Text.ColourBack</u>
 - <u>Text.Locate</u>
 - <u>Text.LocateXY</u>
 - <u>Text.WhatCol</u>
 - <u>Text.WhatColor</u>
 - <u>Text.WhatColorBack</u>
 - <u>Text.WhatColour</u>
 - <u>Text.WhatColourBack</u>
 - <u>Text.WhatRow</u>
 - <u>Text.maxcol</u>
 - <u>Text.maxrow</u>
- <u>Time</u>
 - <u>Time.Date</u>
 - <u>Time.DateSec</u>
 - <u>Time.Delay</u>
 - <u>Time.Elapsed</u>
 - <u>Time.ElapsedCPU</u>
 - <u>Time.PartsSec</u>

- <u>Time.Sec</u>
- <u>Time.SecDate</u>
- <u>Time.SecParts</u>
- <u>TypeConv</u>
 - From Integer
 - intreal integer-to-real function
 - intstr integer-to-string function
 - From Nat
 - natreal natural number to real function
 - <u>natstr natural-number-to-string function</u>
 - From Real
 - ceil real-to-integer function
 - erealstr real-to-string function
 - <u>floor real-to-integer function</u>
 - frealstr real-to-string function
 - realstr real-to-string function
 - round real-to-integer function
 - From String
 - strint string-to-integer:function
 - strintok string-to-integer:function
 - strnat string to natural number function
 - strnatok string to natural number function
 - strreal string-to-real function
 - strrealok string-to-real function
 - To/From ASCII
 - <u>chr integer-to-character function</u>
 - ord character-to-integer function
- <u>View</u>
 - <u>View.ClipAdd</u>
 - <u>View.ClipOff</u>
 - <u>View.ClipSet</u>
 - <u>View.Set</u>
 - <u>View.Update flicker-free animation</u>
 - <u>View.WhatDotColor</u>
 - <u>View.WhatDotColour</u>
 - <u>View.maxcolor</u>
 - <u>View.maxcolour</u>
 - <u>View.maxx</u>
 - <u>View.maxy</u>

- <u>Window</u>
 - <u>Window.Close</u>
 - <u>Window.GetActive</u>
 - <u>Window.GetPosition</u>
 - <u>Window.GetSelect</u>
 - <u>Window.Hide</u>
 - <u>Window.Open</u>
 - <u>Window.Select</u>
 - <u>Window.Set</u>
 - <u>Window.SetActive</u>
 - <u>Window.SetPosition</u>
 - <u>Window.Show</u>
 - <u>Window.Update</u>
- Miscellaneous Subprograms
 - Addresses and Sizes
 - <u>addr address of a variable</u>
 - size of size of a type
 - Arrays
 - lower bound
 - upper bound
 - Bit Manipulation
 - <u>bits extraction</u>
 - Classes
 - <u>objectclass of a pointer</u>
 - self pointer to current object
 - Enumerated Types
 - pred predecessor function
 - <u>succ successor function</u>
 - Type Cheats
 - <u>cheat type cheating</u>

The GUI Module

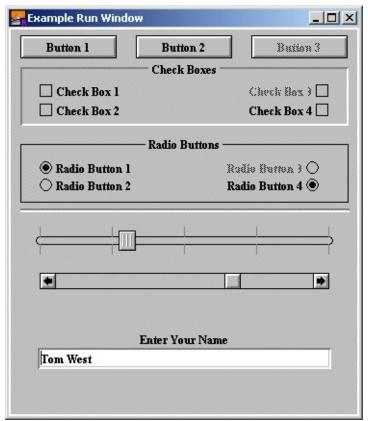
Introduction

Since the introduction of the Macintosh, graphical user interfaces (GUI) have been becoming more and more common. Most commercial programs written for either the Macintosh or Microsoft Windows make use of GUI elements to make their program easier to use.

For some time, students have been requesting methods of putting GUI elements such as buttons, check boxes, radio buttons, etc, into their Turing programs. Turing now includes a new set of predefined subprograms that allow students to add numerous GUI elements to their programs quickly and easily. These subprograms allow students to create: buttons, check boxes, radio buttons, sliders, scroll bars, picture buttons, radio picture buttons, text fields, lines, text labels, and frames.

The entire GUI Library is written in Turing and the source is included with the Turing distribution. The GUI library is completely procedure-oriented. This means that it is not necessary to know object-oriented programming or concepts in order to be able to use the library. Advanced students are welcome (in fact, encouraged) to look at the programs as an example of a large project written in Turing. We hope that there will be enterprising students who will be inspired to add new widgets to the library and encourage those who do so to submit them to Holt Software for possible inclusion into the next version of the library.

Here is a window with a few widgets.



Some GUI Widgets (Output of Example.dem)

The GUI library is usable by students who understand the concept of subprograms. In order to use the GUI library, students must write procedures (although they may be as simple as the student desires). We therefore suggest that teachers introduce students to the GUI library in a Grade 11 computer science course.

Note:Turing has not changed. It is not a visual building language. Students wishing to use the GUI library will be writing programs to create and use these GUI element, not spending their time visually building user interfaces (which may be fun, but teaches very little). In keeping with the tradition of Turing, the more the students learn about computer science, the more interesting their

programs will be, GUI or no GUI!

Terminology

The term "Widget" means any graphical user interface element. A button, a check box or radio button are all examples of widgets.

The term "Event" means either a keystroke or a mouse button being pressed in a window.

Example Programs

All example programs shown here are located in the **[Turing Directory]/Examples/GUI** directory. (In other words, start in the same directory as the Turing executable, move to the Examples folder and then the GUI folder.) All the available GUI widgets have example programs to demonstrate their use.

General Principles of the GUI Library

Here are some general instructions for the use of the GUI library. Read this section before looking at the specifics of various routines.

- All the subprograms for the GUI library are placed in a module called GUI. To call any of the subprograms, preface the name of the subprogram with *GUI*. For example, the *CreateLabel* subprogram would be called using *GUI*.*CreateLabel*.
- In general, most widgets have a Create subprogram. For example, buttons have a *CreateButton* subprogram, radio buttons have a *CreateRadioButton* subprogram, and so on. The Create subprogram takes as parameters things such as the location, the size of the GUI element, and the name of a procedure to be called when the widget is clicked. This procedure must be declared before the call to the Create subprogram.
- For most widgets, there are two forms of the Create subprogram. The Create subprogram and the CreateFull subprogram. The difference between the two is that the CreateFull subprogram allows the user to define more parameters that are otherwise set to default values. For example, the *GUI.CreateButton* procedure allows the user to specify the x and y location of the button, the width of the button, the text that appears in the button, and the procedure to call when the button is clicked. The *GUI.CreateButtonFull* routine specifies those and also allows the user to specify the height of the button (otherwise set to a height that will fit the label), a short cut keyboard character that allows the user to specify if this button is the "default" button (the one "pressed" if the user presses the Enter key).
- All Create subprograms return an integer. This number is the ID of the widget that has been created. You need to use this ID if you want to do anything to the widget later, such as move it, change its size, hide it, and so on. Most simple programs can safely ignore the widget ID, although they will need to handle the return value from the function.
- After all the widgets have been created, the program must repeatedly call *GUI*.*ProcessEvent* until the function returns true.

% Now process events until the user aborts the program. **loop exit when** *GUI.ProcessEvent*

end loop

GUI.ProcessEvent checks for user input from the mouse or the keyboard and then checks to see if the user has clicked on a widget. If the user has, then it responds appropriately (toggling the check box, pressing the button, etc.) and then if appropriate, calls the procedure the user supplied in the Create subprogram. *GUI.ProcessEvent* returns **true** when the *GUI.Quit* has been called, otherwise it returns **false**.

• When a program is finished execution (for example if the user selected "Quit" or "Exit" from the file menu), it should call the *GUI.Quit* procedure. This will cause the *GUI.ProcessEvent* loop to exit. The program should have any clean up code placed after the end loop.

Here is a very simple example of a program that puts "Hello" every time a button is pressed.

```
% The "Hello" program.
import GUI
View.Set ("graphics:200;200") % Shrink the run window
% The procedure called when the button is pushed.
procedure PutHello
    put "Hello"
end PutHello
% Create the button. The number returned is the ID number of the but
var b : int := GUI.CreateButton (100, 100, 0, "Say Hello", PutHello)
% Now process events until the user aborts the program.
loop
    exit when GUI.ProcessEvent
end loop
```

Here is the output window after the user has pressed the button twice.

Hello Run Window	
Hello	
Hello	
S	ay Hello

Output of Hello.dem

Active and Passive Widgets

Widgets come in two forms. Active widgets are ones that respond to keystrokes and button clicks. Passive widgets do not respond to anything. Examples of passive widgets are lines, frames, labels, labelled frames and pictures. Passive widgets are generally used to organize the output window.

Here is an example of a small program that show some passive widgets.

```
% The "Passive" program
% This demonstrates some of the passive widgets such as:
% Lines, Frames, Labelled Frames, Labels and Pictures.
import GUI
% We'll need a picture for our Picture widget.
                                                Most likely
% you would normally have it saved in an external file and
% use Pic.FileNew to read it into a picture. For the example
% program we'll construct it by hand.
Draw.FillOval (50, 50, 50, 50, blue)
Draw.FillBox (17, 17, 83, 83, brightred)
Draw.FillStar (17, 17, 83, 83, brightgreen)
Draw.FillMapleLeaf (37, 37, 63, 63, brightpurple)
var pic := Pic.New (0, 0, 100, 100)
View.Set ("graphics:310;335")
% The background must be gray for indented and exdented
% items to be visible.
GUI.SetBackgroundColor (gray)
% Now place the widgets.
% Three lines of the different types with labels
var line1 := GUI.CreateLine (70, 10, maxx - 10, 10, GUI.LINE)
var label1 := GUI.CreateLabelFull (60, 10, "Line", 0, 0,
    GUI.RIGHT + GUI.MIDDLE, 0)
var line2 := GUI.CreateLine (70, 30, maxx - 10, 30, GUI.INDENT)
var label2 := GUI.CreateLabelFull (60, 30, "Indent", 0, 0,
    GUI.RIGHT + GUI.MIDDLE, 0)
var line3 := GUI.CreateLine (70, 50, maxx - 10, 50, GUI.EXDENT)
var label3 := GUI.CreateLabelFull (60, 50, "Exdent", 0, 0,
    GUI.RIGHT + GUI.MIDDLE, 0)
% Now place the frames
var frame1 := GUI.CreateFrame (10, 70, 100, 120, GUI.LINE)
var label4 := GUI.CreateLabelFull (10, 70, "Line", 90, 50,
```

```
GUI.CENTER + GUI.MIDDLE, 0)
var frame2 := GUI.CreateFrame (110, 70, 200, 120, GUI.INDENT)
var label5 := GUI.CreateLabelFull (110, 70, "Indent", 90, 50,
    GUI.CENTER + GUI.MIDDLE, 0)
var frame3 := GUI.CreateFrame (210, 70, 300, 120, GUI.EXDENT)
var label6 := GUI.CreateLabelFull (210, 70, "Exdent", 90, 50,
    GUI.CENTER + GUI.MIDDLE, 0)
% Now place the labelled frames
var frame4 := GUI.CreateLabelledFrame (10, 140, 100, 190, GUI.LINE,
var frame5 := GUI.CreateLabelledFrame (110, 140, 200, 190, GUI.INDEN
    "Indent")
var frame6 := GUI.CreateLabelledFrame (210, 140, 300, 190, GUI.EXDEN
    "Exdent")
% Place the picture
var label7 := GUI.CreateLabel (30, 315, "Picture without merge")
var pic1 := GUI.CreatePicture (30, 210, pic, false)
var label8 := GUI.CreateLabel (maxx - 130, 315, "Picture with merge"
var pic2 := GUI.CreatePicture (maxx - 130, 210, pic, true)
% This loop doesn't do much since none of the widgets have any actio
1000
    exit when GUI.ProcessEvent
end loop
```

Here is the output window from the program with some labels, a line, a picture, and a labelled frame.

Passive	e Run Win	dow					
Pause	Stop	Print	Save	Paste			
Pictu	Picture without merge Picture with merge						
	٠						
			Y				
Liı	ne —	Ident		Exdent			
Liı	ne	Inden	t	Exdent			
Exden							
Inden Line							

Output of Passive.dem

When an active widget is initialized, usually an action procedure must be specified. This is the name of a procedure that will be called when the widget is selected. For example, in the Hello program, the PutHello procedure was specified as the action procedure of the button. Whenever the button was pressed, the PutHello procedure was called.

Some action procedures have arguments. For example, the action procedure for a slider has a parameter of the current value. This allows the procedure to use the current value without having to call a GUI subprogram to get the current slider value.

Keyboard Shortcuts

Several types of widgets can have "shortcuts". A shortcut is simply a keystroke that has the same effect as clicking on the widget. When you specify a shortcut to a widget in the CreateFull procedure for the widget, you must specify a single character. The easiest way to do this is to use the chr function with the ASCII value of the character to be used as the shortcut. You can also specify control characters using the "^" notation. For example, the character Ctrl+F can be expressed as "^F" in Turing.

The following characters cannot be used as shortcuts because the Turing environment uses them for various purposes (stopping or rerunning programs, and so on.): Ctrl+C, Ctrl+D, Ctrl+Z, F1, F11 and F12.

Background Color

It is common for windows to have a different background color from the standard white. To change the background color of a window, use the <u>GUI.SetBackgroundColor</u> procedure. This procedure takes one parameter, the new background color. It redraws the window in the background color and then redraws all the widgets. It also notifies the widgets about the new background color so that when the widget is erased or moved, the location of the widget is filled with the new background color instead of white.

Note that Microsoft Windows dialog boxes often have a background color of gray. In order to simulate that, you should give the command **GUI.SetBackgroundColor** (**gray**) before creating widgets.

Several widgets (Canvas, Frame, Labelled Frame, Text Field and Text Box) can have borders of either type INDENT or EXDENT. These borders give a sort of 3-D appearance to the widget. However, they require that the background be set to gray.

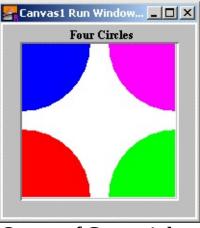
Here is an example of a small program that creates a Canvas with a 3-D appearance and then draws circles in the corner.

% The "Canvas1" program. % Create a canvas and draw four circles on it. import GUI View.Set ("graphics:200;200") % Necessary for a 3-D look for the canvas GUI.SetBackgroundColor (gray) % This procedure is needed as an argument to CreateCanvasFull. procedure DoNothing (mx, my : int) end DoNothing % Create a label for the canvas. We could use CreateLabelFull for m % precise alignment. var label1 := GUI.CreateLabel (70, 182, "Four Circles") % Create the canvas. We need to use CreateCanvasFull in order to % specify the type of border. var canvas := GUI.CreateCanvasFull (20, 20, 160, 160, GUI.INDENT,

DoNothing, DoNothing, DoNothing)

% Draw the four ovals. Notice that they don't extend off the canvas % and the co-ordinates they use are relative to the canvas, not the const radius := 70 % Half the width - 10 GUI.DrawFillOval (canvas, 0, 0, radius, radius, brightred) GUI.DrawFillOval (canvas, 160, 0, radius, radius, brightgreen) GUI.DrawFillOval (canvas, 0, 160, radius, radius, brightblue) GUI.DrawFillOval (canvas, 160, 160, radius, radius, brightblue)

Here is the output window.



Output of Canvas1.dem

Widget Sizes

The size that you specify a widget to be is not necessarily the actual size that the widget will appear. In fact for many widgets, you can specify a width and height of 0 for the widget and lets the initializer decide how large the widget should be. Another example is with check boxes, where if you specify the check box to be right justified, the x and y coordinates indicate the lower-right corner instead of the lower-left corner as usual. This means that you may have to do some experimentation to determine where you want the widgets to be placed. Read the page on each subprogram that you use to find out exactly what you are specifying with the x, y, width and height parameters.

If you are trying to align widgets together (for example aligning scroll bars with a canvas), use the <u>GUI.GetX</u>, <u>GUI.GetY</u>, <u>GUI.GetWidth</u>, and <u>GUI.GetHeight</u> functions to determine the size of the object.

Positioning Text Labels (Aligning Labels with Widgets)

It is very common to want to align text labels with widgets on the screen. There are a few tips and tricks to doing so successfully. To align a text label with a widget, it is simply a matter of using the GUI.CreateLabelFull function with the appropriate x, y, width, height and alignment arguments.

If you are left or right aligning a label, then generally you will want the x coordinate to specify the edge to be aligned from and the width parameter should be set to 0. Similarly, if you are top or bottom aligning a label, then the y coordinate should specify the edge to be aligned from and the height parameter should be set to 0.

To align a widget horizontally with a widget, choose GUI.CENTER for the horizontal alignment and the use the x coordinate and width of the widget as the label's x coordinate and width. You can get the x coordinate and width of a widget using GUI.GetX and GUI.GetWidth.

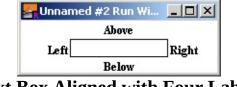
Likewise, to align a widget vertically with a widget, choose GUI.MIDDLE for the vertical alignment and the use the y coordinate and height of the widget as the label's y coordinate and height. You can get the y coordinate and height of a widget using GUI.GetY and GUI.GetHeight.

Here is an example illustrating the placement of a label at the center of each of four sides of a widget called w:

```
import GUI
View.Set ("graphics:200;50,nobuttonbar")
procedure DoNothing (text : string)
end DoNothing
var w : int := GUI.CreateTextField (50, 15, 100, "", DoNothing)
% These following lines are the important part of the program.
var left := GUI.CreateLabelFull (GUI.GetX (w) - 2, GUI.GetY (w),
        "Left", 0, GUI.GetHeight (w), GUI.RIGHT + GUI.MIDDLE, 0)
var above := GUI.CreateLabelFull (GUI.GetX (w),
        GUI.GetY (w) + GUI.GetHeight (w) + 2, "Above", GUI.GetWidth (w),
        GUI.CENTER + GUI.BOTTOM, 0)
```

```
var right := GUI.CreateLabelFull (GUI.GetX (w) + GUI.GetWidth (w) +
    GUI.GetY (w), "Right", 0, GUI.GetHeight (w), GUI.LEFT + GUI.MIDD
var below := GUI.CreateLabelFull (GUI.GetX (w), GUI.GetY (w) - 2,
    "Below", GUI.GetWidth (w), 0, GUI.CENTER + GUI.TOP, 0)
```

Here's the result. Note that the formula for aligning a label with a widget is the same for any type of widget.



Text Box Aligned with Four Labels

Here's an example illustrating aligning a widget with the top of the window. Notice that the label is center aligned with x of 0 and width of maxx, and top aligned with a y of maxy and a height of 0.

```
var title := GUI.CreateLabelFull (0, maxy, "Title", maxx, 0,
GUI.CENTER + GUI.TOP, 0)
```

Finally, here's an example illustrating the placement of a label in the center of the screen. Notice that the label is center aligned with x of 0 and width of maxx, and middle aligned with a y of 0 and a height of maxy.

```
var title := GUI.CreateLabelFull (0, 0, "Title", maxx, maxy,
GUI.CENTER + GUI.MIDDLE, 0)
```

Note that if a label's position or size is changed with <u>GUI.SetPosition</u>, <u>GUI.SetSize</u> or <u>GUI.SetPositionAndSize</u>, it still retains its alignment with respect to its new x, y, width, and height values.

Canvases

The canvas is a rather unique widget. It is essentially a drawing surface that you place in the window. There are calls using a canvas widget that essentially duplicate all the standard Draw module calls, along with calls corresponding to Font.Draw and various Pic module calls.

The difference is that the calls using the Canvas widget use (0, 0) to mean the bottom left corner of the canvas (not the window) and all drawing is clipped to the canvas (meaning that if you accidentally draw off the canvas, the part of the picture outside the bounds of the canvas will not appear). One of the most common bugs is to accidentally use the actual Draw module routines instead of the GUI.Draw routines when drawing in a canvas. If the drawing is goes outside the bounds of the Canvas, you have made this mistake.

Another feature of the Canvas widget is that you can specify a procedures to be called whenever a user clicks in the Canvas widget, drags the mouse with the mouse button down and then lets go of the mouse button. These procedures allow your program to respond to mouse activity taking place in the canvas widget.

Here is a program that uses a Canvas to allow the user to draw and a button to allow the user to erase the drawing.

```
% The "Draw" program
import GUI
View.Set ("graphics:300;300")
var oldx, oldy : int
var canvas : int % The drawing canvas.
var clear : int % The clear button.
% Called when the user presses the mouse button in the canvas.
% Sets the initial mouse position.
procedure MouseDown (mx, my : int)
        oldx := mx
        oldy := my
end MouseDown
```

% Called as the user drags the mouse with the button down in the can % Draws a line from the previous mouse position to the current posit

```
procedure MouseDrag (mx, my : int)
    GUI.DrawLine (canvas, oldx, oldy, mx, my, colorfg)
    oldx := mx
    oldy := my
end MouseDrag
% Called when the mouse button is released.
procedure DoNothing (mx, my : int)
end DoNothing
% Called when the clear button is pressed.
procedure Clear
    GUI.DrawCls (canvas)
end Clear
% Create the canvas
canvas := GUI.CreateCanvasFull (10, 30, maxx - 20, maxy - 40, 0,
    MouseDown, MouseDrag, DoNothing)
% Create the clear button
clear := GUI.CreateButton (maxx div 2 - 20, 0, 40, "Clear", Clear)
loop
    exit when GUI.ProcessEvent
end loop
```

Here is the output window after the user has drawn some lines.

Unnamed #3 Run Window						
Pause	Stop	Print	Save	Paste		
	R,	u	y Le	-		
Clear						

Output of Draw.dem

Multiple Windows

Turing allows for multiple run windows. This can be used to add extra functionality to programs, however there are a few issues that must be understood before multiple windows can be successfully used.

Turing uses the concept of selected windows and active windows. A selected window is determined by the program and is changed by Window.Select. The selected window is the window in which all output appears. When a widget is created, it is automatically created in the selected window.

An active window is last window on which the user clicked. The active window is shown by having its title bar highlighted. When a user types, all keystrokes are sent to the active window.

It is entirely possible to have the selected window and the active window be to different windows.

When you call getch, Mouse.ButtonWait, or any other input routine, Turing checks only the selected window. The GUI Library works around this by selecting all the windows that have widgets in them (one at a time, starting with the active window) and checking each for events.

If you are processing an event from one of several windows, make certain that the correct window is selected before you output your results. Note that the widgets automatically select the correct window, so there is no need to change the selected window before making any calls to the GUI module.

If you close a window with widgets in it, use <u>GUI.CloseWindow</u> to close the window. This removes all the widgets in the window before closing it. If you have several windows with widgets and want some windows to be hidden and then shown later, use the <u>GUI.ShowWindow</u> and <u>GUI.HideWindow</u> to show and hide the windows with widgets in them.

The GUI Library Internals

While it is not necessary to know the internals of the GUI Library to use it, we provide this brief overview for those who wish to understand the inner workings of the library.

The GUI Library consists of four parts. The only part visible to the user is the GUI module. This is located in "%oot/lib/GUI", where %oot is the directory in which the OOT executable is located. The GUI module is essentially a series of procedures that provide a front end to the Widget Module and the Widget Class Library.

The Widget Module is a module called WidgetModule that consists of a series of subprograms that cover all the aspects of GUI's that do not pertain to a particular widget. For example, the procedure to change the background color is here, as well as the procedure to process an event. It is located in "%oot/lib/GuiClass/wdgtmod.tu"

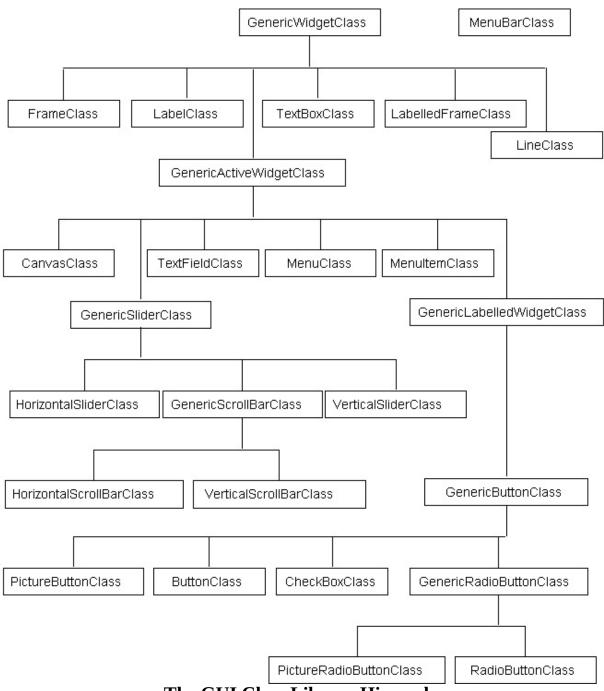
The GUI Class Library consists of a series of classes arranged in a hierarchy illustrated in the following figure. Most of the actual Turing code for the GUI Library is located in the Widget Class Library. Each different type of widget has its own class. Widgets that share common behavior have the same parent. For example, both the vertical and horizontal slider have a slider class as a parent. Those classes whose names start with Generic are abstract classes. They should not be instantiated themselves. They are used to define common behavior among their subclasses. The Turing source for the classes can be found in the directory "%oot/lib/GuiClass"

The fourth part is the WidgetGlobals module. This module that consists mostly of global variables used by the GUI Class Library and the Widget module. It is located in "%oot/lib/GuiClass/wdgtglob.tu"

Here is an example of how the GUI module works: when you create a button using GUI.CreateButton, the CreateButton function in the GUI module creates an object of type ButtonClass. (ButtonClass is found in the Widget Class Library discussed further down). It then calls the Initialize procedure of the ButtonClass to initialize the button with the specified parameters. Finally it assigns an ID number to the widget and arranges it in a table for future reference. Here is another example: when you call a procedure like GUI.Show, the Show procedure takes the widget ID, looks up the object that it represents, and then calls the Show procedure of the object.

Students who wish to add new widgets to the GUI library will have to understand the principles of object oriented programming, as they will be adding a new class to the GUI Class Library and then adding new subprograms in the GUI module that will call their new classes. (At the very least, a Create subprogram will be required for the new widget.)

A suggested project would be to create new versions of the ButtonClass, CheckBoxClass and RadioButtonClass classes that are buttons, check boxes and radio buttons with the new Windows 95/NT appearance. They could be called the Button95Class, CheckBox95Class and RadioButton95Class. Properly written, these new classes should inherit from ButtonClass, etc. and contain only those procedures that differ from the base class (presumably the procedures that display the widget).



The GUI Class Library Hierarchy

GUI Module Routines Summary

The routines in the GUI module are divided into several different types. There are the routines to create various widgets, the routines to create menus and menu items, the routines to do general activities (such as processing an event, changing the background color, etc.) and the routines that act on various types of widgets.

Here is the list of the routines that create widgets:

Create a button.
Create a check box.
Create a radio button.
Create a picture button.
Create a picture radio button.
Create a horizontal slider.
Create a vertical slider.
Create a horizontal scroll bar.
Create a vertical scroll bar.
Create a canvas.
Create a text field.
Create a text box.
Create a multi-line selector.
Create a line.
Create a frame.
Create a labelled frame.
Create a label.
Create a picture.

Here is the list of routines that create menus and menu items:

GUI.CreateMenuCreate a menu.GUI.CreateMenuItem, GUI.CreateMenuItemFullCreate a menu item.

Here is the list of general routines:

GUI.ProcessEvent	Process a single keyboard or mouse down event.
<u>GUI.Quit</u>	Tell the program to exit the event loop.
<u>GUI.ResetQuit</u>	Reset the "quit" flag so a program can reenter the event loop.
<u>GUI.Refresh</u>	Redraw all the widgets on the screen.
GUI.SetBackgroundColor	Change the window's background colour.
<u>GUI.SetNullEventHandler</u>	Set the null event handler.
<u>GUI.SetKeyEventHandler</u>	Set the keystroke event handler.
GUI.SetMouseEventHandler	Set the mouse event handler.
<u>GUI.HideMenuBar</u>	Hide the menu bar in the window.
GUI.ShowMenuBar	Show the menu bar in the window.
<u>GUI.GetEventWidgetID</u>	Get the selected widget's ID (used in a widget's action procedure).
<u>GUI.GetEventWindow</u>	Get the window that the event took place in (used in a widget's action procedure).
<u>GUI.GetEventTime</u>	Get the time (in milliseconds) that the event took place (used in a widget's action procedure).
GUI.GetScrollBarWidth	Return the width of a scroll bar.
<u>GUI.GetMenuBarHeight</u>	Return the height of a menu bar.
GUI.GetVersion	Return the current version number of the GUI module.

Here is a list of routines that act on the widgets and the sort of widgets they act on.

	Display the widget.	All
<u>GUI.Hide</u>	Hide the widget.	All

<u>GUI.GetX</u>	Return the x coordinate of the widget's left edge.	All
<u>GUI.GetY</u>	Return the y coordinate of the widget's bottom edge.	All
<u>GUI.GetWidth</u>	Return the widget's actual width.	All
<u>GUI.GetHeight</u>	Return the widget's actual height.	All
<u>GUI.Dispose</u>	Dispose of the widget.	All
<u>GUI.SetPosition</u>	Set the widget's position.	All
<u>GUI.SetSize</u>	Set the widget's size.	All
GUI.SetPositionAndSize	Set the widget's position and size.	All
<u>GUI.Enable</u>	Enable the widget to respond to events.	Active Widgets
<u>GUI.Disable</u>	Disable the widget from responding to events.	Active Widgets
<u>GUI.SetLabel</u>	Set the widget's text label.	Button, Check Box, Radio Button, Label, Labelled Frame
	Set the button's	

<u>GUI.SetColor</u>	color.	Button
<u>GUI.SetDefault</u>	Make the button the default button.	Button
<u>GUI.GetCheckBox</u>	Get whether a check box is filled.	Check Box
<u>GUI.SetCheckBox</u>	Set a check box to be filled or not.	Check Box
<u>GUI.SelectRadio</u>	Select a radio button.	Radio Button, Picture Radio Button
<u>GUI.GetSliderValue</u>	Return the current value of the slider.	-
<u>GUI.SetSliderValue</u>	Set the value of the slider.	Slider, Scroll Bar
<u>GUI.SetSliderMinMax</u>	Set the slider's minimum and maximum.	Slider, Scroll Bar
<u>GUI.SetSliderSize</u>	Set the slider's length (or height).	Slider, Scroll Bar
<u>GUI.SetSliderReverse</u>	Reverse the direction of the slider.	Slider, Scroll Bar
<u>GUI.SetScrollAmount</u>	Set the scroll bar's thumb size and the scroll amount for arrows/page up and down.	Scroll Bar
DrawArc, DrawBox, DrawCls, DrawDot, DrawFill, DrawFillArc, DrawFillBox, DrawFillMapleLeaf, DrawFillOval, DrawFillPolygon, DrawFillStar, DrawLine, DrawMapleLeaf, DrawOval, DrawPolygon,	Routines that perform the same function as the Draw module for the Canvas	Canvas

<u>DrawStar, DrawText</u>	widget.	
	"Font.Draw" for	
<u>GUI.FontDraw</u>	the Canvas	Canvas
	widget.	
<u>PicDraw, PicNew, PicScreenLoad,</u> PicScreenSave	Routines that perform the same function as the Pic module for the Canvas widget.	Canvas
<u>GUI.SetXOR</u>	Performs View.Set ("xor") for the Canvas Widget.	Canvas
<u>GUI.SetText</u>	Set the text of a text field.	Text Field
<u>GUI.SetEchoChar</u>	Set the character that appears when user enters a keystroke.	Text Field
<u>GUI.SetSelection</u>	Set the selection in the text field.	Text Field
<u>GUI.SetActive</u>	Make the text field the active one (where keystrokes will go and where the cursor blinks).	Text Field
<u>GUI.ClearText</u>	Clear a text box.	Text Box
<u>GUI.AddText</u>	Add text to a text box.	Text Box
<u>GUI.AddLine</u>	Add a line of text to a text box.	Text Box
<u>GUI.SetTopLine</u>	Scrolls text box to make line appear at top.	Text Box

<u>GUI.SetScrollOnAdd</u>	Sets whether text box scrolls when text added.	Text Box
---------------------------	--	----------

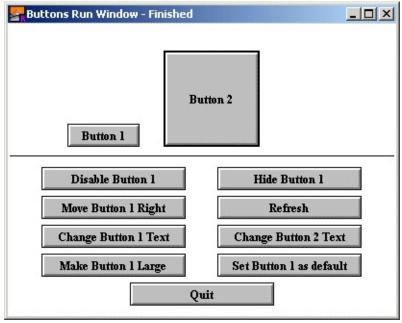
Widgets - Common Routines

All of the procedures in this section can be used with any widget, although some may have no effect (for example GUI.GetX on a menu item is meaningless).

GUI.Show	Displays a widget. Used in conjunction with Hide to hie widgets. Hidden widgets cannot get events (i.e. respond keystrokes or mouse clicks).
<u>GUI.Hide</u>	Hides a widget. Used in conjunction with Show to hide widgets. Hidden widgets cannot get events (i.e. respond keystrokes or mouse clicks). If an active text field (see hidden, then any keystrokes in the window will be igno
	Returns the x (y) coordinate of the left edge of a widget this may be different from the x coordinate specified in Create call. For example, if a radio button is created wir justification, the x coordinate in the Create method spec right edge.
<u>GUI.GetX</u> <u>GUI.GetY</u>	Here is a small subprogram that should draw a rectangl around a widget (i.e. no part of the widget should stick
	<pre>procedure WidgetRect (widgetID : int) const x : int := GUI.GetX (widgetID) const y : int := GUI.GetY (widgetID) const width : int := GUI.GetWidth (widgetI const height : int := GUI.GetHeight (widge Draw.Box (x, y, x + width - 1, y + height end WidgetRect</pre>
<u>GUI.GetWidth</u> <u>GUI.GetHeight</u>	Returns the actual width (height) of a widget. Note that different from the width specified in the Create call (esj since many widgets allow you to specify 0 for the widtl GUI module determine the necessary width).
<u>GUI.Dispose</u>	Eliminates a widget. It should be called in order to free memory that the widget might have allocated. Note that use the widget after it has been disposed of. If you wish temporarily get rid of a widget, consider using the Hide then the Show method when you want to use it again.

	Moves a widget to the specified location. If the widget
	is moved immediately to the new location. If the widge
	will appear at the new location when the Show procedu
GUI.SetPosition	Note that the location specified in GUI.SetPosition are
	in the Create method. For example, if you had specified
	to be right justified in the CreateCheckBoxFull function
	location in a call to SetPosition would specify the lowe
	as opposed to the lower-left corner.
	Changes the size of a widget. If the widget is visible, its
	changed immediately, otherwise the widget will appear
	size when the widget is next made visible. Note that the
<u>GUI.SetSize</u>	height parameters are not necessarily the actual width a
	the widget. For example, the TextField widget ignores t
	parameter, calculating the widget's actual height from tl
	the text in the TextField.
	Changes the position and size of the widget simultaneo
GUI.SetPositionAndSize	works the same way as the SetPosition and SetSize pro-

Widgets - Buttons



Output of Buttons.dem

The button widget is used to implement a textual button. When you click on a button, the button's action procedure is called. If a button is given a short cut, then entering the keystroke will cause the action procedure to be called. It will not visibly cause the button to depress.

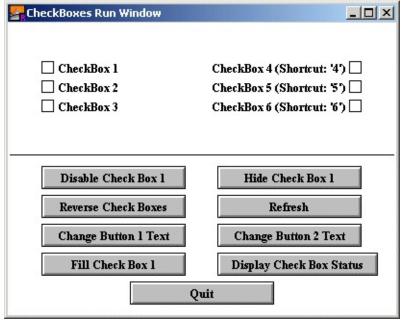
If a button's width or height is set to zero (or not specified at all), then the button is shaped to fit the text.

A button can be the default button for a window. If that is the case, then the button will be drawn with a ticker border around it and if the user presses ENTER, then the button's action procedure will be called.

When a button is not enabled, the text in the button is grayed out and the button no longer responds to any mouse clicks or keystrokes until the button is enabled again.

<u>GUI.CreateButton</u>	Creates and displays a button. GUI.CreateButton specifies the location, width, text and action procedure of the button.
<u>GUI.CreateButtonFull</u>	Creates and displays a button. GUI.CreateButtonFull also specifies the height, keyboard shortcut and default status of the button.
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a button. Disabled buttons have their text grayed out and cannot get events (i.e. respond to keystrokes or mouse clicks).
<u>GUI.SetLabel</u>	Changes the text of a button.
<u>GUI.SetDefault</u>	Sets the "default status" of a button. If a button is the default button, then it is drawn with a heavy outline and it is activated when the user presses ENTER (RETURN on a Macintosh).

Widgets - Check Boxes



Output of CheckBoxes.dem

The check box widget is used to implement a check box that can be set or unset. When you click on a check box, the status of the check box flips from set to unset and back again and the check box's action procedure is called with the new status as a parameter. If a check box is given a short cut, then entering the keystroke will cause the check box to change status and the action procedure to be called. The new status will be displayed immediately.

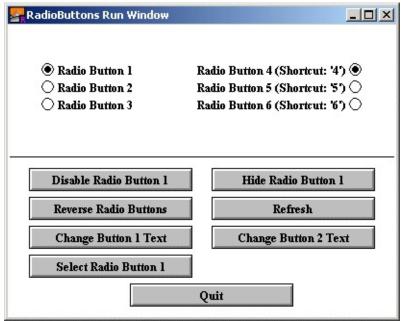
A check box's size is not specified during creation. It is determined based on the size of the text. Instead the user specifies the lower-left corner of the check box (or the lower-right if the check box is right justified).

When a check box is not enabled, the label beside the check box is grayed out

and the check box no longer responds to any mouse clicks or keystrokes until the check box is enabled again.

<u>GUI.CreateCheckBox</u>	Creates and displays a left aligned (check box to the left of the label) check box. GUI.CreateCheckBox specifies the location, text and action procedure of the check box.
<u>GUI.CreateCheckBoxFull</u>	Creates and displays a check box. GUI.CreateCheckBoxFull also specifies the alignment of the check box (whether the checkbox is the right or left of the text) and the check box's keyboard shortcut.
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a check box. Disabled check boxes have their text grayed out and cannot get events (i.e. respond to keystrokes or mouse clicks).
<u>GUI.SetLabel</u>	Changes the text of a check box.
<u>GUI.GetCheckBox</u>	Returns a check box's status. If the check box is set (has an X in it), GUI.GetCheckBox returns true, otherwise it returns false.
<u>GUI.SetCheckBox</u>	Sets the status of a check box. It calls the check box's action procedure with the new status and redraws the widget with the new status.

Widgets - Radio Buttons



Output of RadioButtons.dem

The radio button widget is used to implement a set of buttons of which one and only one button must be selected at all times. (Think old-style radio station button. Selecting one "deselects" the previously-selected station.) When you click on a radio button, any other radio button that is part of the set is deselected and the radio button's action procedure is called. If a radio button is given a short cut, then entering the keystroke will cause the radio button to be selected (and any other radio button in the group to be de-selected) and the action procedure to be called. The newly-selected or deselected radio buttons will be displayed immediately.

When a radio button is created, the widget ID of another radio button must be supplied. A value of zero for the widget ID indicates that this radio button is part

of a new group. The widget ID must be the ID of the last radio button added to the group. Because radio buttons are almost always placed in groups you can specify -1 for the x and y coordinates and the radio button will be placed just below the previous radio button and retain the same alignment. When a group of radio buttons is selected, the first radio button created in the group will be the selected one. You can change this by using the GUI.SelectRadio procedure to select a different one.

A radio button's size is not specified during creation. It is determined based on the size of the text. The user specifies the lower-left corner of the radio button (or the lower-right if the radio button is right justified).

When a radio button is not enabled, the label beside the radio button is grayed out and the radio button no longer responds to any mouse clicks or keystrokes until the radio button is enabled again.

<u>GUI.CreateRadioButton</u>	Creates and displays a left aligned (circle to the left of the label) radio button. GUI.CreateRadioButton specifies the location, text, the radio button to be joined to and the action procedure of the radio button.
<u>GUI.CreateRadioButtonFull</u>	Creates and displays a radio button. GUI.CreateRadioButtonFull also specifies the alignment of the radio button (whether the circle is the right or left of the text) and the radio button's keyboard shortcut.
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a radio button. Disabled radio buttons have their text grayed out and cannot get events (i.e. respond to keystrokes or mouse clicks).
<u>GUI.SetLabel</u>	Changes the text of a radio button.
GUI.SelectRadio	Selects a radio button. The previously-selected radio button is "de-selected". The action procedure of the radio button is called.

Widgets - Picture Buttons

PictureButtons Run Window	
Maple Pressed	
* 🕈 🖲 🔳	
Disable Buttons	Hide Star
Move Star Right	Refresh
Make Star Large	
Qu	it

Output of PictureButtons.dem

The picture button widget (hereafter simply called a button) is simply a button with a picture on it instead of text. The picture must be created by the program beforehand using Pic.New or Pic.FileNew. The resulting picture can then be used as a parameter to GUI.CreatePictureButton. In general, pictures should be a maximum of about 30 pixels high and wide, although there is no built in limit in the GUI library.

When you click on a picture button, the picture button's action procedure is called. If a picture button is given a short cut, then entering the keystroke will cause the action procedure to be called. It will not visibly cause the button to depress.

If a button's width or height is set to zero (or not specified at all), then the button

is shaped to fit the picture.

When a picture button is not enabled, the picture button is grayed out and the picture button no longer responds to any mouse clicks or keystrokes until the button is enabled again.

	Creates and displays a picture button. The button
	is automatically sized to fit the picture. If you
GUI.CreatePictureButton	need to know the precise size of the button, use
<u>GOI. Cleater Icture Dutton</u>	the GUI.GetWidth and GUI.GetHeight functions.
	GUI.CreatePictureButton specifies the location,
	picture id and action procedure of the button.
	Creates and displays a picture button.
	GUI.CreatePictureButtonFull also specifies the
GUI.CreatePictureButtonFull	width, height and keyboard shortcut of the button.
	It also specifies whether the button picture should
	be merged with the background color or not.
	Enables (disables) a picture button. Disabled
<u>GUI.Enable</u>	picture buttons are grayed out and cannot get
<u>GUI.Disable</u>	events (i.e. respond to keystrokes or mouse
	clicks).

Widgets - Picture Radio Buttons

Picrbtns Run Window	
Circle Pressed	
Disable Buttons	Hide Star
Move Star Left	Refresh
Make Star Large	luit

Output of PictureRadioButtons.dem

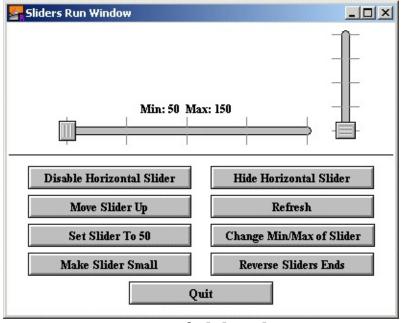
The picture radio button widget (hereafter simply called a button) is simply a picture button (see Widget - Picture Button) that has the behavior of a radio button. This means that one and only one picture radio button of a group is selected at any time. A selected picture radio button is displayed as being pressed.

When you click on a picture button, the previously-selected picture radio button will be de-selected and the new picture button's action procedure is called. If a picture button is given a short cut, then entering the keystroke will cause the action procedure to be called and the picture radio button will be drawn selected.

Creates and displays a picture radio button. The button is automatically sized to fit the

<u>GUI.CreatePictureRadioButton</u>	picture. If you need to know the precise size of the button, use the GUI.GetWidth and GUI.GetHeight functions. GUI.CreatePictureRadioButton specifies the location, picture id and action procedure of the button as well as the radio picture button to be joined to.
<u>GUI.CreatePictureRadioButtonFull</u>	Creates and displays a picture radio button. GUI.CreatePictureRadioButtonFull also specifies the width, height and keyboard shortcut of the button. It also specifies whether the button picture should be merged with the background color or not.
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a picture radio button. Disabled picture buttons are grayed out and cannot get events (i.e. respond to keystrokes or mouse clicks).

Widgets - Sliders



Output of Sliders.dem

Sliders are the equivalent of a volume control on a stereo. To control a slider, the user simply clicks on the control knob and slides the control left and right (up and down for a vertical slider). Whenever the user slides the control knob, the action procedure of the widget is called with the current value as a parameter.

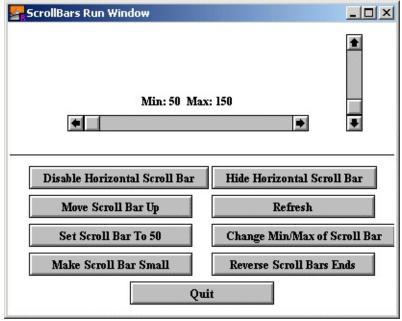
The range of values that the slider will give is determined by the min and max parameters in the Create call. The left side of the slider (bottom for vertical sliders) represents the minimum value, while the right (top) represents the maximum value.

In some instances, you will want the reverse to be true (right/top is minimum). In that case, call the GUI.SetSliderReverse procedure to flip the values of the slider.

Sliders always have a fixed height (for horizontal sliders) or width (for vertical sliders). The length parameter in the Create call specifies how long the slider should be.

<u>GUI.CreateHorizontalSlider</u>	Creates and displays a horizontal (left-right) slider. GUI.CreateHorizontalSlider specifies the location, length, minimum and maximum values for the slider, the initial value of the slider, and the action procedure of the slider.
<u>GUI.CreateVerticalSlider</u>	Creates and displays a vertical (up-down) slider. GUI.CreateVerticalSlider specifies the location, length, minimum and maximum values for the slider, the initial value of the slider, and the action procedure of the slider.
<u>GUI.Enable</u>	Enables (disables) a slider. Disabled sliders cannot
<u>GUI.Disable</u>	get events (i.e. respond to mouse clicks).
<u>GUI.GetSliderValue</u>	Returns the current value of a slider.
<u>GUI.SetSliderValue</u>	Sets the value of a slider. It moves the control knob on the slider to the appropriate location and calls the slider's action procedure with the new value.
<u>GUI.SetSliderMinMax</u>	Sets the minimum and maximum values of a slider. It redraws the control knob to take into account the new minimum and maximum. If the current value of the slider is outside the new min/max, then the value is adjusted appropriately.
<u>GUI.SetSliderSize</u>	Changes the length of a slider. Redraws the slider and changes the position of the control knob to take into account the new size of the slider.
<u>GUI.SetSliderReverse</u>	Sets a slider into (or out of, if already into) "reverse mode". Normally, a slider is at its minimum value when the control knob is on the left side (bottom for a vertical slider). This reverses it, so the minimum value is when the slider is at the right side (top for vertical sliders) of the track. Calling this routine a second time reverses it back to normal. This procedure redraws the slider to move the control knob to its new location.

Widgets - Scroll Bars



Output of ScrollBars.dem

Scroll bars are usually used to allow a user to see a piece of a document that cannot be displayed on the screen in its entirety. The picture above shows the scroll bars appearance. To control a scroll bar, there are a few choices: the user can click on the thumb (the box in the scroll bar) and slide it up and down, or the user can click in the scroll bar itself above or below the thumb (in which case the thumb is moved up or down one "page"), or the user can click on the up or down arrows at the ends of the scroll bar (in which case the thumb is moved up one "line"). The programmer defines a page or a line. When the value of the scroll bar changes, the action procedure of the scroll bar is called with the new value as a parameter.

The range of values that the scroll bar will give is determined by the min and

max parameters in the Create call. The left side of the scroll bar (bottom for vertical scroll bars) represents the minimum value, while the right (top) represents the maximum value. There is also the "thumb size". This represents the range of values that can be seen at once on the screen.

For example, if you have a window that can display 20 lines of text at once and there are 100 lines of text, you would set min to 1, max to 100, and thumbSize to 20. The value returned by the scroll bar would then be the line number of the first line on the screen to be displayed. When the scroll bar was at its maximum value, it would return 81, since by doing so, lines 81-100 would be displayed.

When a scroll bar is disabled or the scroll bar's thumb size is greater than the difference between the minimum and maximum values (i.e. the item being scrolled fits in the window), the scroll bar is deactivated. The bar is drawn in white rather than gray and the arrows are grayed out. The scroll bar does not respond to mouse clicks.

In some instances, you will want the minimum and maximum values of the scroll bar to be reversed (right/top is minimum). In that case, call the GUI.SetSliderReverse procedure to flip the values of the scroll bar.

Scroll bars always have a fixed height (for horizontal scroll bars) or width (for vertical scroll bars). To get the scroll bar's width, use the GUI.GetScrollBarWidth function. The length parameter in the Create call specifies how long the scroll bar should be.

Creates and displays a horizontal (leftright) scroll bar. GUI.CreateHorizontalScrollBar specifies the location, length, minimum and maximum values for the scroll bar, the initial value of the scroll bar, and the scroll bar's action procedure.

By default, the arrow increment (the amount the value is changed when the scrolling arrows are pressed) is set to one. The page up/down increment (the amount the value is changed when the user clicks in

GUI.CreateHorizontalScrollBar

	the bar to the right or left of the thumb) is set to one quarter the difference between the minimum and the maximum. The "thumb size" is set to zero (see the description of scroll bars for an explanation of the thumb size).
	Creates and displays a vertical (up-down) scroll bar. GUI.CreateVerticalScrollBar specifies the location, length, minimum and maximum values for the scroll bar, the initial value of the scroll bar, and the scroll bar's action procedure.
<u>GUI.CreateVerticalScrollBar</u>	By default, the arrow increment (the amount the value is changed when the scrolling arrows are pressed) is set to one. The page up/down increment (the amount the value is changed when the user clicks in the bar to the right or left of the thumb) is set to one quarter the difference between the minimum and the maximum. The "thumb size" is set to zero (see the description of scroll bars for an explanation of the thumb size).
GUI.CreateHorizontalScrollBarFull	Creates and displays a horizontal (left- right) scroll bar. GUI.CreateHorizontalScrollBarFull also
<u>GUI.CreateVerticalScrollBarFull</u>	specifies the arrow increment, page increment, and thumb size for the scroll bar. Creates and displays a horizontal (left- right) scroll bar. GUI.CreateVerticalScrollBarFull also specifies the arrow increment, page increment and thumb size for the scroll bar.
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a scroll bar. Disabled scroll bars cannot get events (i.e. respond to

	mou
GUI.GetSliderValue	Retu
	Sets
GUI.SetSliderValue	cont
	appr
	bar's
	Sets
	a sci
GUI.SetSliderMinMax	take
Sensetsnachminnan	max
	bar i
	valu
	Cha
GUI.SetSliderSize	the s
	the c
	new
	Sets
	into
	is at
	knot
	verti
<u>GUI.SetSliderReverse</u>	mini
	the r
	the t
	reve redra
	knot
	Sets incre
GUI.SetScrollAmount	SCro
	SCIU.

nouse clicks).

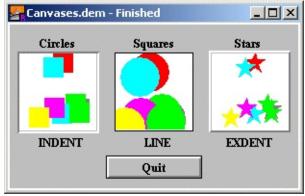
Returns the current value of a scroll bar. Sets the value of a scroll bar. It moves the control knob on the scroll bar to the appropriate location and calls the scroll bar's action procedure with the new value. Sets the minimum and maximum values of a scroll bar. It redraws the control knob to cake into account the new minimum and maximum. If the current value of the scroll bar is outside the new min/max, then the value is adjusted appropriately.

Changes the length of a scroll bar. Redraws he scroll bar and changes the position of he control knob to take into account the new size of the scroll bar.

Sets a scroll bar into (or out of, if already into) "reverse mode". Normally, a scroll bar is at its minimum value when the control knob is on the left side (bottom for a vertical scroll bar). This reverses it, so the minimum value is when the scroll bar is at the right side (top for vertical scroll bars) of the track. Calling this routine a second time reverses it back to normal. This procedure redraws the scroll bar to move the control knob to its new location.

Sets a scroll bar's arrow increment, page increment, and thumb size. Redraws the scroll bar to take into account the new thumb size.

Widgets - Canvases



Output of Canvases.dem

A canvas is a drawing surface for use by the program. It differs from just using the window surface to draw on in that (0, 0) represents the lower-left corner of the canvas and all drawing is clipped to the canvas. (This means that if you accidentally attempt to draw outside of the canvas, it will not actually draw beyond the border of the canvas.)

Canvases have procedures that emulate all the procedures in the Draw module as well as a procedure to emulate Font.Draw, Pic.Draw, Pic.New, Pic.ScreenLoad, and Pic.ScreenSave.

You can get mouse feedback from a canvas. Using the GUI.CreateCanvasFull method, you can specify three routines that are called when the mouse button is depressed while pointing in a canvas. One routine will be called when the user presses the mouse button down in a canvas. Another routine will be called while the user drags the mouse with the mouse button down. This routine is repeatedly called whenever the mouse changes position while the mouse button is down. The last routine is called when the mouse button is released. All three routines

take an x and y parameter, which is the location of the mouse with respect to the canvas (i.e. (0, 0) is the lower-left corner of the canvas).

<u>GUI.CreateCanvas</u>	Creates and displays a canvas. GUI.CreateCanvas specifies the location and size of the canvas. The canvas will have a line border around it.
<u>GUI.CreateCanvasFull</u>	Creates and displays a canvas. GUI.CreateCanvasFull also specifies the type of border and three procedures to be called when a mouse is pressed, dragged or released on the canvas.
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a canvas. Disabled canvases cannot get events (i.e. respond to mouse clicks). If no mouse routines were specified (i.e. the canvas was created with GUI.CreateCanvas and not GUI.CreateCanvasFull) this routine essentially does nothing.
<u>GUI.DrawArc</u>	
<u>GUI.DrawBox</u>	
<u>GUI.DrawCls</u>	
<u>GUI.DrawDot</u>	
<u>GUI.DrawFill</u>	
GUI.DrawFillArc	
<u>GUI.DrawFillBox</u>	
GUI.DrawFillMapleLea	<u>f</u>
GUI.DrawFillOval	All these routines draw to a canvas in the same
<u>GUI.DrawFillPolygon</u>	manner as the similarly named $Draw $, Pic Φ and
GUI.DrawFillStar	Font.Draw subprograms.
<u>GUI.DrawLine</u>	All coordinates are based on the canvas and all
<u>GUI.DrawBox</u>	drawing is clipped to the canvas drawing surface. If
<u>GUI.DrawMapleLeaf</u>	the canvas is in "xor mode", all the drawing will be
GUI.DrawOval	done with "xor" set. (See View.Set for more
<u>GUI.DrawPolygon</u>	information about "xor".)
<u>GUI.DrawStar</u>	
<u>GUI.DrawText</u>	
GUI.FontDraw	
<u>GUI.PicDraw</u>	
<u>GUI.PicNew</u>	
GUI.PicScreenLoad	

GUI.PicScreenSave

GUI.SetXOR

Sets the "xor mode" of a canvas. When in "xor mode", all the Draw... procedures of a canvas are treated as if the View.Set ("xor") statement had been executed before the Draw procedure.

Widgets - Text Fields

FextFields Run Window 💶 🗵 🗙			
Pause	Stop	Print	Save
	Record 1	ofl	
Name	Tom West		
Address	203 College	e St	
Age	41		
	Add Re	cord	*

Output of TextFields.dem

A text field is a box for entering one line of text. When the user presses ENTER, the text field's action procedure is called.

Only one text field is active at a time. The active text field has a blinking cursor (or its selection highlighted). If a keystroke occurs when a window has an active text field in it, the keystroke will be directed to the active text field. You can change which text field is active with the GUI.SetActive procedure or by simply clicking on another text field with the mouse.

When multiple text fields are created in a window, the first text field created is active when the program begins.

The current version of the text field does not support cut and paste or keyboard commands to extend the selection.

Because strings are a maximum of 255 characters, this is the maximum number of characters in a text field.

The TAB character cycles between different text fields in a window. It cycles through the text fields in the order in which they were created. BACK TAB (shift+TAB) cycles through the fields in reverse order.

<u>GUI.CreateTextField</u>	Creates and displays a text field. GUI.CreateTextField specifies the location, width, initial text string, and action procedure of the text field. The height of the text field is determined by the height of the font used by the text field. The text field will have a line border around it.
<u>GUI.CreateTextFieldFull</u>	Creates and displays a text field. GUI.CreateTextFieldFull also specifies the type of border, font for entered text, and kind of input restriction (integer only, etc.)
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a text field. Disabled picture buttons are grayed out and cannot get events (i.e. respond to keystrokes or mouse clicks).
<u>GUI.SetText</u>	Sets the text of a text field. The cursor is set the beginning of the text.
<u>GUI.GetText</u>	Returns the current text of a text field.
GUI.SetSelection	Sets the selection (the selected text) in a text field.
<u>GUI.SetActive</u>	Makes a text field active.

Widgets - Text Boxes



Output of TextBoxes.dem

A text box is a box used for displaying larger quantities of text. It has both vertical and horizontal scroll bars to allow the user to scroll through all the text in the box.

<u>GUI.CreateTextBox</u>	Creates and displays a text box. GUI.CreateTextBox specifies the location and size of the text box. The text box will have a line border around it.
GUI.CreateTextBoxFull	Creates and displays a text box. GUI.CreateTextBoxFull also specifies the type of border and the font for displayed text.
<u>GUI.ClearText</u>	Clears all the text in a text box.
	Adds text to the current line of the text box. Does not

<u>GUI.AddText</u>	add a newline after the text. Equivalent to put text This scrolls the text box (if necessary) so that the added text is now visible. To move the cursor to the end of the text without adding any extra text, call GUI.AddText with "" for the text parameter.
GUI.AddLine	Adds text to the current line of the text box followed by a newline. Equivalent to put text. This scrolls the text box (if necessary) so that the added text is now visible.

Widgets - Lines

🔄 Lines Run Window		
Line Widgets		
Line Indent Exdent		
Hide Lines	Move Line Right	
Refresh	Make Line Longer	
Quit		

Output of Lines.dem

Lines are organizational elements that make the window look better and help organize the GUI elements.

Creates and displays a line. GUI.CreateLine specifies the end GUI.CreateLine points of the line (which must be either vertical or horizontal) and the type of line.

Widgets - Frames

Frames Run Window		
Frame Widgets		
Line	Indent	Exdent
Hide Frames		Move Frame Right
Refresh		Make Frame Taller
Quit		

Output of Frames.dem

Frames are organizational elements that make the window look better and help organize the GUI elements. Frames and labelled frames are the only widgets in which other widgets can be placed.

Creates and displays a frame. GUI.CreateFrame specifies the <u>GUI.CreateFrame</u> coordinates of the lower-left and upper-right corner of the frame and the type of border of the frame.

Widgets - Labelled Frames

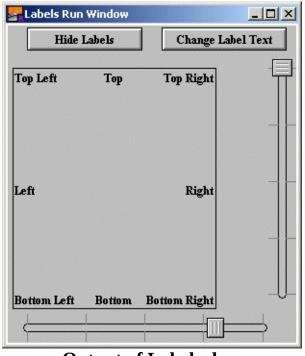
LabelledFrames Run Window				
Pause Stop Print	Save Paste			
Labelled Fran	me Widgets			
Hide Frames Move Frame Right				
Refresh	Make Frame Taller			
Change Labels				
Quit				

Output of LabelledFrames.dem

Labelled frames are organizational elements that make the window look better and help organize the GUI elements. Frames and labelled frames are the only widgets in which other widgets can be placed.

	Creates and displays a labelled frame.
	GUI.CreateLabelledFrame specifies the coordinates
GUI.CreateLabelledFram	e of the lower-left and upper-right corner of the frame,
	the type of border of the frame, and the text of the
	frame's label.
<u>GUI.SetLabel</u>	Changes the text of a labelled frame.

Widgets - Labels



Output of Labels.dem

Labels are organizational elements that make the window look better and help organize the GUI elements. They are simply text placed in a window. To aid in aligning text with various widgets, it is possible to align text in a larger region (as shown in the figure).

	Creates and displays a label. GUI.CreateLabel specifies
<u>GUI.CreateLabel</u>	the lower-left corner of the text and the text itself. The
	system font is used to display the label.
	Creates and displays a label. GUI.CreateLabelFull also
	specifies the width, height, alignment, and font for the
<u>GUI.CreateLabelFull</u>	label. The width and height are specified for alignment

purposes. Changes the text of a label.

GUI.SetLabel

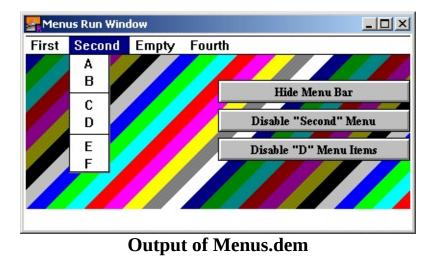
Widgets - Pictures

Pictures Run Window	
Ficture (no merge)	
Hide Pictures Move Pictures Quit	Right

Output of Pictures.dem

Pictures are organizational elements that make the window look better and help organize the GUI elements. They are simply a picture placed in a window. The pictures are specified using a picture ID from any of the Pic subprograms. Creates and displays a picture. GUI.CreatePicture specifies GUI.CreatePicture the location, picture ID, and whether the picture should be merged with the background.

Widgets - Menus



Menus are used in most modern interfaces. In order to create a full set of menus, you must create the menu and then create the menu items in that menu. The

menus are automatically added to the menu bar of the selected menu.

Menu items are the individual entries of a menu. To create menus for a window, you must create a menu, then create the menu items for that menu, then create the next menu, then the items for that menu, etc. All menu items are automatically added to the last menu and after the last menu item of the currently selected (not active!) window.

When you select an item in a menu, the action procedure of the item is called. The action procedure has no parameters.

As of the v1.0 release of the GUI Library, it is an error to create a menu item without having created a menu first. In future releases it will be possible to create menus and attach and remove them from menu bars when desired.

Menus and menu items can be enabled and disabled. A disabled menu item is grayed out. When the user selects the menu, all items in the menu appear disabled and cannot be selected. A disabled menu item is grayed out when the menu is displayed. The user cannot select the menu item.

Separators in a menu appear as a solid line across the menu. These are created by creating a menu item whose text is three dashes "---".

<u>GUI.CreateMenu</u>	Creates and displays a menu. The menu will be added after the other menus in the menu bar. If there are no previous menus, then a menu bar is automatically created and the menu added. GUI.CreateMenu specifies the text that will appear in the menu bar. It is suggested that the text not have any spaces in it.
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a menu. Disabled menus are grayed out in the menu bar. If selected, all the menu items in the menu bar appear disabled and cannot be selected.
<u>GUI.CreateMenuItem</u>	Creates a menu item. GUI.CreateMenuItem specifies the text of the menu item and the action procedure to be called when the menu item is selected. The menu item will be added to the last menu after the other menu items in the menu. If there are no menus defined, an error results.
GUI.CreateMenuItemFul	Creates a menu item. GUI.CreateMenuItemFull also specifies a shortcut keystroke.
<u>GUI.Enable</u> <u>GUI.Disable</u>	Enables (disables) a menu item. Disabled menu items are grayed out when the menu is displayed and cannot be selected by the user.

Widgets - General Routines

The following procedures are included in the GUI module but do not relate to a specific widget.

This function processes a single event (a mouse button press or a keystroke). If the event activates a widget, then the action procedure of the widget is called. To find out which widget was activated and called the action procedure (necessary if several widgets have the same action procedure), you can call GUI.GetEventWidgetID. To get the exact time that the event occurred, you can call GUI.GetEventTime. To get the window in which the event took place, you can call GUI.GetEventWindow.

If a mouse click occurred, but did not activate any widget, then the default mouse event handler is called. By default, this does nothing. However, if you want your program to respond to mouse events that do not affect a widget, call GUI.SetMouseEventHandler to specify your own default mouse event handler.

If a keystroke occurred, but did not activate any widget (i.e. it wasn't a short cut for a widget and there are no text fields in the window) then the default keystroke handler is called. By default, this does nothing. However, if you want your program to respond to keystroke events that do not affect a widget, call

GUI.SetKeyEventHandler to specify your own default key event handler.

If no event occurred, then the null event handler is called. By default, this does nothing. However, if you want your program to perform some action repetitively when it is not doing anything else, then call GUI.SetNullEventHandler to specify your own null event handler. The null event

GUI.ProcessEvent

handler is often used for such things as updating a clock and making certain that music is playing in the background.

This procedure causes GUI.ProcessEvent to return true. If the program is structured properly with a

loop exit when GUI.ProcessEvent end loop

at the end of the program, then the program will exit the loop after finishing the current action procedure. This procedure is usually called from the action procedure of a Quit button or Exit menu item.

This routine redraws all the widgets in the currently-selected window. This is used when some form of drawing may have overwritten the widgets in a window. It is used by the GUI Library to redraw all the widgets when the background color of a window has changed. Changes the background color of the currently-selected window. (Both spellings of color are acceptable.) This does not change the value of color 0 in the window. Instead it fills the entire window with the new background color and then redraws all the widgets. The usual background color outside of white is gray.

Sets the new null event handler. The specified procedure will be called every time GUI.ProcessEvent is called and there is no keystroke or mouse button pressed.

Sets the new default mouse event handler. The specified procedure will be called every time GUI.ProcessEvent is called and there is a mouse button pressed which is not handled by any widget.

GUI.Quit

GUI.Refresh

<u>GUI.SetBackgroundColor</u> <u>GUI.SetBackgroundColour</u>

GUI.SetNullEventHandler

GUI.SetMouseEventHandler

<u>GUI.SetKeyEventHandler</u>	Sets the new default keystroke event handler. The specified procedure will be called every time GUI.ProcessEvent is called and there is a keystroke which is not handled by any widget.
<u>GUI.HideMenuBar</u>	Hides the menu bar in the selected window. No menu items can be selected when the menu bar is hidden. (Menu item shortcuts will be ignored when the menu bar is hidden.)
GUI.ShowMenuBar	Shows the menu bar in the selected window.
<u>GUI.GetEventWidgetID</u>	Returns the widget ID of the widget that was activated by the mouse button press or the keystroke. This function should only be called in an action procedure, as it will return -1 when there is no event that activated a widget being processed.
<u>GUI.GetEventWindow</u>	Returns the window ID of the window in which the event (mouse button or keystroke) took place. This function should only be called in an action procedure or in a default mouse or keystroke event handler, as it will return -1 when there is no event being processed.
<u>GUI.GetEventTime</u>	Returns the time in milliseconds when the event (mouse button or keystroke) took place. This value is the same value that Time.Elapsed returns if called when the event was processed. This function should only be called in an action procedure or in a default mouse or keystroke or null event handler, as it will return -1 when there is no event being processed.
<u>GUI.SetDisplayWhenCreated</u>	Sets whether widgets are automatically displayed when created, or whether GUI.Show must be called first. By default, this is set to true (widgets are displayed when created). However, there may be times when you want to create a widget and then make several additional calls before displaying the widget. Closes a window with widgets in it. This

GUI.ShowWindowShows a hidden window with widgets in it. This procedure makes certain that the GUI Library recognizes that the window is visible. This procedure will call Window.Show, so there is no need for the user to do so.GUI.HideWindowHides a visible window with widgets in it. This procedure makes certain that the GUI Library recognizes that the window is no longer visible. This procedure will call Window.Hide, so there is no need for the user to do so.GUI.GetScrollBarWidthReturns the width of a scroll bar. Useful when placing a scroll bar widget beneath another object.GUI.GetMenuBarHeightReturns the height of the menu bar. Useful when placing widgets to make certain that they do not overlap the menu.GUI.GetVersionReturns the current version of the GUI module. Because the GUI module is expected to grow, new versions will probably be made available at Holt Software's web site located at http://www.holtsoft.com/turing. If you wish to use features that do not appear in earlier versions of the library, you can have your program check that the current available version returns an integer from 100 - 999 and is read as 1.00 to 9.99.	GUI.CloseWindow	procedure automatically disposes of any widgets in the window and makes certain that the GUI Library recognizes that the window no longer exists. This procedure will call Window.Close, so there is no need for the user to do so.
GUI.HideWindowprocedure makes certain that the GUI Library recognizes that the window is no longer visible. This procedure will call Window.Hide, so there is no need for the user to do so. Returns the width of a scroll bar. Useful when placing a scroll bar widget beneath another object.GUI.GetMenuBarHeightReturns the height of the menu bar. Useful when placing widgets to make certain that they do not overlap the menu. Returns the current version of the GUI module. Because the GUI module is expected to grow, new versions will probably be made available at Holt Software's web site located at 	GUI.ShowWindow	procedure makes certain that the GUI Library recognizes that the window is visible. This procedure will call Window.Show, so there is no
GUI.GetScrollBarWidthplacing a scroll bar widget beneath another object.GUI.GetMenuBarHeightReturns the height of the menu bar. Useful when placing widgets to make certain that they do not overlap the menu.Returns the current version of the GUI module. Because the GUI module is expected to grow, new versions will probably be made available at Holt Software's web site located at 	<u>GUI.HideWindow</u>	procedure makes certain that the GUI Library recognizes that the window is no longer visible. This procedure will call Window.Hide, so there is
GUI.GetMenuBarHeightplacing widgets to make certain that they do not overlap the menu.Returns the current version of the GUI module. Because the GUI module is expected to grow, new versions will probably be made available at Holt Software's web site located at http://www.holtsoft.com/turing. If you wish to use features that do not appear in earlier versions 	GUI.GetScrollBarWidth	placing a scroll bar widget beneath another
GUI.GetVersionBecause the GUI module is expected to grow, new versions will probably be made available at Holt Software's web site located at http://www.holtsoft.com/turing. If you wish to use features that do not appear in earlier versions of the library, you can have your program check that the current available version meets the programs needs. GUI.GetVersion returns an	<u>GUI.GetMenuBarHeight</u>	placing widgets to make certain that they do not
	<u>GUI.GetVersion</u>	Because the GUI module is expected to grow, new versions will probably be made available at Holt Software's web site located at http://www.holtsoft.com/turing. If you wish to use features that do not appear in earlier versions of the library, you can have your program check that the current available version meets the programs needs. GUI.GetVersion returns an

Predefined Functions and Procedures

abs	addr	arctan	arctand	anyclass
break	buttonchoose	buttonmoved	buttonwait	ceil
chr	clock	cls	color	colorback
colour	colourback	COS	cosd	date
delay	drawarc	drawbox	drawdot	drawfill
drawfillarc	drawfillbox	drawfillmapleleaf	drawfilloval	drawfillpo
drawfillstar	drawline	drawmapleleaf	drawoval	drawpic
drawpolygon	drawstar	empty	eof	erealstr
exp	fetcharg	floor	frealstr	getch
getchar	getenv	getpid	getpriority	hasch
index	intreal	intstr	length	ln
locate	locatexy	lower	max	maxcol
maxcolor	maxcolour	maxint	maxnat	maxrow
maxx	maxy	min	minint	minnat
mousehide	mouseshow	mousewhere	nargs	natreal
natstr	nil	ord	palette	play
playdone	pred	rand	randint	randnext
randomize	randseed	realstr	repeat	round
setpriority	setscreen	sign	simutime	sin
sind	sizeof	sizepic	sound	sqrt
strint	strintok	strnat	strnatok	strreal
strrealok	SUCC	sysclock	sysexit	system
takepic	time	upper	wallclock	whatcol
whatcolor	whatcolorback	whatcolour	whatcolourback	whatdotco
whatdotcolour	whatpalette	whatrow	whattextchar	whattextc
whattextcolorback	« whattextcolour	whattextcolourback	X	

Predefined Modules

Brush	Button	CheckBox	Comm	Concurrency
Config	Dir	Draw	DropBox	EditBox
Error	ErrorNum	Event	File	Font
GUI	Input	Joytick	Keyboard	Limits
ListBox	Math	Menu	Mouse	Music
Net	Obsolete	PC	Pen	Pic
Print	RadioButton	Rand	RGB	Sound
Sprite	Str	Stream	Student	Sys
Text	Time	TypeConv	Video	View
Window				

Predefined OOT Constants

(... means several constants with the prefix, see the module for a complete list)

black	blue	brightblue	brightcyan
brightmagenta	brightpurple	brightred	brightwhite
brushErrorBase	cdMaxNumColors	cdMaxNumColours	cdMaxNumP
cdScreenWidth	clLanguageVersion	clMaxNumDirStreams	clMaxNumR
clRelease	cmFPU	cmOS	cmProcessor
colourbg	colorfg	colourfg	configErrorB
darkgray	darkgrey	defFontID	defWinID
e (ErrorNum)	errWinID	excp (Exceptions)	fileErrorBase
fontErrorBase	fsysErrorBase	generalErrorBase	gray
grey	guiErrorBase	joystick1	joystick2
magenta	mouseErrorBase	musicErrorBase	ootAttr (Fil
penErrorBase	pic (Pic)	picXor	
placeCenterDisplay	v placeCentreWindow	printerErrorBase	purple
red	rgbErrorBase	spriteErrorBase	streamErrorE
timeErrorBase	unixSignalToExceptior	n viewErrorBase	white
yellow			

Keywords

		_		
addressint	all	and	array	asm
assert	begin	bind	bits	body
boolean	break	by	case	char
cheat	checked	class	close	collection
condition	const	decreasing	def	deferred
div	else	elseif	elsif	end
endfor	endif	endloop	enum	exit
export	external	false	fcn	flexible
for	fork	forward	free	function
get	handler	if	implement	import
in	include	inherit	init	int
int1	int2	int4	invariant	label
loop	mod	module	monitor	nat
nat1	nat2	nat4	new	not
objectclass	of	opaque	open	or
packed	pause	pervasive	pointer	post
pre	priority	proc	procedure	process
put	quit	read	real	real4
real8	record	register	rem	result
return	seek	self	set	shl
shr	signal	skip	string	tag
tell	then	timeout	to	true
type	unchecked	union	unqualified	var
wait	when	write	xor	

Operators

Mathematical Operators

Operator	Operation	Result Type
Prefix +	Identity	As Operands
Prefix -	Negative	As Operands
+	Addition	As Operands
-	Subtraction	As Operands
*	Multiplication	As Operands
/	Division	As Operands
div	Integer Division	int
mod	Modulo	int
rem	Remainder	int
**	Exponentiation	As Operands
<	Less Than	boolean
V A	Greater Than	boolean
=	Equals	boolean
<=	Less Than or Equal	boolean
>=	Greater Than or Equal	boolean
not=	Not Equal	boolean

Boolean Operators

Operator	Operation	Result Type
Prefix not	Negation	boolean
and	And	boolean
or	Or	boolean
xor	Exclusive Or	boolean
=>	Implication	boolean

Set Operators

Operator	Operation	Result Type
+	Union	set
-	Set Subtraction	set
*	Intersection	set
=	Equality	boolean
not=	Inequality	boolean
<=	Subset	boolean
<	Strict (Proper) Subset	boolean
>=	Superset	boolean
>	Strict (Proper) Superset	boolean

Operators on Members and Sets

Operator	Operation	Result Type
in	Member of Set	boolean
not in	Not Member of Set	boolean
xor	Exclusive Or	set

Bit Manipulation Operators

Operator	Operation	Result Type			
shl	Shift left	nat			
shr	Shift right	nat			
and	Bit-wise And	nat			
or	Bit-wise Or	nat			
xor	Bit-wise Exclusive Or	nat			

Pointer Operators

Operator	Operation	Result Type
\wedge	Follow pointer	Target type

Type Cheats

Operator	Operation	Result Type
#	Type cheat	nat

Operator Short Forms

These can be used in place of the above notation.

not ~ not= ~= not in ~in and & or |

Operator Precedence

Highest precedence operators first.

```
    **, ^, #
    prefix + and -
    * , / , div , mod , rem , shl , shr
    4. + , -, xor
    5. < , > , = , <= , >= , not= , in , not in
    6. not
    7. and
    8. or
    9. =>
```

File Statements

File Commands

open	open a file
close	close a file
put	write alphanumeric text to a file
get	read alphanumeric text from a file
write	binary write to a file
read	binary read from a file
seek	move to a specified position in a file
tell	report the current file position
eof	check for end of file

File Command Syntax

open : streamNo, fileName, ioCapability {, ioCapability }
 ioCapability is one of get, put, read, write, seek, mod

put or **write** capability will cause any existing file to be truncated to zero length unless the **mod** capability is also specified.

seek capability is needed to use seek or tell commands.

```
close : streamNo
get : streamNo , getItem { , getItem }
put : streamNo , putItem { , putItem }
read : streamNo [ : fileStatus ] , readItem { , readItem }
write : streamNo[ : fileStatus ] , writeItem {, writeItem }
seek : streamNo , filePosition or seek : streamNo , *
tell : streamNo , filePositionVar
eof ( streamNo ) : boolean (This is a function)
```

Appending to a Text File

% Open the file for put, but do not erase it
open : streamNo, fileName, put { , get }, mod
% Move the file pointer to the end of the file
seek : streamNo, *

Control Constructs

for [**decreasing**] *variable* : *startValue* .. *endValue*

... statements ...

FOR

exit when *expn* ... *statements* ...

end for

loop

LOOP

... statements ... exit when expn ... statements ...

end loop

if condition then ... statements ... { elsif condition then IF ... statements ... } [else ... statements ...] end if

case expn of ... statements ... { label expn {, expn ... statements ... } [label : ... statements ...] end case

Any number of **exit** and **exit when** statements can appear at any place inside **for** .. **end for** constructs and **loop** .. **end loop** constructs.

Key Codes

Ordinal values of keystrokes returned by getch and getchar.

	0	(space)	32	@	64	``	96
Ctrl-A	1	!	33	А	65	a	97
Ctrl-B	2		34	В	66	b	98
Ctrl-C	3	#	35	С	67	С	99
Ctrl-D	4	\$	36	D	68	d	100
Ctrl-E	5	%	37	E	69	е	101
Ctrl-F	6	&	38	F	70	f	102
Ctrl-G	7	1	39	G	71	g	103
Ctrl-H / BS	8	(40	Н	72	h	104
Ctrl-I / Tab	9)	41	Ι	73	i	105
Ctrl-J / Enter	10	*	42	J	74	j	106
Ctrl-K	11	+	43	K	75	k	107
Ctrl-L	12	,	44	L	76	1	108
Ctrl-M	13	-	45	М	77	m	109
Ctrl-N	14	•	46	Ν	78	n	110
Ctrl-O	15	/	47	0	79	0	111
Ctrl-P	16	0	48	Р	80	р	112
Ctrl-Q	17	1	49	Q	81	q	113
Ctrl-R	18	2	50	R	82	r	114
Ctrl-S	19	3	51	S	83	S	115
Ctrl-T	20	4	52	Т	84	t	116
Ctrl-U	21	5	53	U	85	u	117
Ctrl-V	22	6	54	V	86	v	118
Ctrl-W	23	7	55	W	87	W	119

Ctrl-X	24	8	56		X	88		Х	120
Ctrl-Y	25	9	57		Y	89	Í	у	121
Ctrl-Z	26	:	58		Z	90	Í	Z	122
Ctrl-[/ Esc	27	;	59		[91	Í	{	123
Ctrl-\	28	<	60		\	92	Í		124
Ctrl-]	29	=	61]	93	Í	}	125
Ctrl-^	30	>	62		Λ	94	Í	~	126
Ctrl	31	?	63		_	95	ĺ	Ctrl-BS	127
		 	, <u> </u>	,		, _,			
Alt-9	128	Alt-D	160		F6	192		Ctrl-F3	224
Alt-0	129	Alt-F	161		F7	193		Ctrl-F4	225
Alt	130	Alt-G	162		F8	194		Ctrl-F5	226
Alt-=	131	Alt-H	163		F9	195		Ctrl-F6	227
Ctrl-PgUp	132	Alt-J	164		F10	196		Ctrl-F7	228
F11	133	Alt-K	165			197		Ctrl-F8	229
F12	134	Alt-L	166			198		Ctrl-F9	230
Shift-F11	135		167		Home	199		Ctrl-F10	231
Shift-F12	136		168		Up Arrow	200		Alt-F1	232
Ctrl-F11	137		169		PgUp	201		Alt-F2	233
Ctrl-F12	138		170			202		Alt-F3	234
Alt-F11	139		171		Left Arrow	203		Alt-F4	235
Alt-F12	140	Alt-Z	172			204		Alt-F5	236
Ctrl-Up Arrow	141	Alt-X	173		Right Arrow	205		Alt-F6	237
	142	Alt-C	174			206	Í	Alt-F7	238
Back Tab	143	Alt-V	175		End	207		Alt-F8	239
Alt-Q	144	Alt-B	176		Down Arrow	208		Alt-F9	240
Ctrl-Down Arrow	145	Alt-N	177		PgDn	209		Alt-F10	241
Ctrl-Insert	146	Alt-M	178		Insert	210	Í		242

	<u> </u>		ļļ					ĮI
Ctrl-Delete	147		179		Delete	211	Ctrl-Left Arrow	243
Alt-T	148		180		Shift-F1	212	Ctrl-Right Arrow	244
Alt-Y	149		181		Shift-F2	213	Ctrl-End	245
Alt-U	150		182		Shift-F3	214	Ctrl-PgDn	246
Alt-I	151		183		Shift-F4	215	Ctrl-Home	247
Alt-O	152		184		Shift-F5	216	Alt-1	248
Alt-P	153		185		Shift-F6	217	Alt-2	249
	154		186		Shift-F7	218	Alt-3	250
	155	F1	187		Shift-F8	219	Alt-4	251
	156	F2	188		Shift-F9	220	Alt-5	252
	157	F3	189		Shift-F10	221	Alt-6	253
Alt-A	158	F4	190		Ctrl-F1	222	Alt-7	254
Alt-S	159	F5	191		Ctrl-F2	223	Alt-8	225

Ordinal values of characters that are set by Input.KeyDown.

Backspace	8	(space)	32	@	64	`	96
Tab	9	!	33	A	65	a	97
Enter	10	· !	34	B	66	b	98
Escape	27	· #	35	C	67	C	99
Цзсаре	/	 \$	36	D	67 68	d	100
F11	133	 %	30 37	E	69		100
						e f	
F12	134	&	38 20	F	70 71		102
Shift	180		39 10	G	71	g	103
Control	181	(40	H	72	h	104
Alt	182)	41	Ι	73	i	105
Keypad 5	183	*	42	J	74	j	106
F1	187	+	43	K	75	k	107
F2	188	,	44	L	76	1	108
F3	189	-	45	Μ	77	m	109
F4	190	•	46	Ν	78	n	110
F5	191	/	47	0	79	0	111
F6	192	0	48	Р	80	р	112
F7	193	1	49	Q	81	q	113
F8	194	2	50	R	82	r	114
F9	195	3	51	S	83	S	115
F10	196	4	52	Т	84	t	116
		5	53	U	85	u	117
Home	199	6	54	V	86	v	118
Up Arrow	200	7	55	W	87	W	119
PgUp	201	8	56	Х	88	х	120
Left Arrow	203	9	57	Y	89	у	121
Right Arrow	205	:	58	Z	90	Z	122

End	207	•	59	[91	{	123
Down Arrow	208	<	60	\	92		124
PgDn	209	=	61]	93	}	125
Insert	210	>	62	\wedge	94	2	126
Delete	211	?	63	_	95		

Run Window Character Set

The Turing character set and their ASCII values

chr (8) = backspace chr (9) = tab char (10) = newline

	1 1	0.01		1 1	60	0	1	0.0		1	100	0	l i	1.00		2	100	1	1	004	2
0		32			64	0		96			128	€		160			192	À		224	à
1		33	!		65	A		97	a		129			161	i		193	Á		225	á
2		34	"		66	в		98	b		130	,		162	¢		194	Â		226	â
3		35	#		67	С		99	С		131	f		163	£		195	Ã		227	ã
4		36	\$		68	D		100	d		132	"		164	×		196	Ä		228	ä
5		37	÷		69	Е		101	е		133			165	¥		197	Å		229	å
6		38	æ		70	F		102	f		134	+		166	1		198	Æ		230	æ
7		39	1		71	G		103	g		135	ŧ		167	s		199	Ç		231	ç
8		40	(72	н		104	h		136	^		168			200	È		232	è
9		41)		73	I		105	i		137	48		169	©		201	É		233	é
10		42	*		74	J		106	j		138	š		170	2		202	Ê		234	ê
11		43	+		75	К		107	k		139	<		171	«		203	Ë		235	ë
12		44	,		76	L		108	1		140	Œ		172	7		204	Ì		236	ì
13		45	-		77	М		109	m		141			173	-		205	Í		237	í
14		46			78	Ν		110	n		142	ž		174	8		206	Î		238	î
15		47	1		79	0		111	о		143			175	-		207	Ϊ		239	ï
16		48	Ο		80	Ρ		112	р		144			176	۰		208	Ð		240	ð
17		49	1		81	Q		113	q		145	1		177	±		209	Ñ		241	ñ
18		50	2		82	R		114	r		146	,		178	2		210	ò		242	ò
19		51	3		83	s		115	з		147	~		179	з		211	Ó		243	ó
20		52	4		84	т		116	t		148	"		180	1		212	ô		244	ô
21		53	5		85	U		117	u		149	•		181	μ		213	õ		245	õ
22		54	6		86	v		118	v		150	-		182	IP		214	ö		246	ö
23		55	7		87	W		119	w		151	_		183	•		215	×		247	÷
24		56	8		88	х		120	х		152	~		184	د		216	ø		248	ø
25		57	9		89	Y		121	У		153	125		185	1		217	Ù		249	ù
26		58	:		90	Z		122	z		154	š		186	۰		218	Ú		250	ú
27		59	;		91	Γ		123	{		155	>		187	»		219	Û		251	û
28		60	<		92	1		124	1		156	œ		188	1 ₄		220	Ü		252	ü
29		61	=		93]		125	}		157			189	₩2		221	Ý		253	ý
30		62	>		94	~		126	~		158	ž		190	36		222	Þ		254	þ
31		63	2		95			127			159	Ϋ́		191	ć		223	ß		255	ÿ
	 I I			I I		_	1			1			1		-				1		-

You can also get (on most systems), the MS-DOS character set by using the command setscreen ("msdos").

This is the character set after the setscreen ("msdos") command was given.

	-	 0.01		 6.0	0	1	0.0		1.0	100		1	1.00	1	0	100		000		٦
0		32		64	0		96			128	ç		160	á		192	1	224	α	l
1		33	!	65	A		97	a		129	ü		161	í		193	T	225	ß	l
2		34	"	66	в		98	b		130	é		162	Ó		194	Т	226	Г	l
3		35	#	67	С		99	С		131	â		163	ú		195	+	227	п	l
4		36	\$	68	D		100	d		132	ä		164	ñ		196	-	228	Σ	l
5		37	*	69	Е		101	e		133	à		165	Ñ		197	+	229	σ	l
6		38	&	70	F		102	f		134	å		166	2		198	F	230	μ	l
7		39	T.	71	G		103	g		135	ç		167	۰		199	┠	231	τ	l
8		40	(72	н		104	h		136	ê		168	5		200	L	232	Φ	l
9		41)	73	I		105	i		137	ë		169	-		201	F	233	Θ	l
10		42	*	74	J		106	j		138	è		170	~		202	[234	ឆ	l
11		43	+	75	К		107	k		139	ï		171	₩		203	T	235	δ	l
12		44	,	76	L		108	1		140	î		172	4		204	ŀ	236	~	l
13		45	-	77	М		109	m		141	ì		173	i		205	=	237	φ	l
14		46		78	Ν		110	n		142	Ä		174	~		206	₽	238	ε	l
15		47	1	79	0		111	ο		143	Å		175	»		207	Ŧ	239	Π	l
16		48	0	80	Ρ		112	р		144	É		176			208	Ш	240	≡	l
17		49	1	81	Q		113	q		145	æ		177			209	ᆕ	241	±	l
18		50	2	82	R		114	r		146	Æ		178			210	I	242	≥	l
19		51	3	83	s		115	s		147	ô		179	T		211	L	243	≤	l
20		52	4	84	Т		116	t		148	ö		180	-		212	F	244	ſ	l
21		53	5	85	U		117	u		149	ò		181	=		213	F	245		l
22		54	6	86	v		118	v		150	û		182			214	Г	246	÷	l
23		55	7	87	W		119	w		151	ù		183	п		215	#	247	~	l
24		56	8	88	Х		120	х		152	ÿ		184	Ŧ		216	ŧ	248	۰	l
25		57	9	89	Y		121	У		153	Ö		185	4		217	1	249	•	l
26		58	:	90	Ζ		122	z		154	Ü		186			218	г	250	•	l
27		59	;	91	[123	{		155	¢		187	٦.		219		251	V	l
28		60	<	92	١.		124	L		156	£		188	ſ		220		252	ъ	
29		61	=	93]		125	}		157	¥		189	Ш		221	[]	253	2	
30		62	>	94	~		126	~		158	R		190]		222		254		
31		63	?	95			127			159	f		191	٦		223		255		
					-			_									_			L

Click the button to launch the program to display the character set in Turing.

Turing Teacher/Administrator Quick Start Guide

This guide is designed to be the minimal information that you should read if you are teaching courses using Turing 4.0 or are in charge of administering Turing 4.0

This guide provides information on the Turing environment (the editor, the run windows, etc.) It does **not** provide any information on the Turing programming language itself.

Note, if you have any questions not answered here, please check with the <u>Turing Teacher/Administrator's Complete Guide</u> for more information.

Table of Contents

- 1. Making Certain You Have the Lastest Version
- 2. <u>Mandatory Administration Issues</u>
- 3. <u>Using the Parallel Ports in Turing 4</u>
- 4. <u>Student Distribution</u>
- 5. <u>Technical Support</u>

1. Making Certain You Have the Lastest Version

Turing 4.0 is updated regularly. To find out what version you are currently running, click the **About Turing** menu item in the **Help** menu. To find out the current version of Turing, go to

<u>http://www.holtsoft.com/turing/support/#currentversion</u>. This lists the current version of the software.

This page also lists which bugs have been fixed and when new features have been added. The page also contains a section (<u>http://www.holtsoft.com/turing/support/#turing4patches</u>) where a file can be downloaded that will update most versions of Turing to the latest version.

If you are a teacher or an administrator at a school and require a complete installation file (as opposed to the publically available patch), contact Tom West at (416) 978-8363.

2. Mandatory Administration Issues

Turing 4.0 has been designed to run on either a network or installed on individual stand-alone machines. Running on a network is often preferred because of ease of administration. However, Turing, like any other application will load somewhat faster when it is launched if it is stored locally.

There are only two administration issues that must be addressed to configure Turing properly.

1. Set the start-in (often called the "working") directory properly.

It is important that the **start-in** directory for Turing be set to the student's home directory. This is done by setting the **start-in** (or **working**) directory of the shortcut or menu item used to launch Turing to the drive mapped to the student's home directory.

The shortcut's working directory can be modified by changing it's start-in or working directory property. (To display the **Properties** dialog box, right click on the shortcut or menu item and select Properties from the pop-up menu.)

ring 4.0 Prope	rties	? 🗙
General Shorto	ut Compatibility	1
	uring 4.0	
Target type:	Application	
Target location	Turing	
Target:	D:\Apps\Turing\turing.exe	
Run in sep	arate memory space 🛛 🧖 Run as different use	er
Start in:	H:N	
Shortcut <u>k</u> ey:	None	
<u>B</u> un:	Normal window	
C <u>o</u> mment:		
	<u>Find Target</u> <u>Change Ico</u>	on
	OK Cancel <u>A</u> r	Properties for a Shor

under Windows 2000

This requires that you have write access to the shortcut. Note that commercial menu front-ends have their own way of setting the start-in directory of applications.

If there is no drive mapped to the student's home directory, or this method is not applicable to your system setup, please check with the <u>Turing</u> <u>Teacher/Administrator's Complete Guide</u> for more information.

2. Set either Beginner or Regular Mode for Students.

Turing can be set to **Beginner** mode, in which case only one file is open at a time. Opening up another file closes the first one. When the program is running, the Editor window minimizes. In this way, only one window is usually visible at a time. In this fashion, **Beginner** mode is similar to the old DOS Turing.

In **Regular** mode, any number of files can be opened and each appears in its own editor window.

In general, we suggest **Beginner** mode for grade 9 and 10 and **Regular** mode for all others. If you are using **Beginner** mode, we suggest that you point out to the students that they can use the Prefences to change to **Regular** mode when they feel comfortable with the environment.

To change the mode (or to make any other change to the system preferences), you must open a DOS window, **cd** to the directory in which the Turing executable (**turing.exe**) is located and issue the command:

turing -admin

This will cause Turing to start in **Administrator mode**, displaying the following dialog box:



Administration Mode Notification

Selecting **Preferences** from the **File** menu will display a tabbed dialog box of all the Turing preferences.

Preferences	?×
General Editor Window Run Window	Printing Admin
Beginner Mode (One file at a time)	Spaces per Tab: 8
Display Full Pathnames Save Previous File as ".BAK"	".t" Files Are Currently Associated With:
 Confirm on Quit Skip Splash Screen 	Turing
Add ".t" to File Names Convert Leading Spaces into Tabs	No Associated Application
Don't Save Recent Files Menu Find Uses Selected Text	Associate ".t" files With Turing
Start in Last Active Directory	Reset ".t" File Association
	OK Cancel Apply

Preferences in Administrator Mode

Set (or unset) the **Beginner Mode** preference, click the **OK** button and quit Turing. Of course, you can change any other preferences that you would like. Any changes that you make here will affect all those using Turing.

3. Using the Parallel Ports in Turing 4

Turing 4.0 supports the **parallelput** and **parallelget** commands to allow you to set the pins on the parallel port low and high. If you are running under Windows NT, 2000 or XP, you will need to install a device driver on each machine. Instructions for doing so can be found <u>here</u>. If **parallelput** and **parallelget** do not seem to function, you may need to change their BIOS settings. You can get information on how to do that <u>here</u>.

4. Student Distribution of Turing 4.0

The Ontario Ministry license does not include the right to redistribute the software to students. If you wish to give the software to students, a separate license must be obtained directly from Holt Software.

If your school does not have such a license, students can purchase the software for home use by printing and then mailing an order form found <u>http://www.holtsoft.com/studentbuy</u>. This order form can also be used to allow interested students to purchase textbooks.

If your school has purchased a redistribution license (almost 200 schools have done so), then you may redistribute the Turing software to your students. You can either burn copies of the CD to distribute to students or place the Turing installer file onto your school Internet server and give your students the URL.

If you place this file on your school internet server, you must not make any links from your web pages to the software.

This will prevent other users of the web from downloading your software. If you place links from your school's web site, then at some point a web search engine will find the software and post its location to the internet at large, allowing for large scale piracy (and a very overloaded school web server). If there are no links to the software, then only those who have been told the exact URL will be able to download the software.

Note that if you are copying a number of different programs onto a single CD for redistribution, you may rename the Turing installer program to something more descriptive.

Permanent student redistribution licenses can be purchased for \$500-\$750 for most schools. If you are interested in purchasing a student redistribution license, contact Chris Stephenson at (416) 978-6476 or chris@hsa.on.ca

5. Technical Support

If you have problems, questions or suggestions about the Turing software, contact technical support at:

Telephone:	(416) 978-8363
Toll free:	1-800-361-8324
Fax:	(416) 978-1509
E-mail:	<u>west@hsa.on.ca</u>

Turing Teacher/Administrator Guide

This is the Turing Teacher/Administrator guide. It contains most of the information that you will need to administer the Turing 4.0 software. It should be read by those who are teaching courses using Turing 4.0 or are in charge of administering Turing 4.0

This guide provides information on the Turing environment (the editor, the run windows, etc.) It does **not** provide any information on the Turing programming language itself.

Note that there is some overlap with the <u>Turing Teacher/Administrator Quick</u> <u>Start Guide</u>.

Table of Contents

- 1. Making Certain You Have the Lastest Version
- 2. <u>Mandatory Administration Issues</u>
- 3. <u>Changing System Preferences</u>
- 4. <u>Turing Preferences</u>
- 5. Parallel Port and Turing 4
- 6. Missing Features from Turing 4
- 7. <u>The Turing Debugger</u>
- 8. Student Distribution
- 9. Technical Support

1. Making Certain You Have the Lastest Version

Turing 4.0 is updated regularly. To find out what version you are currently running, click the **About Turing** menu item in the **Help** menu. To find out the current version of Turing, go to

<u>http://www.holtsoft.com/turing/support/#currentversion</u>. This lists the current version of the software.

This page also lists which bugs have been fixed and when new features have been added. The page also contains a section (http://www.holtsoft.com/turing/support/#turing4patches) where a file can be downloaded that will update most versions of Turing to the latest version.

If you are a teacher or an administrator at a school and require a complete installation file (as opposed to the publically available patch), contact Tom West at (416) 978-8363.

2. Mandatory Administration Issues

Turing 4.0 has been designed to run on either a network or installed on individual stand-alone machines. Running on a network is often preferred because of each of installation, however, Turing will load somewhat faster when it is launched if it is stored locally.

Turing 4.0 has been designed to run on either a network or installed on individual stand-alone machines. Running on a network is often preferred because of ease of administration. However, Turing, like any other application will load somewhat faster when it is launched if it is stored locally.

There are a few administration issues that must be addressed to configure Turing properly.

1. Set the start-in (often called the "working") directory properly.

It is important that the start-in directory for Turing be set to the student's home directory. This can be done in a variety of ways, from least to most complicated.

 Many schools have the student's home directory mapped to a drive (often H:\). If this is the case, then set the start-in (or working) directory of the shortcut or menu item used to launch Turing to the mapped drive.

The shortcut's working directory can be modified by changing it's start-in or working directory property. (To display the **Properties** dialog box, right click on the shortcut or menu item and select Properties from the pop-up menu.)

ng no rrope	rties	? ×
neral Shortc	ut Compatibility	
	uring 4.0	
Target type:	Application	
Target location:	Turing	
Target:	D:\Apps\Turing\turing.exe	
and the local of the state of the state of the	Construction of the second	
Start in: Shortcut <u>k</u> ey:	H:\ None	
Shortcut <u>k</u> ey:	None	
Shortcut <u>k</u> ey: <u>R</u> un:	None	
Shortcut <u>k</u> ey: <u>R</u> un:	None Normal window	

Shortcut under Windows 2000

This requires changing the shortcut's properties requires write access to the shortcut. Note that commercial menu front-ends have their own way of setting the start-in directory of applications.

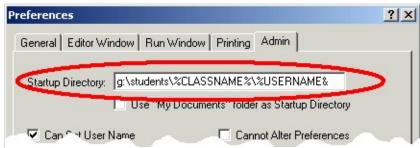
2. If the "My Documents" folder is mapped to the students home directory, you can set the preferences to make Turing use the "My Documents" folder as the start-in folder. Note that this option is the preferred option for home users, where the "My Documents" folder is the home directory.

Preferences	<u>?</u> ×
General Editor Window Run Window Printing Ad	min
Startup Directory: Use "My Documents" folder as	Startup Directory
Pan Set Pser Name Cannot A	Pref inces

The Use "My Documents" Folder in the Admin Pane

To change the preference, enter Administrator mode (see <u>Changing</u> <u>System Preferences</u> below) and set the **Use** "**My Documents**" Folder checkbox in the Admin pane of the Turing Preferences (see <u>Turing</u> <u>Preferences</u> below).

- 3. If each student's home directory is specified differently (i.e. there is no mapping of the home directory to a drive), then you may need to use environment variables to create a useable path. For example, if students files are stored in g:\class108\[Student Name] and there is an environment variables USERNAME that contains the student's name, then the start-in directory for the shortcut should be set to g:\class108\%USERNAME%.
- 4. Occasionally users have reported situations like the previous case, except that Windows seems unable to set the start-in directory properly using environment variables.



The Startup Directory Text Field in the Admin Pane

To set the start-in directory, enter Administrator mode (see <u>Changing</u> <u>System Preferences</u> below) and set the **Startup Directory** text field in the **Admin** pane of the Turing Preferences appropriately (see <u>Turing</u> <u>Preferences</u> below) including specifying the directory using the environment variables.

2. Set either Beginner or Regular Mode for Students.

Turing can be set to **Beginner** mode, in which case only one file is open at a time. Opening up another file closes the first one. When the program is running, the Editor window minimizes. In this way, only one window is usually visible at a time. In this fashion, **Beginner** mode is similar to the old DOS Turing.

In **Regular** mode, any number of files can be opened and each appears in its own editor window.

In general, we suggest **Beginner** mode for grade 9 and 10 and **Regular** mode for all others. If you are using **Beginner** mode, we suggest that you point out to the students that they can use the Prefences to change to **Regular** mode when they feel comfortable with the environment.

To change the preference, you will need to enter Administrator mode (see <u>Changing System Preferences</u> below) and change the **Beginner Mode** preference in the **General** pane appropriately (see <u>Turing Preferences</u> below).

3. Windows 95 and the Turing Net module

If you are using Windows 95 (not Windows 98/Me/NT/2000/XP) and you wish to use the Turing **Net** module in order to write programs that communicate with other computers, then you may need update the Window 95 networking code.

This can be done by running the program located at:

[Turing directory]\support\Microsoft\W95ws2setup.exe

This program from Microsoft patches Windows 95 networking code. It must be run on every computer running Windows 95 unless the machines boot from a server.

4. **Parallel Port Usage** If you are intend to use Turing to control devices attached to the parallel port (using the **parallelput** and **parallelget**, and are running under Windows NT/XP/2000, you will need to install a device driver. (Windows NT/XP/2000 do not allow direct access to the system hardware.). The device driver can be installed by running the program located at:

[Turing directory]\support\dlportio_install.exe

Further information can be found in <u>Parallel Port and Turing 4</u>.

3. Changing System Preferences

Turing uses a two level preference model. There is a central set of preferences that are kept in

[Turing Directory]\Support\Ini\turing_admin.ini

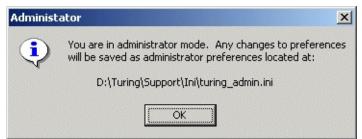
These preferences are the same for all users and are called the **System Preferences**. A second set of preferences is stored in each student's home directory in a file called **turing.ini**. These preferences override the central preferences and allow students to individually modify the behaviour of Turing.

When a student modifies the preferences, the **turing.ini** file in the student's home directory is changed. To change the System preferences (i.e. the contents of the **turing_admin.ini** file), Turing must be started up with the command line

turing -admin

This can be done by starting up a DOS window, doing a **cd** to the directory in which **turing.exe** is stored and then typing **turing -admin**. The user must have write access to the directory in which the **turing_admin.ini** file is stored.

When Turing is launched, it will display a message box on start-up indicating that it is in **Administrator mode** and where the preferences will be stored.



Administration Mode Notification

Selecting **Preferences** from the **File** menu will display a tabbed dialog box of all the Turing preferences. If you are in **Administrator mode**, then an extra tab labeled **Admin** will appear with extra preferences that can only be set by the Administrator. Note that individual students cannot override the preferences that appear on the **Admin** pane.

Preferences	<u>? ×</u>
General Editor Window Run Window	Printing Admin
 Beginner Mode (One file at a time) One Window Mode Display Full Pathnames Save Previous File as ".BAK" 	Spaces per Tab: 8 Indent Size: 4 ''.t'' Files Are Currently Associated With: Turing
 Confirm on Quit Skip Splash Screen Add ".t" to File Names Convert Leading Spaces into Tabs 	".t" Files Were Associated With: No Associated Application
 Don't Save Recent Files Menu Find Uses Selected Text Start in Last Active Directory 	Associate ".t" files With Turing Reset ".t" File Association
	OK Cancel Apply

Preferences in Administrator Mode

Once the Preferences have been set, the Administrator can exit Turing.

4. Turing Preferences

This section lists all the preferences available in Turing sorted by the pane

"General" Pane

Preferences	<u>? ×</u>
General Editor Window Run Window	Printing Admin
 Beginner Mode (One file at a time) One Window Mode Display Full Pathnames Save Previous File as ".BAK" Confirm on Quit 	Spaces per Tab: 8 Indent Size: 4 ''.t'' Files Are Currently Associated With: Turing
 Skip Splash Screen Add ".t" to File Names Convert Leading Spaces into Tabs 	I ".t" Files Were Associated With: No Associated Application
 Don't Save Recent Files Menu Find Uses Selected Text Start in Last Active Directory 	Associate ".t" files With Turing Reset ".t" File Association
	OK Cancel Apply

The "General" Pane

• Beginner Mode

When **Beginner** mode is set, only one file can be can be loaded into the Turing environment at a time. The Editor window is minimized every time the Run window is active and vice-versa. The debugger cannot be displayed or used. If this box is not set, then Turing is in **Regular** mode and multiple files can be open simultaneously. **Beginner** mode is recommended for students who are overwhelmed or confused by too many Turing windows (especially students in grades 9 and 10).

• **One Window Mode** This option is not available in Turing 4.0. It will be used to switch between MDI and SDI modes in a future version of the software.

• Display Full Pathnames

The full path name and not just the file name are displayed in the Editor window's title bar and recent files submenu.

• **Save Previous File as ".BAK**" When a file is saved over top of an older file, the older file is renamed with a .BAK suffix (i.e. **test.t** to **test.bak**)

• Confirm on Quit

When the user closes the last editor window, Turing displays a dialog box asking whether the user wishes to Quit. On systems where loading is slow (usually over a network), this can stop the users from accidentally quitting Turing.

• Skip Splash Screen

When Turing starts up, it normally displays a "Splash" screen with the version number. The Splash screen disappears when the user clicks a button or after 3 seconds. Turning this option off causes the Splash screen not to be displayed.

• Add ".t" to File Names

When a Save File dialog box appears and the user specifies a file name without a file suffix, Turing adds ".t". This can be removed by the user.

• **Convert Leading Spaces into Tabs** When Turing saves a file, it normally converts groups of leading spaces into TAB characters, saving disk space. The number of spaces per tab is specified by the **Spaces per Tab** text field. If this check box is not selected, then the spaces are not converted.

• Don't Save Recent Files Menu

Normally Turing lists the last several files opened. However, if students do not have individual accounts, then this feature may not be useful and can be turned off.

• Find Uses Selected Text

By default, Turing uses the Windows behaviour that when a Find command is given and there is selected text, the selected text is copied to the Find dialog box. If you don't like this behaviour, then unsetting this option will stop this behaviour. If it is not set, then an additional menu item is placed in the **Search** menu that finds the next occurance of the currently selected text.

• Start in Last Active Directory

If students are writing large projects in many subdirectories, then this option can be enabled to automatically start Turing in the last directory in which a file was loaded or saved. The Turing preferences file is still stored in the student's home directory.

• Spaces per Tab

This specifies the number of spaces that a Tab is converted to and from. When Turing reads a text file into the editor, all tabs are converted on to spaces. If you created the Turing file on an editor where Tabs are a different number of spaces, then change this value appropriately.

• Indent Size

When Tab is pressed in the Turing editor, it is converted into spaces. The number in the **Indent Size** text field determines how many spaces the Tab key is converted to. Note the **Spaces per Tab** is used when Turing reads/writes files. The **Indent Size** field is used when Turing responds to the Tab key.

• File Association

Microsoft Windows has the ability to associate files with a particular suffix with specific programs. The Turing environment can be associated with the following file suffixes: .t, .tur, .dem, .ti and .tu. To associate these file suffixes with Turing, click the Associate ".t" files with Turing button. To reset the file associations back to what they were before, click the Reset ".t" File Association.

"Editor Window" Pane

Preferences	? ×
General Editor Window Run Window Printing Admin Editor Font Font Name: Courier New Syntax Coloring Font Size: 10 Always Show 'Debugger' Menu	
Editor Window Size Full Screen for Beginner Mode Full Screen in Regular Mode Editor Window Rows: 25 Editor Window Columns: 80	
OK Cancel App	ly

The "Editor Window" Pane

• Editor Font

You can change the font size and typeface used in the editor window. If you change the typeface, you must choose a monospaced font where bold and non-bold letters are the same size. Many monospaced fonts have different sizes for bold and non-bold letters.

• Full Screen for Beginner Mode If in beginner mode, editor windows are automatically in full screen mode. You can change this behaviour buy unsetting this checkbox.

• Full Screen for Regular Mode

Editor windows are normally 25x80 columns (or the size specified by the **Editor Window Rows/Columns** text fields). If this checkbox is set, editor windows are automatically maximized to fill the screen. This option is suggested for systems with 640x480 screen resolution. Users can change the window size by clicking the full screen/normal window button in the window's title bar.

• Editor Window Rows/Columns

These text fields control the size of editor windows when first created.

• Syntax Coloring

When selected, parts of Turing programs appear in different colors: comments appear in green, strings in red, keywords in bold face, predefined identifiers in black, and user identifiers in blue.

• Always Show 'Debugger' Menu

In the interest of keeping the interface as simple as possible, and because most students do not use the debugging features, Turing does not display the Debugger menu by default. Instead, the user selects **Show Debugger Menu** from the **Run** menu. Setting this checkbox causes the Debugger menu to always be displayed.

• Text Cursor Width

By default, the cursor (correctly called the caret) is 2 pixels wide, allowing it to be seen easily in the Editor window. However, some users prefer the caret to be 1 pixel wide and some prefer the caret to overlay an entire character (in a similar manner to DOS editors). Legal values are 1 (which represent a 1 pixel thick caret), 2 (which represent a 2 pixel thick caret), and 3 (which represents a text cursor that covers the entire character.

"Run Window" Pane

Preferences	<
General Editor Window Run Window Printing Admin Default Run Window Font	1
Font Name: Courier New Font Size: 10 Change Font Default Run Window Size V Start in Graphics Mode	
Full Screen Run Window Run Window Rows: 25	
Run Window Columns: 80 OK Cancel	

The "Run Window" Pane

• Default Run Window Font

You can change the font size and typeface used in the Turing run window. If you change the typeface, you must choose a monospaced font. Note that changing the size of the run window font may cause odd output in Turing programs that were written assuming a particular character size.

• Start in Graphics Mode

This makes the initial run window a graphics window. The user can use graphics without the setscreen ("graphics") command in the program. Output that scrolls off the top of the screen is lost. In text mode, graphics are not allowed, but all text output is kept and can be scrolled, saved, and printed at any time.

• **Full Screen Run Window** This causes the default run window to occupy the entire screen.

• Run Window Rows/Columns

This sets the size of the default run window. Changing the size may cause programs that assume a 25x80 window to malfunction.

• Use Small Fonts on 640x480 Screen

A standard 25x80 window does not quite fit on a 640x480 screen with a standard size task bar. As a result, without this option, the default graphics mode run window appears with scroll bars. This option causes Turing to use

a slightly smaller font for 640x480 run windows so the entire window fits on the screen. This option may cause odd output in Turing programs that were written assuming a particular character size.

"Printing" Pane

Preferences	<u>? ×</u>
General Editor Window Run Wind Printing Text Font Font Name: Courier New Font Size: 10 ☐ Change Font ☐ Bold Keywords ☑ Italicize Identifiers ☑ Print Page Header ☑ User Name in Header ☐ Two-Up Printing ☐ Line Numbers ☐ Border Around Text	dow Printing Admin Printing Header Font Font Name: Courier New Font Size: 10 Change Font Margins (Inches) Left: 0.5'' Right: 0.5'' Top: 0.5'' Bottom: 0.5'' User Name:
[OK Cancel Apply

The "Printing" Pane

Note that the options marked with an asterisk (*) can be set in the printing dialog.

• Printing Font/Printing Header Font

You can change the font size and typeface used in printing. If you change the typeface, you must choose a monospaced font where bold and non-bold letters are the same size. Many monospaced fonts have different sizes for bold and non-bold letters.

• **Bold Keywords/Italicize Identifiers** * This allows for "syntax coloring" of printouts. Keywords can be placed in bold and identifiers italicized.

• Print Page Header

On some systems, a header page is automatically printed out and there is little reason for a page header. Unsetting this checkbox will cause printouts

not to have any header on the printed programs.

• User Name in Header

Normally the user name is displayed in the header. However, on some systems the user name is non-descriptive or non-existent. This allows you to disable the printing of the user name on such systems.

• Two-Up Printing *

This is a paper saving measure. Text is printed in landscape mode with two "pages" per piece of paper. This means that listings use half the amount of paper. Of course, the font is much smaller.

• Line Numbers *

This prints out line numbers in front of each line of the program. This is useful if you are printing out a listing to hand out to the class.

• Border Around Text

Some users like to have a thin border printed around the text for ease of determining the margins, etc. Setting this checkbox causes Turing to display a hairline border around program printouts.

• Margins

If you need to change the margins on printouts (for example, to facilitate the insertion of program listings into 3-ring binders), then you can do by changing the values in the text fields. Margins are expressed in inches.

• User Name

On systems where the user name is non-descriptive or non-existent, the user can enter a name that will appear on the printout allowing it to be identified. If students are choosing inappropriate user names, this preference can be disabled in the Admin tab preference dialog.

"Admin" Pane

Preferences	? ×
General Editor Window Run Window Printing Admin	
Startup Directory: Use "My Documents" folder as Startup Directory	
Can Set User Name Cannot Alter Preferences	
🔽 Can Enter Regular Mode 📃 Prohibit use of 'Sys.Exec'	
Debugger Available No Sound or Music	
Use Only Startup Directory and Below Use Only Allowed Drives	
Allowed Drives: ABCDEFGHIJKLMNOPQRSTUVWXYZ	
Forbid Use of Specified Drives	
Forbidden Drives: ABCDEFGHIJKLMNOPQRSTUVWXYZ	
OK Cancel) Apply

The "Admin" Pane

Note, the **Admin** pane is only available in administrator mode. Students starting Turing will not have access to this dialog box.

• Startup Directory

Users will start in the directory listed in this text field. If left blank, Turing will use the working directory of the shortcut used to start Turing. If no working directory for the shortcut is specified, the directory where Turing is installed is used. Environment variables prefixed and suffixed with % signs can be used here (example: g:\students\%username%). Note that in most cases, this text field should be blank.

- Use "My Documents" folder The "My Documents" folder can be used as the startup directory instead of the contents of the Startup Directory text field. This is most commonly used in single user systems.
- **Can Set User Name** Allows the user to specify the user name to appear on top of printouts. This checkbox can be unset if students are choosing inappropriate user names.

• **Can Enter Regular Mode** Allows the user to turn off the beginner mode preference. This checkbox can be unset if it is desirable to have all students working in beginner mode.

• Debugger Available

This checkbox can be unset in order to stop students getting access to the debugger.

• Cannot Alter Preferences

When set, this stops students from accessing the Preferences dialog. It also stops Turing from reading the student's **turing.ini** file. This can be set when it is important to have a uniform set of preferences on all student machines, or when there is no log-on procedure and students are sharing machines.

• **Prohibit use of 'Sys.Exec'** When set, the **Sys.Exec** and **system** procedures do nothing. This can be set if there is a system security concern about students starting arbitrary programs.

• No Sound or Music

When set, the Turing sound and music commands do nothing.

• Use only Startup Directory and Below

CAUTION! This option only provides rudimentary security and is to be used only on systems where the operating system cannot provide proper security. This option will not allow students to save or load to any directory other that their startup directory or directories created in the startup directory. This option does not stop students from browsing the names of files in other directories, only from loading or saving files in them.

• Use Only Allowed Drives

CAUTION! This option only provides rudimentary security and is to be used only on systems where the operating system cannot provide proper security. This option allows students to load files only from the set of drives specified in the text field. The '*' represents network drives.

• Forbid Use of Certain Drives

CAUTION! This option only provides rudimentary security and is to be used only on systems where the operating system cannot provide proper security. This option prevents students from loading files from the set of drives specified in the text field. The '*' represents network drives.

5. Using the Parallel Ports in Turing 4

Under Turing 4.0 and , it is possible to access the IBM PC's parallel port for reading and writing. This is normally done using the **parallelput** and **parallelget** commands. (In Turing 4.0, you can use **PC.ParallelPut** and **PC.ParallelGet** to access ports other than LPT1.)

Under Windows 95/98 and Me, the operating system allows direct access to the parallel port. Under Windows NT, 2000 and XP, access to the parallel port is restricted by the operating system. In order to access the port, under Windows NT, 2000 and XP, it is necessary to install a device driver on each machine that will be accessing the parallel port. The needed device driver installer is located at:

[Turing directory]\support\dlportio_install.exe

where **[Turing directory]** is the directory in which Turing was installed. Executing this program will install the device driver in the Windows System directory of the machine upon which it is executed.

Lastly, before accessing the parallel port, check that the parallel port is in **normal** mode and not in **bi-directional**, **PS/2**, **EPP** or **ECP** mode. You can determine which mode the parallel port is in using the Device Manager. In the list of devices in the Device Manager window, double click **Ports (COM and LPT)**, and then the **LPT** (or **PRN**) port. This displays a property sheet about the parallel port. Take a look at what it says about it. If it is in a non-original mode (i.e. mentions **bi-directional**, **PS/2**, **EPP** or **ECP** mode), then you will probably need to adjust the systems BIOS settings.

(Under Windows 2000 the Device Manager can be displayed by selecting the System Control Panel from Settings in the Start menu. Select the Hardware tab and then click the Device Manager button.)

If the setting for the parallel port needs to be changed, restart the machine, and press the appropriate key to enter the BIOS Set Up Program (the key varies between machines, but is usually noted on the screen). Often the parallel port settings are found in **Advanced Settings** of the program. The parallel port should be set to **Normal**, **Standard**, **AT** or **Unidirectional** mode. (All four

names are different descriptions of the same mode.) Once this is done and the new settings are saved, then start the machine into Windows. Once in Windows, select **Shut Down** and turn the machine completely off. Once the machine is off, wait at least 10 seconds in the powered-off state. Then restart the machine. (While shutting down the machine should not be necessary, we have had several cases where the change only *took* when the machine had been powered-down and restarted.)

6. Missing Features from Turing 4

Turing 4.0 does not yet incorporate all the features that currently exist in Object Oriented Turing 3.1.1 and are intended for future Turing releases. Over the next few months, new features will be added and updates to Turing will be made available on our web site. The features to be added are:

- View Variables
- The Sprite module

If you have immediate need of any of these features, please continue to use the Object Oriented Turing 3.1.1 software. Ontario users can obtain the software through their OESS software representative.

To determine the current release of Turing, check the Turing web site at

http://www.holtsoft.com/turing/support

7. The Turing Debugger

The current version of Turing has a Debugger facilities that allow students to follow execution of a program either pausing after each line of execution or executing slowly while highlighting each line of execution. Because of the confusing nature of the debugger for many students, the default configuration of Turing does not display the debugger menu until the student requests it by selecting the **Show Debugger Menu** from the **Run** menu.

Preferences	? X
General Editor Window Run Window Printing Admin	
Editor Font Font Name: Courier New Font Size: 10	

Always Show 'Debugger' Menu in the Editor Window Pane

If you wish students to have a more regular exposure to the Debugger, you can set a System preference to cause Turing to always display the Debugger menu. To set the this preference, enter Administrator mode (see <u>Changing System</u> <u>Preferences</u> below) and set the **Always Show 'Debugger' Menu** checkbox in the **Editor Window** pane of the Turing Preferences.

8. Student Distribution of Turing 4.0

The Ontario Ministry license does not include the right to redistribute the software to students. If you wish to give the software to students, a separate license must be obtained directly from Holt Software.

If your school does not have such a license, students can purchase the software for home use by printing and then mailing an order form found <u>http://www.holtsoft.com/studentbuy</u>. This order form can also be used to allow interested students to purchase textbooks.

If your school has purchased a redistribution license (almost 200 schools have done so), then you may redistribute the Turing software to your students. You can either burn copies of the CD to distribute to students or place the Turing installer file onto your school Internet server and give your students the URL.

If you place this file on your school internet server, you must not make any links from your web pages to the software.

This will prevent other users of the web from downloading your software. If you place links from your school's web site, then at some point a web search engine will find the software and post its location to the internet at large, allowing for large scale piracy (and a very overloaded school web server). If there are no links to the software, then only those who have been told the exact URL will be able to download the software.

Note that if you are copying a number of different programs onto a single CD for redistribution, you may rename the Turing installer program to something more descriptive.

Permanent student redistribution licenses can be purchased for \$500-\$750 for most schools. If you are interested in purchasing a student redistribution license, contact Chris Stephenson at (416) 978-6476 or chris@hsa.on.ca

9. Technical Support

If you have problems, questions or suggestions about the Turing software, contact technical support at:

Telephone:	(416) 978-8363
Toll free:	1-800-361-8324
Fax:	(416) 978-1509
E-mail:	<u>west@hsa.on.ca</u>

Turing Debugger Guide

Introduction

Turing supports a number of tools to enable users to debug their programs and examine program execution. Some of the available tools are:

- <u>Stepping execution</u> a line at a time.
- <u>Tracing execution</u> with a set pause between lines of execution
- <u>Setting breakpoints</u> to pause execution at specified points in the program.
- <u>Viewing all open resources</u> of a program including files, fonts, pictures, etc.

As of May 2002, the Turing 4.1 software does not contain the following feature:

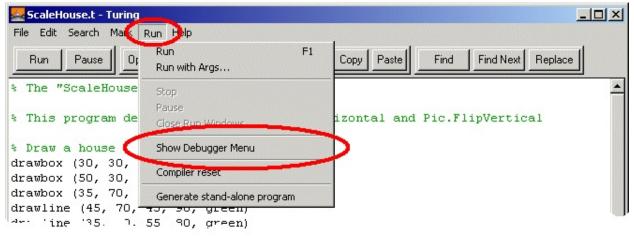
• View variables.

This feature will be implemented as time and developer resources permit.

Displaying The Debugger Menu

In order to use any of the debugger commands, you need to have the Debugger menu showing in the Editor window. By default, the Debugger menu is hidden in order to simplify the Turing environment for the majority of students not using the debugger.

There are two ways that the Debugger menu can be displayed. You can select the **Show Debugger Menu** command in the **Run** menu.



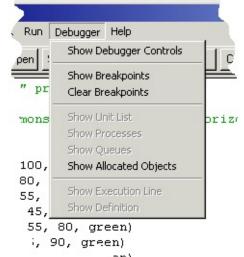
Show Debugger Menu command in the Run menu

If you are consistenly using the debugger, then you should change the Preferences so that the Debugger menu is always displayed. This can be done by bringing up the **Editor Window** pane of the **Preferences** dialog box and setting the **Always Show 'Debugger' Menu** check box.

Preferences	<u>?</u> ×
General Editor Window Run Win	ndow Printing Admin
Editor Font Font Name: Courier New Font Size: 10	 ✓ Suptaw Coloring ✓ Always Show 'Debugger' Menu

Always Show 'Debugger' Menu in the Editor Window Pane

Using The Debugger Menu

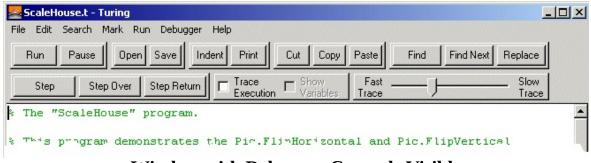


Show Debugger Menu command in the Run menu

The Debugger menu gives you access to all the debugging features in Turing 4.0. The following menu options are available:

• Show Debugger Controls

This command causes each Editor window to display a second row of controls at the top of the window (called the Debugger controls) that allow the user to step through a program and trace execution of the program. When the Debugger controls are visible, then this menu item changes to **Hide Debugger Controls**.



Window with Debugger Controls Visible

• **Show Breakpoints** This command causes each Editor window to dislay an area on the left where breakpoints can be placed. A breakpoint is a marker that causes the program to pause when execution reaches the marked line.

😾 ScaleHouse.t - Turing	
File Edit Search Mark Run Debugger Help	
Run Pause Open Save Indent Print Cut Copy Paste Find	Find Next Replace
Step Step Over Step Return Trace Show Fast Trace	Slow Trace
% The "ScaleHouse" program.	
This program demonstrates the Pic.FlipHorizontal and Pic.F Draw a house drawbox (30, 30, 100, 100, black) drawbox (50, 30, 80, 60, brightred) drawbox (35, 70, 55, 90, green) drawline (45, 70, 45, 90, green) drawline (35, 80, 55, 80, green) drawline (35, 70, 95, 90, green) drawline (85, 70, 85, 90, green) drawline (75, 80, 95, 80, green) drawline (75, 80, 95, 80, green) drawline (30, 100, 65, 135, brightblue) drawline (65, 135, 100, 100, brightblue) drawline (90, 110, 90, 150, black) drawline (75, 125, 75, 150, black)	flipvertical
locate (maxrow, 2)	
<pre> put "Home sweet home" % Create the three pictures.</pre>	
var pic := Pic.New (0, 0, 130, 160)	• •
78 Rows by 22 Columns	Line 1 of 41 Col 1

Window with Breakpoint Visible

- **Clear Breakpoints** This command quickly eliminates all breakpoints that have been set in any Editor window.
- Show Allocated Objects This command shows all the files, fonts, pictures, etc that have been opened or created by the program but have not been closed. As the program executes, this window updates every time a resource is allocated or freed.

P Allocated Objects			
Туре	ID	Where Allocated	Description
폐 File	1	TestDebugAllocator.t - Line 33	D:\Turing\junk1 [put]
🛤 File	3	TestDebugAllocator.t - Line 33	D:\Turing\junk2 [put]
📰 Window	-2	During Initialization	Standard Input/Output: graphics, positio
🛅 Window	-1	During Initialization	Standard Input/Output: graphics,positio
🛅 Window	0	During Initialization	Standard Error: text:3,20,popup,positio
📰 Window	2	TestDebugAllocator.t - Line 38	title:My Window #1
🛅 Window	4	TestDebugAllocator.t - Line 38	title:My Window #2
A Font	5001	TestDebugAllocator.t - Line 40	Serif:10
A Font	5004	TestDebugAllocator.t - Line 40	Serif:12
Directory	6002	TestDebugAllocator.t - Line 41	D:\Turing*
Directory	6005	TestDebugAllocator.t - Line 41	D:\Turing*
Picture	7003	TestDebugAllocator.t - Line 42	[51×101] [21KB] From screen location (5
Picture	7006	TestDebugAllocator.t - Line 42	[51×101] [21KB] From screen location (1
•			<u> </u>

Allocated Objects Window

Turing Execution

The rest of this document will use some terms that must be understood properly.

- **Running** A Turing program is running when statements are being continuously executed without human intervention. However, running may be stalled while the program is waiting for keyboard input, executing a delay statement, etc.
- **Paused** A Turing program is paused when execution is temporarily halted. No input is accepted by the program while it is paused. A program can become paused in a variety of ways including:
 - The user pressing the **Pause** button.
 - The user pressing a **Step**, **Step Over** or **Step Return** button.
 - Execution reaching a breakpoint.
 - Execution reaching the **break** statement in a program.

When a Turing program is paused, the line in the source code on which the Turing program was about to execute is highlighted.

In each case, the program can resume running by clicking the **Resume** button. The program can also temporarily resume running by the user pressing the **Step**, **Step Over** or **Step Return** button. Once the statement or statements are finished running, the program is paused again.

- **Executing** A program is executing when it is either **Running** or **Paused**.
- **Halted** A Turing program is halted when it can no longer executing. It cannot be resumed and must be started from the beginning in order to run it again. A program can be halted by:
 - The user pressing the **Stop** button.
 - A run-time error.
 - The user closing a Run window.

Stepping a Program

Stepping a program consists of executing a Turing program a line at a time. After each step command, execution pauses, highlighting the next line to be executed. To step a program, the program must not be running. The Debugger controls in the Editor window must be visible. To step the program, the user clicks one of the three buttons in the Debugger Controls:

- **Step** If the program is halted, this executes the very first line of code and then pauses. If the program is paused, it executes the highlighted line of code and then pauses. If it line of code to be executed is a subprogram call, then it enters the subprogram and pauses at the first line of code in the subprogram. If it is at the end of the subprogram, then it leaves the subprogram and pauses at the subprogram call. This command is sometimes called **Step Into**.
- **Step Over** This executes a single line of code. However, if the program is paused at a subprogram call, then the entire subprogram is executed and execution is paused at the next line after the subprogram call.
- **Step Return** This resumes execution until the execution returns from a subprogram. The next line after the subprogram call. Note that if this command is given in the main program, then execution will continue until the program terminates.

Tracing a Program

Tracing a program is simply executing a program slowly with each line of code in the program being highlighted as it is executed. Tracing allows users to see how control constructs operate. Unlike stepping, with tracing, execution continues until it is paused or halted in some other fashion.

To start a program tracing, the Debugger Controls must be visible. The user then clicks the **Trace Execution** check box. When the program is next running, each time a statement is executed, the line in the source code will be highlighted. The speed at which the tracing occurs is controlled by the **Trace Speed** slider found in the Debugger Controls. When the slider is on the left, there is essentially no delay between execution of statements. (Note that execution will still be much slower than not tracing due to the time taken to highlight each executing statement.) When the slider is on the right, each statement will wait for about 5 seconds after executing before the next statement is executed.

Setting Breakpoints

Often only a particular section of code is of interest to the user. Breakpoints provide a mechanism for pausing a program only when execution reaches a specified statement. To set breakpoints, they need to be displayed in the Editor window. This is done with the **Show Breakpoints** command in the **Debugger** menu. Once breakpoints are visible, they can be set (and removed) by simply clicking the mouse in the breakpoint area. When the mouse is over the breakpoint area, it turns into a diamond. When the breakpoint is set, a red diamond appears in the breakpoint area. To remove the breakpoint, simply click the red diamond and the breakpoint disappears.

ScaleHouse.t - Turing - U × File Edit Search Mark Run Debugger Help Сору Pause Open Save Print Cut Paste **Find Next** Replace Run Indent Find Fast Trace Slow □ Variable
 Г Step Over Step Return Step Execution Trace Trace * The "ScaleHouse" program. * This program demonstrates the Pic.FlipHorizontal and Pic.FlipVertical 🕆 Draw a house drawbox (30, 30, 100, 100, black) drawbox (50, 30, 80, 60, brightred) drawbox (35, 70, 55, 90, green) drawline (45, 70, 45, 90, green) drawline (35, 80, 55, 80, green) drawbox (75, 70, 95, 90, green) drawline (85, 70, 85, 90, green) drawline (75, 80, 95, 80, green) drawline (30, 100, 65, 135, brightblue) drawline (65, 135, 100, 100, brightblue) drawline (90, 110, 90, 150, black) drawline (75, 125, 75, 150, black) drawline (75, 150, 90, 150, black) (locate (maxrow, 2) put "Home sweet home" ... % Create the three pictures. var pic := Pic.New (0, 0, 130, 160) Line 1 of 41 78 Rows by 22 Columns Col 1

Window with Two Breakpoints Set

When execution reaches a line with a breakpoint on it, the program pauses and

the line is highlighted. The program can then be resumed or stepped as desired. Note that it is possible to set a breakpoint on a line that is not executed. If you place a breakpoint on a blank line, it is quite likely that execution will not stop. Likewise, in lines broken over multiple lines, it is best to put the breakpoint on the last line in the statement.

To clear all the breakpoints, select **Clear Breakpoints** from the **Debugger** menu.

Viewing Open Files/Pictures/Fonts, etc.

A common error in Turing programs is to neglect to close open files (using the **Close** statement) or free pictures or fonts (using the **Pic.Free** or **Font.Free** procedures). The **Allocated Objects** window displays all allocated resources until they are freed. This window is displayed by selecting the **Show Allocated Objects** command from the **Debugger** menu. The command can be given even after a program has halted. It is a useful command to use when am **open**, Pic.New or Font.New has failed because of too many open files, pictures or fonts.

The window will also display just how much memory each picture takes up.

Allocated	Objects		
Туре	ID	Where Allocated	Description
🛥 File	1	TestDebugAllocator.t - Line 33	D:\Turing\junk1 [put]
🛥 File	3	TestDebugAllocator.t - Line 33	D:\Turing\junk2 (put)
🛅 Window	-2	During Initialization	Standard Input/Output: graphics,positio
🛅 Window	-1	During Initialization	Standard Input/Output: graphics,positio
🛅 Window	0	During Initialization	Standard Error: text:3,20,popup,positio
🗂 Window	2	TestDebugAllocator.t - Line 38	title:My Window #1
🗂 Window	4	TestDebugAllocator.t - Line 38	title:My Window #2
A Font	5001	TestDebugAllocator.t - Line 40	Serif:10
A Font	5004	TestDebugAllocator.t - Line 40	Serif:12
Directory	6002	TestDebugAllocator.t - Line 41	D:\Turing*
🚞 Directory	6005	TestDebugAllocator.t - Line 41	D:\Turing*
🕋 Picture	7003	TestDebugAllocator.t - Line 42	[51×101] [21KB] From screen location (5
🛋 Picture	7006	TestDebugAllocator.t - Line 42	[51×101] [21KB] From screen location (1
•			Þ

Allocated Objects Window

Clicking on the line containing the allocated object will highlight the line in the code where the object was allocated. The headings on each of the columns can be clicked allowing the objects to be sorted either by their ID, or by where in the program they were allocated.

Error.TripMsg

Part of <u>Error</u> module

Syntax	Error.TripMsg (errorCode : int, errorMessage : string)
Description	Error.TripMsg is a procedure that sets the error number and error is returned by Error.Last and Error.LastMsg . It does not halt the
	You can find a list of constants for the legal error codes in the mod ErrorNum . Any call to Error.TripMsg should use a constant four ErrorNum module or zero, if the error code is not relevant (i.e. on message is relevant).
	This program sets an error code. The program outputs 201 for the ϵ and "File *really* not found" for the message.
Example	Error.TripMsg (<i>eFsysFileNotFound</i> , "File *really* put "Error code = ", Error.Last put "Error message = ", Error.LastMsg

Execute

Exported qualified.

StatusThis means that you can only call the function by calling Error.Tri
calling TripMsg.

Error.Halt

Syntax	Error.Halt (errorMessage : string)
Description	Error.Halt is a procedure that immediately halts execution of the program and shows the specified error message on the line in the program that calls Error.Halt .
	This procedure is useful for allowing a quick halt to a program whi indicating an error condition.
Example	This program halts on the first line with an error message stating "Execution terminated by the program."
	Error.Halt ("Execution terminated by the program

Execute

Exported qualified.

Status This means that you can only call the function by calling **Error.Ha** not by calling **Halt**.