

ETAC: Visual Interaction System



Other Related ETAC Documents

ETAC_Preliminaries.pdf	Preliminaries before using ETAC
ETACOverview.pdf	An Overview of ETAC
ETACProgLang(Official).pdf	The Official ETAC Programming Language
RunETAC.chm	Run ETAC Scripts Help
ETACWithCPP.pdf	ETAC: Interacting with C++
ETACCompiler.pdf	The ETAC Compiler
ETACCompiler.chm	ETAC Compiler Help
ETACErrorCodes.pdf	ETAC Compilation and Run-time Error Codes
FunctionsETACScriptLib.pdf	Functions ETAC Script Library
SplineScriptFnts.pdf	Spline ETAC Script Library
PathScriptFnts.pdf	Path ETAC Script Library
VISShapePathFnts.pdf	VIS Shape Path Functions
VISPointSetFnts.pdf	VIS Point Set Functions

Legal Information

ETAC and  (the ETAC logo) are unregistered trademarks (™) of Victor Vella for *computer software incorporating an implementation of a computer programming language*. There may be other owners of the “ETAC” trademark used for other purposes.

The author of this document shall not be liable for any direct or indirect consequences arising with respect to the use of all or any part of the information in this document, even if such information is inaccurate or in error. The information in this document is subject to change without notice.

Visual Interaction System

Victor Vella

Published by Victor Vella
(1 December 2025)

First Published: 1 December 2025

Copyright © Victor Vella (2025). All rights reserved.

Permission is hereby granted to make any number of exact electronic copies of this document without any remuneration whatsoever. Permission is also granted to make annotated electronic copies of this document for personal use only. Except for the permissions granted, and apart from any fair dealing as permitted under the relevant Copyright Act, no part of this document may be reproduced or transmitted in any form or by any means without the express permission of the author. The copyright of this document shall remain entirely with the original copyright holder.

Preface

The ETAC™ interpreter now intrinsically contains a system that allows user visual interactions with computer software. The system is called VIS (Visual Interaction System), and is implemented in both ETAC code and the ETAC interpreter executable program (C++) and can be used to create fully-fledged interactive 2-D graphics computer programs via the ETAC programming language. The system incorporates mainly vector graphics that the user can interact with, without the application programmer writing substantial ETAC code in many cases. No third-party software is used in or by VIS; all of the VIS and ETAC software is an original design by me at all levels.

Indeed, I designed and incorporated the graphics system along with the unreleased predecessor of ETAC (pronounced: E-tack) called TAC, beginning in the year 2000. However, I did not release the graphics system part of TAC to the public along with the release of ETAC (see the Preface of the ETACProgLang(Official).pdf document for a short history of the development of ETAC). I needed to make significant changes to the unreleased graphics system to make it compatible with the ETAC programming language, and suitable for public use.

In fact, the VIS version of ETAC was pre-release in November 2024. Since then, I made only minor changes to VIS; most of my spare time was involved in creating some software tools and documentation for the official VIS release of the ETAC programming language. The full documentation itself takes considerable time to produce.

Unlike other current graphics systems, I did not intend VIS to be a glorified dialog box. Instead, my intention was to create a graphics system that is more oriented towards developing interactive diagrammatic type application programs, with common features like object selection, scrolling, zooming, etc, built into the interpreter rather than having the programmer toil with such features. I also wanted the system to be a more accurate representation of the graphics structure of a graphics program. This requires, among other things, that the visible graphics objects be defined by description (declarative) rather than by an algorithm that needs to be applied each time the operating system wants to refresh a window. VIS does not define any dialog box controls; these can already be created under ETAC.

This document represents the first official release of the Visual Interaction System for the ETAC programming language.

Victor Vella

Perth, Western Australia
1 December 2025

Contents

Preface.....	v
Contents.....	vi
Tables and Diagrams.....	ix
Document Conventions.....	x
Introduction.....	1
1 Visual Interaction System.....	2
1.1 Introduction (VIS).....	2
1.2 Autonomous Features.....	3
1.3 VIS Components.....	3
1.3.1 Base Component.....	4
1.3.2 Project Component.....	4
1.3.3 Drawing Plane Component.....	4
1.3.4 VIEW Component.....	5
1.3.5 Graphics Object Component.....	6
1.3.6 Line Item Component.....	6
1.3.7 Curve Item Component.....	6
1.3.8 Text Item Component.....	6
1.3.9 Image Item Component.....	7
1.3.10 Formatted Text Item Component.....	7
1.3.11 Graphics Context Component.....	7
1.3.12 Pen Attributes Component.....	7
1.3.13 Brush Attributes Component.....	8
1.3.14 Bitmap Attributes Component.....	8
1.3.15 Text Style Attributes Component.....	8
1.3.16 Matrix Object Component.....	8
1.3.17 Matrix Object Definition Component.....	9
1.3.18 Measurement Object Component.....	9
1.3.19 Measurement Object Definition Component.....	10
1.3.20 Data Pack Component.....	10
1.4 The VIS Framework.....	11
1.5 VIS Frames.....	11
1.5.1 Drawing Frame.....	12
1.5.2 Window Frame.....	12
1.5.3 Client Rectangle.....	13
1.5.4 Window Boundary.....	13
1.5.5 Reference Frame.....	13
2 Programming Guide.....	14
2.1 Introduction (Guide).....	14
2.2 Preliminaries for Executing VIS Code.....	14
2.3 Default Features of VIS.....	15

2.4	Drawing Plane Features.....	15
2.4.1	Coordinate System.....	15
2.4.2	Distance Units.....	15
2.4.3	Background Colour.....	16
2.4.4	Mouse Cursor Image.....	16
2.4.5	Drawing Frame Visibility.....	16
2.4.6	Smoothing.....	16
2.5	Window Features.....	16
2.5.1	Co-VIEWS.....	17
2.5.2	Scrolling.....	17
2.5.3	Zooming.....	18
2.5.4	Menu.....	18
2.5.5	Status Bar.....	18
2.6	Creating Components.....	18
2.7	Component Dependencies.....	22
2.8	Graphics Objects.....	23
2.8.1	Graphics Items.....	24
2.8.2	Fixed Size Items.....	25
2.8.3	Scale Range Visibility.....	25
2.8.4	Graphics Object Groups.....	26
2.9	Vector Graphics.....	27
2.9.1	Shape Path Functions.....	27
2.9.2	Point Set Functions.....	28
2.9.3	Custom Arrowheads.....	28
2.10	Bitmap Graphics.....	29
2.11	Text Graphics.....	29
2.11.1	Regular Text Item.....	30
2.11.2	Formatted Text Item.....	31
2.12	Selecting Graphics Objects.....	32
2.12.1	Manipulating a Graphics Object.....	33
2.12.2	Selecting Many Graphics Objects.....	34
2.12.3	Manipulating Many Graphics Objects Together.....	34
2.13	Matrix Objects.....	35
2.14	Measurement Objects.....	36
2.15	Events and Event Handlers.....	38
2.15.1	Events and Their Handler Functions.....	38
2.15.2	Setting and Creating a Handler Function.....	40
2.15.3	Creating a Graphics Object from within a Handler Function.....	42
2.16	Idle-time Processing.....	42
2.17	Colour and Font Dialog Boxes.....	43
2.18	Clipboard.....	44
2.19	Animation Emulation.....	44
3	Programming Reference.....	46
3.1	Introduction (Reference).....	46
3.2	Mathematical Constants.....	46
3.3	Pre-processor Definitions.....	47

3.3.1	System Definitions for Windows.....	47
3.3.2	Visual Interaction System Definitions.....	62
3.3.3	Operating System Messages.....	79
3.4	VIS Component Data Definitions.....	85
3.4.1	Variable Naming Conventions.....	85
3.4.2	Component Member Summary.....	86
3.4.3	Notational Conventions.....	95
3.4.4	All Components.....	96
3.4.5	Project.....	99
3.4.6	Drawing Plane.....	100
3.4.7	VIEW.....	124
3.4.8	Graphics Object.....	156
3.4.9	Line Item.....	168
3.4.10	Curve Item.....	174
3.4.11	Text Item.....	180
3.4.12	Formatted Text Item.....	188
3.4.13	Image Item.....	194
3.4.14	Graphics Context.....	199
3.4.15	Pen Attributes.....	204
3.4.16	Brush Attributes.....	209
3.4.17	Bitmap Attributes.....	214
3.4.18	Text Style Attributes.....	217
3.4.19	Matrix Object.....	226
3.4.20	Matrix Object Definition.....	230
3.4.21	Measurement Object.....	239
3.4.22	Measurement Object Definition.....	250
3.4.23	Data Pack.....	258
3.5	VIS Application Programming Interface.....	258
3.5.1	Variable Naming Conventions.....	259
3.5.2	Command and Function Summary.....	259
3.5.3	Notational Conventions.....	261
3.5.4	Menu.....	262
3.5.5	Status Bar.....	270
3.5.6	Idle-time Job.....	275
3.5.7	API Commands.....	280
3.5.8	API Functions.....	289
	Bibliography.....	315
	Glossary.....	316

Tables and Diagrams

Document Conventions.....	x
Component Creation Hierarchy.....	19
Component Types and Prefix.....	21
Event Registration Functions.....	39
Mathematical Constants.....	46
Common Colour Table.....	68

Document Conventions

The following conventions are used in this document.

Document Conventions

<u>Symbol</u>	<u>Meaning</u>
$\langle x \rangle$	separates x as a unit of information from the surrounding text.
$x \cdots$	middle ellipsis means zero, one, or more of the same kind as x .
$[x]$	means that x optional.
$\{x\}$	means that x is the default value.
(x)	groups x as a unit.
$x \mid y$	means that only x or y applies, but not both (could have more than two options).
$\dots$	ellipsis represents omitted text (as usual).
<i>text</i>	maroon coloured italic text is a link to the text's definition.
<u>text</u>	underlined green text is a link into the document.
text	bold green text is a link into the document.
♦	indicates the end of a block of document text.

Introduction

This document is for version **1-1** of the ETAC programming language implemented in RunETAC.exe and AppETAC.dll version **4-0-6-ena**.

(Australian English)

Version 4-0 of the ETAC™ interpreter introduces an interactive graphics system. The system is called “VIS”, which stands for “Visual Interaction System”. VIS is a 2-D graphics system that processes user interactions with computer software, allowing a computer programmer to create event-driven windowed graphical application programs in the ETAC programming language. The implementation of VIS is shared between ETAC system code and the ETAC interpreter. VIS is implemented in both RunETAC.exe and AppETAC.dll, and the ETAC system code is implemented in visETAC.btac and visETAC-D.btac (the latter being the debug version of the implementation) which are loaded by the *loader script*.

Unlike traditional graphics systems (at the time of this writing), VIS does not have the ability to draw graphics objects or display them explicitly. Instead, the programmer only needs to specify the desired features of graphics objects, and VIS will display those objects automatically. Because VIS owns the graphics objects, it can facilitate some interactions between the user and those objects automatically, removing the need for the programmer to explicitly program those interactions. Some of those interactions include automatic rendering of a prescribed drawing into a window, scrolling the image in a window, zooming in and out of the image in a window, mouse click and drag selection of graphics objects, graphics object resizing, graphics object movement, automatic highlighting of graphics objects when selected, *elastic windows*, scale range drawing for items, and fixed-size items when scaling.

This document assumes that the reader is familiar with the ETAC programming language. Familiarity with the basic principles of graphics programming would also be an advantage.

Chapter 1, Visual Interaction System, describes the main conceptual features of VIS.

Chapter 2, Programming Guide, describes various aspects of using VIS in ETAC programs. This chapter should be read before a programmer writes their first VIS ETAC program.

Chapter 3, Programming Reference, is a reference to the pre-processor definitions, *components*, functions, and commands that an ETAC programmer may need to consult while writing VIS ETAC programs.

Visual Interaction System

This chapter presents the fundamental concepts of the Visual Interaction System (VIS). A programmer should read this chapter before beginning their first VIS ETAC™ program to gain a conceptual understanding of VIS. The reader is assumed to have some knowledge of basic graphics programming concepts. Familiarity with the ETAC programming language is assumed.

1.1 Introduction (VIS)

VIS (Visual Interaction System) is a system that enhances the ETAC™ interpreter and ETAC system code providing capabilities to an application program involving user visual interactions with computer software. VIS is implemented in both *ETAC code* and the ETAC interpreter executable program (RunETAC.exe and AppETAC.dll), and can be used to create fully-fledged interactive 2-D graphics computer programs using the ETAC programming language. The system incorporates mainly vector graphics (using a Cartesian coordinate system) that the user can interact with, without the application programmer writing *ETAC text script* in some cases. Optional autonomous features of VIS include mouse zooming (with zoom limits), mouse and arrow key scrolling (with optional scroll bars), multiple graphics object mouse click and drag selection and highlighting, multiple graphics object resizing, and multiple graphics object dragging. Graphics object drawing into a *VIEW* (Visual Interactive ETAC Window) is automatic; there are no functions to explicitly draw graphics objects. However, if necessary, graphics objects can be invalidated, allowing VIS to draw those objects at a later time. Currently, VIS does not support alpha-channel transparencies.

VIS allows an ETAC programmer to create a graphics application program by constructing the program via predefined (by VIS) *components*. There are *components* for projects, *drawing planes* (the “canvas” by which graphics objects are defined), *VIEWs*, graphics objects and sub-objects (the programmer-defined graphics to be drawn), five types of graphics items (which belong to graphics objects), graphics contexts, pen attributes, brush attributes, bitmap attributes, text style attributes, and data packs. The top-level *components* are projects, and all other *components* form a hierarchy under, and belong to, a project *component*. There could be more than one project *component* in an application program. A project *component* typically corresponds to the information that might be saved in a file for later loading. VIS does not support dialog boxes or dialog box controls at all; such controls can only be processed via a dialog box data object returned by the existing `etac.@NewDialog()` function.

The programmer can specify six levels of smoothness for rendering graphics objects into a *VIEW*. VIS uses a *supersampling* method to achieve smoothness; the rendering may be slow, and hairlines may be fainter. By default, VIS does not do smooth rendering, which produces the fastest rendering. However, because the *supersampling* method is slow when smoothing, VIS may optimise the rendering process by temporarily doing default rendering on some occasions, such as during scrolling or zooming the drawing in a *VIEW*.

VIS *components* are not accessed by (or stored in) variables, but accessed by a unique (programmer defined) text name (called a *name identifier*) or corresponding (VIS generated) numerical identifier (called a “CID” which stands for “*crucial identifier*”). VIS owns the *components* and maintains them where required. Any *component* under a project *component* can access any other *component* under that same project (and, indirectly, any *component* of another project) at any time and place within the program via the `@VIS()` and `@VISX()` functions.

To an ETAC application program, VIS is a set of ETAC API functions and commands, most of which create and set up the *components* of the application program as configured by the application programmer. However, VIS does not have direct drawing functions; the application program defines what to draw, but not how to draw. Therefore, the drawings that appear in *VIEWs* are defined by the application programmer via graphics *components*, and VIS automatically takes care of rendering a drawing onto the appropriate *VIEW* (VIS can do that because it owns and has access to the *components*).

1.2 Autonomous Features

VIS implements some autonomous features, alleviating programmers themselves from explicitly programming those features. Of course, those features are optional, and the programmer can disable them and explicitly program their own desirable versions of those features instead. The autonomous features are typically activated by the programmer setting appropriate flags or calling appropriate functions; no other programming is necessary. Some of those features are on by default.

Autonomous features include automatic rendering of a prescribed drawing into a *VIEW*, scrolling the image in a *VIEW*, zooming in and out of the image in a *VIEW*, mouse click and drag selection of graphics objects, graphics object resizing, graphics object movement, automatic highlighting of graphics objects when selected, *elastic windows*, scale range drawing for items, and fixed-size items when scaling.

An “*elastic window*” keeps the same image in the window (*VIEW*) regardless of the window’s size. Therefore, the image shrinks with the window if the user makes the window smaller, or expands with the window if the user makes the window larger (so the window acts as a perfect elastic, shrinking and expanding the image along with the window’s size).

Scale range drawing for items allow parts of graphics objects to be visible only within a specified range of the *VIEW* scale (zoom level). Outside that range, the object parts become invisible.

Some parts of graphics object can be created for displaying at a fixed size regardless of the *VIEW* scale. This feature is useful for displaying control handles in a diagram, which need to be displayed at a constant size when the user zooms in or out of the diagram.

1.3 VIS Components

A graphics application program written in ETAC requires the creation of VIS *component* instances (which are actually ETAC data objects). There are altogether 19 *component* types from which instances can be created. *Component* instances are created by the ETAC program via appropriate functions, but owned and maintained (where required) by VIS. The created *components* form a tree structure with a project *component* at the top. Certain *component* pairs have a parent-child relation between them; when the parent *component* is deleted, the child *component* is also automatically deleted.

Each *component* instance is identified by a unique numerical identifier (called the *crucial identifier*), generated internally by VIS, and a corresponding unique textual identifier (called the *name identifier*) optionally created by the application program or automatically by VIS. Each project *component* instance has its own unique system of textual identifiers for the *components* under it; a *component* belonging to project *component* can have the same textual identifier, but not the same numerical identifier, as a *component* belonging to a different project *component*. A program assesses a *component* either by its numerical or textual identifier.

A *component* instance is created by the relevant VIS function (depending on the *component* type) which takes a (possibly empty) string name and a TAC procedure (called the “*initialiser procedure*”) as its two arguments. The function does not return a value, but registers the *component* with VIS. The actual *component* is created after the *initialiser procedure* ends its

processing but before the function ends. The string argument is the *component*'s textual identifier, which may be altered by VIS if it is not unique within the relevant project *component*. If the string argument is empty, VIS creates a unique *name identifier* for the *component*. A unique numerical identifier (*CID*) for the *component* is also automatically created by VIS; the programmer can read the identifier but cannot alter it. VIS recognises the textual identifiers, but internally uses the numerical identifiers, not the textual identifiers, to access the *component* instances. The textual identifiers are mainly for the convenience of the programmer, and can be altered at any time by the application program.

Components are created with pre-allocated member variables and functions. The member variables serve as parameters to some of the member functions. The programmer sets the values of desired variables from within the *initialiser procedure* or before calling the relevant member function.

A summary of the VIS *component* types is presented in the subsections below.

1.3.1 Base Component

This *component* is not a *component* in its own right, but forms the common basis of all other *components*. This *component* manages and allows access to the *CID*, *name ID*, parent *CID*, ancestor *drawing plane CID*, and *component* type for the other *components*.

1.3.2 Project Component

This is the top-level *component*. There can be any number of project *components*. A project *component* instance owns the instances of all other types of *components*, and all project *components* are owned by VIS. The VIS ETAC interpreter creates one default project *component* instance, uniquely identified by “@VIS@”, before it starts any ETAC program.

A project *component* is typically used to represent the data contained in a file of an application program; the data of each loaded file is represented by its own project *component* instance. Of course, the application program can use project *components* for whatever purpose it likes.

A project *component* instance handles the **F1** help key event.

1.3.3 Drawing Plane Component

The drawing plane *component* defines the “canvas” into which graphics objects are conceptually drawn. The “canvas” is not part of the window rendering area but is independent of any window. A drawing plane *component* instance is the parent of most of the other *components*, but is itself a child of a project *component*. There can be more than one drawing plane *component* instance for the same project *component* instance.

The drawing plane *component* manages an imaginary Cartesian coordinate system by which graphics objects are defined. This is the only coordinate system that VIS recognises; there is no “world coordinate system” or “device coordinate system” in VIS. Note that with a Cartesian coordinate system, the positive Y value is in the upward direction not the downward direction (as is the case for traditional graphics systems). The *drawing plane* origin is at the centre of the *drawing plane*.

A *drawing plane* has only one distance unit, the “*base unit*”, which is a decimal unit defined as follows. One *base unit* is equal in length to one *logical pica* under default conditions. A *logical pica* is defined to be one-sixth of a logical inch, and one logical inch has the pixel length as defined by the DPI (dots per inch) setting of the operating system (which can be modified by the user). Therefore, a length of one *base unit* is DPI/6 pixels on the screen under default conditions. The default DPI is 96, therefore, one *base unit* is 16 pixels by default. Note that one *logical pica* is equivalent to 12 logical font points. Therefore, one *base unit* corresponds to the character

height of a 12-point font. The application program is responsible for converting *base units* to other desired physical units.

Distance measurements (dimensions and positions) of all graphics objects are defined in terms of *base units*. The accuracy of a distance measurement depends on the *drawing plane* density as defined by the programmer. The default density is 48 dots per *base unit*. A distance measurement, therefore, has an accuracy of $1/48$ ($= 0.0208\bar{3}$) *base unit* by default.

The whole *drawing plane* is quite large (approximately 89,478,483 *base units* both wide and high under default settings), so a drawing of only a few *base units* wide would barely be visible if the whole *drawing plane* were to be displayed in a screen window. Therefore, the presentation of a drawing is limited to a reasonably sized rectangle, called the “*drawing frame*”, within the *drawing plane*. The *drawing frame* position and size can be set (and made visible) by an ETAC program.

The drawing plane *component* determines the background colour of its *co-VIEWS*, and also the pen and brush attributes of the *drawing frame* rectangle if it is made visible.

A drawing plane *component* instance handles the following events: mouse button, mouse movement, graphics object *hit testing*, timer, keyboard key operation, *VIEW* close, menu command, and miscellaneous activities (includes user object tracking and menu highlight events).

1.3.4 VIEW Component

The *VIEW component* represents a screen window into which part of the graphics image in the associated *drawing plane* is presented to the user. A *VIEW component* is a child of a drawing plane *component*, and can present only the drawing in the associated *drawing plane*. There can be more than one *VIEW component* instance for a particular drawing plane *component* instance; each *VIEW component* instance can have its own features, and can present different parts of the *drawing plane* image. Two or more *VIEWs* for the same *drawing plane* are called “*co-VIEWS*”.

The client area of a *VIEW* is mapped into an imaginary rectangle (*client rectangle*) within the *window frame* of the associated *drawing plane*. The *window frame* may be larger than the *client rectangle* depending on certain conditions. However, the top-left position of the *client rectangle* is always at the top-left position of the *window frame*, and the width of the *client rectangle* is always relative to the width of the *window frame*. Only that part of the drawing that is within the *client rectangle* appears in the window’s client area. For a typical situation, the whole width of a maximised window client area maps onto the whole width of the *window frame*. The smaller the *window frame*, the larger the drawing will appear in the window’s client area. So, when the user zooms into a drawing, or the programmer increases the drawing scale, VIS reduces the size of the *window frame* (and therefore the *client rectangle*, being proportional to the *window frame*), making the drawing appear larger in the *VIEW*. Similarly, zooming out causes the *window frame* to increase in size, making the drawing appear smaller. The programmer can adjust only the position of the *window frame*; the *window frame* sizing is automatically handled by VIS when the user or programmer changes the *VIEW* scale. Each *co-VIEW* has its own *window frame*.

The *client rectangle* will always exist within the *drawing frame* rectangle plus an outside margin (*window boundary*) set by the programmer. That margin is zero by default, therefore, the *client rectangle* will be within the *drawing frame* rectangle under typical conditions. In the case that the *client rectangle* becomes larger than the *drawing frame*, VIS automatically, and temporarily, increases the margin to accommodate the *client rectangle*. When a user scrolls the client area, VIS automatically adjusts the *window frame* position as appropriate.

The *VIEW component* maintains the visible *VIEW* features, such as the title, position, size, and style. A *VIEW* can be set with any of the styles and extended styles, except for a few that make no sense, available to the operating system.

A *VIEW component* instance handles the following events: active window, *VIEW* size, *VIEW* close, system command, and keyboard key operation.

1.3.5 Graphics Object Component

The graphic object *component* represents an element of a drawing that can be manipulated by the user. A graphics object *component* is a child of a *drawing plane component*, or a child of another graphics object *component*; graphics object *component* instances can be nested to any level. More than one graphics object *component* instance can exist for the same *drawing plane component* or other graphics object *component* instances.

A whole diagram in an application program is typically defined by a number of graphics objects. Each graphics object has an origin (relative to the *drawing plane* origin) and an automatically calculated bounding box (defined in *base units*). However, a graphics object by itself does not display anything; it is an empty object. To present an image onto the *VIEW*, a graphics object *component* needs one or more child graphics item *components*. There are five types of graphics item *components*: line, curve, text, formatted text, and image. Every graphics item *component* instance is a child of, and needs, a parent graphics object *component* instance. These item *components* are described in the subsections below. The graphics items define some kind of image to be displayed relative to the parent graphics object's origin. The bounding box of a graphics object is the smallest rectangle that contains all the images of its children graphics items. The bounding box is automatically calculated by VIS as necessary.

A graphics object *component* instance handles the mouse button and graphics object *hit testing* events.

1.3.6 Line Item Component

The line item *component* represents a joined sequence of line segments (typically called a “poly-line”), and is a child of a graphics object *component*. More than one line item *component* instance can exist for the same graphics object *component* instance.

The poly-line of a line item is defined by a sequence of points (in *base units*) indicating the vertices of the poly-line relative to the parent graphics object's origin. A line item has pen and brush attributes. These are defined by the pen attributes *component* and the brush attributes *component*, respectively, described in a later section.

1.3.7 Curve Item Component

The curve item *component* represents a joined sequence of Bezier curve segments (sometimes called a “poly-curve”), and is a child of a graphics object *component*. More than one curve item *component* instance can exist for the same graphics object *component* instance.

The poly-curve of a curve item is defined by a sequence of points (in *base units*) indicating the control points of joined Bezier curves relative to the parent graphics object's origin. A curve item has pen and brush attributes. These are defined by the pen attributes *component* and the brush attributes *component*, respectively, described in a later section.

1.3.8 Text Item Component

The text item *component* represents plain text, and is a child of a graphics object *component*. More than one text item *component* instance can exist for the same graphics object *component* instance.

A text item is defined by a text string and a rectangle within which the text can be displayed at various positions. The rectangle is defined relative to the parent graphics object's origin. A text item has attributes defined by the text style attributes *component* described in a later section.

1.3.9 Image Item Component

The image item *component* represents a bitmap image, and is a child of a graphics object *component*. More than one image item *component* instance can exist for the same graphics object *component* instance.

An image item is defined by a rectangle within which the image can be displayed. The rectangle is defined relative to the parent graphics object's origin. An image item has attributes defined by the bitmap attributes *component* described in a later section. The actual bitmap data is specified in the bitmap attributes. VIS lacks the ability to manipulate bitmap pixels.

1.3.10 Formatted Text Item Component

The formatted text item *component* represents formatted (RTF) text, and is a child of a graphics object *component*. More than one formatted text item *component* instance can exist for the same graphics object *component* instance.

A formatted text item is defined by an RTF (**R**ich **T**ext **F**ormat) coded text string and a rectangle within which the text can be displayed. The rectangle is defined relative to the parent graphics object's origin. A formatted text item has no attributes other than being able to set the background colour in the item *component* itself. Other attributes are specified within the actual RTF text string.

1.3.11 Graphics Context Component

The graphics context *component* represents the highlighting features of a graphics object, and is a child of a drawing plane *component*. More than one graphics context *component* instance can exist for the same drawing plane *component* instance. Alternatively, a graphics context *component* instance can be created within the *initialiser procedure* of a graphics object *component* instance. In this case, the graphics context *component* will be a child *component* of the drawing plane *component*, rather than the *component* under which it was created.

A graphics context is defined by the type of highlighting required: border, background, tracking rectangle highlighting, or any combination of the three. Border highlighting requires attributes defined by the pen attributes *component*; background highlighting requires attributes defined by the brush attributes *component*; and tracking rectangle highlighting requires attributes defined within the graphics context *component* itself. If the required attributes are not specified, then default attributes will take effect. The pen and brush attributes *components* are described in the sections below. A tracking rectangle may have sizing handles, and the user can make the rectangle itself or its handles track a moving mouse pointer.

1.3.12 Pen Attributes Component

The pen attributes *component* represents the attributes of a poly-line or poly-curve, and is a child of a drawing plane *component*. More than one pen attributes *component* instance can exist for the same drawing plane *component* instance. Alternatively, a pen attributes *component* instance can be created within the *initialiser procedure* of a line item, curve item, or graphics context *component* instance. In each case the pen attributes *component* will be a child *component* of the drawing plane *component*, rather than the *component* under which it was created.

The pen attributes *component* specifies the following attributes of a poly-line or poly-curve: foreground colour, style (solid, dashed, dotted, etc), line or curve thickness, background colour, background mode (transparent or opaque), and drawing mode (specifies how the foreground colour of the pen is combined with the existing screen colour being replaced).

1.3.13 Brush Attributes Component

The brush attributes *component* represents the attributes of the area within a closed shape, and is a child of a drawing plane *component*. More than one brush attributes *component* instance can exist for the same drawing plane *component* instance. Alternatively, a brush attributes *component* instance can be created within the *initialiser procedure* of a line item, curve item, or graphics context *component* instance. In each case the brush attributes *component* will be a child *component* of the drawing plane *component*, rather than the *component* under which it was created.

A brush attributes *component* specifies the following attributes: foreground colour, brush style (solid, hatched, etc), hatch style (type of hatching), background colour, background mode (transparent or opaque), and drawing mode (specifies how the foreground colour of the brush is combined with the existing screen colour being replaced).

1.3.14 Bitmap Attributes Component

The bitmap attributes *component* represents the attributes of a bitmap image, and is a child of a drawing plane *component*. More than one bitmap attributes *component* instance can exist for the same drawing plane *component* instance. Alternatively, a bitmap attributes *component* instance can be created within the *initialiser procedure* of an image item or brush attributes *component* instance. In each case the bitmap attributes *component* will be a child *component* of the drawing plane *component*, rather than the *component* under which it was created.

A bitmap attributes *component* specifies the following attributes of a bitmap image: BMP file data (as a TAC memory object) or the file path of a BMP file, raster operation mode (specifies how the bitmap pixels are combined with the existing screen pixels being replaced), and stretch mode (specifies how the bitmap data is temporarily modified before being rendered at a different size).

1.3.15 Text Style Attributes Component

The text style attributes *component* represents the attributes of text, and is a child of a drawing plane *component*. More than one text style attributes *component* instance can exist for the same drawing plane *component* instance. Alternatively, a text style attributes *component* instance can be created within the *initialiser procedure* of a text item *component* instance. In this case, the text style attributes *component* will be a child *component* of the drawing plane *component*, rather than the *component* under which it was created.

A text style attributes *component* specifies the following attributes of all the characters of a piece of text: height, width, weight, italic, underline, strikethrough, character set, pitch and family, face name, foreground colour, background colour, and background mode (transparent or opaque).

1.3.16 Matrix Object Component

The matrix object *component* incorporates a graphics object *component* and represents a group of specified graphics objects laid out in relation to each other in a specified way. A matrix object *component* is a child of a drawing plane *component*, or a child of a graphics object *component*. More than one matrix object *component* instance can exist for the same drawing plane or graphics object *component* instance.

A matrix object *component* is specified by a sequence of graphics object *component* identifications, and a matrix object definition *component*. The matrix object definition *component*, described in the section below, contains the specification for how the said sequence of graphics objects are laid out. Note that a matrix object *component* has nothing whatsoever to do with mathematical matrices used for coordinate transformations, as is the case in some traditional graphics libraries. It is called a “matrix” object because the default layout of the specified graphics objects is in the form of a matrix (ie: a grid).

1.3.17 Matrix Object Definition Component

The matrix object definition *component* represents the layout specifications of the graphics objects specified for the associated matrix object. A matrix object definition *component* is a child of a drawing plane *component*. More than one matrix object definition *component* instance can exist for the same drawing plane *component* instance. The same matrix object definition *component* instance can be specified for more than one matrix object *component* instance. Alternatively, a matrix object definition *component* instance can be created within the *initialiser procedure* of a matrix object *component* instance. In this case, the matrix object definition *component* will be a child *component* of the drawing plane *component*, rather than the *component* under which it was created, and can be used with other matrix objects.

A matrix object definition *component* is specified by:

- the number of rows and columns in the displayed matrix object,
- the nominal horizontal and vertical spacing between the rows and columns,
- a sequence for each cell of the matrix object specifying whether or not that cell will contain a graphics object,
- a sequence containing the order that the cells in the matrix are processed,
- a sequence for each cell of the matrix object specifying the horizontal position of the graphics object in that cell relative to the vertical borders of graphics objects in one or two other cells,
- a sequence for each cell of the matrix object specifying the vertical position of the graphics object in that cell relative to the horizontal borders of graphics objects in one or two other cells.

The layout of the graphics objects of the associated matrix object is maintained even when those graphics objects change their size.

1.3.18 Measurement Object Component

The measurement object *component* incorporates a graphics object *component* and represents the image of one of four types of measurements. A measurement object *component* is a child of a drawing plane *component*, or a child of a graphics object *component*. More than one measurement object *component* instance can exist for the same drawing plane or graphics object *component* instance.

There are four possible types of measurements that a measurement object can display.

- **Standard measurement:** measures the distance between two parallel lines.
- **Angle measurement:** measures the angle between two intersecting lines.
- **Radial measurement:** measures the distance between two points.
- **Right-angle measurement:** indicates a right angle (using the right-angle symbol, $\sphericalangle$).

In each case, a measurement object displays the visual measurement image along with the actual value of the measurement in *base units*. For example, with a typical standard measurement, a measurement object would display two parallel lines with an arrowed line between those two lines along with a number beside the arrowed line. The programmer has an enormous amount of control over the displayed measurement image features.

A measurement object *component* is specified by:

- two or four points (depending on the measurement type) for calculating the measurement value and drawing the appropriate measurement lines,
- a measurement object definition *component*,
- the number of significant digits of the measurement value,
- a measurement value multiplier,
- the size (length) of all arrowheads,

- pen attributes for the measurement lines,
- pen attributes for the arrow lines,
- pen attributes for the arrowhead outline,
- text style for the value object,
- the identification of an optional custom value graphics object (must have a text item as the first item).

The measurement object definition *component*, described later, contains the specifications for the desired type of measurement object and its configuration. The pen attributes and the text style are defined by the pen attributes and text style *components*, respectively, described previously. If the identification of a custom value graphics object is specified, then that object can be used as the measurement value instead of the default one. This allows the measurement value to be presented as a textual symbol rather than the actual measurement.

1.3.19 Measurement Object Definition Component

The measurement object definition *component* represents the configuration of the associated measurement object. A measurement object definition *component* is a child of a drawing plane *component*. More than one measurement object definition *component* instance can exist for the same drawing plane *component* instance. The same measurement object definition *component* can be specified for more than one measurement object *component*. Alternatively, a measurement object definition *component* instance can be created within the *initialiser procedure* of a measurement object *component* instance. In this case, the measurement object definition *component* will be a child *component* of the drawing plane *component*, rather than the *component* under which it was created, and can be used with other measurement objects.

A measurement object definition *component* is specified by:

- the measurement type (standard, angle, radial, or right-angle),
- an indication of which measurement lines and arrows are visible, whether the arrowheads are filled, and whether to show the measurement in degrees rather than radians for angle measurements,
- the shape of the arrowheads (custom, standard, circle ●, slash \, simple >, cup ☐, or wide ►),
- the distance before the start of the measurement line, arrow line, or size of the right-angle symbol (⊥),
- the length of the measurement lines beyond the arrowhead tip,
- the perpendicular distance of the value object from the centre point of the arrow line,
- the longitudinal distance of the value object from the centre point of the arrow line,
- whether the arrow line is between or on the outside of the measurement lines,
- the length of the outside arrowed lines,
- a poly-curve (sequence of Bezier curve points) defining the outline of the custom arrowheads (for custom arrowheads only),
- the arrow base number for the arrowheads (for custom arrowheads only),
- the anchor point number for the arrowheads (for custom arrowheads only).

A custom arrowhead can be defined by a poly-curve, with the base of the arrowhead to the left and the tip of the arrowhead to the right. An arrowhead is used in the arrow line for standard, angle, and radial measurement types.

1.3.20 Data Pack Component

The data pack *component* represents a place where the programmer can allocate variables for their own use. A data pack *component* is a child of a project *component*, or a child of another data

pack *component*; data pack *component* instances can be nested to any level. More than one data pack *component* instance can exist for the same project *component* or other data pack *component* instances.

There is no visible representation of a data pack *component*.

1.4 The VIS Framework

The VIS framework is implemented at two levels (although, to the programmer, the two levels are seamless). The lower level is designed in the C++ programming language, and retains the details of all the *component* instances except for the project and data pack *component* instances. The higher level is designed in the ETAC programming language, and retains all the parameters of all the *component* instances. The higher level also defines all the member functions of all the *components*, and all the VIS functions and commands called by the programmer. Those VIS functions can internally call lower-level functions as required. Note that this arrangement of the two levels may change with a future release of VIS.

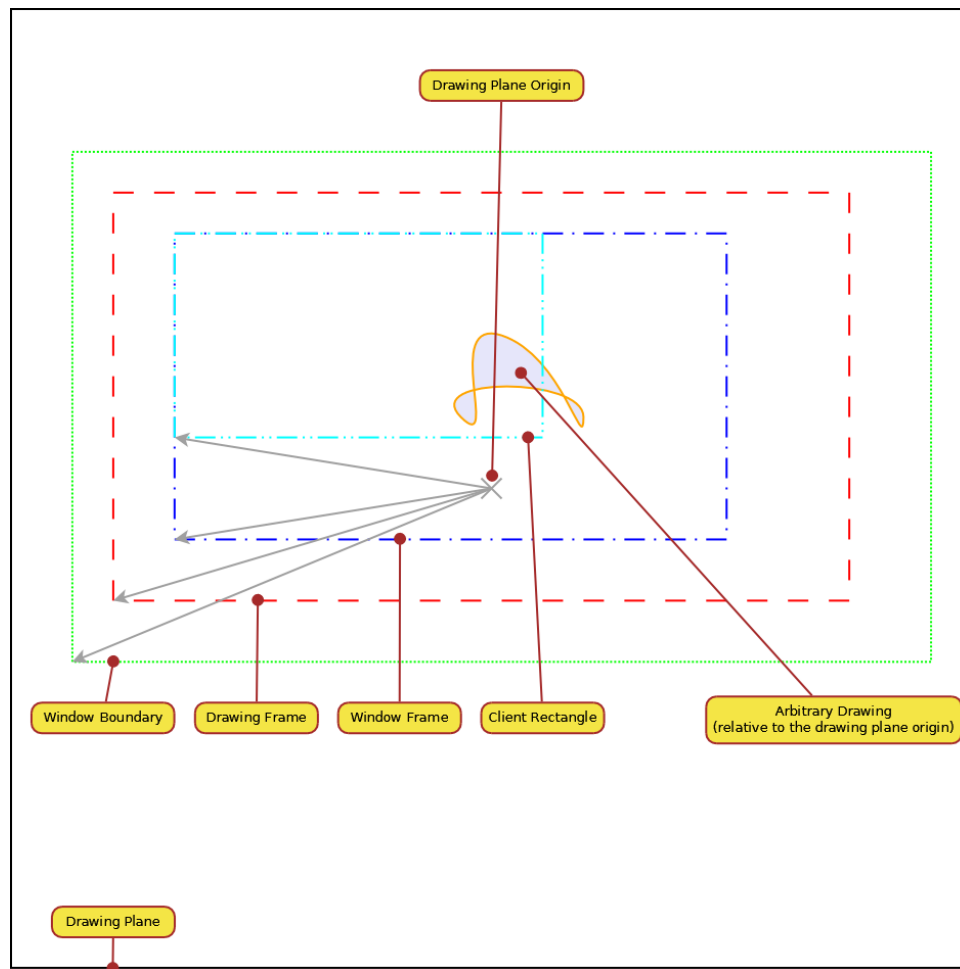
In VIS, there is this concept of the *current* project, *current drawing frame*, and *current VIEW*. The lower-level part of the framework maintains a list of all *VIEW*s in an application program, along with their corresponding *drawing planes*. When a user activates a *VIEW*, that *VIEW* and its corresponding *drawing plane*, along with the appropriate project, are automatically made *current* so that user events are processed appropriately. The VIS functions called by the programmer at the higher-level part of the framework must also correspond to the *current VIEW* and *current drawing frame* existing at the lower-level, and also to the corresponding project, for proper operation. This correspondence is achieved by providing the programmer with the ability to access the *component* instances corresponding to the *current* project, *current drawing plane*, and *current VIEW*.

1.5 VIS Frames

A graphics application program would be seriously deficient if it were not capable of scrolling and zooming. For a graphics ETAC program, much of the scrolling and zooming action is automatic and implemented within VIS. To facilitate that implementation, VIS uses a concept of rectangular frames defined on the *drawing plane*. The frames limit the extent and definition of scrolling and zooming in an ETAC program via program settings.

There are five frames altogether, which are not all independent of each other. They can be directly altered by the program and indirectly altered by the user, but there are complex interrelations and automatic adjustments of frames as described in the following subsections.

A frame has a position and size, both defined in *base units*. The position is always defined as the bottom-left corner point of the frame.



The diagram above illustrates four frames defined within a *drawing plane*. An arbitrary drawing, relative to the *drawing plane* origin at centre, is also shown. The *reference frame* is not shown because it is not really a frame as such. The frames are described below.

1.5.1 Drawing Frame

The whole *drawing plane* is quite large (approximately 89,478,483 *base units* both wide and high under default settings), so a drawing of only a few *base units* in size would barely be visible if the whole *drawing plane* were to be displayed in a *VIEW*. Therefore, the area of the *drawing plane* available for display is limited to a reasonably sized rectangle (the *drawing frame*), involving the *window boundary*, enclosing the drawing. However, a drawing may exist outside the *drawing frame*. The *drawing frame* can also be used as a “page area” for the drawing, and made visible in the *VIEW*.

1.5.2 Window Frame

For a non *elastic window*, the *window frame* width represents the maximum possible width of the client area of a *VIEW* with no scroll bars. For an *elastic window*, the *window frame* width represents the actual current width of the *VIEW*’s client area. The *window frame* width and height ratio is, by default, the same as that of the maximum possible client area of a *VIEW* with a title bar but no scroll bars.

The *VIEW* scale is a positive decimal number defined as the ratio of the *reference frame* width to the *window frame* width. For example, if both widths are identical, then the scale is 1.0 (100%). If the *window frame* becomes twice as wide as the *reference frame*, then the scale becomes 0.5 (50%). The size of the drawing presented in a *VIEW* is proportional to the *VIEW* scale.

If the *window frame* is attempted to be positioned such that the *client rectangle* ends up being outside the *window boundary*, then VIS will automatically adjust the position of the *window frame* so that the *client rectangle* remains within the *window boundary*.

1.5.3 Client Rectangle

Every *VIEW* (operating system window) has a ‘client area’. This is the (typically white) inner rectangular part of the window where a program displays output, such as text and graphics. The title bar, menu bar, sizing borders, and scroll bars are collectively referred to as the window’s ‘non-client area’ (note that the status bar is within the *client area*, not the *non-client area*). The operating system manages most aspects of the *non-client area*; an application program, along with VIS, manages the appearance and behaviour of the drawing in the *client area*.

In the context of VIS, a *VIEW*’s *client area* is automatically mapped onto the top-left corner of the *window frame* as an imaginary rectangle called the *client rectangle*. For an *elastic window*, the *client rectangle* always has the same width as that of the *window frame*. For a non *elastic window*, the ratio of the *client rectangle* width to the *window frame* width is the same as the ratio of the *VIEW*’s actual current *client area* width to the maximum possible *client area* width (which is typically the screen width). There is no necessary relation between the *client rectangle* height and *window frame* height. However, the *client rectangle* height is always automatically kept correctly proportioned to its width. For example, for a maximised non *elastic window without* scroll bars, the *client rectangle* width is the same as the *window frame* width. If the *VIEW* is resized so that its actual *client area* width becomes, say, half the maximum possible *client area* width, then the *client rectangle* width will also become half the *window frame* width. The portion of the drawing covered by the *client rectangle* will always appear in the *VIEW*’s *client area*.

1.5.4 Window Boundary

The *window boundary* is a rectangular region in the *drawing plane* within which the *client rectangle* is restricted to move. This means that only the portions of the drawing within the *window boundary* can be displayed in a *VIEW*. No part of the *client rectangle* can ever exist outside the *window boundary*, except if the *VIEW* was created with no *boundary checking* (in which case, only the right or bottom borders of the *client rectangle* can be outside the *window boundary*). Whenever the *client rectangle* cannot fit entirely within the *window boundary* for a *VIEW* created with *boundary checking*, VIS will automatically extend the *window boundary* appropriately to keep the *client rectangle* within the *boundary*.

The *window boundary* is not specified by position and size. Instead, it is specified by positive or negative border offsets relative to each corresponding border of the *drawing frame*. For example, if the four borders of the *window boundary* are set to zero, then the position and size of the *boundary* becomes identical to the position and size of the *drawing frame* (this is the default setting). A positive value for a border offset is defined to be towards the outside of the *drawing frame*, and a negative value is defined to be towards the inside of the *drawing frame* (negative offsets are rarely used). For example, if the top *window boundary* border is set at an offset of three *base units*, then that border will be three *base units* above the top border of the *drawing frame*; if the left *boundary* border is set to two *base units*, then that border will be two *base units* to the left of the left border of the *drawing frame*. If the *drawing frame* changes position or size, then the *window boundary* will change with it to maintain the specified border offsets.

1.5.5 Reference Frame

The *reference frame* serves as a reference for VIS to calibrate the *VIEW* scale. The *VIEW* scale is defined as the ratio of the *reference frame* width to the *window frame* width. If both widths are identical, the scale is 1.0 (100%). The height of the *reference frame* is not currently used in the calculation.

Internally to VIS, only the size of the *reference frame* is stored. The *reference frame* position is a pseudo position used to set the *window frame* position during *VIEW* scale calibration by the ETAC program.

Programming Guide

This chapter is about how a programmer can use VIS in the ETAC programming language to create a graphics application program. The information in this chapter consists mostly of references to other parts of this document. Familiarity with the ETAC programming language described in the document ETACProgLang(Official).pdf is assumed.

2.1 Introduction (Guide)

Source *ETAC text script* files for VIS are distributed with the ETAC system. Those files use variable names and techniques that are defined for VIS only. The programmer must not use variable names or techniques contrary to those specified in this document for programming, even if those names and techniques result in a successfully operating program. Also, the VIS source files are considered to be system code, and are available for informational purposes only — the programmer must not alter system code.

2.2 Preliminaries for Executing VIS Code

Every ETAC application program requires a *loader script*. In addition, ETAC programs using VIS require the file visETAC.etac or one of visETAC.btac or visETAC-D.btac. The said visETAC files are considered to be part of the *loader script*, and must be in the same directory as the *loader script*, RunETAC.etac or RunETAC.btac.

When the *loader script* is executing, by default, it will also check for the existence of visETAC.etac in the same directory as the *loader script*. If visETAC.etac exists, it is executed, loading the VIS environment. If visETAC.etac is not found, then, if RunETAC.exe is executing with the non-debug option, the existence of visETAC.btac is checked and executed, otherwise (if RunETAC.exe is executing with the debug option) the existence of visETAC-D.btac is checked and executed. If none of visETAC.etac, visETAC.btac, or visETAC-D.btac exists in the *loader script* director, then RunETAC.exe will terminate with an error message.

visETAC.btac is visETAC.etac compiled without the debug option, and visETAC-D.btac is visETAC.etac compiled with the debug option. With the debug option, visETAC.etac (or visETAC-D.btac) contains code that does extra checking for correct arguments for certain VIS functions (resulting in higher overheads).

VIS can be prevented from being loaded by setting the NO_VIS option for RunETAC.exe. This option prevents any of the said visETAC files from being executed. In this case, no VIS commands or function can be used in the ETAC application program. With the default setup of ETAC, RunETAC.exe implicitly runs with the VIS option, whether or not the application program uses graphics.

An ETAC program using VIS must include the file VISGlobal.PTAC, which contains all the pre-processor definitions required for the VIS commands and functions. (In addition, TACGlobal.PTAC is required for all ETAC programs as usual.)

Before a VIS ETAC program begins, the default empty project named “@VIS@” will have already been created but not made *active*. The default project can be accessed via the **vis_Project()** function. The application program can create appropriate *components* under the default project or a new project as desired (see [2.6_Creating Components](#)).

In order for VIS to receive and process user events, the function `vis_WaitForEvents()` needs to be called.

2.3 Default Features of VIS

One of the design principles of VIS is to reduce coding done by the programmer for typical situations. So, for example, a *VIEW* can be created simply by the code fragment `vis_VIEW("") {};`. There is no mention of the *VIEW* features in the code fragment; the created *VIEW* will have default features. Of course, the programmer can set desired features. All *components* have default features but only some of them will be mentioned here.

By default, the *drawing frame* and the *window frame* are the same size, which correspond to the maximum size of the client area on the screen. They are both centred on the *drawing plane* origin. The *VIEW* scale is 100%. The *VIEW* position is at the top left of the screen. The *VIEW* is a visible, maximised, overlapped window with a sizing frame and vertical and horizontal scroll bars (visible only when required) with the title “RunETAC”. The default *VIEW* is an *elastic window*, meaning that it keeps the same visible drawing in the *VIEW* regardless of the *VIEW*’s size (the drawing stretches or shrinks with the *VIEW* size).

The arrow and page keys, and the mouse wheel, by default, scroll the drawing automatically without any programmer coding.

The origin of graphics objects is at the *drawing plane* origin, and the objects are visible, by default. Text items and formatted text items, by default, are displayed at the parent graphics object’s origin.

Part or all of the drawing on the *drawing plane* is rendered into the *VIEW* automatically since VIS retains the details of all *components*.

Of course, the defaults mentioned above can be overwritten or cancelled by the programmer.

2.4 Drawing Plane Features

A VIS ETAC program can create any number of *drawing planes*. Each *drawing plane* has a single *drawing frame* which defines the limits of the drawing that can be presented in a *VIEW*. The *drawing frame* position and size can be set and read by an ETAC program. The functions `@dpSetDrawFrame()`, `@dpExtDFrameBord()`, and `@dpFitDrawFrame()` set the *drawing frame*. The `@dpGetDrawFramePos()` function gets the position, and the `@dpGetDrawFrameDim()` function gets the size of the *drawing frame*.

2.4.1 Coordinate System

Graphics objects can only be defined on a *drawing plane* which represents a Cartesian coordinate system with the positive X values to the right, and the positive Y values upwards (not downwards), and with the absolute origin (coordinate (0, 0)) at the centre. Measurements are in *base units* not pixels.

2.4.2 Distance Units

Only one type of distance unit is used with a *drawing plane*: the *base unit* (a decimal number). The accuracy of a distance measurement depends on the *drawing plane* density as defined by the programmer via the `@dpSetDrawFrame()` function (which also sets the location and size of the *drawing frame*).

2.4.3 Background Colour

A *drawing plane* also has a background colour (set when the *drawing plane* is created or at any other time) which is used as the background colour of all *VIEWs* of that *drawing plane*. The `@dpBkClr` variable sets the background colour of a *drawing plane*.

2.4.4 Mouse Cursor Image

A default mouse cursor image can be set on a *drawing plane* by a VIS program at any time via the `@dpSetCursor()` function. The default cursor is displayed on all *VIEWs* of that *drawing plane* whenever the mouse enters those *VIEWs*. Different *drawing planes* can have their own default cursor images.

2.4.5 Drawing Frame Visibility

The borders of the *drawing frame* are not visible to the user by default. However, a VIS program can set and change the line (pen) and fill (brush) attributes of a *drawing frame* so that it becomes visible in the *VIEW* via `@dpDFramePen` and `@dpDFrameBrush` variables. The *drawing frame* would thus appear as though it were a rectangle graphics object even though it is not (a *drawing frame*'s border and brush are rendered via a different system than are graphics objects). A VIS program can use the `@dpFitDrawFrame()` function to adjust the size of the *drawing frame* to enclose all displayable graphics objects.

2.4.6 Smoothing

By default, VIS renders a drawing onto a *VIEW* in a very fast and efficient way that does not smooth the edges of the drawing's shapes. Unfortunately, the system that VIS uses to render graphics (GDI) cannot perform anti-aliasing (smoothing) on the drawing, so VIS needs to do that itself. VIS uses a technique called *supersampling* to perform smoothing (not actually anti-alias). This technique involves rendering the drawing into a larger bitmap then reducing the size of the bitmap, based on some kind of smoothing algorithm, to the proper size before transferring the bitmap onto the *VIEW*.

VIS uses two smoothing algorithms to perform smoothing. The first algorithm reduces the size of a bitmap using a mode called "HALFTONE" by the operating system's GDI (Graphics Device Interface) documentation. The explanation of HALFTONE is not clearly defined in the GDI documentation, but appears to be very similar to a bi-linear reduction followed by slight sharpening. The second algorithm, not available in GIS, is a high quality bi-cubic reduction as defined by GDI Plus (GDI+). Neither algorithm is ideal for smoothing a drawing; sometimes the algorithms produce unsightly artefacts. Another disadvantage of *supersampling* is that line and fill patterns become tighter, making them appear different than intended. A second (and probably worse) disadvantage of *supersampling* is that it is extremely slow compared with no smoothing.

There are six levels of smoothing in VIS, plus the default non-smoothing level (level 0). Levels 1, 2, and 3 use the first algorithm with a *supersampling* size of two, three, and four times, respectively. Levels 4, 5, and 6 use the second algorithm also with a *supersampling* size of two, three, and four times, respectively. A higher level smoothing does not necessarily mean better smoothing than a lower level one; a VIS programmer will need to try out all levels to determine the best one for the intended application. The `@dpSetSmooth()` function sets the smoothing level for the *VIEWs* of a *drawing plane*. The `@dpSmoothShot()` function can temporarily alter the smoothing level.

2.5 Window Features

A user-interface window in VIS is called a *VIEW* (Visual Interactive ETAC Window). The client area of a *VIEW* displays a portion of the drawing existing in the associated *drawing plane* as

determined by the VIS program or user. The client area is automatically mapped onto the top-left location of a *window frame* rectangle which has a position that can be altered by the program or user. As the *window frame* is moved (by the program or user), the mapped area, called the *client rectangle*, moves along with it, thus showing a different portion of the drawing. The *client rectangle* can exist only within a margin around the *drawing frame* forming a rectangle called the *window boundary*. If the mapped area becomes larger than the *window boundary*, VIS will automatically, and temporarily, expand the *window boundary* so that the area remains within the *boundary* (applicable only if *boundary checking* is in effect, set by the `@vNoBoundChk()` function).

The programmer sets the visible *VIEW* (window) features, such as the title (`@vTitle`), position (`@vXPos`, `@vYPos`), size (`@vWidth`, `@vHeight`), and style (`@vStyle`, `@vExStyle`), via the *VIEW component*. The *VIEW* position and size are set in *logical picas*, not pixels. The programmer can use any of the styles and extended styles, except for a few that make no sense, available to the operating system.

The `@vGetWinClientPos()` function gets the position, and the `@vGetWinClientDim()` function gets the size of the *client rectangle*.

The four *window boundary* border offsets can be set and read by an ETAC program. The `@vSetWinBound()` function sets the four border offsets of the *window boundary*. The `@vGetWinBoundPos()` function gets the position, and the `@vGetWinBoundDim()` function gets the size of the *window boundary*.

By default, the *drawing frame* and *window frame* are both the same size as the maximum *VIEW* client area mapped onto the *drawing plane*. The mapping is one *logical pica* (client area) to one *base unit* (*drawing plane*). That situation defines a *VIEW* scale of 100%, which can be altered via the `@vRepScale()` function or the user via *zoom dragging*.

2.5.1 Co-VIEWS

A *drawing plane* can be associated with more than one *VIEW*; each such *VIEW* is called a *co-VIEW*. Each *co-VIEW* has its own window features, *window frame*, and *window boundary*. This implies that different portions of the drawing can be viewed, scrolled, and zoomed independently by the user (and VIS program). In addition, each *co-VIEW* can have its own menu system and status bar. *Co-VIEWS* are created simply by creating more than one *VIEW* for the same *drawing plane*.

2.5.2 Scrolling

When a user zooms into a drawing, other parts of that drawing may need to be scrolled to come into view. This can be achieved by moving the scroll bars to the desired position. However, sometimes it is more convenient for the user to drag a mouse button to scroll the drawing. But that traditionally requires some programming by the programmer to redraw the drawing at the new desired position based on the mouse movement.

An autonomous feature of VIS is to simply specify an *image dragging* flag (`@vFlags`) to a *VIEW component* instance, causing the drawing to be scrolled by the user dragging the mouse, using the right (menu) button, to the desired position; no other programming is necessary. If *zoom dragging* is also in effect, then the right mouse button must be dragged horizontally at first then in any direction for scrolling. VIS takes care of the interactions between the mouse movement and the drawing during the dragging operation.

The *window frame* position can be altered by an ETAC program via the `@vMoveWinFrame()` function. Moving the *window frame* scrolls the drawing in the *VIEW*. The *window frame* position can also be altered by the user scrolling the *VIEW*'s client area via the scroll bars and up and down arrow keys. Whether a *VIEW* has scroll bars is determined by the `@vFlags` variable.

2.5.3 Zooming

Parts of large drawings may need to be zoomed in for accurate modifications. Traditional graphics programs achieve zooming by having a control button to increase or decrease the *VIEW* scale, or they use menu items for the user to set the desired scale. In any case, some programming is required by the programmer to make the drawing appear larger or smaller in the window.

An autonomous feature of VIS is to simply specify a *zoom dragging* flag (`@vFlags`) to a *VIEW component*, causing the drawing to be zoomed by the user initially dragging the mouse, using the right mouse button, vertically; no other programming is necessary. VIS takes care of the interactions between the mouse movement and the drawing during the zooming operation. Initially dragging the mouse upwards zooms into the drawing, and initially dragging it downwards zooms out of the drawing. The *VIEW* scale is automatically adjusted to be roughly proportional to the distance between the current mouse cursor position and its position when dragging began. That proportionality can be changed by the programmer so that a drawing zooms in faster or slower for the same mouse movement (`@dpSetScaleMult()`).

There is a limit to the maximum and minimum *VIEW* scale range activated by *zoom dragging*. Under default conditions, the minimum *VIEW* scale is 100% and the maximum is 300%. The programmer can alter the range via the `@vMinWinScale` and `@vMaxWinScale` variables. The range limit does not apply when the program explicitly changes the *VIEW* scale through the appropriate function (`@vRepScale()`).

The position or size of the *window frame* is automatically altered either by the user *zoom dragging* or via the function `@vRepScale()`, both of which change the *VIEW* scale.

The *window frame* can be made identical to the *reference frame* (making the *VIEW* scale 1.0) via the function `@vCalibrateScale()`.

2.5.4 Menu

As with most windowing systems, a *VIEW* can have one menu bar (`@vMenuID`). However, the operating system does not allow child windows to contain a menu bar. Any *VIEW* can show a context (or “popup”) menu that can be made to appear anywhere on the screen via the `@vShowPopup()` function. A menu can have the usual features of checkmarks, radio buttons (bullet marks), separators, highlighting, and disability. A bitmap can be shown next to menu items. However, menu items themselves cannot be bitmaps. Menu items can be added (`@mPutMItem()`) and removed (`@mRemoveMItem()`) at any time. A menu can be loaded (`vis_NewMenuData()`) from a resource file (*.res) or programmed via *ETAC code* (`@MENU`).

2.5.5 Status Bar

Any *VIEW* can have one status bar which can be positioned at the top or bottom, or even on the left or right side of the *VIEW*. A status bar always has two modes — a “simple mode” and a “parts mode” — but only one mode can be displayed at a time. The simple mode can accept only one line of text and/or bitmap; the parts mode can have more than one partition, each of which can accept one line of text and/or bitmap. The borders of each partition can have their own limited style. The minimum height, colour, border style, and text style of a status bar can be set and changed by the ETAC program. And, of course, a status bar can be destroyed and recreated anew (perhaps with new features). (Refs: `@vNewSBar()`, `@sbSetMinHeight()`, `@sbRepBkColr()`, `@sbRepText()`, `@sbDestroySBar()`)

2.6 Creating Components

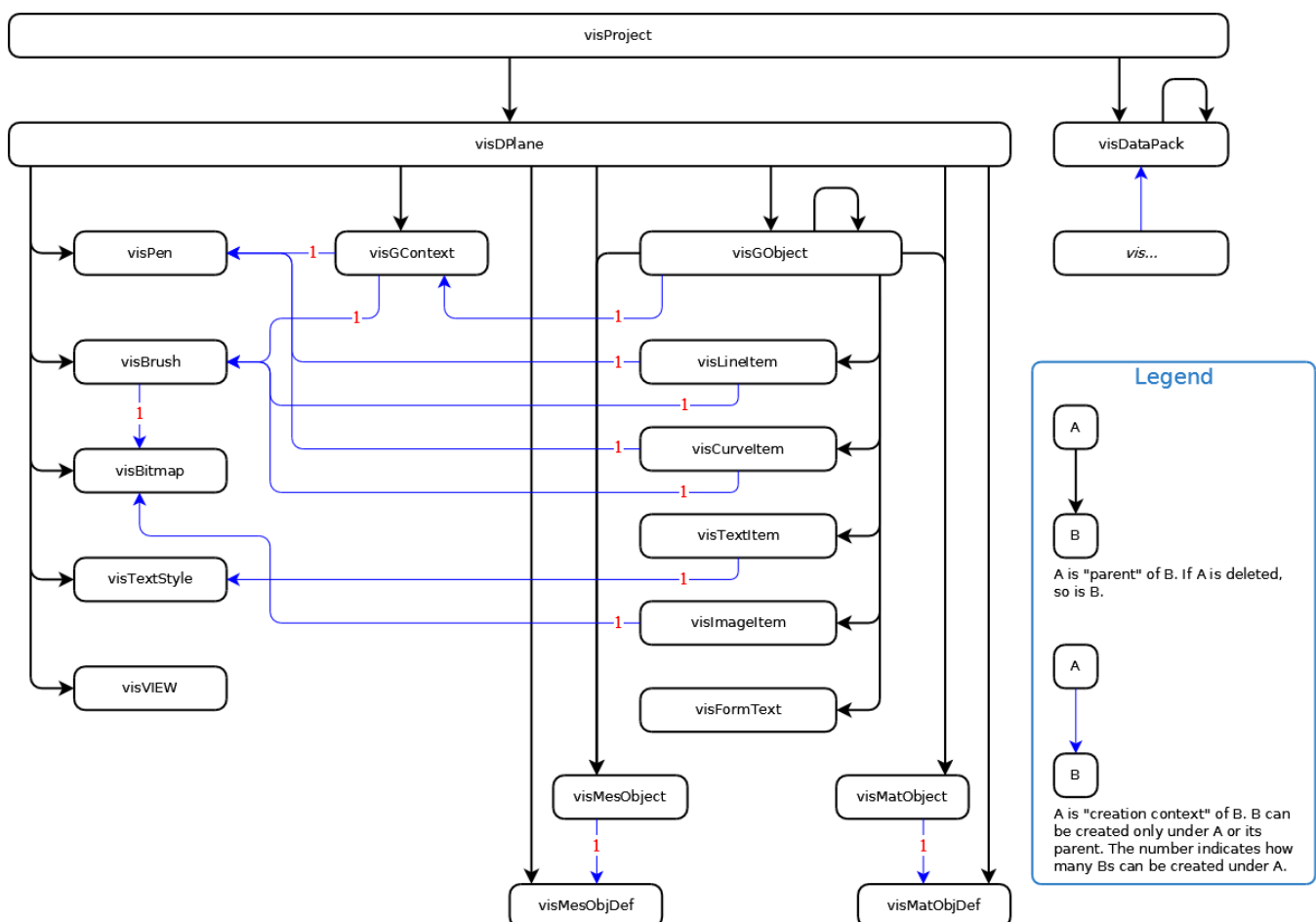
When a *component* instance is being created, it must have an appropriate *creation context* or *active parent component*. For example, a pen attributes *component* can only be created within the

initialiser procedure of a graphics context, line item, or curve item *component*, or when a drawing plane *component* is *active*. Also, the desired project *component* must be *active* when a *component* is being created so that it can be (automatically) registered with that project *component*. A new item *component* instance can be added to a graphics *component* instance only when the graphics *component* itself is in the process of being created, but not when it is only being referenced. A data pack *component* can be created as a child *component* of any *active component*. Each project *component* instance maintains a list of all the *components* created while that *component* instance was *active*.

A *component* instance must NEVER be duplicated implicitly (for example, using the ETAC command `dupl` or `@`). Doing so will have unpredictable consequences. A *component* instance can be duplicated explicitly by copying its features to the parameters of a new *component* instance of the same type.

The following diagram shows all the VIS *components* along with their parent *components* (thick black arrows) and *creation contexts* (thin blue arrows). The arrows point towards the child *component*. If a parent *component* instance gets deleted, then all its children *component* instances automatically get deleted as well. The relevant parent *component* instance of a *component* instance being created must be *active*.

Component Creation Hierarchy



The following code fragment example illustrates how a pen attributes *component* instance can be created. The parent *component* of a pen attributes *component* is a drawing plane *component*, so the desired drawing plane *component* instance must be made *active* (either by creating a new drawing plane *component* via the `via_DPlane()` function or referencing an existing one via the `@VIS()` or `@VISX()` functions).


```

[* An existing drawing plane instance is being accessed in this case. *]
[* 1*]  @VIS("MyDPlane").
        {
            [* The drawing plane instance is active (and current). *]
            [* A new graphics object instance is required for a new line item
               instance. *]
[* 2*]  vis_GObject("")
        {
            [* Creation context for the pen attributes. *]
[* 3*]  vis_LineItem("")
        {
            [* The pen can only be created in the initialiser procedure of the
               creation context. *]
[* 4*]  vis_Pen("")
        {
            [* Creation details. *]
        };
    };
};
};

```

At line **1** above, an existing drawing plane *component* instance is being accessed, which will be the parent *component* of a new graphics object (line **2**) and pen attributes *component* (line **4**) instances. A new drawing plane *component* instance could have been created instead. In this example, the pen attributes *component* instance is required only for the line item *component* instance (line **3**), therefore it is created in the *initialiser procedure* of that line item instance which is the *creation context* of the pen attributes. Also, a line item *component* instance can only be created in the *initialiser procedure* of a graphics object *component* instance, so a new graphics object instance must be created (line **2**). Note that the parent *component* of the pen attributes *component* is the drawing plane *component*, but its *creation context* is the line item *component*.

The following code fragment is an alternative to the code fragment above, having the same effect.

```

@VIS("MyDPlane").
{
    [* The drawing plane instance is active (and current). *]
    [* The pen attributes can be created in the initialiser procedure of the drawing
       plane component. *]
    vis_Pen("MyPen")
    {
        [* Creation details. *]
    };

    [* A new graphics object instance is required for a new line item instance. *]
    vis_GObject("")
    {
        [* Creation of the line item component instance. *]
        vis_LineItem("")
        {
            @liPenID := "MyPen"; [* Pen attributes allocated for the line item. *]
        };
    };
};

```

In this case, the pen attributes *component* instance is created under its parent *component* instance (the drawing plane *component*), and then assigned to the line item *component* instance via the `@liPenID` variable. This method is useful if the pen attributes *component* instance is to be used with other *component* instances requiring the same pen attributes.

In summary, when creating a *component* instance:

- its parent *component* must be *active*,
- the *component* can be created only under its parent *component* or under its *creation context*,

- if the *component* is to be created under its *creation context*, then it can only be created from within the *initialiser procedure* of that *creation context* (except for a data pack *component*, which need not be created from within the *initialiser procedure* of its *creation context*),
- an item *component* instance can only be created from within the *initialiser procedure* of a graphics *component* instance,
- if a graphics object *component* instance is to be created under a parent graphics object *component*, then the instance can only be created from within the *initialiser procedure* of that parent *component*,
- not all member functions of a *component* instance being created can be executed from within the instance's *initialiser procedure*.
- A *component* instance must NEVER be duplicated implicitly.

The type name of a created *component* instance is automatically stored in the @Name member of the *component*'s ETAC data object. The following table shows each *component* type with its corresponding type definition, type name, and prefix. The type definition is a pre-processor name and retrieved via the @visGetType() function. The prefix is used for the member names of the *component*.

Component Types and Prefix

Component	Type Definition	Type Name	Prefix
Project	visC_Project	visProject	prj
Drawing Plane	visC_DPlane	visDPlane	dp
VIEW	visC_VIEW	visVIEW	v
Graphics Object	visC_GObject	visGObject	go
Line Item	visC_LineItem	visLineItem	li
Curve Item	visC_CurveItem	visCurveItem	ci
Text Item	visC_TextItem	visTextItem	ti
Formatted Text Item	visC_FormText	visFormText	fi
Image Item	visC_ImageItem	visImageItem	ii
Graphics Context	visC_GContext	visGContext	gc
Pen Attributes	visC_Pen	visPen	pn
Brush Attributes	visC_Brush	visBrush	br
Bitmap Attributes	visC_Bitmap	visBitmap	bm
Text Style Attributes	visC_TextStyle	visTextStyle	ts
Matrix Object	visC_MatObject	visMatObject	mato
Matrix Object Definition	visC_MatObjDef	visMatObjDef	matd
Measurement Object	visC_MesObject	visMesObject	meso
Measurement Object Definition	visC_MesObjDef	visMesObjDef	mesd
Data Pack	visC_DataPack	visDataPack	dpk

Component instances of the various types are created via functions having the same name as the type name but with an underscore after the vis part. For example, the type name of a text item *component* is visTextItem, therefore the function name creating an instance of that *component* type is **vis_TextItem()**. The *component* creation functions require two arguments, the second

of which is the *initialiser procedure*. The created *component* instance becomes *active* during the execution of the *initialiser procedure*, but not necessarily outside it.

2.7 Component Dependencies

For proper operation, the *component* member functions called in a VIS ETAC program at the higher-level part of the VIS framework must correspond appropriately with the *current VIEW* and *current drawing frame* existing at the lower-level part of the framework, and also correspond appropriately with the *current* project. It is the responsibility of the programmer to ensure that the appropriate project, *drawing frame*, and *VIEW* are *current* before accessing *component* instances or calling member functions of certain *components*, otherwise failure is guaranteed. If VIS is running in debug mode, and the wrong project, *drawing frame*, and *VIEW* are *current* when accessing *component* instances or their member functions, then an error message is issued (in most cases) and the program ends; if VIS is not running in debug mode, then the consequence is unpredictable. (Hint: a VIS ETAC program should always be developed in debug mode for additional error checking.)

When accessing a *component* instance via the `@VIS()` or `@VISX()` function, the *component*'s parent project *component* must be *current* (but not necessarily *active*). This is required so that the *component* instance being accessed can be found in the list of *component* instances owned by the parent *component* instance. A project can be made *current* via the `vis_SetCurProject()` function.

If a drawing plane *component* instance is accessed via the `@VIS()` or `@VISX()` function, then the corresponding *drawing plane* will become temporarily *current* during the execution of the said function's procedure. This allows the member functions of the *component* instance, and of the child *component* instances, to access the correct lower-level structures of the VIS framework. Similarly, a *VIEW* is made *current* via the same functions.

When accessing an instance of a graphics object *component*, a graphics context *component*, all item *components*, all attribute *components*, a matrix *component*, or a measurement *component*, then the parent *drawing plane* of those *components* must be made *current* before the access, if not already *current*. For example, the code fragment to access a graphics object *component* instance called "MyObj" that has a parent drawing plane *component* instance called "MyDPlane" is `@VIS("MyDPlane").{... @VIS("MyObj").{...}; ...};`. Note that `@VIS("MyDPlane").{...}` makes the relevant *drawing plane* temporarily *current* (and *active*) within the procedure, allowing the correct graphics object ("MyObj") to be accessed at the lower-level part of the VIS framework. If the *drawing plane* corresponding to the "MyDPlane" *component* instance is already *current*, then `@VIS("MyDPlane")` need not be called; the code fragment to access "MyObj" would then just be `@VIS("MyObj").{...};`.

When calling certain function members of a *VIEW component* instance, the *VIEW* corresponding to that instance must be *current*. For the *VIEW* to become *current*, the instance's parent *drawing plane* must be *current* when accessing the *VIEW component* instance. If the required *drawing plane* is not *current*, VIS automatically makes it *current* when accessing a *VIEW component* member via the "dot" (member selection) operator. However, if a *VIEW*'s member function is exported via the ETAC `@ExportFnt()` function, then the "dot" operator is not evoked and the relevant *drawing plane* and *VIEW* will not automatically become *current* when the exported function is called. Variable members of a *VIEW component* instance do not need the corresponding *VIEW* to be *current*.

The restrictions above may seem complicated, but most code is executed under an event handler function, in which case the appropriate project, *VIEW*, and *drawing plane* are automatically made *current* by VIS (as determined by the *current VIEW* selected by the user). It is only when accessing a *different* project, *VIEW*, or *drawing plane* within an event handler do the restrictions above need to be observed.

The functions `vis_Project()`, `vis_DPlane()`, or `vis_VIEW()` can be used with a NULL stack object first argument to make the *component* instance of the corresponding *current* project, *current drawing plane*, or *current VIEW* temporarily *active* during the execution of the said function's procedure. For example, to access the *component* instance associated with the *current drawing plane*, the programmer calls `vis_DPlane(?) {...}`. Note that, with the NULL argument, a new drawing plane *component* instance is not created, but an existing one corresponding to the *current drawing plane* is made *active*. Therefore, the code within the procedure (represented by the ellipsis) will execute with the drawing plane *component* instance of the *current drawing plane* being *active*. A similar situation applies for project and VIEW *component* instances.

2.8 Graphics Objects

See [1.3.5 Graphics Object Component](#) for general information on graphics object *components*.

A graphics object *component* instance can be created via the `vis_GObject()` function. When a graphics object *component* instance is created, `@goDisplay` is true by default, causing the graphics object to be visible. If `@goDisplay` is set to false in the *initialiser procedure*, the graphics object *component* instance is still created, but the graphics object itself will be invisible and not take up space in the *drawing plane*. The functions `@goRelocateTo(0)` and `@goShow(true)` are called to later make the graphics object itself visible (and take up space in the *drawing plane*). (Refs: `@goRelocateTo()`, `@goShow()`)

A group graphics object *component* instance is created by creating at least one graphics object *component* instance within the *initialiser procedure* of the first instance. Alternatively, a group graphics object *component* instance can be created by setting `:!GO_GROUP_OBJ:` in `@goFlags` within the *initialiser procedure*. Once created, a group graphics object cannot be converted to a regular graphics object, and vice versa. See [2.8.4 Graphics Object Groups](#) for further details.

The function `@vZoomToAll()` zooms a *VIEW* to all graphics objects defined on the corresponding *drawing plane*. However, any graphics object *component* instance that was set with the `:!GO_SYSTEM:` flag (via `@goFlags` or `@goRepFlags()`), will be excluded from zoomed graphics objects. Likewise, the `@dpFitDrawFrame()` function excludes graphics objects with the `:!GO_SYSTEM:` flag set.

To allow automatic highlighting of a graphics object when the user selects it, a graphics context needs to be created for the graphics object *component* instance either by setting `@goContextID` to a pre-created graphics context *component* instance or by creating one explicitly (`vis_GContext()`), both within the *initialiser procedure* of the graphics object.

A graphics object can only be resized directly by a user; it cannot be directly resized programmatically. However, a graphics object can effectively be resized programmatically by modifying the object's graphics items or sub-objects appropriately.

Graphics objects can be copied to the clipboard as an enhanced metafile, or written to a disk file as *.emf, *.bmp, *.jpg, *.gif, *.png, or *.tif files via the `vis_WriteGObjs()` function.

A graphics object *component* instance can be removed (not deleted) from the internal list of displayed graphics objects ('display list'), and transferred to another internal list for temporary storage ('storage list'). While the object instance is on the *storage list*, it is not displayed on the screen and does not take up space in the *drawing plane* (it is as though the graphics object were non-existent). The object instance can later be transferred back to the *drawing plane* to be displayed as usual. The function `@goRelocateTo()` can relocate an object as a sub-object of a group object, or to the front of the object *display list*, or to the object *storage list*.

A graphics object *component* instance (it could be a sub-object) on the *storage list* can be swapped with another graphics object instance on the *display list* via the `@goReplaceWith()` function.

A *CID* sequence of the sub-object *component* instances of a displayed graphics object *component* instance can be obtained via the `@goGetSubObjIDs()` function. The position of a graphics object in relation to its sibling objects can be changed via the `@goReorder()` function.

A *CID* sequence of all the object *component* instances of the graphics objects being displayed (ie: on the *display list*) can be obtained via the `@dpGetDispObjIDs()` function.

The bounding box of a graphics object is obtained via the `@goGetBoundingBox()` function which returns a TAC sequence containing the bottom-left and top-right coordinates of the bounding box in *base units*.

(Refs: [3.4.8_Graphics Object](#) ▪ [3.4.2_Component Member Summary](#).)

2.8.1 Graphics Items

There are five types of graphics item *components*: [Line Item](#), [Curve Item](#), [Text Item](#), [Formatted Text Item](#), and [Image Item](#). A graphics item *component* instance must have a graphics object *component* instance as parent; the item instance can only be selected via that parent. A graphics object *component* instance can contain (be a parent of) any number (including zero) of graphics item *component* instances of any type; the graphics items cannot be selected individually.

The features of a graphics item *component* instance are specified by the instance itself (intrinsic features) and an appropriate attributes *component* instance. The parent of an attributes *component* is always a drawing plane *component*. However, an attributes *component* instance can be created within the *initialiser procedure* of an appropriate graphics item *component* instance or within the *initialiser procedure* of a drawing plane *component* instance, as desired by the programmer. The same attributes *component* instance can be shared among several graphics item *component* instances regardless of where it was created; features that are changed for that attributes instance will therefore affect all the graphic item instances that share it. For example, if the colour of a pen attributes *component* instance that is shared among different line and curve item *component* instances (which can be children of different graphics *component* instances) changes, then the corresponding displayed curves of all those item instances will display the new colour. The attributes *component* instance of an item *component* instance can be replaced at any time via a member function of the item *component* (see the references in the table below for the details of the members of each graphics item *component* type).

The following table summarises the attributes of each graphics item *component* type. Each item *component* also has additional features set by flags which are not shown below.

Item Type	Attributes Component	Intrinsic Features
Line Item	Pen Attributes ▪ Brush Attributes	Point sequence ▪ Scale range
Curve Item	Pen Attributes ▪ Brush Attributes	Point sequence ▪ Scale range
Text Item	Text Style Attributes	Text string ▪ Bounding box ▪ Scale range
Formatted Text Item	<i>none</i>	Text string ▪ Bounding box ▪ Background colour ▪ Scale range
Image Item	Bitmap Attributes	Bounding box ▪ Scale range

The following table summarises the features of each attributes *component* type.

<u>Attributes Type</u>	<u>Features</u>
<u>Pen Attributes</u>	Pen style ▪ Pen width ▪ Foreground colour ▪ Background colour ▪ Background mode ▪ Drawing mode
<u>Brush Attributes</u>	Brush style ▪ Hatch style ▪ Foreground colour ▪ Background colour ▪ Background mode ▪ Drawing mode ▪ <u>Bitmap Attributes</u>
<u>Text Style Attributes</u>	Foreground colour ▪ Background colour ▪ Background mode ▪ Font style (height, width, weight, italic, underline, strikethrough) ▪ Character set ▪ Pitch and family ▪ Facename
<u>Bitmap Attributes</u>	Image source ▪ Raster operation mode ▪ Stretch mode

The attributes and features above can be replaced for the appropriate *component* instance at any time within a program.

2.8.2 Fixed Size Items

In a graphics application program, there may be parts of a drawing that need to be displayed at a constant size regardless of the *VIEW* scale. For example, a selection box typically has little square boxes on the perimeter called “handles” which are displayed at a constant size. In traditional graphics programs, some programming is required by the programmer to keep the handle size constant as the user zooms in and out of the drawing.

An autonomous feature of VIS is to simply create fixed-size graphics item *components* which display images retaining their visible size when the *VIEW* scale is altered; no other programming is necessary. Fixed-size graphics item *component* instances are created in the following way. If a graphics item *component* is created by the **vis_***X*() function, where *X* is the name of the item, then a fixed-size version of that *component* is created by the **vis_Fix***X*() function. For example, a normal curve item *component* instance is created by the **vis_CurveItem**() function, but a fixed-size curve item *component* instance is created by the **vis_FixCurveItem**() function. The positions and dimensions specified for a fixed-size graphics item *component* instance are in *screen picas* not *base units*. However, under default conditions, there is no difference between the two. When the scale of the *VIEW* containing a fixed-size graphics item is altered, that item will be displayed in its original size (in *screen picas*), but not necessarily at its original relative position.

Once created, a fixed-sized graphics item *component* instance cannot be converted to a (corresponding) regular graphics item *component* instance, and vice versa.

2.8.3 Scale Range Visibility

In some graphics application programs, parts of a diagram need to be visible only within a given window scale range. In traditional graphics programs, some programming is required by the programmer to achieve such a feature.

An autonomous feature of VIS is for the programmer to specify a lower and upper *VIEW* scale value for graphics item *component* instances, which makes those items visible only within the specified *VIEW* scale; no other programming is necessary. A graphics item *component* includes two parameters (ie: member variables) to specify the *VIEW* scale range. The **@xLowerScale** parameter contains the lower *VIEW* scale value below which the item becomes invisible, and the **@xUpperScale** parameter contains the upper *VIEW* scale value above or at which the item becomes invisible; *x* is the item type name prefix. For example, if the **@ciLowerScale** parameter of a curve item *component* instance is set to 1.5, and the **@ciUpperScale** parameter is set to 2.5, then the item will be visible only within the *VIEW* scale 1.5 or above and below 2.5.

2.8.4 Graphics Object Groups

Graphics object *component* instances need to contain item *component* instances to display anything. However, they can also contain other graphics object *component* instances in addition (or exclusively). In other words, graphics object *component* instances can be nested to any level. When a graphics object has sub-objects, it is called a ‘group object’. When a *group object* moves, its sub-objects move along with it. When a *group object* is deleted, its sub-objects are automatically deleted as well.

VIS has two kinds of *group objects*:

- **Regular Group Object:** The sub-objects of this kind of *group object* can be selected, highlighted, and moved by the user or program, and resized individually by the user.
- **Unit Group Object:** The sub-objects of this kind of *group object* act as a single unit together with the parent object, and cannot be selected, highlighted, moved, or resized individually by the user. However, they can be moved, but not highlighted nor selected, by the program.

There are two ways to create a *group object*:

- When a graphics object *component* instance is created from within the *initialiser procedure* of another graphics *component* instance, that other instance is automatically created as a *group object*.
- A graphics object *component* instance is created as a *group object* when it has no sub-object instances, but has the `!GO_GROUP_OBJ:` flag set in `@goFlags` when it is being created (ie: within the *initialiser procedure*).

A unit *group object* is created by creating a *group object* and setting the `!GGO_UNIT_OBJ:` flag in `@goGrpFlags` in the *group object* during creation or at any time afterwards. Once created, a *group object* cannot be converted to a non-*group object*, and vice versa. However, a unit *group object* can be converted to a regular *group object* at any time by removing the `!GGO_UNIT_OBJ:` flag via the `@goRepGrpFlags()` function.

The following code fragment illustrates how *group objects* can be created.

```
[* 1*] vis_GObject("GrpObj1") [* Group object. *]
      {
[* 2*]   vis_GObject("Obj1") [* Regular object. *]
      {
          [* Creation details. *]
      };
  };

[* 3*] vis_GObject("GrpObj2") [* Group object. *]
      {
[* 4*]   @goFlags := :!GO_GROUP_OBJ;;
  };

[* 5*] vis_GObject("Obj2") [* Regular object. *]
      {
          [* Creation details. *]
      };

      [* Relocate "Obj2" to "GrpObj2" as a sub-object. *]
[* 6*] void @VIS("Obj2").@goRelocateTo("GrpObj2");

      [* Convert "GrpObj1" to a unit group object. *]
[* 7*] void @VIS("GrpObj1").@goRepGrpFlags(:!GGO_UNIT_OBJ: true);
```

Line 1 creates a *group object* because a sub-object is created (line 2) within the *initialiser procedure* of that *group object*. Line 3 creates a *group object* because `@goFlags` explicitly sets it as a group object (line 4). Line 5 creates a regular graphics object to be later relocated (line 6) to the *group object* (at line 3) as a sub-object. Line 7 converts the first *group object* (at line 1) to a

unit *group object*. Note that the first *group object* (at line 1) could have been created as a unit *group object* directly by setting the `:!GGO_UNIT_OBJ:` flag in `@goGrpFlags` instead of the code at line 7.

A *group object* and any of its sub-objects cannot both simultaneously be in the selected state; the sub-objects of a unit *group object* cannot be selected at all.

The images of sub-object *component* instances are drawn in the order that they are specified; the first ones specified are drawn first. An ETAC program can change the order that they are drawn at any time via the `@goReorder()` function.

When the user resizes a *group object*, the sub-objects are affected in one of a number of various ways as determined by the program. A sub-object can be resized proportionally from its border or its origin, or only its origin is stretched proportionally. If the `:!GO_NO_RESIZE:` flag is set in `@goFlags` of a sub-object, then that sub-object is not resized along with the parent *group object* (otherwise the sub-object is resized by the same proportion as its parent *group object*). If the `:!GGO_KEEP_PROP:` flag is set in `@goGrpFlags` of the *group object*, then all the sub-object origins will stretch or shrink proportionally with the *group object* (otherwise the sub-object origins will be moved such that one or two adjacent sides of the sub-objects will be fixed in position as for the parent *group object*). Note that the two flags provide for four options.

The position of a sub-object is not defined relative to the sub-object's parent *group object*. It is defined relative to the *drawing plane* origin.

In practice, regular *group objects* are typically used to temporarily group graphics objects to move or resize them together, while unit *group objects* are typically used to create compound graphics objects as a unit (in this case, the `:!GGO_KEEP_PROP:` flag is typically set in `@goGrpFlags` of the unit *group object*).

2.9 Vector Graphics

There are two graphics item *components* in VIS that create vector graphics. These are the line item and the curve item *components*. A line item *component* defines joined straight lines, and a curve item *component* defines joined Bezier curves. The shapes displayed by both types of item *components* can be: automatically closed, filled or outlined, have alternative filling (if filled), and keep their shape when the parent object is resized. A pen attributes *component* instance can be associated with each item *component* type specifying the displayed foreground colour, style, line or curve thickness, background colour, background mode, and drawing mode. (Refs: `@liFlags`, `@ciFlags`, [Pen Attributes](#))

The most important property of a line or curve item is the sequence of points specifying the displayed shape. The sequence consists of the coordinates of points in the form $[x_1, y_1, x_2, y_2, \dots]$. In other words, the sequence is a flat sequence of coordinates. For a line item, the points represent the vertices of the line segments; for a curve item, the points represent the control points of Bezier curve segments (the common end points of adjacent segments are not doubled).

2.9.1 Shape Path Functions

An ETAC program can specify the points in the point sequence of a curve (or line) item *component* instance explicitly, or typically, by using a combination of functions from a set of (currently) 42 functions called 'SPF functions'. "SPF" stands for **S**hape **P**ath **F**unctions. (We will ignore the fact that the term *SPF functions* translates directly to "Shape Path Functions functions".) *SPF functions* are mainly used with curve items because they produce the points of a sequence of Bezier curves, but can also be used with line items with the proper manipulations.

SPF functions begin with the prefix `spf_`. Various *SPF functions* can use appropriate parameters in a particular ETAC data object, and possibly passed arguments in addition. The particular data

object is called an “SPF context”, and is always used with *SPF functions*. The @SPF command creates a new temporary *SPF context* initialised by a procedure which calls the desired *SPF functions*. The command then returns the final sequence of points that can be used with curve (and sometimes line) item *component* instances.

An *SPF context* is created with default parameter values. There (currently) 21 parameters beginning with the prefix @spf. All the *SPF functions* use the current path (the @spfPath parameter containing a sequence of points) within the *SPF context*. The most typically used parameters are explained below. Note that not all *SPF functions* use these particular parameters (except for @spfPath).

- @spfPath: Contains the current path used by all *SPF functions* within the *SPF context*. The value of this path is returned by the @SPF command and typically assigned to @ciPoints of a curve item *component* instance.
- @spfOrigin: Contains a sequence containing the X and Y values by which the calculated path is shifted.
- @spfWidth: Contains the width of a shape. For some *SPF functions*, the width can be negative, resulting in some variation or orientation of the shape.
- @spfHeight: Contains the height of a shape. For some *SPF functions*, the height can be negative, resulting in some variation or orientation of the shape.
- @spfApexRadius: Contains the radius of curved apices of some shapes. Some shapes have vertices that can be curved by the specified radius.

A list of the *SPF functions* and their arguments is presented in the SPF functions document VISShapePathFnts.pdf.

2.9.2 Point Set Functions

In addition to the SPF functions, there is a set of (currently) 20 functions called “PTS functions”. “PTS” stands for Point Set. *PTS functions* map a set of points or values to another set of points or values. For example, there is a *PTS function* that calculates the point that is projected a given distance along a line segment defined by two points.

PTS functions begin with the prefix pts_. *PTS functions* convert given points or values to other points or values (each point is a TAC sequence with X and Y values). For example, the pts_Mult() function multiplies a given point by a given value, returning the resultant point. The pts_IsPerp() function determines whether two position vectors (given as two points) are perpendicular (ie: their dot product is zero).

A list of the *PTS functions* and their arguments is presented in the PTS Functions document VISPointSetFnts.pdf.

2.9.3 Custom Arrowheads

Paths with arrowheads typically feature in diagrammatic application programs. In VIS, a programmer-defined arrowhead can optionally be attached to the current path using the SPF spf_Arrowhead() function which returns a new arrowhead path from the given one. The arrowhead is connected (or positioned) to the end of the current path at any one of nine positions relative to the bounding box of the arrowhead, or to the arrowhead’s start or end points. The details for creating arrowhead paths is presented in the SPF functions document VISShapePathFnts.pdf.

2.10 Bitmap Graphics

An image item *component* instance defines the bounding box of a bitmap image. When displayed, the image is stretched or compressed to fit the bounding box. To be displayed, an image item *component* instance requires a bitmap attributes *component* instance which specifies the file path to the BMP bitmap, or a TAC memory object containing the contents of such a file. The bitmap attributes *component* also specifies the raster operation mode (how the bits of the image combine with the background when displayed) and the stretching mode (how the bits of the image are internally modified when the image is stretched or compressed). No other operations can be performed on images themselves; especially, they cannot be rotated. (Refs: [vis_ImageItem\(\)](#), [Bitmap Attributes](#))

Note that the bitmap image data must be in BMP format; other formats, such as PNG, JPEG, GIF, TIFF, etc, are not currently recognised. If a file path is specified, it must be to a BMP file.

The general VIS *ETAC code* to load and display a BMP image is illustrated below.

```
vis_GObject("")
{
    [* Set the position of the graphics object. *]
    @goOriginX := 10.0; @goOriginY := -5.0;

    /** Other item component definitions. **]

    [* Create an image item component instance. *]
    vis_ImageItem("")
    {
        [* Specify the bottom-left and top-right corners of the bounding box to
            contain the image. *]
        @iiBoundingBox := [-5.0, -3.5, 2.0, 4.7]; [* Optional. *]

        [* Create a bitmap attributes component and specify the file path of the
            image. *]
        vis_Bitmap("") {@bmImage := 'MyImages\Image.bmp'}};
    };

    /** Other item component definitions. **]
};
```

If `@iiBoundingBox` is omitted, the image is displayed in its natural size determined by the image data.

2.11 Text Graphics

VIS has two kinds of text item *components*:

- **Regular Text:** All the characters of this kind of text have the same features. The `vis_TextItem()` and `vis_FixTextItem()` functions create text item *component* instances of this kind. The `vis_TextStyle()` function creates a text style attributes *component* to define the character style of the text characters. The text is specified simply by a text string, and is aligned at one of nine positions relative to its bounding box.
- **Formatted Text:** Different groups of characters of this kind of text can have different features. The `vis_FormText()` and `vis_FixFormText()` functions create text item *components* of this kind. The text is specified by a text string coded in RTF (Rich Text Format). RTF is an industry standard, and can be processed by most good word processors. There is no attributes *component* for formatted text since the formatting and attributes are coded in the RTF string itself. Note that only a very small subset of the RTF codes, as determined by the operating system, is recognised by VIS.

Both kinds of text items are necessarily displayed only horizontally and upright. The displayed text of either type of text item cannot be directly edited by the user.

2.11.1 Regular Text Item

The general VIS *ETAC code* to create and display regular text is illustrated below.

```
vis_GObject("")
{
    [* Set the position of the graphics object. *]
    @goOriginX := 10.0; @goOriginY := -5.0;

    /** Other item component definitions. **]

    [* Create a text item component instance. *]
    vis_TextItem("")
    {
        [* Specify some flags (optional). *]
        @tiFlags := (!GTI_WORD_WRAP: &or :!GTI_TJUST:);

        [* Specify the bottom-left and top-right corners of the bounding box to
            contain the text. *]
        @tiBoundingBox := [-5.0, -3.5, 2.0, 4.7]; [* Optional. *]

        [* Specify the text to be displayed. *]
        @tiText := "Hello World!";

        [* Specify the attributes of the text (optional). *]
        vis_TextStyle("")
        {
            @tsHeight := -2.5;
            @tsFaceName := "Times New Roman";
            @tsColr := vis_RGB(200 100 255);
            [* etc *]
        }
    };

    /** Other item component definitions. **]
};
```

By default, the text is centred within the specified bounding box. If **@tiBoundingBox** is omitted, the text is centred at the origin of the parent graphics object. If text wrapping is specified, the text is wrapped to the width of the bounding box, or to the longest word if a bounding box was not specified.

There are a few aspects to regular text. The text can be aligned to one of nine positions within the bounding box which can optionally be automatically fitted to the width and height of the text whenever the text changes.

Bounding Box and Alignment

The bounding box is specified by the lower-left and top-right corner positions, relative to the parent graphics object, within a single TAC sequence. The default bounding box is all zeros. The left edge of the text can be specified to be aligned to the left border of the bounding box, the midline of the text to the midline of the bounding box, and the right edge of the text to the right border of the bounding box. Analogous alignments can be made in the vertical direction. These alignments are made regardless of the bounding box size. With a default bounding box, therefore, if the text is specified to be aligned at the left and bottom borders of the bounding box, the text's bottom-left corner will be at the parent graphics object's origin. The appearance of the text is not constrained to the bounding box by default. (Refs: **@tiBoundingBox**, **@tiFlags**)

Automatic Bounding Box Fitting

The bounding box can be automatically fitted to the displayed text. This is the default. If the bounding box is not automatically fitted (**@tiAutoFitBound** is false) then any part of the text that is outside the specified bounding box is clipped. When the bounding box is fitted, the text is

first aligned to the bounding box specified by the `@tiBoundingBox` parameter, and then the internal bounding box is wrapped around the displayed text (the value of the `@tiBoundingBox` parameter remains unaltered). When the text is later altered while automatic bounding box fitting is set (`@tiAutoFitBound` is true), the text is automatically realigned to the internal bounding box, and then that bounding box is refitted. The programmer can explicitly cause the bounding box to be fitted by calling the `@tiFitBoundingBox()` function.

Text Features

The text features for regular text are specified in a text style attributes *component* instance. A text style attributes *component* specifies the following attributes of all the characters of a piece of regular text: height, width, weight, italic, underline, strikethrough, character set, pitch and family, face name, foreground colour, background colour, and background mode (transparent or opaque). The default character height is one *base unit* (equivalent to 12 points with default screen DPI). A height of zero results in an operating system default text that is fixed in size during zooming. A width of zero results in the correct width for the height. The default face name is an operating system defined face name. The default background colour is white, but the default background mode is transparent. (Ref: [Text Style Attributes](#))

There are a few more characteristics of regular text, such as word wrapping, whether the bounding box is stretched if the user stretches the parent object, whether text line overhang or leading is taken into account, and whether to automatically realign the text with its bounding box each time it is displayed. In addition, the bounding box can be made to automatically fit changed text. (Refs: `@tiFlags`, `@tiAutoFitBound`)

2.11.2 Formatted Text Item

Formatted text is specified by a text string containing some formatting and alignment features encoded within the string itself as RTF codes. There is no bounding box alignment with formatted text as there is with regular text. The text is wrapped, and displayed at the top left of its bounding box which can be made to fit around the text by the programmer via the `@ftFitBoundingBox()` function, or automatically when the text is changed if `@ftAutoFitBound` is true (the default). When displayed, the text outside the bounding box is clipped (this can happen if the bounding box is not fitted). The RTF text can contain embedded OLE objects which cannot be edited by the user.

The general VIS *ETAC code* to create and display formatted text (RTF) is illustrated below.

```
vis_GObject("")
{
    [* Set the position of the graphics object. *]
    @goOriginX := 10.0; @goOriginY := -5.0;

    /** Other item component definitions. **]
```

[continued ...]

```

[* Create a formatted text item component instance. *]
vis_FormText("")
{
    [* Specify the bottom-left and top-right corners of the bounding box to
        contain the text. *]
    @ftBoundingBox := [-5.0, -3.5, 2.0, 4.7];

    [* Specify the text as RTF to be displayed. *]
    @ftText :=
' {\rtf1\ansi\ansicpg1252\deff0\deflang3081{\fonttbl{\f0\fswiss\fcharset0
Arial;}}\viewkind4\uc1\pard\f0\fs24 Hello World!\par}';
};

[** Other item component definitions. **]
};

```

The text is necessarily displayed at the top left of its bounding box, and necessarily wrapped to the bounding box width. If the bounding box is not specified, the text will not appear because the default width of the bounding box is zero.

The RTF text itself can be obtained by first writing the desired text using an RTF word processor, like WordPad.exe, and then saving the file. The file is then reopened again in a regular text editor, like NotePad.exe, to see the actual RTF codes (any NULL bytes at the end of the text need to be removed). The text in the text editor can then be copied to its destination (either an ETAC program, or a file that the program reads).

Technical Note

Unfortunately, there are some display issues with formatted text when the *VIEW* is zoomed in and out. The actual rendering of RTF text is controlled by the operating system, not by VIS. It is a “feature” of the operating system that the actual displayed size of RTF text is not necessarily proportional to the *VIEW* scale. So, when the text wraps to the bounding box width, it may wrap differently for different *VIEW* scales. The result is that the number of text lines in the bounding box may change as the *VIEW* is zooming in and out. This situation is unavoidable — RTF text (via RTF controls) is designed to be displayed only in its own window, not in the same *drawing plane* as the rest of the graphics. VIS displays RTF text in the *drawing plane* by doing some “cleverness” which has its limitations. Traditional third-party graphics systems typically render their own text, thus avoiding the issue, but that text cannot normally contain embedded OLE objects, as it can for VIS formatted text.

2.12 Selecting Graphics Objects

A graphics application program containing graphics objects typically requires that the user selects the desired object for manipulation. In traditional graphics programs, it is the responsibility of the application program to implement code to detect whether the (usually left) mouse button has been clicked while the mouse cursor is over a graphics object, thereby selecting the object. The application program would also typically implement highlighting code for the selected object.

Selecting a graphics object involves *hit selection*. An autonomous feature of VIS is for the programmer to simply specify a *hit selection* flag (*@goFlags*) to a graphics *component* instance, causing VIS to do the *hit selection* and highlighting itself when the graphics object is clicked; no other programming is necessary. VIS holds a TAC sequence of the *CIDs* of all the graphics object *component* instances corresponding to the currently selected graphics objects. An ETAC program simply calls the *@dpGetSelObjIDs()* function to get the list of the selected graphics object *CIDs* at any time.

There are two types of selection that a programmer can specify for a graphics object. If a programmer specifies *hit selection* alone (*: !GO_HIT_TEST_ON:*), then the graphics object is selected when the mouse cursor is clicked anywhere within the bounding box of the object. If the

programmer also specifies *micro-hit selection* (: !GO_MICRO_SELN:), then the graphics object is selected when the mouse cursor is clicked on any one of the object's items, but not selected when the cursor is clicked outside all of those items. The left or right mouse button can be used to select a graphics object. (Ref: @goFlags)

A selected graphics object is not necessarily highlighted. VIS offers a few ways to highlight a graphics object when selected. A program only needs to specify the highlighting features of a selected object via its graphics context *component* instance; VIS does the actual highlighting automatically. Alternatively, a program can call the @goRepSelect() function to explicitly modify the selection and highlighting of the desired object. If automatic highlighting is not enabled, a program can manage graphics object highlighting itself as desired.

To enable automatic highlighting of a graphics object, a pen attributes *component* instance can be created within the graphics context *initialiser procedure* for the border highlighting features, and a brush attributes *component* can be specified for the background highlighting features. Alternatively (or additionally), the graphics context *component* instance associated with the graphics object can specify a tracking rectangle for highlighting. (Ref: Graphics Context)

If the left or right mouse button is pressed over more than one graphics object, then the front object is selected (if no keyboard key is down and no object was selected prior). If one or more graphics objects are already selected, and the left or right mouse button is pressed over one or more of those objects, then all those objects will remain selected.

2.12.1 Manipulating a Graphics Object

In most graphics application programs, graphics objects are selected for manipulation. Typically, such manipulation consists of moving and resizing the object. In traditional graphics programs, it is the responsibility of the programmer to write code for the manipulation.

An autonomous feature of VIS is for the programmer to specify whether the user is allowed to alter the width, alter the height, or move a graphics object (Ref: @goFlags). The programmer also needs to specify tracking rectangle highlighting for the object (Ref: @gcOHFlags). And, of course, the object has to be selectable (Ref: @goFlags). No other programming is necessary for resizing or moving the object.

The user can move or resize the object only via a tracking rectangle (of course, the programmer can create their own code for other types of manipulations). A tracking rectangle is one of the graphics object selection options, and is displayed as a rectangular border around the object. The border can have thin solid or dotted lines, or it can have thick hatched lines. Sizing handles can be placed either on the inside or on the outside of the border. The graphics object itself can also be overlaid by a hatched pattern. Either two horizontal, two vertical, or eight sizing handles can appear, depending on whether the user is allowed to alter (respectively) the width, height, or both dimensions of the object. (Ref: Graphics Context, @gcOHFlags, @gcOTStyles)

The default tracking rectangle has a thick hatched border, allowing the user to only move a graphics object if the program allows. A graphics object is moved either by the user dragging the tracking rectangle, or dragging the object itself (subject to *hit selection* or *micro-hit selection*). The user can resize a graphics object by dragging one of the sizing handles of the tracking rectangle.

Text and bitmap items do not stretch or shrink when their parent graphics object is resized. VIS is designed that way so as not to distort those items. The programmer, however, can distort those items explicitly by redefining some of their properties if desired. Vector (curve and line) items do resize proportionally to the size of their parent graphics object. See [2.8.4 Graphics Object Groups](#) for the effects of a user resizing or moving a graphics object containing sub-objects.

2.12.2 Selecting Many Graphics Objects

Sometimes more than one graphics object needs to be manipulated at the same time by the user. The desired objects first need to be selected. There are two ways for a user to select more than one graphics object.

- **Clicking:** Additional graphics objects can be selected by holding down the **Ctrl** key while clicking on those objects if they are not already selected. **Ctrl** clicking an object actually toggles the selection state so that any object, from a set of selected objects, can be deselected by **Ctrl** clicking it.
- **Drag Selection:** Graphics objects can be selected by holding down the left mouse button away from any object and dragging the mouse to form a visible selection rectangle. When the left mouse button is released, only the objects entirely within the selection rectangle are selected. If the **Shift** key is down before drag selection begins, then only the objects touching or entirely within the selection rectangle are selected. If the **Ctrl** key is down before drag selection begins, then existing selected objects remain selected in addition to the drag-selected objects. Note that both the **Shift** and **Ctrl** key can be down before drag selection begins, which has the combined effect of both keys. To cancel the drag selection process, the user clicks the right mouse button, or taps the **ESC** (escape) key on the keyboard.

Drag selection is an autonomous feature of VIS. The programmer simply needs to set the `:!DPV_DRAG_SELECT:` flag to the `@vFlags` variable of a VIEW *component* instance; no other programming is necessary.

A program can select multiple graphics objects individually via the `@goRepSelect()` function. That function uses the *CID* of each one of the graphics object *component* instances corresponding to the graphics objects desired for selection.

2.12.3 Manipulating Many Graphics Objects Together

The graphics objects to be manipulated by the user must first be selected (see [2.12.2 Selecting Many Graphics Objects](#)). Only objects that an application program sets to be moveable can be moved by the user (Ref: `@goFlags`). To move those objects, the user simply presses and holds down the left mouse button on one of those selected objects that has a tracking rectangle, and then moves the mouse. The left mouse button can then be released at the desired position. To cancel the movement process, the user clicks the right mouse button, or taps the **ESC** (escape) key on the keyboard. When a user moves graphics objects, only the tracking rectangle moves with the mouse pointer until the mouse button is released. Then the graphics objects will be repositioned at their new position.

There is a special selection mode enabling more than one graphics object to be simultaneously resized (or moved). With more than one object selected, the user holds down the left mouse button on one of those selected objects, and then, without moving the mouse, holds down the right mouse button for at least the double-click period (approximately 0.4 seconds) before releasing it. The left mouse button is then released. A thin dashed tracking rectangle will remain (called the ‘object group tracker’). The user can then move the sizing handles on that rectangle to simultaneously resize the objects within the rectangle proportionally to the distance moved by the mouse. Note that only objects marked for user resizing will be resized. The tracking rectangle can also be dragged to move the enclosed objects. To remove the tracking rectangle, the user clicks anywhere outside it. To cancel the movement or resizing process, the user clicks the right mouse button, or taps the **ESC** (escape) key on the keyboard.

A program can allow the user to manipulate two or more ungrouped graphics objects (as described in the paragraph above) by transferring those objects into a temporary new regular *group object* (see [2.8.4 Graphics Object Groups](#)) created with a tracking rectangle.

2.13 Matrix Objects

A VIS application program may require that a number of graphics objects retain certain relations among themselves when those objects change their size. To achieve this, the program would normally need to have those relations explicitly coded. However, there is a way for the programmer to only specify what the relations are, and then let VIS internally maintain those relations among the objects automatically when requested. This automatic maintenance is achieved by using a matrix object *component*.

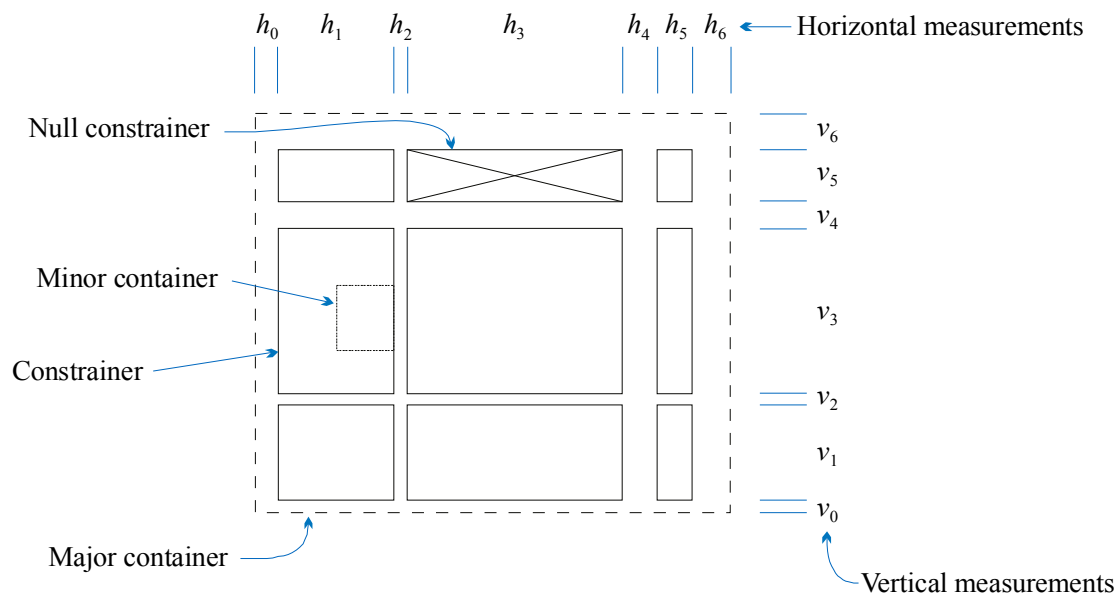
See [1.3.16_Matrix Object Component](#) for general information on matrix object *components*.

A matrix object *component* instance is created by the `vis_MatObject()` function, and represents a group of specified graphics objects laid out in relation to each other in a specified way. Traditional graphics systems use the concept of a layout object to achieve something similar, but a matrix object allows for the explicit specification of the position and size of each object in relation to other objects.

A matrix object *component* instance requires a list of graphics object identifications (*name identifiers* or *CIDs*) and a matrix object definition *component* instance (created by the `vis_MatObjDef()` function) which contains the desired layout arrangements of those objects. An instance of a matrix object *component* is actually a group object internally, except that its sub-objects are subject to the specified layout.

A simple use of a matrix object is to have a number of graphics objects laid out in a grid formation with a specified amount of spacing between the cells of the grid. Each graphics object exists within its own cell and aligned in one of nine ways. With this arrangement, if one of the graphics objects changes size, the grid formation is maintained with the same inter-cell spacing and alignment of the graphics objects.

The diagram below represents a simple 3×3 grid layout of a matrix object presented in a *VIEW*. The constrainers are numbered sequentially, beginning with zero at the bottom left, and increasing left to right from bottom to top.



The horizontal measurements are specified by the `@matdHorSpaces` variable of the associated matrix object definition *component* instance; the vertical measurements are specified by the `@matdVerSpaces` variable. Whether or not a cell contains a null constrainer is specified by the `@matdConsTypes` variable. A null constrainer is not associated with a graphics object. A “minor container” represents the bounding box of the corresponding graphics object (a “minor container” is typically just called a “container”). The “major container” contains the whole matrix object (which is a group graphics object).

When the matrix object is updated via the `@matoUpdateLayout()` function, if a minor container turns out to be larger than its constrainer horizontally or vertically, then the constrainer will automatically enlarge in the relevant dimension to tightly contain the minor container. This causes all the constrainers in the same row or column to also enlarge in the same dimension. For example, if v_3 (in the diagram) were 5.0 *base units*, and the minor container's height (ie: the corresponding graphics object's height) were 7.0 *base units*, then the value of v_3 would be changed to 7.0 *base units* internally. Therefore, the three constrainers in the middle row would also become 7.0 *base units* high internally.

After adjusting the widths and heights of the constrainers where necessary, the next stage of the matrix object update is container justification and wrapping as specified by the flags (first element of each subsequence) of the `@matdHorShift` and `@matdVerShift` variables. Where specified in the flags, container justification is applied first then constrainer wrapping is applied. Container justification positions the container in one of nine positions within its constrainer. Constrainer wrapping then wraps the constrainer tightly around its container. This stage is applied to each cell in the order specified in the `@matdShiftOrd` variable.

After justification and wrapping, the next stage of the matrix object update is constrainer shifting and post wrapping as specified by the flags (first element of each subsequence) of the `@matdHorShift` and `@matdVerShift` variables. Where specified in the flags, the borders or midlines of any number of constrainers are each positioned in relation to the specified borders or midline of any other constrainer, both horizontally and vertically. Horizontal and vertical stretching of the minor containers is also performed as part of the shifting process where specified. Post wrapping is then performed where specified. This stage is applied to each constrainer in the order specified in the `@matdShiftOrd` variable.

The final update stage is calculating the size of the major container. The bottom-left vertex point of the major container is the matrix object origin point.

The minimum coding for a matrix object is shown below. The code produces a matrix object with default values, and the graphics objects in a grid formation with no gaps between the rows and columns.

```
vis_MatObject("")
{
    [* Set the list of graphics objects for the matrix object to lay out. *]
    @matoGObjIDList := [[* Sequence of graphics object IDs. *]];

    vis_MatObjDef("")
    {
        @matdNumRows := [* Number of rows. *];
        @matdNumCols := [* Number of columns. *];
        @matdConsTypes := [* Specify that all cells will have objects. *]
            [do_loops :!MOD_DYNAMIC_CONS: (@matdNumRows * @matdNumCols)];
    };
};
```

(Refs: [3.4.19_Matrix Object](#) - `vis_MatObject()` - [3.4.20_Matrix Object Definition](#) - `vis_MatObjDef()`)

2.14 Measurement Objects

Technical drawing application programs typically involve the display of measurements. VIS has the ability to create and present measurement objects to inform the user of the measurements of parts of a technical diagram. There are four types of measurements: standard (the default), angle, radial, and right angle.

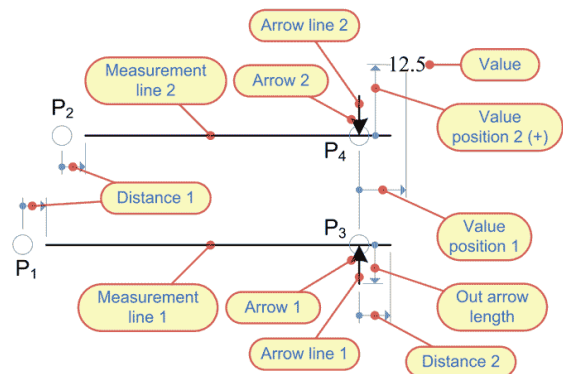
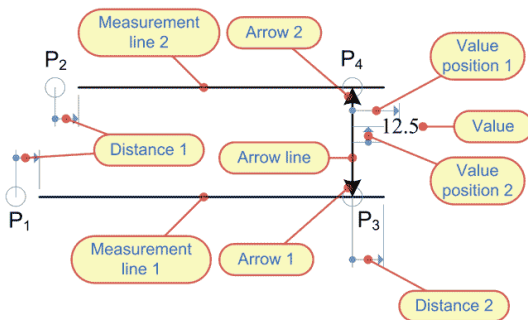
See [1.3.18_Measurement Object Component](#) for general information on measurement object *components*.

A measurement object *component* instance is created by the `vis_MesObject()` function, which also requires a measurement object definition *component* instance created by the `vis_MesObjDef()` function. The same definition instance can be used with more than one measurement object instance.

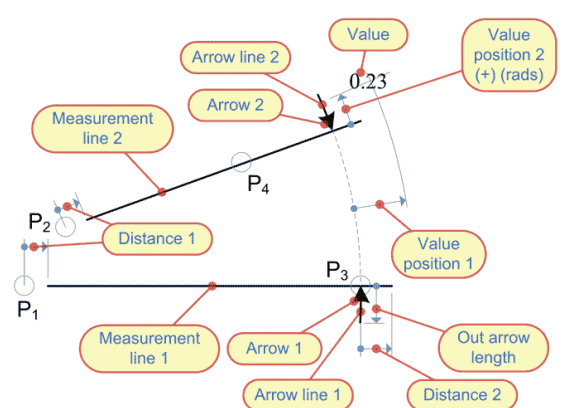
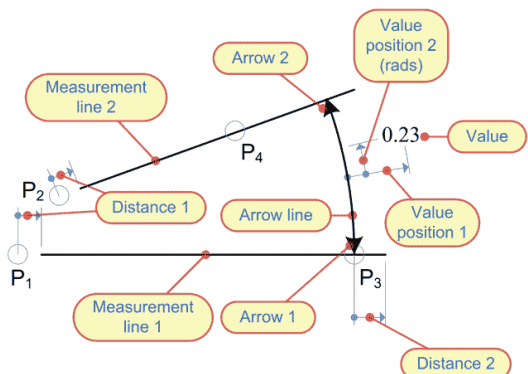
A measurement object requires two to four points (depending on the measurement type) to set the measurement calculation and display. If the points are changed by the program, the appearance of the measurement object is updated by the `@mesoUpdateLayout()` function.

The diagrams below represent the four types of measurement objects indicating the programmable features. The three diagrams in the left column show inner arrow lines, and the ones in the right column show outer arrow lines. P_1 to P_4 represent the points that define the measurement.

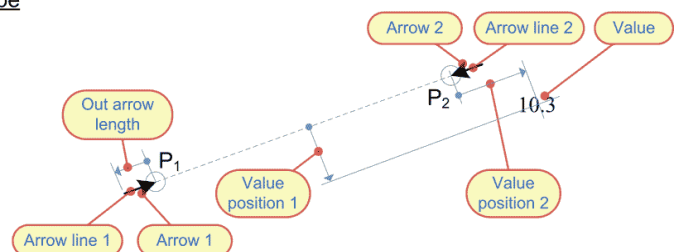
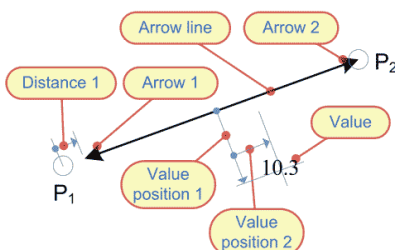
Standard Type



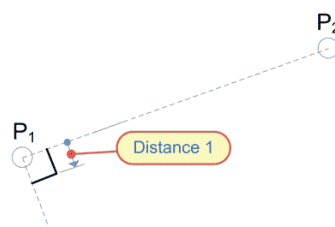
Angle Type



Radial Type



Right-angle Type



The appearance of almost every aspect of a measurement object can be controlled programmatically, including being able to create custom arrowheads for the arrow lines.

The absolute minimum coding for a measurement object is shown below. The code produces a standard measurement object with default values.

```
vis_MesObject("")
{
    @mesoPointList := [[* Four points in a flat sequence. *]];
    vis_MesObjDef("") {};
};
```

A measurement object can also have a custom graphics object in place of the measurement value to show any desired graphics (not applicable to a right-angle measurement). For example, a graphics object containing a variable name (as a text item) can be shown instead of the measurement value. And, of course, other graphics objects can be added to a measurement object since, internally, it is actually a group graphics object with some additional capabilities. But not only that, the programmer can create a custom arrowhead for use in a measurement object in place of one of the pre-defined ones (not for a right-angle measurement).

(Refs: [3.4.21_Measurement Object](#) ▪ [vis_MesObject\(\)](#) ▪ [3.4.22_Measurement Object Definition](#) ▪ [vis_MesObjDef\(\)](#))

2.15 Events and Event Handlers

In VIS, user and other events are handled by certain ETAC functions that are registered with appropriate *components*. Different kinds of events require different handler functions with the appropriate predefined parameters. The functions are defined by the ETAC application program, and then registered as handler functions, allowing VIS to call those functions when the appropriate events occur.

Only certain *components* in VIS can handle user events, as follows:

- A project *component* instance handles the **F1** help key event.
- A drawing plane *component* instance handles the following events: mouse button, mouse movement, graphics object *hit testing*, timer, keyboard key operation, *VIEW* closing, menu command, and miscellaneous activities (includes user object tracking and menu highlight events).
- A VIEW *component* instance handles the following events: *VIEW* activation, *VIEW* sizing, *VIEW* closing, system command, and keyboard key operation.
- A graphics object *component* instance handles the mouse button and graphics object *hit testing* events.

Notice that some different types of *components* handle the same event. Which of the *component* instances gets to handle the event is determined by certain fixed rules.

In order for VIS to capture events, the [vis_WaitForEvents\(\)](#) function must be called.

Note that there is no handler for detecting the rolling of the mouse wheel.

2.15.1 Events and Their Handler Functions

An event handler is an ETAC function, having predefined parameters, that handles particular user and system events. The function is passed as an argument to an appropriate event registration function of the appropriate *component*. When an appropriate event occurs, the function is automatically called by VIS. VIS supports, but does not directly implement, cascaded event handling.

Before an event handler is called, VIS automatically makes the appropriate project, *drawing plane*, and *VIEW current* for that handler. The following table shows the registration functions for the listed event.

Event Registration Functions

<u>Event</u>	<u>Registration Function</u>	<u>Comment</u>
Mouse button.	@dpSetMseBtnFnt ()	Triggered when a mouse button event has occurred.
	@goSetMseBtnFnt ()	Triggered when a mouse button event has occurred. This handler has priority if only one graphics object was chosen by the user.
Mouse movement.	@dpSetMseMoveFnt ()	Triggered when <u>no</u> graphics object is <i>hit</i> while a mouse movement event occurs.
Graphics object <i>hit</i> .	@dpSetMseHitFnt ()	Triggered when a graphics object is <i>hit</i> while a mouse movement event occurs.
	@goSetMseHitFnt ()	Triggered when a graphics object is <i>hit</i> while a mouse movement event occurs. This handler has priority if only one graphics object was <i>hit</i> during the mouse movement event.
Keyboard key.	@dpSetKeyFnt ()	Triggered when the user activates a keyboard key.
	@vSetKeyFnt ()	Triggered when the user activates a keyboard key. This handler has priority if set.
<i>VIEW</i> activation.	@vSetVIEWActFnt ()	Triggered when a <i>VIEW</i> is being activated or inactivated.
<i>VIEW</i> sizing.	@vSetVIEWSizeFnt ()	Triggered when a <i>VIEW</i> changes its size as a result of user or programmed action.
<i>VIEW</i> closing.	@dpSetCloseVIEWFnt ()	Triggered when a <i>VIEW</i> is closed as a result of user action.
	@vSetCloseVIEWFnt ()	Triggered when a <i>VIEW</i> is closed as a result of user action. This handler has priority if set.
Timer.	@dpSetTimerFnt ()	Triggered when a program-specified timing event occurs.
F1 help.	@prjSetHelpFnt ()	Triggered when the F1 key is tapped.
Menu command.	@dpSetCommandFnt ()	Triggered when a menu item is selected directly or by a menu shortcut.
System command.	@vSetSysCmdFnt ()	Triggered when the user releases the left mouse button after performing certain operations on a <i>VIEW</i> .
Miscellaneous.	@dpSetMiscFnt ()	Triggered for various events: menu highlighting, object tracking, <i>image dragging</i> , <i>zoom dragging</i> , drag selection, and client area mouse hit. ♦

The details for each handler function are presented under the corresponding registration function heading for each *component* type.

2.15.2 Setting and Creating a Handler Function

Each handler function needs to be registered by an appropriate handler registration function. Only one handler function can be registered at a time for the same handler registration function. That is to say that the registration of a handler function is not queued.

The following points apply.

- The format of a handler registration function is `@xSetEvtFnt()`, where *x* is the *component* type name prefix of the registering *component*, and *Evt* is a name indicating the event which is to be handled.
- All handler registration functions take an event handler function as their only argument, and return the previous handler function (which can be used for cascading events) or a NULL stack object if there was none. (The handler registration functions replace the pre-existing handler function.) A handler function can be removed by setting the argument to a NULL stack object.
- An event handler function must be defined as follows: `fnt:(pCompID[*int*] ...) {...}`, where `pCompID` will contain the *crucial identifier* (CID) of the *component* instance that registered the handler function when the handler function is called. The handler function is actually the argument to the `@xSetEvtFnt()` function.
- The handler function parameters must be as defined by VIS for the particular handler function. The actual parameter names can be different, but the semantics of the parameters is defined by VIS.
- The parent *drawing plane* of the *component* instance for which the handler function is registered must be *current*. This applies only for registering handler functions for VIEW and graphics object *component* instances.
- Before the handler function is called (by VIS), the *active component* is the one by which the handler function was registered. Therefore, the *CIDs* of the *active component* and that of `pCompID` are the same.

A handler function can be registered outside of the *initialiser procedure* of the *component* instance for which the handler function is registered.

The following example shows how to handle mouse button events for a graphics object *component* instance. The handler function is registered with the graphics object *component* instance via the `@goSetMseBtnFnt()` function which returns the existing handler function or a NULL stack object if there is no such handler.

```
vis_GObject("")
{
    [* Register the mouse button event handler for the graphics object. *]
    void @goSetMseBtnFnt()
        fnt:(pCompID[*int*] pMseFlags[*int*] pXPos[*dec*] pYPos[*dec*] pTime[*int*])
        {
            [* Event handler code. *]
        };
};
```

Note that the handler function, `fnt:(pCompID[*int*] ...) {...}`, is actually the argument to the `@goSetMseBtnFnt()` function. ETAC language syntax allows the function to be passed as shown in the example above.

The `@goSetMseBtnFnt()` function returns the previously registered handler function (if one exists), replacing it with the specified handler function. In this case, there is no existing handler function, so the `@goSetMseBtnFnt()` function returns a NULL stack object which is discarded.

When the handler function in the example above is called, `pCompID` will contain the *CID* of the graphics object *component* instance; that instance will also be the *active component* during the execution of the function.

Cascading Handler Functions

VIS is not capable of directly cascading handler functions. However, it is possible to emulate cascading handler functions.

Handler functions are cascaded when a number of such functions are set up to be called for the same event in some particular order. Since VIS can only call one handler function for a particular event, that handler function itself would need to call other handler functions to process the same event if required. All event handler registration functions return the previously registered handler function (if any) for each registration. That returned handler function can be saved by the program and called at any time during the processing of the event.

The following code fragment illustrates how to create two handler functions that handle mouse button events.

```
vis_DPlane("")
{
    dpMseBtnHndlr :- ?; /* Variable to hold the previous event handler. */

    /* Register the first mouse button event handler for the drawing plane. */
    void @dpSetMseBtnFnt()
        fnt:(pCompID[*int*] pChoiceList[*seq*] pMseFlags[*int*] pXPos[*dec*]
            pYPos[*dec*] pTime[*dec*])
        {
            /* Event handler code. */
        };

    /* Register the second mouse button event handler for the drawing plane. */
    dpMseBtnHndlr := @dpSetMseBtnFnt() /* Save the first event handler. */
        fnt:(pCompID[*int*] pChoiceList[*seq*] pMseFlags[*int*] pXPos[*dec*]
            pYPos[*dec*] pTime[*dec*])
        {
            /* Event handler code. */

            /* Call the previously saved mouse button handler (if it exists). */
            if @NotNull("dpMseBtnHndlr") then
                {dpMseBtnHndlr(pCompID pChoiceList pMseFlags pXPos pYPos pTime);}
            endif;

            /* Event handler code. */
        };
};
```

In the illustration above, the first mouse button event handler is registered as usual because there is no other handler already registered for that event. When the second event handler is registered, the registration function returns the first registered handler which is saved to the `@dpMseBtnHndlr` variable. Note that the returned handler is no longer registered because VIS holds only one registered handler at a time for the same event. When a mouse button event occurs, the second handler is called by VIS because it is the only handler that has been registered. However, that second handler calls the first handler, where desired, via the variable `@dpMseBtnHndlr`.

Any number of handler functions can be cascaded using the technique illustrated above. Of course, a programmer may decide to use a different technique than the one above to cascade handler functions. For example, a global variable containing a sequence of handler functions can be established, with a single handler function being registered to process that sequence.

2.15.3 Creating a Graphics Object from within a Handler Function

In a typical graphics program, graphics objects can be created from within event handler functions. VIS also allows graphics objects, as well as other *components*, to be created from within any event handler function as long as the appropriate parent *component* instance of the new *component* is *active*. The appropriate project *component* instance must also be *active* to register the new *component*.

The following points apply.

- The desired project *component* instance must be made *active* before a new *component* instance can be created. This is because the project *component* needs to register the new *component* instance.
- Next, the desired drawing plane *component* instance must be made *active* before the new *component* instance can be created. This associates the new *component* instance with that *drawing plane* at the lower level of the VIS framework. The exception is when creating new data pack or project *component* instances which do not need a *drawing plane*.
- New item *component* instances can only be created during the creation of a new graphics object instance; they cannot be created for an existing graphics object instance.

The following example illustrates how to create a new graphics object from within a handler function.

```
vis_GObject("")
{
    /* Set the mouse button event handler for the graphics object. */
    void @goSetMseBtnFnt()
        fnt:(pCompID[*int*] pMseFlags[*int*] pXPos[*dec*] pYPos[*dec*] pTime[*int*])
        {
            /* Event handler code. */
            /* The appropriate project, VIEW, and drawing plane are automatically
               current. */
            vis_Project(?) /* Make the current project component active. */
            {
                vis_DPlane(?) /* Make the current drawing plane component active. */
                {
                    vis_GObject("") /* Create the new graphics object. */
                    {
                        /* Code for the new graphics object. */
                    }
                }
            }
        }
    };
};
```

Of course, graphics objects can be created for other existing (non-*current*) project and drawing plane *component* instances via the `vis_SetCurProject()` and `@VIS()` functions, respectively. Also, new project and drawing plane *component* instances can be created via the `vis_Project()` and `vis_DPlane()` functions, respectively.

2.16 Idle-time Processing

The operating system queues user and some system events for an ETAC application program to process them. When all the queued events have been processed, the program waits for the next

event to occur. That waiting period is typically called ‘idle time’. Most events are user events, so *idle time* represents the period that the user is not triggering events and the program is not doing anything useful. A VIS ETAC program can utilise that *idle time* to do some processing that is not suitable to be done during event handling.

In VIS, code to be executed at *idle time* is called an ‘idle-time job’. An ETAC application program sends *idle-time jobs* to VIS which queues them to a list to be activated in turn at *idle time*. VIS removes the job from the list before activating it.

A program specifies the details of an *idle-time job* in a data object created by a call to the `vis_NewIdleJobData()` function. The program then sets the parameters of that data object as desired, and possibly allocates some member variables, before posting it to VIS via the `vis_PostIdleJob()` function. The `vis_PurgeIdleJobs()` function purges all *idle-time jobs* from the *idle-time job* queue. The `vis_DeleteIdleJob()` function deletes an *idle-time job* from the *idle-time job* queue.

If an *idle-time job* is too demanding to be performed at one time, then it can be performed recursively. This involves splitting the job into smaller sections that are executed in turn with the help of the ETAC `uses_parts` command. `uses_parts` allows a function to be executed in parts. If the job function (assigned to `@itjJobFnt`) sets `@itjMoreTime` to `true`, then that same job will be called by VIS during the next *idle-time* period with the value of `@itjJobCount` incremented.

The general three-step process of creating and posting an *idle-time job* is illustrated in the code fragment below.

```
IdleJobData :- ?;

[* 1. Create the data object for the idle-time job. *]
IdleJobData := vis_NewIdleJobData();

[* 2. Initialise the idle-time data object. *]
IdleJobData.
{
    /** Member variables (itj...) that are needed for the job can be allocated
        here. **]

    /** Initialise the desired parameters (@itj...). **]

    [* Define and assign the function that will perform the idle-time job. *]
    @itjJobFnt := fnt:()
    {
        /** Code for the job goes here. **]
    };
};

/** Somewhere in the program ... **]

[* 3. Post the idle-time job. *]
vis_PostIdleJob(IdleJobData);
```

Note that the ETAC programming language is not currently capable of concurrent programming, so *idle-time job* processing can sometimes be used as an alternative.

2.17 Colour and Font Dialog Boxes

An ETAC program can obtain, from the user, a colour value via the `vis_ColorDlg()` function, and font details via the `vis_FontDlg()` function. If successful, `vis_ColorDlg()` returns an RGB colour which can be used directly as a colour value where such is required, and `vis_FontDlg()` returns a sequence containing the user-selected font attributes.

2.18 Clipboard

Graphics objects can be copied to the clipboard (or to a file) via the `vis_WriteGObjs()` function. Plain text can be copied to the clipboard via the `vis_WriteClipStr()` function, and read from the clipboard via the `vis_ReadClipStr()` function.

2.19 Animation Emulation

Animating graphics objects is a useful feature of graphics programs. However, VIS does not intrinsically support animation; in VIS, animated graphics objects can only be emulated.

An emulated animation typically consists of a defined path, as a sequence of points, along which an animated object will move or be altered. Periodically, the object will be positioned along each point of the path in sequence, giving the impression that the object is moving. To achieve realistic movement, some simple physics is used. In physics, the movement of an object along a path is given by the equation: $speed = distance \div time$. If *time* is the total time that the object moves from the beginning to the end of the path, then *speed* is the average speed of the moving object. To make the coding simple, one “unit” of *distance* is taken to be the average distance between consecutive points on the path. Therefore, the object’s position at *n* “units” of *distance* from its original position is the point at index *n* of the path.

The average speed of the animated object is first taken: $avg-speed = \langle total\ distance \rangle \div \langle total\ time \rangle$ ($= \langle \text{number of points in the path minus one} \rangle \div \langle total\ time \rangle$). The value of *total time* is the time that the programmer decided for the object to complete its path. Next, a timer is set up to trigger periodically; for efficiency, this is at $\langle 1 \div avg-speed \rangle$ time intervals. Each time a timer event occurs, the following process is performed: the current distance is calculated from a deduction of the same physics equation: $\langle current\ distance \rangle = avg-speed \times \langle current\ time \rangle$, where *current time* is the elapsed time since the start time of the object (the ETAC `sys_ticks` command can be used to obtain time values). *Current distance* is the distance in “units” from the original position, and therefore an index into the path. (*Current distance* may need to be rounded or truncated to a positive integer.) The position at that index is consequently the position of the object at the *current time*. The animation object is moved from its previous position to the one just calculated.

The following code fragment illustrates how to emulate a moving graphics object in real time along a path using a timer event handler. Additional code can be used to change the characteristics of the object as it moves. For example, the graphics object can be made to blink as it moves along the path by showing and hiding the object at appropriate points along the path. Or the graphics object’s shape can be altered as it moves.

```

vis_DPlane("")
{
    dpPoints[*seq*] :- [[* Decimal path points: x1,y1, x2,y2, x3,y3, ... *]];
    dpPeriod[*int*] :- [* Animation period in milliseconds. *];
    dpNumPoints[*int*] :- ?; dpAvgSpeed[*dec*] :- ?; dpStartTime[*int*] :- ?;

    [* Create and set the timer event handler function. *]
    void @dpSetTimerFnt()
        fnt:(pCompID pVIEWCID pTimerID)
        {
            Time[*int*] :- ?; Idx[*int*] :- ?;

            [* Calculate the time (Time) since the object started (dpStartTime) and its
               index position (Idx) on the path (dpPoints) at that calculated time. *]
            Time := (sys_ticks - dpStartTime);
            Idx := trunc ((dpAvgSpeed * Time) + 1); Idx := ((2 * Idx) - 1);

            if ( Idx <= (2 * dpNumPoints) ) then [* Idx is within range. *]
            {
                [* Change the animated object's position to the current position. *]
                @VIS("Animated Object").
                {
                    [* The new position is the point (x,y) at Idx of dpPoints. *]
                    void void @goRepOrigin(@goOriginX := dpPoints%[Idx]
                                           @goOriginY := dpPoints%[(Idx + 1)]);

                    /** Other alterations to the animated object if desired. **]
                };
            }
            endif;

            [* Cancel the timer when the animation has completed. *]
            if ( Time > dpPeriod ) then {@dpCancelTimer("Main VIEW" 1000);} endif;
        };

    [* Create the VIEW to display the animation. *]
    vis_VIEW("Main VIEW") {[* Initialise the VIEW. *]};

    [* Calculate the number of points (x,y) in the animation path (dpPoints). *]
    dpNumPoints := trunc (|dpPoints| / 2);

    [* Calculate the average speed of the animated object. *]
    dpAvgSpeed := ((dpNumPoints - 1) / dpPeriod);

    [* Set the start time. *]
    dpStartTime := sys_ticks;

    [* Create the graphics object to animate. *]
    vis_GObject("Animated Object")
    {
        [* Creation details. *]
    };

    [* Set the timer (1000 => timer ID). *]
    void @dpSetTimer("Main VIEW" 1000 round (1.0 / dpAvgSpeed));
};

```

More than one graphics object can be animated simultaneously by providing code similar to the one above for each object.

3

Programming Reference

This chapter contains a reference of the functions, commands, and *components* that are installed with the standard installation of the VIS version of ETAC. The chapter is intended for programmers who are familiar with the ETAC programming language and VIS.

3.1 Introduction (Reference)

The information in this reference pertains to the ETAC interpreter version **4-0-6-ena** implemented in RunETAC.exe and AppETAC.dll. That interpreter provides programmers with the ability to create interactive 2-D graphics programs in their ETAC application programs. The programmer needs to include the header file VISGlobal.PTAC within a program to use VIS (Visual Interaction System).

In addition to the information in this reference, there are other auxiliary libraries containing code that relates to VIS. Those libraries exist in their own separate documentations. The document VISShapePathFnts.pdf contains functions to create the paths of various basic shapes, and the document VISPointSetFnts.pdf contains functions to map a set of points or values to another set of points or values. In addition, there is an *ETAC script* library (SplineScriptFnts.pdf) to process Bezier splines and B-splines, and also an *ETAC script* library (PathScriptFnts.pdf) relating to geometric paths.

3.2 Mathematical Constants

The following pre-processor definition names are defined as mathematical constants. The source code definitions exist in the file TACGeneralIncl.TAC, which are included in the file TACGlobal.PTAC.

Mathematical Constants

<u>Symbol</u>	<u>Value</u>	<u>Meaning</u>
#SUCCESS	0	Successful return code.
#min_int	-2147483648	Minimum positive integer.
#max_int	2147483647	Maximum positive integer.
#+min_dec	2.2250738585072014E-308	Minimum positive decimal.
#+max_dec	1.7976931348623158E308	Maximum positive decimal.
#-min_dec	-2.2250738585072014E-308	Negated minimum positive decimal.
#-max_dec	-1.7976931348623158E308	Negated maximum positive decimal.
#mdec	1.1102230246251566636E-16	The largest decimal for which ((1.0 + :#mdec:) = 1.0) is true.

#pi	3.14159265358979323846	Ratio of circumference of a circle to its diameter (π).
#e	2.71828182845904523536	Natural base of logarithms (e).
#sqrt2	1.41421356237309504880	Square root of 2 ($\sqrt{2}$).
#log2e	1.44269504088896340736	Logarithm base 2 of e ($\log_2 e$).
#log10e	0.434294481903251827651	Logarithm base 10 of e ($\log_{10} e$).
#ln2	0.693147180559945309417	Natural logarithm of 2 ($\ln 2$).
#ln10	2.30258509299404568402	Natural logarithm of 10 ($\ln 10$). ♦

3.3 Pre-processor Definitions

The following subsections list all the pre-processor definitions that can be used in a VIS ETAC program. Note that not all definitions are effective; the list of definitions is provided for completeness. All the VIS pre-processor definitions are included in the file VISGlobal.PTAC. Additional non-VIS pre-processor definitions are included in the file TACGlobal.PTAC required for every ETAC program.

3.3.1 System Definitions for Windows

The descriptions of the pre-processor definitions below are based on the descriptions provided by the operating system documentation. The operating system descriptions may be misleading or inaccurate.

System Definitions Reference

- [3.3.1.1 Normal Window Styles](#)
- [3.3.1.2 Extended Window Styles](#)
- [3.3.1.3 Window Visibility](#)
- [3.3.1.4 Pen Styles](#)
- [3.3.1.5 Brush Styles](#)
- [3.3.1.6 Hatch Styles](#)
- [3.3.1.7 Background Modes](#)
- [3.3.1.8 Drawing Modes](#)
- [3.3.1.9 Font Weights](#)
- [3.3.1.10 Character Sets](#)
- [3.3.1.11 Pitch and Family](#)
- [3.3.1.12 ROP Modes](#)
- [3.3.1.13 Stretch Modes](#)
- [3.3.1.14 Mouse Cursor Identifiers](#)
- [3.3.1.15 Menu Item Types](#)
- [3.3.1.16 Menu Item States](#)
- [3.3.1.17 Menu Styles](#)
- [3.3.1.18 Popup Menu](#)
- [3.3.1.19 Status Bar Text](#)
- [3.3.1.20 Status Bar Styles](#)
- [3.3.1.21 Colour Dialog Box](#)
- [3.3.1.22 Font Dialog Box](#)

— Normal Window Styles —

Window styles are used with the `@vStyle` variable and `@vNewSBar()` function of the [VIEW component](#). Note that not all styles are effective; the list of styles is provided for completeness. The following styles can be combined using the `&or` operator.

!WS_@DEFAULTVIEW

This is a special VIS defined style, representing a default *VIEW*, that is equivalent to the `!WS_VISIBLE`, `!WS_MAXIMIZE`, `!WS_OVERLAPPEDWINDOW`, `!WS_THICKFRAME`, `!WS_HSCROLL`, `!WS_VSCROLL`, and `!WS_CLIPCHILDREN` styles. This is the *default*.

!WS_BORDER

Creates a window that has a border.

!WS_CAPTION

Creates a window that has a title bar (implies the `!WS_BORDER` and `!WS_DLGFRAME` styles).

!WS_CHILD

Creates a child window as part of the client area of the parent window. Must not be used with the `!WS_POPUP` style. Used with a child window.

!WS_CLIPCHILDREN

Excludes the area occupied by child windows when a drawing is rendered within the parent window. Used with a parent window.

!WS_CLIPSIBLINGS

Clips child windows relative to each other; that is, when a particular child window renders a drawing, the `!WS_CLIPSIBLINGS` style clips all other overlapping child windows out of the rendering region of the child window. (If `!WS_CLIPSIBLINGS` is not specified, overlapping child windows may possibly draw into each other's client area.) Meaningful with the `!WS_CHILD` style only. Used with a child window.

!WS_DISABLED

Creates a window that is initially disabled.

!WS_DLGFRAME

Creates a window with a double border but no title.

!WS_GROUP

Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined without the `!WS_GROUP` style after the first control belong to the same group. The next control with the `!WS_GROUP` style starts the next group (that is, one group ends where the next begins).

!WS_HSCROLL

Creates a window that has a horizontal scroll bar.

!WS_MAXIMIZE

Creates a window of maximum size.

!WS_MAXIMIZEBOX

Creates a window that has a maximise button.

!WS_MINIMIZE

Creates a window that is initially minimised. For use with the `!WS_OVERLAPPED` style only.

!WS_MINIMIZEBOX

Creates a window that has a minimise button.

!WS_OVERLAPPED

Creates an overlapped window. An overlapped window usually has a caption and a border.

!WS_OVERLAPPEDWINDOW	Creates an overlapped window with the !WS_OVERLAPPED, !WS_CAPTION, !WS_SYSMENU, !WS_THICKFRAME, !WS_MINIMIZEBOX, and !WS_MAXIMIZEBOX styles.
!WS_POPUP	Creates a pop-up window. Must not be used with the !WS_CHILD style.
!WS_POPUPWINDOW	Creates a pop-up window with the !WS_BORDER, !WS_POPUP, and !WS_SYSMENU styles. The !WS_CAPTION style must be combined with the !WS_POPUPWINDOW style to make the control menu visible. Must not be used with the !WS_CHILD style.
!WS_SYSMENU	Creates a window that has a control-menu box in its title bar. Used only for windows with title bars.
!WS_TABSTOP	Specifies one of any number of controls through which the user can select by using the TAB key. The TAB key moves the selection to the next control specified by the !WS_TABSTOP style.
!WS_THICKFRAME	Creates a window with a thick frame that can be used to size the window.
!WS_VISIBLE	Creates a window that is initially visible.
!WS_VSCROLL	Creates a window that has a vertical scroll bar. ♦

— Extended Window Styles —

Extended window styles are used with the **@vExStyle** variable and **@vNewSBar()** function of the **VIEW component**. Note that not all styles are effective; the list of styles is provided for completeness. The following styles can be combined using the **&or** operator.

!WS_EX_ACCEPTFILES	Specifies that a window created with this style accepts drag-and-drop files.
!WS_EX_APPWINDOW	Forces a top-level window onto the taskbar when the window is visible.
!WS_EX_CLIENTEDGE	Specifies that a window has a 3D look — that is, a border with a sunken edge.
!WS_EX_CONTEXTHELP	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message.
!WS_EX_CONTROLPARENT	Allows the user to navigate among the child windows of the window by using the TAB key.
!WS_EX_DLGMODALFRAME	Designates a window with a double border that may (optionally) be created with a title bar when the WS_CAPTION style flag is specified.
!WS_EX_LEFT	Gives the window generic left-aligned properties. This is the default state.
!WS_EX_LEFTSCROLLBAR	Places a vertical scroll bar to the left of the client area.

!WS_EX_LTRREADING	Displays the window text using left-to-right reading order properties. This is the default state.
!WS_EX_NOPARENTNOTIFY	Specifies that a child window created with this style will not send the WM_PARENTNOTIFY message to its parent window when the child window is created or destroyed.
!WS_EX_OVERLAPPEDWINDOW	Combines the WS_EX_CLIENTEDGE and WS_EX_WINDOWEDGE styles.
!WS_EX_PALETTEWINDOW	Combines the WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW, and WS_EX_TOPMOST styles.
!WS_EX_RIGHT	Gives a window generic right-aligned properties. This depends on the window class.
!WS_EX_RIGHTSCROLLBAR	Places a vertical scroll bar (if present) to the right of the client area. This is the default state.
!WS_EX_RTLREADING	Displays the window text using right-to-left reading order properties.
!WS_EX_STATICEDGE	Creates a window with a three-dimensional border style intended to be used for items that do not accept user input.
!WS_EX_TOOLWINDOW	Creates a tool window, which is a window intended to be used as a floating toolbar. A tool window has a title bar that is thinner than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the task bar or in the window that appears when the user presses ALT+TAB .
!WS_EX_TOPMOST	Specifies that a window created with this style should be placed above all non-topmost windows and stay above them even when the window is deactivated.
!WS_EX_TRANSPARENT	Specifies that a window created with this style is to be transparent. That is, the window does not obscure any windows that are beneath it. A window created with this style receives WM_PAINT messages only after all sibling windows beneath it have been updated.
!WS_EX_WINDOWEDGE	Specifies that a window has a border with a raised edge. ♦

— Window Visibility —

Visibility styles are used with the @vShow() function of the VIEW <i>component</i> . Only one of the following values is used.	
#SW_HIDE	Hides the VIEW and activates another window.
#SW_MINIMIZE	Minimises the VIEW and activates the next top-level window in the system's list.
#SW_RESTORE	Activates and displays the VIEW in its original size and position.
#SW_SHOW	Activates the VIEW and displays it in its current size and position.
#SW_SHOWDEFAULT	Shows the VIEW in its initial state.

#SW_SHOWMAXIMIZED
Activates the <i>VIEW</i> and displays it as a maximised window.
#SW_SHOWMINIMIZED
Activates the <i>VIEW</i> and displays it as an icon.
#SW_SHOWMINNOACTIVE
Displays the <i>VIEW</i> as an icon.
#SW_SHOWNA
Displays the <i>VIEW</i> in its current size and position.
#SW_SHOWNOACTIVATE
Displays the <i>VIEW</i> in its most recent size and position.
#SW_SHOWNORMAL
Activates and displays the <i>VIEW</i> . ♦

— Pen Styles —

Pen styles are used with the **@pnStyle** variable of the *Pen Attributes component*. Only one of the following values is used.

#PS_DASH
Creates a dashed pen (– – – –). Applies only when the pen width on the screen is one pixel for a drawing smoothing level of 0 to 3, or when the smoothing level is 4 to 6. Otherwise a solid pen is displayed. See @dpSetSmooth() to set the smoothing level.
#PS_DASHDOT
Creates a pen with alternating dashes and dots (– · – ·). Applies only when the pen width on the screen is one pixel for a drawing smoothing level of 0 to 3, or when the smoothing level is 4 to 6. Otherwise a solid pen is displayed. See @dpSetSmooth() to set the smoothing level.
#PS_DASHDOTDOT
Creates a pen with alternating dashes and double dots (– · · – · ·). Applies only when the pen width on the screen is one pixel for a drawing smoothing level of 0 to 3, or when the smoothing level is 4 to 6. Otherwise a solid pen is displayed. See @dpSetSmooth() to set the smoothing level.
#PS_DOT
Creates a dotted pen (· · · ·). Applies only when the pen width on the screen is one pixel for a drawing smoothing level of 0 to 3, or when the smoothing level is 4 to 6. Otherwise a solid pen is displayed. See @dpSetSmooth() to set the smoothing level.
#PS_NULL
Creates a null pen. A null pen is invisible on the screen.
#PS_SOLID
Creates a solid pen (—). This is the <i>default</i> . ♦

— Brush Styles —

Brush styles are used with the **@brStyle** variable of the *Brush Attributes component*. Only one of the following values is used.

#BS_HATCHED
Creates a hatched brush. See <i>Hatch Styles</i> for available hatch patterns.
#BS_NULL
Creates a null brush. A null brush is transparent on the screen.

#BS_PATTERN

Creates a pattern brush defined by a memory bitmap. Requires a [Bitmap Attributes](#) *component* instance (see [Bitmap Attributes](#) and [@brBitmapID](#)).

#BS_SOLID

Creates a solid brush. This is the *default*. ♦

— Hatch Styles —

Hatch Styles are used with the [@brHatch](#) variable of the [Brush Attributes](#) *component* when [@brStyle](#) has the value :#BS_HATCHED:. Only one of the following values is used.

#HS_BDIAGONAL

A 45-degree upward, left-to-right hatch .

#HS_CROSS

Horizontal and vertical cross-hatch .

#HS_DIAGCROSS

45-degree cross-hatch . This is the *default*.

#HS_FDIAGONAL

A 45-degree downward, left-to-right hatch .

#HS_HORIZONTAL

Horizontal hatch .

#HS_VERTICAL

Vertical hatch . ♦

— Background Modes —

Used with the [@pnBkMode](#), [@brBkMode](#), and [@tsBkMode](#) variables of the [Pen Attributes](#), [Brush Attributes](#), and [Text Style Attributes](#) *components*, respectively. Only one of the following values is used.

#OPAQUE

The background is filled with the respective background colour ([@pnBkColr](#), [@brBkColr](#), or [@tsBkColr](#)) before the text, hatched brush, or pen is drawn.

#TRANSPARENT

The text, hatched brush, or pen has a transparent background (*default*). ♦

— Drawing Modes —

Used with the [@pnDrawMode](#) and [@brDrawMode](#) variables of the [Pen Attributes](#) and [Brush Attributes](#) *components*, respectively. Only one of the following values is used.

#R2_BLACK

Destination is always black (*dest* = BLACK).

#R2_COPYPEN

Destination is the source colour (*dest* = *source*). This is the *default*.

#R2_MASKNOTPEN

Destination is a combination of the colours common to both the inverse of the source and the screen (*dest* = (NOT *source*) AND *dest*).

#R2_MASKPEN

Destination is a combination of the colours common to both the source and the screen (*dest* = *source* AND *dest*).

#R2 _MASKPENNOT	Destination is a combination of the colours common to both the source and the inverse of the screen (<i>dest = source AND (NOT dest)</i>).
#R2 _MERGENOTPEN	Destination is a combination of the inverse of the source colour and the screen colour (<i>dest = (NOT source) OR dest</i>).
#R2 _MERGEPEN	Destination is a combination of the source colour and the screen colour (<i>dest = source OR dest</i>).
#R2 _MERGEPENNOT	Destination is a combination of the source colour and the inverse of the screen colour (<i>dest = source OR (NOT dest)</i>).
#R2 _NOP	Destination colour remains unchanged (<i>dest = dest</i>).
#R2 _NOT	Destination is the inverse of the screen colour (<i>dest = NOT dest</i>).
#R2 _NOTCOPYPEN	Destination is the inverse of the source colour (<i>dest = NOT source</i>).
#R2 _NOTMASKPEN	Destination is the inverse of the #R2_MASKPEN colour (<i>dest = NOT (source AND dest)</i>).
#R2 _NOTMERGEPEN	Destination is the inverse of the #R2_MERGEPEN colour (<i>dest = NOT (source OR dest)</i>).
#R2 _NOTXORPEN	Destination is the inverse of the #R2_XORPEN colour (<i>dest = NOT (source XOR dest)</i>).
#R2 _WHITE	Destination is always white (<i>dest = WHITE</i>).
#R2 _XORPEN	Destination is a combination of the colours in the source and in the screen, but not in both (<i>dest = source XOR dest</i>). ♦

— Font Weights —

Used with the @tsWeight variable of the Text Style Attributes <i>component</i> . The values represent the relative thickness of the characters. Some fonts have only #FW_NORMAL and #FW_BOLD weights. Only one of the following values is used.	
#FW_DONTCARE	Represents an operating system default weight.
#FW_THIN	100.
#FW_EXTRALIGHT	200.
#FW_LIGHT	300.
#FW_NORMAL	400 (<i>default</i>).
#FW_MEDIUM	500.
#FW_SEMIBOLD	600.

#FW_BOLD 700.
#FW_EXTRABOLD 800.
#FW_HEAVY 900. ♦

— Character Sets —

Used with the **@tsCharSet** variable of the [Text Style Attributes](#) *component*. Only one of the following values is used.

#ANSI_CHARSET
ANSI character set (*default*).

#DEFAULT_CHARSET
Represents an operating system value based on the current system locale.

#SYMBOL_CHARSET

#SHIFTJIS_CHARSET

#HANGEUL_CHARSET

#GB2312_CHARSET

#CHINESEBIG5_CHARSET

#OEM_CHARSET ♦

— Pitch and Family —

Used with the **@tsPitchAndFamily** variable of the [Text Style Attributes](#) *component*.

Use the **&or** operator to join one pitch value with one family value.

Pitch values.

#DEFAULT_PITCH
Default pitch.

#FIXED_PITCH
The characters all have the same width.

#VARIABLE_PITCH
The character widths vary (*default*). ♦

Family values.

#FF_DECORATIVE
Novelty fonts. Old English is an example.

#FF_DONTCARE
Operating system default font. This value specifies a generic family name and is used when information about a font does not exist or does not matter. This is the *default*.

#FF_MODERN
Fonts with constant stroke width (monospace), with or without serifs. Monospace fonts are usually modern. Pica, Elite, and Courier New® are examples.

#FF_ROMAN
Fonts with variable stroke width (proportional) and with serifs. MS® Serif, Times New Roman, Century Schoolbook are examples.

#FF_SCRIPT
Fonts designed to look like handwriting. Script and Cursive are examples.

#FF_SWISS

Fonts with variable stroke width (proportional) and without serifs. MS[®] Sans Serif, Helvetica, and Swiss are examples. ♦

— ROP (Raster Operation) Modes —

Used with the [@bmROPMode](#) variable of the [Bitmap Attributes component](#). The “current brush” (*pattern*) is a solid white brush. Only one of the following values is used.

!BLACKNESS

Fills the destination rectangle using the colour associated with index 0 in the physical palette (*dest* = BLACK). (This colour is black for the default physical palette.)

!DSTINVERT

Inverts the destination colours (*dest* = NOT *dest*).

!MERGECOPY

Combines the colours common to both the source and the current brush (*dest* = *source* AND *pattern*).

!MERGEPAINT

Combines the inverted source colours with the destination colours then inverts the result (*dest* = (NOT *source*) OR *dest*).

!NOMIRRORBITMAP

Prevents the bitmap from being mirrored.

!NOTSRCCOPY

Copies the inverted source colours to the destination (*dest* = NOT *source*).

!NOTSRCERASE

Combines the colours of the source and destination then inverts the result (*dest* = NOT (*source* OR *dest*)).

!PATCOPY

Copies the current brush into the destination (*dest* = *pattern*).

!PATINVERT

Combines the colours of the current brush with the colours of the destination that are not in both (*dest* = *pattern* XOR *dest*).

!PATPAINT

Combines the colours of the current brush and the inverted source colours and the destination colours (*dest* = *pattern* OR (NOT *source*) OR *dest*).

!SRCAND

Combines the colours common to both the source and destination (*dest* = *source* AND *dest*).

!SRCCOPY

Copies the source colours directly to the destination (*dest* = *source*). This is the *default*.

!SRCERASE

Combines the colours common to both the source and the inverted destination colours (*dest* = *source* AND (NOT *dest*)).

!SRCINVERT

Combines the colours of the source and destination that are not in both (*dest* = *source* XOR *dest*).

!SRCPAINT

Combines the colours of the source and destination (*dest* = *source* OR *dest*).

!WHITENESS

Fills the destination rectangle using the colour associated with index 1 in the physical palette (*dest* = WHITE). (This colour is white for the default physical palette.) ♦

— Stretch Modes —

Used with the `@bmStretchMode` variable of the [Bitmap Attributes](#) *component*. The stretch mode defines how the system processes pixels of a bitmap when the bitmap is displayed at a smaller size in a *VIEW*. The bitmap data itself is not altered. Only one of the following values is used.

#STRETCH_ANDSCANS

Performs a boolean AND operation using the colour values for the eliminated and existing pixels in the bitmap. If it is a monochrome bitmap, this mode preserves black pixels at the expense of white pixels.

#STRETCH_HALFTONE

Maps blocks of pixels from the bitmap into corresponding pixels in the destination. The colour of the destination pixels approximates the average colour over the bitmap block of pixels. The effect is similar to a bi-linear reduction followed by slight sharpening. This mode is slower and requires more processing of the bitmap image than the other three modes, but produces higher quality images.

#STRETCH_DELETESCANS

This mode deletes all eliminated lines of pixels in the bitmap without trying to preserve their information. This mode is typically used to preserve colour in colour bitmaps. This is the *default*.

#STRETCH_ORSCANS

Performs a boolean OR operation using the colour values for the eliminated and existing pixels in the bitmap. If it is a monochrome bitmap, this mode preserves white pixels at the expense of black pixels. ♦

— Mouse Cursor Identifiers —

Used with the `@dpSetCursor()` function of the [Drawing Plane](#) *component*. Only one of the following values is used.

#IDC_ARROW

Standard arrow.

#IDC_IBEAM

I-beam.

#IDC_WAIT

Hourglass.

#IDC_CROSS

Cross-hair.

#IDC_UPARROW

Vertical arrow pointing upwards.

#IDC_SIZENWSE

Double-pointed arrow pointing north-west and south-east.

#IDC_SIZENESW

Double-pointed arrow pointing north-east and south-west.

#IDC_SIZEWE

Double-pointed arrow pointing west and east.

#IDC_SIZENS

Double-pointed arrow pointing north and south.

#IDC_SIZEALL

Four-pointed arrow pointing north, south, east, and west.

#IDC_NO	Slashed circle.
#IDC_HAND	Pointing hand.
#IDC_APPSTARTING	Standard arrow and small hourglass.
#IDC_HELP	Arrow and question mark. ♦

— Menu Item Types —

Used with the `@mGetMItem()` and `@mPutMItem()` functions of the data object returned by the `vis_NewMenuData()` function, and by the `@MITEM` and `@SMENU` commands. The following flags can be combined using the `&or` operator.

!MFT_MENUBARBREAK	Places the menu item on a new line (for a menu bar) or in a new column (for a submenu or popup menu) separated by a vertical line.
!MFT_MENUBREAK	Places the menu item on a new line (for a menu bar) or in a new column (for a submenu or popup menu) <u>not</u> separated by a vertical line.
!MFT_RADIOCHECK	Displays selected menu items using a radio-button mark instead of a check mark.
!MFT_SEPARATOR	Specifies that the menu item is a separator, appearing as a horizontal dividing line. This value is valid only in a submenu or popup menu.
!MFT_RIGHTJUSTIFY	Right-justifies the menu item and any subsequent items. This value is valid only if the menu item is in a menu bar.
!MFT_RIGHTORDER	Specifies that menus cascade right-to-left (the default is left-to-right). This is used to support right-to-left languages, such as Arabic and Hebrew. ♦

— Menu Item States —

Used with the `@mGetMItem()` and `@mPutMItem()` functions of the data object returned by the `vis_NewMenuData()` function, and by the `@MITEM` and `@SMENU` commands. The following flags can be combined using the `&or` operator.

!MFS_CHECKED	A check mark is placed near the menu item.
!MFS_DEFAULT	Specifies that the menu item is the default. A menu can contain only one default menu item which is displayed in bold.
!MFS_DISABLED	The menu item is disabled and greyed; it cannot be selected.
!MFS_ENABLED	The menu item is enabled; it can be selected. This is the default state.
!MFS_HILITE	The menu item is highlighted.
!MFS_UNCHECKED	No check mark is placed near the menu item. This is the default state.

!MFS_UNHILITE

The menu item is not highlighted. This is the default state. ♦

— Menu Styles —

Used with the `@mRepMInfo()` function of the data object returned by the `vis_NewMenuData()` function, and by the `@SMENU` and `@POPUP` commands. The following flags can be combined using the `&or` operator.

!MNS_AUTODISMISS

The menu automatically ends when the mouse is outside the menu for approximately 10 seconds.

!MNS_CHECKORBMP

The same space is reserved for the check mark and the bitmap. If the check mark is drawn, the bitmap is not. All checkmarks and bitmaps are aligned. Used for menus where some items use checkmarks and some use bitmaps.

!MNS_MODELESS

The menu is modeless; that is, there is no menu modal message loop while the menu is active.

!MNS_NOCHECK

No space is reserved to the left of an item for a check mark. The item can still be selected, but the check mark will not appear next to the item. ♦

— Popup Menu —

Used with the `@vShowPopup()` function of the `VIEW` component.

Combine no more than one entry from each of the following groups using the `&or` operator.

Use one of the following flags to specify how the popup menu is positioned horizontally.

!TPM_CENTERALIGN

The menu is centred horizontally relative to the coordinate specified by the `pHorLoc` parameter.

!TPM_LEFTALIGN

The menu is positioned so that its left side is aligned with the coordinate specified by the `pHorLoc` parameter. This is the default state.

!TPM_RIGHTALIGN

The menu is positioned so that its right side is aligned with the coordinate specified by the `pHorLoc` parameter. ♦

Use one of the following flags to specify how the popup menu is positioned vertically.

!TPM_BOTTOMALIGN

The menu is positioned so that its bottom side is aligned with the coordinate specified by the `pVerLoc` parameter.

!TPM_TOPALIGN

The menu is positioned so that its top side is aligned with the coordinate specified by the `pVerLoc` parameter. This is the default state.

!TPM_VCENTERALIGN

The menu is positioned vertically relative to the coordinate specified by the `pVerLoc` parameter. ♦

Use one of the following flags to specify which mouse button the popup menu tracks.

!TPM_LEFTBUTTON

The user can select menu items with only the left mouse button. This is the default state.

!TPM_RIGHTBUTTON

The user can select menu items with both the left and right mouse buttons. ♦

Use one of the following flags to specify whether to accommodate horizontal or vertical alignment.

!TPM_HORIZONTAL

If the menu cannot be shown at the specified location without overlapping the excluded rectangle (`pRectSeq`), the system tries to accommodate the requested horizontal alignment before the requested vertical alignment. This is the default state.

!TPM_VERTICAL

If the menu cannot be shown at the specified location without overlapping the excluded rectangle (`pRectSeq`), the system tries to accommodate the requested vertical alignment before the requested horizontal alignment. ♦

Use the following flags to determine the user selection without involving the *VIEW* for the menu.

!TPM_NONOTIFY

No notification messages are sent when the user clicks on a menu item.

!TPM_RETURNCMD

The `@vShowPopup()` function returns the menu item identifier of the user's selection. ♦

Use this flag to display a menu when another menu is already displayed. This is intended to support context menus within a menu.

!TPM_RECURSE ♦

— Status Bar Text —

Used with the `@sbRepText()` function of the data object returned by the `vis_NewSBarData()` function. Only one of the following flags is used.

!SBT_@NORMAL

The text is drawn with a border making the text appear sunken (*default*).

!SBT_NOBORDERS

The text is drawn without borders.

!SBT_POPOUT

The text is drawn with a border making the text appear raised.

!SBT_RTLREADING

The text displays in the opposite direction of the text in the parent window. ♦

— Status Bar Styles —

Used with the `@vNewSBar()` function of the *VIEW component*. The following flags can be combined, where meaningful, using the `&or` operator. See also [Normal Window Styles](#).

!SBARS_SIZEGRIP

The status bar will include a sizing grip at the right end. A sizing grip is similar to a sizing border; it is a rectangular area that the user can click and drag to resize the parent *VIEW*. This style is ignored if `!CCS_TOP` is also specified.

!CCS_BOTTOM

Causes the status bar to position itself at the bottom of the parent *VIEW*'s client area and sets the width to be the same as the parent *VIEW*'s client area width. This is the default state.

!CCS_LEFT

Causes the status bar to be displayed vertically on the left side of the parent *VIEW* (implies the !CCS_VERT and !CCS_TOP styles).

!CCS_NODIVIDER

Prevents a two-pixel highlight from being drawn at the top of the status bar.

!CCS_NOMOVEX

Causes the status bar to resize and move itself vertically, but not horizontally, when the parent *VIEW* is resized (implies the !CCS_VERT and !CCS_NOMOVEY styles). This style is ignored if !CCS_NORESIZE is also specified.

!CCS_NOMOVEY

Causes the status bar to resize and move itself horizontally, but not vertically, when the parent *VIEW* is resized. If !CCS_NORESIZE is used, this style does not apply.

!CCS_NORESIZE

Prevents the status bar from using the default width and height when setting its initial size or a new size. Instead, the status bar uses the width and height specified in the request for creation or sizing.

!CCS_RIGHT

Causes the status bar to be displayed vertically on the right side of the parent *VIEW* (implies the !CCS_VERT and !CCS_BOTTOM styles).

!CCS_TOP

Causes the status bar to position itself at the top of the parent *VIEW*'s client area and sets the width to be the same as the parent *VIEW*'s client area width. The !SBARS_SIZEGRIP style is ignored if it is also specified.

!CCS_VERT

Causes the status bar to be displayed vertically. ♦

— Colour Dialog Box —

Used with the **vis_ColorDlg()** function. The following flags can be combined using the **&or** operator.

!CC_ANYCOLOR

Causes the dialog box to display all available colours in the set of basic colours.

!CC_FULLOPEN

Causes the dialog box to display the additional controls that allow the user to create custom colours. If this flag is not set, the user must click the **Define Custom Colour** button to display the custom colour controls.

!CC_PREVENTFULLOPEN

Disables the **Define Custom Colour** button.

!CC_SOLIDCOLOR

Causes the dialog box to display only solid colours in the set of basic colours. ♦

— Font Dialog Box —

Used with the **vis_FontDlg()** function. The following flags can be combined using the **&or** operator.

!CF_EFFECTS

Causes the dialog box to display the controls that allow the user to specify strikeout, underline, and text colour options.

!CF_FIXEDPITCHONLY

Only fixed-pitch fonts are displayed.

!CF_FORCEFONTEXIST

An error condition is indicated if the user attempts to select a font or style that does not exist.

!CF_NOFACESEL

Prevents the dialog box from displaying an initial selection for the font name combo box.

!CF_NOSCRIPTSEL

Disables the **Script** combo box in the dialog box.

!CF_NOSIMULATIONS

Specifies that no font simulations allowed.

!CF_NOSIZESEL

Prevents the dialog box from displaying an initial selection for the font size combo box.

!CF_NOSTYLESEL

Prevents the dialog box from displaying an initial selection for the font style combo box.

!CF_NOVECTORFONTS

No vector font selections allowed.

!CF_NOVERTFONTS

Causes the dialog box to list only horizontally oriented fonts.

!CF_SCALABLEONLY

Allows only the selection of scalable fonts. (Scalable fonts include vector fonts, scalable printer fonts, TrueType fonts, and fonts scaled by other technologies.)

!CF_SCREENFONTS

Causes the dialog box to list only the screen fonts supported by the system. This flag is automatically set.

!CF_SCRIPTONLY

Allows the selection of fonts for all non-OEM and Symbol character sets, as well as the ANSI character set.

!CF_SELECTSCRIPT

Only fonts with the specified character set are displayed. The user will not be allowed to change the character set specified in the **Script** combo box.

!CF_TTONLY

Allows only the selection of TrueType fonts. ♦

3.3.2 Visual Interaction System Definitions

The following pre-processor definitions are defined by VIS.

VIS Definitions Reference

3.3.2.1	Mouse State
3.3.2.2	Key State
3.3.2.3	Miscellaneous Events
3.3.2.4	Standard Colours
3.3.2.5	Common Colours
3.3.2.6	Component Types
3.3.2.7	Drawing Plane Definitions
3.3.2.8	Drawing Frame Borders
3.3.2.9	VIEW Flags
3.3.2.10	Object Highlight Type
3.3.2.11	Object Tracker Styles
3.3.2.12	Graphics Object Values
3.3.2.13	Matrix Object (Constrainer Information)
3.3.2.14	Matrix Object (Constrainer Types)
3.3.2.15	Measurement Object Values
3.3.2.16	Line Item Creation
3.3.2.17	Curve Item Creation
3.3.2.18	Text Item Creation

— Mouse State —

Use with the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.

Use no more than one entry from each of the following groups except “Modifier key states”.

Time span since last mouse button event.

!DP_MSE_TSPAN_MSK

Mask for the time span in milliseconds since the last mouse button event. See the `@vis_MseTimeSpan` command. ♦

Left mouse button states.

!DP_LFT_MSE_MSK

Mask for use with the following three left mouse button states.

!DP_LFT_MSE_DN

The left mouse button is down.

!DP_LFT_MSE_UP

The left mouse button is up.

!DP_LFT_MSE_DCLK

The left mouse button is down on a double-click event. This state is used only with `!DP_LFT_MSE_EVENT`. ♦

Middle mouse button states.

!DP_MID_MSE_MSK

Mask for use with the following three middle mouse button states.

!DP_MID_MSE_DN

The middle mouse button is down.

!DP_MID_MSE_UP

The middle mouse button is up.

!DP_MID_MSE_DCLK

The middle mouse button is down on a double-click event. This state is used only with **!DP_MID_MSE_EVENT**. ♦

Right mouse button states.

!DP_RGT_MSE_MSK

Mask for use with the following three right mouse button states.

!DP_RGT_MSE_DN

The right mouse button is down.

!DP_RGT_MSE_UP

The right mouse button is up.

!DP_RGT_MSE_DCLK

The right mouse button is down on a double-click event. This state is used only with **!DP_RGT_MSE_EVENT**. ♦

Mouse button events. These flags are used with the mouse button states. See also the **@vis_MseMove** command.

!DP_MSE_EVENT_MSK

Mask for use with the following three mouse button events.

!DP_LEFT_MSE_EVENT

Left mouse button event. See the **@vis_LMBPrs**, **@vis_LMBRel**, and **@vis_LMBDclk** commands.

!DP_MID_MSE_EVENT

Middle mouse button event. See the **@vis_MMBPrs**, **@vis_MMBRel**, and **@vis_MMBDclk** commands.

!DP_RGT_MSE_EVENT

Right mouse button event. See the **@vis_RMBPrs**, **@vis_RMBRel**, and **@vis_RMBDclk** commands. ♦

Modifier key states.

!DP_SHIFT_KEY_DN

SHIFT key is down.

!DP_CTRL_KEY_DN

CTRL key is down.

!DP_ALT_KEY_DN

ALT key is down. ♦

— Key State —

Used with the **pFlags** parameter of the event handler functions registered by the **@dpSetKeyFnt()** and **@vSetKeyFnt()** functions.

!DP_KEY_FLAGS

A mask combined with **pFlags** indicating the following information: scan code (bits 0 to 7), extended-key flag (bit 8), context code (bit 13), previous key-state flag (bit 14), and transition-state flag (bit 15) of the key event. The scan code is an OEM-dependent value.

!DP_KEY_CHAR

The event is a character code event (the event occurs only when the key is down).

!DP_KEY_DN

The event is a key code event, and the key is down. If this flag is not set, the key is up. ♦

— Miscellaneous Events —

Used with the `pMiscEvntNum` parameter of the event handler function registered by the `@dpSetMiscFnt()` function.

#VIS_MISC_HTCLIENT

VIEW's client area mouse cursor hit (the mouse cursor enters or leaves the client area).

#VIS_MISC_IMAGEDRAG

User *image dragging*.

#VIS_MISC_MENU_HILITE

Menu highlight.

#VIS_MISC_USRDRAGSEL

User drag selection.

#VIS_MISC_USROBJTRACK

User graphics object tracking.

#VIS_MISC_ZOOMDRAG

User *zoom dragging*. ♦

— Standard Colours —

#RGB_DARK_GREY  (50% grey)

#RGB_LIGHT_GREY  (RGB:227, 227, 227)

#RGB_MEDIUM_GREY  (40% grey)

#RGB_STD_BLUE  (RGB:0, 0, 128)

#RGB_STD_GREY  (25% grey) ♦

— Common Colours —

The following are common colours defined by the operating system. See the table below this one for a sample of the colours.

#RGB_AliceBlue

#RGB_AntiqueWhite

#RGB_Aqua

#RGB_Aquamarine

#RGB_Azure

#RGB_Beige

#RGB_Bisque

#RGB_Black

#RGB_BlanchedAlmond

#RGB_Blue

#RGB_BlueViolet

#RGB_Brown

#RGB_BurlyWood

#RGB_CadetBlue

#RGB_Chartreuse

#RGB_Chocolate

#RGB_Coral
#RGB_CornflowerBlue
#RGB_Cornsilk
#RGB_Crimson
#RGB_Cyan
#RGB_DarkBlue
#RGB_DarkCyan
#RGB_DarkGoldenrod
#RGB_DarkGray
#RGB_DarkGreen
#RGB_DarkKhaki
#RGB_DarkMagenta
#RGB_DarkOliveGreen
#RGB_DarkOrange
#RGB_DarkOrchid
#RGB_DarkRed
#RGB_DarkSalmon
#RGB_DarkSeaGreen
#RGB_DarkSlateBlue
#RGB_DarkSlateGray
#RGB_DarkTurquoise
#RGB_DarkViolet
#RGB_DeepPink
#RGB_DeepSkyBlue
#RGB_DimGray
#RGB_DodgerBlue
#RGB_Firebrick
#RGB_FloralWhite
#RGB_ForestGreen
#RGB_Fuchsia
#RGB_Gainsboro
#RGB_GhostWhite
#RGB_Gold
#RGB_Goldenrod
#RGB_Gray
#RGB_Green
#RGB_GreenYellow
#RGB_Honeydew
#RGB_HotPink
#RGB_IndianRed
#RGB_Indigo

#RGB_Ivory
#RGB_Khaki
#RGB_Lavender
#RGB_LavenderBlush
#RGB_LawnGreen
#RGB_LemonChiffon
#RGB_LightBlue
#RGB_LightCoral
#RGB_LightCyan
#RGB_LightGoldenrodYellow
#RGB_LightGray
#RGB_LightGreen
#RGB_LightPink
#RGB_LightSalmon
#RGB_LightSeaGreen
#RGB_LightSkyBlue
#RGB_LightSlateGray
#RGB_LightSteelBlue
#RGB_LightYellow
#RGB_Lime
#RGB_LimeGreen
#RGB_Linen
#RGB_Magenta
#RGB_Maroon
#RGB_MediumAquamarine
#RGB_MediumBlue
#RGB_MediumOrchid
#RGB_MediumPurple
#RGB_MediumSeaGreen
#RGB_MediumSlateBlue
#RGB_MediumSpringGreen
#RGB_MediumTurquoise
#RGB_MediumVioletRed
#RGB_MidnightBlue
#RGB_MintCream
#RGB_MistyRose
#RGB_Moccasin
#RGB_NavajoWhite
#RGB_Navy
#RGB_OldLace
#RGB_Olive

#RGB_OliveDrab
#RGB_Orange
#RGB_OrangeRed
#RGB_Orchid
#RGB_PaleGoldenrod
#RGB_PaleGreen
#RGB_PaleTurquoise
#RGB_PaleVioletRed
#RGB_PapayaWhip
#RGB_PeachPuff
#RGB_Peru
#RGB_Pink
#RGB_Plum
#RGB_PowderBlue
#RGB_Purple
#RGB_Red
#RGB_RosyBrown
#RGB_RoyalBlue
#RGB_SaddleBrown
#RGB_Salmon
#RGB_SandyBrown
#RGB_SeaGreen
#RGB_SeaShell
#RGB_Sienna
#RGB_Silver
#RGB_SkyBlue
#RGB_SlateBlue
#RGB_SlateGray
#RGB_Snow
#RGB_SpringGreen
#RGB_SteelBlue
#RGB_Tan
#RGB_Teal
#RGB_Thistle
#RGB_Tomato
#RGB_Turquoise
#RGB_Violet
#RGB_Wheat
#RGB_White
#RGB_WhiteSmoke
#RGB_Yellow
#RGB_YellowGreen ♦

The colours are as follows (the values in parentheses are hexadecimal RGB values — do not use these values directly in *ETAC code*).

Common Colour Table

alice blue (#F0F8FF)	antique white (#FAEBD7)	aqua (#00FFFF)	aquamarine (#7FFFD4)
azure (#F0FFFF)	beige (#F5F5DC)	bisque (#FFE4C4)	black (#000000)
blanched almond (#FFEBCD)	blue (#0000FF)	blue violet (#8A2BE2)	brown (#A52A2A)
burly wood (#DEB887)	cadet blue (#5F9EA0)	chartreuse (#7FFF00)	chocolate (#D2691E)
coral (#FF7F50)	cornflower blue (#6495ED)	cornsilk (#FFF8DC)	crimson (#DC143C)
cyan (#00FFFF)	dark blue (#00008B)	dark cyan (#008B8B)	dark goldenrod (#B8860B)
dark gray (#A9A9A9)	dark green (#006400)	dark khaki (#BDB76B)	dark magenta (#8B008B)
dark olive green (#556B2F)	dark orange (#FF8C00)	dark orchid (#9932CC)	dark red (#8B0000)
dark salmon (#E9967A)	dark sea green (#8FBC8B)	dark slate blue (#483D8B)	dark slate gray (#2F4F4F)
dark turquoise (#00CED1)	dark violet (#9400D3)	deep pink (#FF1493)	deep sky blue (#00BFFF)
dim gray (#696969)	dodger blue (#1E90FF)	firebrick (#B22222)	floral white (#FFFAF0)
forest green (#228B22)	fuchsia (#FF00FF)	gainsboro (#DCDCDC)	ghost white (#F8F8FF)
gold (#FFD700)	goldenrod (#DAA520)	gray (#808080)	green (#008000)
green yellow (#ADFF2F)	honeydew (#F0FFF0)	hot pink (#FF69B4)	indian red (#CD5C5C)
indigo (#4B0082)	ivory (#FFFFFF)	khaki (#F0E68C)	lavender (#E6E6FA)
lavender blush (#FFF0F5)	lawn green (#7CFC00)	lemon chiffon (#FFFACD)	light blue (#ADD8E6)
light coral (#F08080)	light cyan (#E0FFFF)	light goldenrod yellow (#FAFAD2)	light green (#90EE90)
light gray (#D3D3D3)	light pink (#FFB6C1)	light salmon (#FFA07A)	light sea green (#20B2AA)
light sky blue (#87CEFA)	light slate gray (#778899)	light steel blue (#B0C4DE)	light yellow (#FFFFE0)
lime (#00FF00)	lime green (#32CD32)	linen (#FAF0E6)	magenta (#FF00FF)

maroon (#800000)	medium aquamarine (#66CDAA)	medium blue (#0000CD)	medium orchid (#BA55D3)
medium purple (#9370DB)	medium sea green (#3CB371)	medium slate blue (#7B68EE)	medium spring green (#00FA9A)
medium turquoise (#48D1CC)	medium violet red (#C71585)	midnight blue (#191970)	mint cream (#F5FFFA)
misty rose (#FFE4E1)	moccasin (#FFE4B5)	navajo white (#FFDEAD)	navy (#000080)
old lace (#FDF5E6)	olive (#808000)	olive drab (#6B8E23)	orange (#FFA500)
orange red (#FF4500)	orchid (#DA70D6)	pale goldenrod (#EEE8AA)	pale green (#98FB98)
pale turquoise (#AFEEEE)	pale violet red (#DB7093)	papaya whip (#FFEFD5)	peach puff (#FFDAB9)
peru (#CD853F)	pink (#FFC0CB)	plum (#DDA0DD)	powder blue (#B0E0E6)
purple (#800080)	red (#FF0000)	rosy brown (#BC8F8F)	royal blue (#4169E1)
saddle brown (#8B4513)	salmon (#FA8072)	sandy brown (#F4A460)	sea green (#2E8B57)
sea shell (#FFF5EE)	sienna (#A0522D)	silver (#C0C0C0)	sky blue (#87CEEB)
slate blue (#6A5ACD)	slate gray (#708090)	snow (#FFFAFA)	spring green (#00FF7F)
steel blue (#4682B4)	tan (#D2B48C)	teal (#008080)	thistle (#D8BFD8)
tomato (#FF6347)	turquoise (#40E0D0)	violet (#EE82EE)	wheat (#F5DEB3)
white (#FFFFFF)	white smoke (#F5F5F5)	yellow (#FFFF00)	yellow green (#9ACD32)

— Component Types —

Component types are read-only and retrieved via the `@visGetType()` function.

visC_Project

Component type of a [Project component](#). Prefix: prj

visC_DPlane

Component type of a [Drawing Plane component](#). Prefix: dp

visC_VIEW

Component type of a [VIEW component](#). Prefix: v

visC_GObject

Component type of a [Graphics Object component](#). Prefix: go

visC_LineItem
Component type of a Line Item <i>component</i> . Prefix: li
visC_CurveItem
Component type of a Curve Item <i>component</i> . Prefix: ci
visC_TextStyle
Component type of a Text Style Attributes <i>component</i> . Prefix: ts
visC_MatObject
Component type of a Matrix Object <i>component</i> . Prefix: mato
visC_MatObjDef
Component type of a Matrix Object Definition <i>component</i> . Prefix: matd
visC_MesObject
Component type of a Measurement Object <i>component</i> . Prefix: meso
visC_MesObjDef
Component type of a Measurement Object Definition <i>component</i> . Prefix: mesd
visC_DataPack
Component type of a Data Pack <i>component</i> . Prefix: dpk ♦

— Drawing Plane Definitions —

#VIS_DP_BG_COLR
Default background colour for <i>VIEWs</i> .
#DP_STD_PICA_PIXELS
The number of pixels in a standard <i>logical pica</i> . The value is 16 pixels (96 DPI / 6).
#CLR_DEFAULT
Default colour value. ♦

— Drawing Frame Borders —

Used with the @dpExtDFrameBord() function of the Drawing Plane <i>component</i> . Only one of the following values is used.
#DP_ALL_BORDERS
#DP_BOTTOM_BORDER
#DP_LEFT_BORDER
#DP_RIGHT_BORDER
#DP_TOP_BORDER ♦

— VIEW Flags —

Used with @vFlags of the VIEW <i>component</i> . The following flags can be combined using the &or operator.
!DPV_AUTO_FOCUS
Causes the <i>VIEW</i> to become active with focus when the mouse cursor is over it.
!DPV_DRAG_SELECT
Sets drag selection on.
!DPV_IMAGE_DRAG
Sets <i>image dragging</i> on.
!DPV_NO_KEY_SCROLL
Prevents scrolling using the scroll keys.

!DPV_NO_WHEEL_SCROLL

Prevents scrolling using the mouse wheel.

!DPV_USE_MAX_WIN

Uses the maximum window size for calculations.

!DPV_ZOOM_DRAG

Sets *zoom dragging* on. ♦

— Object Highlight Type —

Used with the **@gcOHFlags** variable and **@gcRepOHFlags()** function of the [Graphics Context component](#). The following flags can be combined using the **&or** operator.

!OH_BKGRND_HILITE

Fills in the graphics object's background.

!OH_BORD_HILITE

Draws a border around the graphics object.

!OH_TRACKER_HILITE

Draws a tracker rectangle around the graphics object. ♦

— Object Tracker Styles —

Used with the **@gcOTStyles** variable of the [Graphics Context component](#). The following flags can be combined using the **&or** operator.

!OT_BOXES_IN

The resize boxes are shown on the inside of the tracker rectangle (must not be used with **!OT_BOXES_OUT**).

!OT_BOXES_OUT

The resize boxes are shown on the outside of the tracker rectangle (must not be used with **!OT_BOXES_IN**).

!OT_CLEAR

Clears all styles.

!OT_DASH_BORD

The tracker rectangle has a dashed border one pixel wide (must not be used with **!OT_SOL_BORD**).

!OT_HATCH_BORD

The tracker rectangle has a hatched border (the width is specified by **@gcOTBordWidth** with a default value of 0.5 *screen picas*). This is the *default*.

!OT_HATCHED

The interior of tracker rectangle is hatched.

!OT_INVISIBLE

The tracker rectangle has no features visible.

!OT_NULL

Prevents the tracker rectangle style or resize boxes from being modified. This value is used with the **@gcNewObjTracker()** function as a dummy value for **@gcOTStyles** or **@gcOTBordWidth**.

!OT_SOL_BORD

The tracker rectangle has a solid border one pixel wide (must not be used with **!OT_DASH_BORD**). ♦

— Graphics Object Values —

General

Used with the `@goFlags` variable and the `@goRepFlags()` function of the [Graphics Object component](#). The following flags can be combined using the `&or` operator.

!GO_GROUP_OBJ

Indicates that the graphics object *component* instance is a group object. The function `@goRepFlags()` must not set this flag.

!GO_HIDE_MAT_CONT

Ignores the graphics object (making it invisible) if it is a container of a matrix object.

!GO_HIGHLIGHTED

Indicates that the graphics object is highlighted. The highlight state of the graphics object should be altered via the `@goRepSelect()` function.

!GO_HIT_TEST_ON

Specifies that *hit selection* and *hit testing* can be performed on the graphics object when visible.

!GO_IGNORE_LMB

Ignores left mouse button events for the graphics object.

!GO_IGNORE_MBE

Ignores all mouse button events for the graphics object.

!GO_IGNORE_MMB

Ignores middle mouse button events for the graphics object.

!GO_IGNORE_MME

Ignores mouse movement events for the graphics object.

!GO_IGNORE_RMB

Ignores right mouse button events for the graphics object.

!GO_MICRO_SELN

Specifies that the graphics object can be *micro-hit selected*. Requires that `!GO_HIT_TEST_ON` is also specified for this option to be effective.

!GO_NO_RESIZE

Applies when the graphics object is a sub-object of a parent group graphics object. The object is not resized along with the parent object when the user resizes the parent.

!GO_SYSTEM

The graphics object is a system object determined by the programmer.

!GO_USR_HEIGHT

The user can change the height of the graphics object via the object tracker rectangle.

!GO_USR_MOVE

The user can move the graphics object via the object tracker rectangle.

!GO_USR_WIDTH

The user can change the width of the graphics object via the object tracker rectangle.

!GO_VISIBLE

The graphics object is visible. The visibility state of the graphics object should be altered via the `@goShow()` function. ♦

Highlight State
Used with the <code>@goRepSelect()</code> function of the <i>Graphics Object component</i> . Note that a <i>Graphics Context component</i> needs to be set up for the graphics object to effect visual highlighting. Only one of the following values is used.
#GO_HILITE_OFF Deselects and sets highlighting off for the graphics object.
#GO_HILITE_ON Selects and sets highlighting on for the graphics object.
#GO_HILITE_TOGGLE Toggles the selection and highlighting states of the graphics object. ♦
Resize Direction
Used with the <code>pArgs</code> parameter of the event handler function registered by the <code>@dpSetMiscFnt()</code> function, with the value of <code>#VIS_MISC_USROBJTRACK</code> for the <code>pMiscEvtNum</code> parameter.
#GO_DOWN The user altered the bottom border of the tracking rectangle of a graphics object.
#GO_LEFT The user altered the left border of the tracking rectangle of a graphics object.
#GO_RIGHT The user altered the right border of the tracking rectangle of a graphics object.
#GO_UP The user altered the top border of the tracking rectangle of a graphics object. ♦
Group Object
Used by the <code>@goGrpFlags</code> parameter and the <code>@goRepGrpFlags()</code> function of the <i>Graphics Object component</i> . These values apply only to a group graphics object <i>component</i> instance. The following flags can be combined using the <code>&or</code> operator.
!GGO_KEEP_PROP Keeps all the sub-object origins in proportion when resizing the group graphics object.
!GGO_NO_HILITE_SUBOBS Prevents the sub-objects of the group graphics object from being highlighted when the group object itself is highlighted.
!GGO_UNIT_OBJ The group graphics object and its sub-objects form a single unit. Typically used with <code>!GGO_KEEP_PROP</code> . ♦

— Matrix Object (Constrainer Information) —

Used with the <code>@matdHorShift</code> and <code>@matdVerShift</code> variables of the <i>Matrix Object Definition component</i> . Only one item from each of the following headings (except “Default Shift Specification” and “Container and Constrainer Flags”) can be combined using the <code>&or</code> operator.
Left Shift Specification (Target)
!MCI_TARGET_BORD_MSK_L Mask for use with the following three entries.
!MCI_TARGET_BORD1_L Indicates the left or bottom border of the target constrainer.
!MCI_TARGET_BORD2_L Indicates the horizontal or vertical middle line of the target constrainer.

!MCI_TARGET_BORD3_L

Indicates the right or top border of the target constrainer. ♦

Left Shift Specification (Reference)

!MCI_REF_BORD_MSK_L

Mask for use with the following three entries.

!MCI_REF_BORD1_L

Indicates the left or bottom border of the reference constrainer.

!MCI_REF_BORD2_L

Indicates the horizontal or vertical middle line of the reference constrainer.

!MCI_REF_BORD3_L

Indicates the right or top border of the reference constrainer. ♦

Right Shift Specification

The right shift specification target border is always the opposite of the left shift specification target border (not valid with !MCI_TARGET_BORD2_L).

!MCI_REF_BORD_MSK_R

Mask for use with the following three entries.

!MCI_REF_BORD1_R

Indicates the left or bottom border of the reference constrainer.

!MCI_REF_BORD2_R

Indicates the horizontal or vertical middle line of the reference constrainer.

!MCI_REF_BORD3_R

Indicates the right or top border of the reference constrainer. ♦

Minor Container Justification

!MCI_JUST_MSK

Mask for use with the following four entries.

!MCI_JUST1

Indicates centre justification. This is the *default*.

!MCI_JUST2

Indicates left or bottom justification.

!MCI_JUST3

Indicates right or top justification.

!MCI_JUST4

Indicates horizontal or vertical full justification. ♦

Container and Constrainer Flags

The following flags can be combined using the **&or** operator, together with items from the headings above.

!MCI_STRETCH

Stretches container proportionally if necessary. The stretching occurs only if a [Right Shift Specification](#) flag is set.

!MCI_WRAP

Wraps the constrainer around the container.

!MCI_POST_WRAP

Wraps the constrainer around the container after the container is justified, shifted, and (possibly) stretched. ♦

Default Shift Specification

Used for the default elements of the `@matdHorShift` and `@matdVerShift` variables.

`#MCI_DFLT_SHIFT`

Specifies a default shift specification. The value is the TAC sequence [0, 0.0, 0, 0.0, 0]. ♦

— Matrix Object (Constrainer Types) —

Used with the `@matdConstTypes` variable of the [Matrix Object Definition](#) *component*. Only one of the following values is used.

`!MOD_NULL_CONS`

Indicates that the constrainer will not have a corresponding container (ie: will not correspond to a graphics object). This is the *default* for unspecified elements in `@matdConstTypes`.

`!MOD_DYNAMIC_CONS`

Indicates that the constrainer will have a corresponding container (ie: will correspond to a graphics object). ♦

— Measurement Object Values —

Definition Flags

Used with the `@mesdFlags` variable of the [Measurement Object Definition](#) *component*. The following flags can be combined using the `&or` operator.

`!MSOD_@DEFAULT`

This is a special VIS defined flag, representing a default measurement object definition, that is equivalent to the `!MSOD_SHOW_ARROW1`, `!MSOD_SHOW_ARROW2`, `!MSOD_SHOW_LINE1`, `!MSOD_SHOW_LINE2`, and `!MSOD_FILL_AHEAD` flags. This is the *default*.

`!MSOD_DEGREES`

Show value in degrees rather than radians. This flag applies only to angle measurements, it is ignored for other measurements. If set, the measurement value is shown in degrees with a degree symbol. If cleared, the value is shown in radians with no symbol.

`!MSOD_FILL_AHEAD`

Fills in the arrowheads. This flag is ignored for right-angle measurements. If set, the arrowheads are filled in with the same colour as their outline. If cleared, only the outline of the arrow heads are displayed in the measurement object.

`!MSOD_NO_SET_VAL`

Does not set the value object with a value. This flag is ignored for right-angle measurements. If set, the value object is not set to the measurement value. This allows a custom value object to display its own value. For example, it can display a label. If cleared, the value object will be processed normally.

`!MSOD_SHOW_ARROW1`

Shows the first arrow. This flag is ignored for right-angle measurements. If set, the first arrow is visible in the measurement object. For standard and angle measurements, the first arrow is the arrow pointing towards the first measurement line. For radial measurements the first arrow is the arrow pointing towards the first point in `@mesoPointList`. If cleared, the first arrow is invisible in the measurement object.

`!MSOD_SHOW_ARROW2`

Shows the second arrow. This is similar to `!MSOD_SHOW_ARROW1` except that, for standard and angle measurements, the second arrow is the arrow pointing towards the second measurement line. For radial measurements the first arrow is the arrow pointing towards the second point in `@mesoPointList`.

!MSOD_SHOW_LINE1

Shows the first measurement line. This flag applies only to standard and angle measurements, it is ignored for other measurements. If set, the first measurement line is visible in the measurement object. The first measurement line is defined by the first and third points in [@mesoPointList](#). If cleared, the first measurement line is invisible in the measurement object.

!MSOD_SHOW_LINE2

Shows the second measurement line. This is the same as for !MSOD_SHOW_LINE1 except that the second measurement line is defined by the second and fourth argument points in [@mesoPointList](#). ♦

Measurement Types

Used with the [@mesdMeasType](#) variable of the [Measurement Object Definition](#) *component*. Only one of the following values is used.

#MSOD_ANGLE_TYPE

Creates an angle measurement object which measures the angle between two intersecting lines.

#MSOD_RADIAL_TYPE

Creates a radial measurement object which measures the distance between two points.

#MSOD_RIGHT_ANG_TYPE

Creates a right-angle measurement object which indicates a right angle (using the right-angle symbol, $\sphericalangle$).

#MSOD_STD_TYPE

Creates a standard measurement object which measures the distance between two parallel lines. This is the *default*. ♦

Arrowhead Types

Used with the [@mesdArrowType](#) variable of the [Measurement Object Definition](#) *component*. Only one of the following values is used.

#MSOD_CIRCLE_ARR

The shape of the arrowheads is a circle (●). A measurement line runs through the centre of the circle.

#MSOD_CUP_ARR

The shape of the arrowheads is a cup (◐). A cup-shaped arrowhead is a semi-circle with the flat part touching a measurement line.

#MSOD_CUST_ARR

The shape of the arrowheads is a custom shape defined by the poly-curve contained in the [@mesdCustAhead](#) variable.

#MSOD_SIMPLE_ARR

The shape of the arrowheads is an angle (>). A simple arrowhead is two lines joined at one point which touches a measurement line.

#MSOD_SLASH_ARR

The shape of the arrowheads is a slash (\). The centre of the slash crosses a measurement line.

#MSOD_STD_ARR

The shape of the arrowheads is a standard arrow, with the tip of the arrowhead touching a measurement line. This is the *default*.

#MSOD_WIDE_ARR

The shape of the arrowheads is a wide arrow (►). A wide arrowhead is short and wide, with the tip of the arrowhead touching a measurement line. ♦

— Line Item Creation —

Used with the **@liFlags** variable of the *Line Item component*. The following flags can be combined using the **&or** operator.

!GLI_ALTERNATE_FILL

The line item shape is filled using alternate mode.

!GLI_CLOSED_SHAPE

The last point specified for the line item shape will be effectively joined to the first one when the shape is rendered. This is mainly relevant if **!GLI_OUTLINED_SHAPE** is also specified.

!GLI_FILLED_SHAPE

The line item shape is filled in with the item's brush attributes.

!GLI_KEEP_SHAPE

Applies when the parent graphics object stretches or shrinks. The line item shape is not altered, but the first point of the shape stretches or shrinks with the parent graphics object. The effect of this flag is that the shape itself does not stretch or shrink with the parent object, but remains the same size, although the shape's position does stretch or shrink relative to the parent object's origin point.

!GLI_OUTLINED_SHAPE

The line item shape is outlined. ♦

— Curve Item Creation —

Used with the **@ciFlags** variable of the *Curve Item component*. The following flags can be combined using the **&or** operator.

!GCI_ALTERNATE_FILL

The curve item shape is filled using alternate mode.

!GCI_CLOSED_SHAPE

The last point specified for the curve item shape will be effectively joined to the first one when the shape is rendered. This is mainly relevant if **!GCI_OUTLINED_SHAPE** is also specified.

!GCI_FILLED_SHAPE

The curve item shape is filled in.

!GCI_FULL_BBOX

The curve item shape's bounding box will enclose all the shape's defining points (including the control points). Without this flag, the shape's bounding box includes only the actual shape itself.

!GCI_KEEP_SHAPE

Applies when the parent graphics object stretches or shrinks. The curve item shape is not altered, but the first point of the shape stretches or shrinks with the parent graphics object. The effect of this flag is that the shape itself does not stretch or shrink with the parent object, but remains the same size, although the shape's position does stretch or shrink relative to the parent object's origin point.

!GCI_OUTLINED_SHAPE

The curve item shape is outlined. ♦

— Text Item Creation —

Used with the **@tiFlags** variable of the *Text Item component*. Only one item from each of the following headings (except “Text Item Flags”) can be combined using the **&or** operator.

Horizontal Justification

!GTI_HOR_JUST_MASK

Mask for use with the following four entries.

!GTI_CJUST_HOR

Centre justify the text horizontally within the bounding box. This is the *default*.

!GTI_FJUST_HOR

Full justify the text horizontally within the bounding box (implies the **!GTI_LJUST** and **!GTI_RJUST** flags). This option is currently the same as **!GTI_CJUST_HOR**.

!GTI_LJUST

Left justify the text within the bounding box.

!GTI_RJUST

Right justify the text within the bounding box. ♦

Vertical Justification

!GTI_VER_JUST_MASK

Mask for use with the following four entries.

!GTI_BJUST

Bottom justify the text within the bounding box.

!GTI_CJUST_VER

Centre justify the text vertically within the bounding box. This is the *default*.

!GTI_FJUST_VER

Full justify the text vertically within the bounding box (implies **!GTI_TJUST** and **!GTI_BJUST** flags). This option is currently the same as **!GTI_CJUST_VER**.

!GTI_TJUST

Top justify the text within the bounding box. ♦

Text Item Flags

The following flags can be combined using the **&or** operator, together with items from the headings above.

!GTI_CALC_JUST

Recalculates the text justification within the bounding box each time the text is rendered.

!GTI_FILL_BACK

Indicates that the text background is not transparent.

!GTI_OMIT_LEAD

Ignores text internal leading when fitting the text’s bounding box.

!GTI_OVERHANG

Ignores the text overhang when fitting the text’s bounding box.

!GTI_STRETCH_BOUND_BOX

Applies when the parent graphics object stretches or shrinks. Stretches or shrinks the text’s bounding box, and its position, with the parent object. If this flag is not set, only the position of the text’s bounding box stretches or shrinks with the parent object.

!GTI_WORD_WRAP

The text is wrapped to the width of its bounding box. ♦

3.3.3 Operating System Messages

The following messages are sent by the operating system and received by various event handler functions.

Operating System Message Reference

- [3.3.3.1 Virtual Keys](#)
- [3.3.3.2 VIEW Sizing Modes](#)
- [3.3.3.3 System Command Notification Codes](#)

— Virtual Keys —

These codes are received by the event handler function registered by `@dpSetKeyFnt()` function when the user presses or releases a keyboard key.

#VK_BACK
0x08 BACKSPACE key.

#VK_TAB
0x09 TAB key.

#VK_CLEAR
0x0C CLEAR key.

#VK_RETURN
0x0D ENTER key.

#VK_SHIFT
0x10 SHIFT key.

#VK_CONTROL
0x11 CTRL key.

#VK_MENU
0x12 ALT key.

#VK_PAUSE
0x13 PAUSE key.

#VK_CAPITAL
0x14 CAPS LOCK key.

#VK_ESCAPE
0x1B ESC key.

#VK_SPACE
0x20 SPACEBAR key.

#VK_PRIOR
0x21 PAGE UP key.

#VK_NEXT
0x22 PAGE DOWN key.

#VK_END
0x23 END key.

#VK_HOME
0x24 HOME key.

#VK_LEFT
0x25 LEFT ARROW key.

#VK_UP
0x26 UP ARROW key.

#VK_RIGHT
0x27 RIGHT ARROW key.

#VK_DOWN
0x28 DOWN ARROW key.

#VK_SELECT
0x29 SELECT key.

#VK_PRINT
0x2A PRINT key.

#VK_EXECUTE
0x2B EXECUTE key.

#VK_SNAPSHOT
0x2C PRINT SCREEN key.

#VK_INSERT
0x2D INSERT key.

#VK_DELETE
0x2E DELETE key.

#VK_HELP
0x2F HELP key. ♦

Numeric Keys

#VK_0
0x30 0 key.

#VK_1
0x31 1 key.

#VK_2
0x32 2 key.

#VK_3
0x33 3 key.

#VK_4
0x34 4 key.

#VK_5
0x35 5 key.

#VK_6
0x36 6 key.

#VK_7
0x37 7 key.

#VK_8
0x38 8 key.

#VK_9
0x39 9 key. ♦

Alphabetic Keys

#VK_A
0x41 A key.

#VK_B
0x42 B key.

#VK_C
0x43 **C** key.

#VK_D
0x44 **D** key.

#VK_E
0x45 **E** key.

#VK_F
0x46 **F** key.

#VK_G
0x47 **G** key.

#VK_H
0x48 **H** key.

#VK_I
0x49 **I** key.

#VK_J
0x4A **J** key.

#VK_K
0x4B **K** key.

#VK_L
0x4C **L** key.

#VK_M
0x4D **M** key.

#VK_N
0x4E **N** key.

#VK_O
0x4F **O** key.

#VK_P
0x50 **P** key.

#VK_Q
0x51 **Q** key.

#VK_R
0x52 **R** key.

#VK_S
0x53 **S** key.

#VK_T
0x54 **T** key.

#VK_U
0x55 **U** key.

#VK_V
0x56 **V** key.

#VK_W
0x57 **W** key.

#VK_X
0x58 **X** key.

#VK_Y
0x59 **Y** key.

#VK_Z0x5A **Z** key. ♦**Special Keys****#VK_LWIN**

0x5B Left Windows key.

#VK_RWIN

0x5C Right Windows key.

#VK_APPS

0x5D Applications key. ♦

Numeric Keypad Keys**#VK_NUMPAD0**0x60 Numeric keypad **0** key.**#VK_NUMPAD1**0x61 Numeric keypad **1** key.**#VK_NUMPAD2**0x62 Numeric keypad **2** key.**#VK_NUMPAD3**0x63 Numeric keypad **3** key.**#VK_NUMPAD4**0x64 Numeric keypad **4** key.**#VK_NUMPAD5**0x65 Numeric keypad **5** key.**#VK_NUMPAD6**0x66 Numeric keypad **6** key.**#VK_NUMPAD7**0x67 Numeric keypad **7** key.**#VK_NUMPAD8**0x68 Numeric keypad **8** key.**#VK_NUMPAD9**0x69 Numeric keypad **9** key.**#VK_MULTIPLY**0x6A Multiply key **<*>**.**#VK_ADD**0x6B Add key **<+>**.**#VK_SEPARATOR**

0x6C Separator key.

#VK_SUBTRACT0x6D Subtract key **<->**.**#VK_DECIMAL**0x6E Decimal key **<.>**.**#VK_DIVIDE**0x6F Divide key **</>**. ♦**Function Keys****#VK_F1**0x70 **F1** key.

#VK_F2 0x71 F2 key.
#VK_F3 0x72 F3 key.
#VK_F4 0x73 F4 key.
#VK_F5 0x74 F5 key.
#VK_F6 0x75 F6 key.
#VK_F7 0x76 F7 key.
#VK_F8 0x77 F8 key.
#VK_F9 0x78 F9 key.
#VK_F10 0x79 F10 key.
#VK_F11 0x7A F11 key.
#VK_F12 0x7B F12 key.
#VK_F13 0x7C F13 key.
#VK_F14 0x7D F14 key.
#VK_F15 0x7E F15 key.
#VK_F16 0x7F F16 key.
#VK_F17 0x80 F17 key.
#VK_F18 0x81 F18 key.
#VK_F19 0x82 F19 key.
#VK_F20 0x83 F20 key.
#VK_F21 0x84 F21 key.
#VK_F22 0x85 F22 key.
#VK_F23 0x86 F23 key.
#VK_F24 0x87 F24 key. ♦

Control Keys	
#VK_NUMLOCK	0x90 NUM LOCK key.
#VK_SCROLL	0x91 SCROLL LOCK key. ♦
Punctuation Keys	
#VK_OEM_1	0xBA Used for miscellaneous characters; it can vary by keyboard. For the U.S. standard keyboard, the <:;> (comma\colon) key.
#VK_OEM_PLUS	0xBB For any country\region, the <+> key.
#VK_OEM_COMMA	0xBC For any country\region, the <, > key.
#VK_OEM_MINUS	0xBD For any country\region, the <-> key.
#VK_OEM_PERIOD	0xBE For any country\region, the <.> key.
#VK_OEM_2	0xBF Used for miscellaneous characters (varies by keyboard). For the U.S. standard keyboard, the </?> (slash\question) key.
#VK_OEM_3	0xC0 Used for miscellaneous characters (varies by keyboard). For the U.S. standard keyboard, the <~> (accent\tilde) key .
#VK_OEM_4	0xDB Used for miscellaneous characters (varies by keyboard). For the U.S. standard keyboard, the <[> (left-square-bracket\left-brace) key.
#VK_OEM_5	0xDC Used for miscellaneous characters (varies by keyboard). For the U.S. standard keyboard, the < \> (back-slash\vertical-bar) key.
#VK_OEM_6	0xDD Used for miscellaneous characters (varies by keyboard). For the U.S. standard keyboard, the <]> (right-square-bracket\right-brace) key.
#VK_OEM_7	0xDE Used for miscellaneous characters (varies by keyboard). For the U.S. standard keyboard, the <'> (single-quote\double-quote) key.
#VK_OEM_8	0xDF Used for miscellaneous characters (varies by keyboard). ♦

— VIEW Sizing Modes —

These modes are received by the event handler function registered by the **@vSetVIEWSizeFnt()** function when the user resizes a *VIEW*.

#SIZE_MAXHIDE

Some other window has been maximised (received by a popup *VIEW*).

#SIZE_MAXIMIZED

The *VIEW* has been maximised.

#SIZE_MAXSHOW

Some other window has been restored to its former size (received by a popup *VIEW*).

#SIZE_MINIMIZED

The *VIEW* has been minimised.

#SIZE_RESTORED

The *VIEW* has been resized, but neither the #SIZE_MINIMIZED nor #SIZE_MAXIMIZED value applies. ♦

— System Command Notification Codes —

These codes are received by the event handler function registered by the `@vSetSysCmdFnt()` function.

#SC_CLOSE

User has released the left mouse button after closing the *VIEW*.

#SC_HSCROLL

User has released the left mouse button on a horizontal scroll bar.

#SC_KEYMENU

User activated a menu shortcut key.

#SC_MAXIMIZE

User has released the left mouse button after maximising the *VIEW*.

#SC_MINIMIZE

User has released the left mouse button after minimising the *VIEW*.

#SC_MOUSEMENU

User activated the menu via a mouse click.

#SC_MOVE

User has released the left mouse button after moving the *VIEW*.

#SC_RESTORE

User has released the left mouse button after restoring the *VIEW* to its normal position and size.

#SC_SIZE

User has released the left mouse button after sizing the *VIEW*.

#SC_VSCROLL

User has released the left mouse button on a vertical scroll bar. ♦

3.4 VIS Component Data Definitions

Component member functions must not be called from within the *initialiser procedure* of their *component*, except where indicated by the ✨ symbol in the reference pages.

3.4.1 Variable Naming Conventions

The following internal variable naming conventions apply to VIS.

- **Component member (public):** `<@vis...>` and `<@x...>`, where *x* is the lower-case initials of the *component* type name, for example, `@dpGetDispObjIDs()`, `@goRepOrigin()`.
- **Component member (private):** `<_x...>`, where *x* is the initials of the *component* type name. Programmers **MUST NOT ACCESS** these variables in their programs; the names can change in any future release without notice.

NOTE: Programmers must not use the above naming conventions in their own programs.

However, programmers can create public *component* member variable and function names in the form `<x...>`, and private member variable and function names in the form `<_1x...>`, `<_2x...>`, etc,

where **x** is the lower-case initials of the *component* type name. Global variable and function names should begin with a lower-case ‘g’ to avoid any potential conflict VIS variable names.

3.4.2 Component Member Summary

This section contains a look-up summary of all the *component* members in VIS. Variable members are indicated by the ETAC allocation operation (:-) followed by the initial value for the variable. Function members are indicated by an ending pair of parentheses.

Note that “*Initialiser usage*” (marked by ☼) in the table below means that the function can be called within the *initialiser procedure*, otherwise it must not be called from within that procedure.

Member	Description
All Components	Variables and functions common to all <i>components</i> .
@visClass :- ?	For application programmer use.
@visDelete()	Deletes the <i>component</i> instance (except for: project, drawing plane, and all item <i>component</i> instances).
@visGetChildCIDs()	Gets a sequence of the immediate children <i>component CIDs</i> .
@visGetCompID() ☼	Gets the <i>component</i> ’s <i>crucial identifier</i> number (<i>CID</i>). <i>Initialiser usage</i> .
@visGetDPAncCID() ☼	Gets the <i>CID</i> of the <i>drawing plane</i> ancestor <i>component</i> . <i>Initialiser usage</i> .
@visGetName() ☼	Gets the <i>component</i> ’s <i>name ID</i> . <i>Initialiser usage</i> .
@visGetParCID() ☼	Gets the <i>component</i> ’s parent <i>crucial identifier</i> number (<i>CID</i>). <i>Initialiser usage</i> .
@visGetType() ☼	Gets the <i>component</i> ’s type. <i>Initialiser usage</i> .
@visRename()	Renames the <i>component</i> ’s <i>name identifier</i> if possible.
Project	The <i>Project component</i> (<i>vis_Project()</i>).
@prjSetHelpFnt() ☼	Registers the F1 key help handler function. <i>Initialiser usage</i> .
Drawing Plane	The <i>Drawing Plane component</i> (<i>vis_DPlane()</i>).
@dpBkColr :- ?	The <i>drawing plane</i> background colour.
@dpFTBkColr :- ?	Formatted text default background colour.
@dpDFrameBrush :- data	The <i>drawing frame</i> brush attributes <i>component</i> instance (r/o).
@dpDFramePen :- data	The <i>drawing frame</i> pen attributes <i>component</i> instance (r/o).
@dpOGTrackAttr :- data	The object group tracker graphics context <i>component</i> instance (r/o).
@dpSetCloseVIEWFnt() ☼	Registers the close <i>VIEW</i> event handler function. <i>Initialiser usage</i> .
@dpSetCommandFnt() ☼	Registers the command event handler function. <i>Initialiser usage</i> .
@dpSetKeyFnt() ☼	Registers the key event handler function. <i>Initialiser usage</i> .
@dpSetMiscFnt() ☼	Registers the miscellaneous event handler function. <i>Initialiser usage</i> .
@dpSetMseBtnFnt() ☼	Registers the mouse button event handler function. <i>Initialiser usage</i> .
@dpSetMseHitFnt() ☼	Registers the object <i>hit testing</i> event handler function. <i>Initialiser usage</i> .
@dpSetMseMoveFnt() ☼	Registers the mouse movement event handler function. <i>Initialiser usage</i> .
@dpSetTimerFnt() ☼	Registers the timer event handler function. <i>Initialiser usage</i> .
@dpCancelTimer() ☼	Cancels a timer. <i>Initialiser usage</i> .

@dpDeselectGObjs()	Deselects all graphics objects in the <i>drawing plane</i> .
@dpExtDFrameBord() *	Extends a <i>drawing frame</i> border. <i>Initialiser usage</i> .
@dpFitDrawFrame()	Adjusts the size of the <i>drawing frame</i> to enclose all displayable graphics objects.
@dpGetDispObjIDs()	Gets a sequence of the <i>CIDs</i> of the graphics objects in the <i>drawing plane</i> display list.
@dpGetDrawFrameDim() *	Gets the <i>drawing frame</i> dimensions. <i>Initialiser usage</i> .
@dpGetDrawFramePos() *	Gets the <i>drawing frame</i> position. <i>Initialiser usage</i> .
@dpGetDrawPlaneDens() *	Gets the <i>drawing plane</i> density. <i>Initialiser usage</i> .
@dpGetMaxVHeight() *	Gets the nominal height of the client area of a maximised <i>VIEW</i> . <i>Initialiser usage</i> .
@dpGetMaxVWidth() *	Gets the nominal width of the client area of a maximised <i>VIEW</i> . <i>Initialiser usage</i> .
@dpGetMaxVXPos() *	Gets the nominal X position of the client area of a maximised <i>VIEW</i> . <i>Initialiser usage</i> .
@dpGetMaxVYPos() *	Gets the nominal Y position of the client area of a maximised <i>VIEW</i> . <i>Initialiser usage</i> .
@dpGetObjListBBox()	Gets the bounding box enclosing the specified graphics objects.
@dpGetSelObjIDs()	Gets a sequence of the <i>CIDs</i> of the currently selected graphics objects.
@dpOGTrackSel()	Determines whether or not the object group tracker is selected.
@dpSetBkColr() *	Sets the <i>drawing plane</i> background colour. <i>Initialiser usage</i> .
@dpSetBusyCursor() *	Sets the busy cursor (hour glass) on or off. <i>Initialiser usage</i> .
@dpSetCursor() *	Sets the default cursor. <i>Initialiser usage</i> .
@dpSetDrawFrame() *	Sets the <i>drawing frame</i> size, position, and density. <i>Initialiser usage</i> .
@dpSetMinZoomDist() *	Sets the minimum distance (pixels) that the mouse cursor needs to move before a zoom drag event is detected. <i>Initialiser usage</i> .
@dpSetScaleMult() *	Sets the scale multiplier for multiplying the distance that the mouse cursor has moved for <i>zoom drag</i> . <i>Initialiser usage</i> .
@dpSetSmooth() *	Replaces the smoothing level of the <i>VIEWs</i> of the <i>drawing plane</i> . <i>Initialiser usage</i> .
@dpSetTimer() *	Sets a timer. <i>Initialiser usage</i> .
@dpSetZoomAngle() *	Sets the angle (radians) from the vertical that triggers <i>zoom drag</i> . <i>Initialiser usage</i> .
@dpSmoothShot() *	Sets the smooth shot value of the <i>VIEWs</i> of the <i>drawing plane</i> . <i>Initialiser usage</i> .
@dpUpdateVIEWs()	Updates all co- <i>VIEWs</i> but the <i>current VIEW</i> .
VIEW	The <i>VIEW component</i> (vis_VIEW()).
@vExStyle :- ?	The extended styles for a new <i>VIEW</i> .
@vFlags :- 0	The flags for a new <i>VIEW</i> .
@vForceRef :- false	Determines whether to force a full immediate refresh when @vRefresh() is called.
@vHeight :- ?	The <i>VIEW</i> 's height.
@vMaxWinScale :- 3.0	The maximum <i>zoom dragging VIEW</i> scale.
@vMenuID :- ?	The menu bar identifier (HMENU).
@vMinWinScale :- 1.0	The minimum <i>zoom dragging VIEW</i> scale.

@vParVIEWID :- ?	The parent <i>VIEW</i> 's identifier if applicable.
@vParVIEWProjID :- ?	The parent <i>VIEW</i> 's project <i>component</i> identifier if the parent <i>VIEW</i> does not belong to the <i>current</i> project.
@vSizeMode :- 0	Determines the interpretation of @vWidth and @vHeight.
@vStyle :- ?	The styles for a new <i>VIEW</i> .
@vTitle :- ?	The <i>VIEW</i> 's title in the title bar.
@vWidth :- ?	The <i>VIEW</i> 's width.
@vXPos :- ?	The <i>VIEW</i> 's left position relative to the screen or parent <i>VIEW</i> .
@vYPos :- ?	The <i>VIEW</i> 's top position relative to the screen or parent <i>VIEW</i> .
@vSetCloseVIEWFnt() ✱	Registers the <i>VIEW</i> close event handler function. <i>Initialiser usage</i> .
@vSetKeyFnt() ✱	Registers the key event handler function. <i>Initialiser usage</i> .
@vSetSysCmdFnt() ✱	Registers the system command event handler function. <i>Initialiser usage</i> .
@vSetVIEWActFnt() ✱	Registers the <i>VIEW</i> activation event handler function. <i>Initialiser usage</i> .
@vSetVIEWSizeFnt() ✱	Registers the <i>VIEW</i> size change event handler function. <i>Initialiser usage</i> .
@vCalcMaxVIEWScale() ✱	Calculates the maximum <i>VIEW</i> scale so that one <i>drawing plane</i> dot maps to one screen pixel. <i>Initialiser usage</i> .
@vCalcMinVIEWScale() ✱	Calculates the minimum <i>VIEW</i> scale to retain the <i>VIEW</i> 's client area within the <i>window boundary</i> . <i>Initialiser usage</i> .
@vCalibrateScale()	Sets the specified <i>window frame</i> size and position for a <i>VIEW</i> scale of 1.0.
@vDPTtoVIEWPos()	Converts a <i>drawing plane</i> position in <i>base units</i> to a <i>VIEW</i> client position.
@vDrawMBar()	Draws the menu bar associated with the <i>VIEW</i> .
@vEnable()	Sets and returns the current enableity state of the <i>VIEW</i> .
@vGetWinBoundDim()	Gets the <i>window boundary</i> dimensions.
@vGetWinBoundPos()	Gets the <i>window boundary</i> position.
@vGetWinClientDim()	Gets the <i>VIEW</i> 's client area dimensions mapped onto the <i>window frame</i> .
@vGetWinClientPos()	Gets the <i>VIEW</i> 's client area position mapped onto the <i>window frame</i> .
@vGetWinFrameDim()	Gets the <i>window frame</i> dimensions.
@vGetWinFramePos()	Gets the <i>window frame</i> position.
@vMoveWinFrame()	Moves the <i>window frame</i> to the desired position.
@vNewSBar()	Creates a status bar in the <i>VIEW</i> .
@vNewScaleLimits()	Replaces the minimum and maximum <i>zoom dragging VIEW</i> scales.
@vNoBoundChk()	Specifies whether to check that the right and bottom borders of the <i>VIEW</i> 's client area are within the <i>window boundary</i> .
@vRefresh()	Refreshes the <i>VIEW</i> .
@vRepMBar()	Replaces the menu bar assigned to a <i>VIEW</i> .
@vRepScale()	Replaces and returns the existing <i>VIEW</i> scale.
@vRepTitle() :	Replaces and returns the existing <i>VIEW</i> title in the title bar.
@vRepVIEWRect()	Replaces and returns the existing <i>VIEW</i> 's rectangle on the screen.
@vSetKBFocus()	Sets the keyboard focus to the <i>VIEW</i> , activating it or its parent <i>VIEW</i> .
@vSetWinBound()	Sets the <i>window boundary</i> borders.

@vShow()	Shows a <i>VIEW</i> in the specified state.
@vShowPopup()	Displays a pop-up menu anywhere on the screen.
@vUpdateVIEWRect()	Updates the member variables @vXPos, @vYPos, @vWidth, @vHeight with the position and size of the <i>VIEW</i> rectangle on the screen.
@vZoomToAll()	Zooms to all visible graphics objects.
@vZoomToSel()	Zooms to the selected graphics objects.
Graphics Object	The <i>Graphics Object component</i> (<i>vis_GObject()</i>).
@goContextID :- ?	The identifier of the graphics context <i>component</i> instant for this object.
@goDisplay :- true	Determines whether the graphics object is displayed or put on the object storage list invisibly.
@goFlags :- 0	The flags for the graphics object.
@goGrpFlags :- 0	The flags for the group graphics object (if it is one).
@goOriginX :- 0.0	The graphics object's X position.
@goOriginY :- 0.0	The graphics object's Y position.
@goSetMseBtnFnt() ⚠	Registers the mouse button event handler function. <i>Initialiser usage.</i>
@goSetMseHitFnt() ⚠	Registers the object <i>hit testing</i> event handler function. <i>Initialiser usage.</i>
@goGetBoundingBox()	Gets the actual bounding box of this graphics object.
@goGetOrigin()	Gets the position of this graphics object.
@goGetSubObjIDs()	Gets a sequence of the <i>CIDs</i> of the sub-objects of this graphics object.
@goRelocateTo()	Relocates this graphics object as a sub-object of a group object, or to the front of the object display list, or to the object storage list.
@goReorder()	Reorders the depth position of this graphics object in relation to its sibling objects.
@goRepContext()	Replaces and returns the existing graphics context <i>component</i> instance of this graphics object.
@goRepFlags()	Replaces and returns the existing flags of this graphics object.
@goRepGrpFlags()	Replaces and returns the existing flags of this group object (if it is one).
@goReplaceWith()	Replaces this graphics object with a top-level one in the object storage list.
@goRepOrigin()	Replaces and returns the existing position of this graphics object.
@goRepSelect()	Replaces and returns the existing selection state of this graphics object.
@goSetRedraw()	Sets this graphics object to be redrawn at a suitable time.
@goShow()	Shows or hides this graphics object, or just returns its visibility state.
Line Item	The <i>Line Item component</i> (<i>vis_LineItem()</i>).
@liBrushID :- ?	The identifier of the brush attributes <i>component</i> instance for this item.
@liFixed :- false	If true, this is a fixed item.
@liFlags :- 0	The flags for the line item.
@liLowerScale :- 0.0	The lowest <i>VIEW</i> scale below which this item becomes invisible.
@liPenID :- ?	The identifier of the pen attributes <i>component</i> instance for this item.

@liPoints :- ?	The sequence of points of a poly-line.
@liUpperScale :- :#+max_dec:	The highest <i>VIEW</i> scale equal to or above which this item becomes invisible.
@liNewPoints()	Replaces the existing points of this item.
@liRepBrush()	Replaces and returns the existing brush attributes.
@liRepFlags()	Replaces and returns the existing flags of this item.
@liRepLwrScale()	Replaces and returns the lower <i>VIEW</i> scale.
@liRepPen()	Replaces and returns the existing pen attributes.
@liRepUprScale()	Replaces and returns the upper <i>VIEW</i> scale.
Curve Item	The <i>Curve Item component</i> (<i>vis_CurveItem()</i>).
@ciBrushID :- ?	The identifier of the brush attributes <i>component</i> instance for this item.
@ciFixed :- false	If true, this is a fixed item.
@ciFlags :- 0	The flags for the curve item.
@ciLowerScale :- 0.0	The lowest <i>VIEW</i> scale below which this item becomes invisible.
@ciPenID :- ?	The identifier of the pen attributes <i>component</i> instance for this item.
@ciPoints :- ?	The sequence of points of a poly-Bezier.
@ciUpperScale :- :#+max_dec:	The highest <i>VIEW</i> scale equal to or above which this item becomes invisible.
@ciNewPoints()	Replaces the existing points of this item.
@ciRepBrush()	Replaces and returns the existing brush attributes.
@ciRepFlags()	Replaces and returns the existing flags of this item.
@ciRepLwrScale()	Replaces and returns the lower <i>VIEW</i> scale.
@ciRepPen()	Replaces and returns the existing pen attributes.
@ciRepUprScale()	Replaces and returns the upper <i>VIEW</i> scale.
Text Item	The <i>Text Item component</i> (<i>vis_TextItem()</i>).
@tiAutoFitBound :- true	Indicates whether the internal bounding box is fitted around the text when the item is created after it is aligned with the specified bounding box.
@tiBoundingBox :- [0.0, 0.0, 0.0, 0.0]	The bounding box into which the text is aligned.
@tiFixed :- false	If true, this is a fixed item.
@tiFlags :- 0	The flags for the text item.
@tiLowerScale :- 0.0	The lowest <i>VIEW</i> scale below which this item becomes invisible.
@tiText :- ""	The text for this item.
@tiTextStyleID :- ?	The identifier of the text style attributes <i>component</i> instance for this item.
@tiUpperScale :- :#+max_dec:	The highest <i>VIEW</i> scale equal to or above which this item becomes invisible.
@tiFitBoundingBox()	Fits the bounding box around the text of this item.
@tiGetBoundingBox()	Gets the internal bounding box of this item.
@tiRepBoundingBox()	Replaces and returns the actual bounding box of this item.
@tiRepFlags()	Replaces and returns the existing flags of this item.
@tiRepLwrScale()	Replaces and returns the lower <i>VIEW</i> scale.
@tiRepText()	Replaces and returns the text of this item.

@tiRepTextStyle()	Replaces and returns the text style attributes.
@tiRepUpScale()	Replaces and returns the upper <i>VIEW</i> scale.
Formatted Text Item	The <u>Formatted Text Item</u> <i>component</i> (vis_FormText()).
@ftAutoFitBound :- true	Indicates whether the internal bounding box is fitted around the text when the item is created.
@ftBkColr :- ?	The background colour of the text.
@ftBoundingBox :- [0.0, 0.0, 0.0, 0.0]	The bounding box for the text.
@ftFixed :- false	If true, this is a fixed item.
@ftLowerScale :- 0.0	The lowest <i>VIEW</i> scale below which this item becomes invisible.
@ftText :- ""	The RTF text containing RTF codes for this item.
@ftUpperScale :- :#+max_dec:	The highest <i>VIEW</i> scale equal to or above which this item becomes invisible.
@ftFitBoundingBox()	Fits the bounding box around the text of this item.
@ftGetBoundingBox()	Gets the actual bounding box of this item.
@ftRepBkColr()	Replaces and returns the background colour of the text.
@ftRepBoundingBox()	Replaces and returns the actual bounding box of this item.
@ftRepLwrScale()	Replaces and returns the lower <i>VIEW</i> scale.
@ftRepText()	Replaces and returns the text of this item.
@ftRepUpScale()	Replaces and returns the upper <i>VIEW</i> scale.
Image Item	The <u>Image Item</u> <i>component</i> (vis_ImageItem()).
@iiBitmapID :- ?	The identifier of the bitmap attributes <i>component</i> instance for this item.
@iiBoundingBox :- ?	The bounding box for the image.
@iiFixed :- false	If true, this is a fixed item.
@iiLowerScale :- 0.0	The lowest <i>VIEW</i> scale below which this item becomes invisible.
@iiUpperScale :- :#+max_dec:	The highest <i>VIEW</i> scale equal to or above which this item becomes invisible.
@iiFitBoundingBox()	Fits the bounding box around the natural size of the image of this item.
@iiGetBoundingBox()	Gets the actual bounding box of this item.
@iiRepBitmap()	Replaces and returns the bitmap attributes.
@iiRepBoundingBox()	Replaces and returns the actual bounding box of this item.
@iiRepLwrScale()	Replaces and returns the lower <i>VIEW</i> scale.
@iiRepUpScale()	Replaces and returns the upper <i>VIEW</i> scale.
Graphics Context	The <u>Graphics Context</u> <i>component</i> (vis_GContext()).
@gcBrushID :- ?	The identifier of the brush attributes <i>component</i> instance of this graphics context for background highlighting.
@gcOHFlags :- 0	The highlighting flags.
@gcOTBordWidth :- 0.5	The object tracker border and resize boxes width.
@gcOTStyles :- :!OT_HATCH_BORD:	The object tracker styles.
@gcPenID :- ?	The identifier of the pen attributes <i>component</i> instance of this graphics context for foreground highlighting.
@gcNewObjTracker()	Sets new object tracker details.

@gcRepBrush ()	Replaces and returns the existing brush attributes.
@gcRepOHFlags ()	Replaces and returns the existing highlighting flags of this graphics context.
@gcRepPen ()	Replaces and returns the existing pen attributes.
Pen Attributes	The Pen Attributes <i>component</i> (vis_Pen()).
@pnBkColr :- ?	The pen background colour.
@pnBkMode :- ?	The background mode.
@pnColr :- :#RGB_Black:	The pen foreground colour.
@pnDrawMode :- ?	The drawing mode.
@pnStyle :- :#PS_SOLID:	The pen style.
@pnWidth :- 0.0	The pen width.
@pnNewPen ()	Sets new pen details.
@pnRepBkColr ()	Replaces and returns the background colour of the pen.
@pnRepBkMode ()	Replaces and returns the background mode of the pen.
@pnRepDrawMode ()	Replaces and returns the drawing mode of the pen.
Brush Attributes	The Brush Attributes <i>component</i> (vis_Brush()).
@brBitmapID :- ?	The identifier of the bitmap attributes <i>component</i> instance for this brush attributes.
@brBkColr :- ?	The brush background colour.
@brBkMode :- ?	The background mode.
@brColr :- :#RGB_MEDIUM_GREY:	The brush foreground colour.
@brDrawMode :- ?	The drawing mode.
@brHatch :- :#HS_DIAGCROSS:	The hatch style.
@brStyle :- :#BS_SOLID:	The brush style.
@brNewBrush ()	Sets new brush details.
@brRepBitmap ()	Replaces and returns the bitmap attributes.
@brRepBkColr ()	Replaces and returns the background colour of the brush.
@brRepBkMode ()	Replaces and returns the background mode of the brush.
@brRepDrawMode ()	Replaces and returns the drawing mode of the brush.
Bitmap Attributes	The Bitmap Attributes <i>component</i> (vis_Bitmap()).
@bmImage :- ?	The BMP file data or file path of a BMP file.
@bmROPMode :- :!SRCCOPY:	The ROP (Raster Operation) mode.
@bmStretchMode :- :#STRETCH_DELETESCANS:	The stretching mode.
@bmNewBitmap ()	Sets new bitmap details.
Text Style Attributes	The Text Style Attributes <i>component</i> (vis_TextStyle()).
@tsBkColr :- ?	The text background colour.
@tsBkMode :- ?	The text background mode.
@tsCharSet :- :#ANSI_CHARSET:	The character set.
@tsColr :- ?	The text foreground colour.
@tsFaceName :- ""	The face name.
@tsHeight :- -1.0	The font height.

@tsItalic :- false	If true, the text is italic.
@tsPitchAndFamily :- :#VARIABLE_PITCH:	The pitch and family.
@tsStrikeThru :- false	If true, the text is struck through.
@tsUnderline :- false	If true, the text is underlined.
@tsWeight :- :#FW_NORMAL:	The font weight.
@tsWidth :- 0.0	The font width.
@tsGetValues ()	Places the text style attributes of the <i>component</i> into a sequence.
@tsNewFont ()	Sets new font details.
@tsPutValues ()	Assigns the text style attributes in a sequence to the <i>component</i> .
@tsRepBkColr ()	Replaces and returns the background colour of the text.
@tsRepBkMode ()	Replaces and returns the background mode of the text.
@tsRepColr ()	Replaces and returns the foreground colour of the text.
Matrix Object	The <u>Matrix Object</u> <i>component</i> (vis_MatObject ()).
@matoGObjIDList :- []	A sequence of graphics object <i>component</i> instance identifiers.
@matoMatDefID :- ?	The identifier of a matrix object definition <i>component</i> instance.
@matoAttachDef ()	Attaches a matrix object definition <i>component</i> instance.
@matoNumDynCons ()	Returns the number of dynamic constrainers.
@matoPurgeGObjs ()	Purges all sub-objects to the display list.
@matoUpdateGObjs ()	Updates the graphics objects for this matrix object.
@matoUpdateLayout ()	Updates the layout of this matrix object.
Matrix Object Definition	The <u>Matrix Object Definition</u> <i>component</i> (vis_MatObjDef ()).
@matdConsTypes :- []	The constrainer type of each constrainer in a matrix object.
@matdHorShift :- ?	The horizontal shift specifications of each constrainer in a matrix object.
@matdHorSpaces :- ?	The nominal horizontal spacing of a matrix object's columns.
@matdNumCols :- ?	Indicates the number of columns in the matrix object.
@matdNumRows :- ?	Indicates the number of rows in the matrix object.
@matdShiftOrd :- ?	The shift order of each constrainer in a matrix object.
@matdVerShift :- ?	The vertical shift specifications of each constrainer in a matrix object.
@matdVerSpaces :- ?	The nominal vertical spacing of a matrix object's rows.
Measurement Object	The <u>Measurement Object</u> <i>component</i> (vis_MesObject ()).
@mesoArrowHdPenID :- ?	The identifier of a pen attributes <i>component</i> instance for the outline of the arrowheads.
@mesoArrowHdSize :- 0.5	The size (length) of all arrowheads.
@mesoArrowLnPenID :- ?	The identifier of a pen attributes <i>component</i> instance for the arrow lines.
@mesoFail :- 0	Indicates a failure code for @mesoUpdateLayout () .
@mesoLinePenID :- ?	The identifier of a pen attributes <i>component</i> for the measurement lines.
@mesoMesDefID :- ?	The identifier of a measurement object definition <i>component</i> instance.

<code>@mesoPointList :- []</code>	A sequence containing two or four points that determine the measurement value of this measurement object.
<code>@mesoSigDigits :- 3</code>	The number of significant digits of the measurement value.
<code>@mesoVal :- ?</code>	The value of the measurement object after initial creation.
<code>@mesoValidate :- false</code>	If true, validates the point list in <code>@mesoPointList</code> when <code>@mesoUpdateLayout()</code> is called.
<code>@mesoValueMult :- 1.0</code>	The measurement value multiplier.
<code>@mesoValueObjID :- ?</code>	The identifier of a custom value graphics object <i>component</i> instance.
<code>@mesoValueStyleID :- ?</code>	The identifier of a text style <i>component</i> instance for the value object.
<code>@mesoCalcVal()</code>	Calculates the measurement value rather than using the assigned value.
<code>@mesoNewArrowHdAttr()</code>	Attaches a pen attributes <i>component</i> instance to this measurement object for the outline of the arrowheads.
<code>@mesoNewArrowHdSize()</code>	Sets the size (length) of all arrowheads.
<code>@mesoNewArrowLnAttr()</code>	Attaches a pen attributes <i>component</i> instance to this measurement object for the arrow lines.
<code>@mesoNewLineAttr()</code>	Attaches a pen attributes <i>component</i> instance to this measurement object for the measurement lines.
<code>@mesoNewSigDigits()</code>	Sets the number of significant digits of the displayed measurement value.
<code>@mesoNewValueAttr()</code>	Attaches a text style <i>component</i> instance for the value object.
<code>@mesoNewValueMult()</code>	Assigns a measurement value multiplier.
<code>@mesoSetValue()</code>	Assigns a measurement value.
<code>@mesoShowVal()</code>	Determines whether the measurement value is visible or not.
<code>@mesoUpdateLayout()</code>	Updates the layout of this measurement object.
Measurement Object Definition	The <u>Measurement Object Definition</u> <i>component</i> (<code>vis_MesObjDef()</code>).
<code>@mesdAncPoint :- ?</code>	The anchor point number for the arrowheads.
<code>@mesdArrowBase :- ?</code>	The arrow base number for the arrowheads.
<code>@mesdArrowType :- ?</code>	The shape of the predefined arrowheads to be displayed by this measurement object.
<code>@mesdCustAhead :- ?</code>	A poly-curve sequence for the definition of a custom arrowhead.
<code>@mesdDist1 :- ?</code>	The distance before the start of the measurement line, arrow line, or size of the right-angle symbol.
<code>@mesdDist2 :- ?</code>	The length of the measurement lines beyond the arrowheads.
<code>@mesdFlags :- : !MSOD_DEFAULT:</code>	The flags for this measurement object definition.
<code>@mesdInArrow :- ?</code>	Specifies whether the arrow line is between or on the outside of the measurement lines.
<code>@mesdMeasType :- : #MSOD_STD_TYPE:</code>	The type of measurement to be displayed by this measurement object.
<code>@mesdOutALen :- ?</code>	The length of the outside arrows.
<code>@mesdValPos1 :- ?</code>	The perpendicular distance of the value object from the centre point of the arrow line.
<code>@mesdValPos2 :- ?</code>	The longitudinal distance of the value object from the centre point of the arrow line.

@mesdAddFlags ()	Adds flags to this measurement object definition.
@mesdClearFlags ()	Clears flags from this measurement object definition.

3.4.3 Notational Conventions

The following conventions are used in the description boxes of the following sections.

- For member variables, the top row contains the name of the member variable (in **bold**), followed by an assignment symbol (**:=**), followed by the label representing the required argument (in *blue*), followed by the initial value of the variable (within **bold**) in parentheses.
- For member functions, the top row contains the name of the member function (in **bold**), with the labels representing the required input arguments (in *blue*) within parentheses, followed by a red arrow (→), and the labels representing the produced output arguments (in *blue*) after the arrow. The red arrow itself is not part of the syntax and is not to be entered.
- The symbol ✨ after a member function indicates that the function can be used within the *initialiser procedure* of the *component* creation function.
- Input argument labels will not be present if the member function does not require input arguments.
- Output argument labels and the red arrow will not be present if the member function does not produce output arguments.
- Each argument is a stack object on the object stack.
- The leftmost argument in a series of input argument labels is the topmost stack object on the object stack; the next argument label is the second top stack object, and so on. Likewise for output argument labels.
- Black square brackets ([]) enclose member variables (beginning with an @ symbol) that are used by the function. This means that the variables between the brackets can be read or modified by the function. If the brackets are on the left side of the red arrow (→), or if there is no red arrow, then the values of the member variables are read by the function; if the brackets are on the right side of the red arrow, then values of the member variables may be modified by the function. The brackets and the enclosed member variables themselves are not part of the syntax and are not to be entered when calling the function.
- Red vertical bars (|) separate one of a number of options to be entered. The bars themselves are not part of the syntax and are not to be entered.
- The description of each argument (if any) exists in the rows following the top row.

In the following example of the top row of a description box,

```
@gcRepOHFlags (pFlags pAdd) → prev-flags [@gcOHFlags]
```

when the function **@gcRepOHFlags ()** is called, it expects two arguments on the object stack (represented by *pFlags* and *pAdd*), and returns a stack object (represented by *prev-flags*). The function also writes to the member variable *@gcOHFlags*.

In the following example,

```
@pnNewPen () [@pnColr @pnWidth @pnStyle]
```

when the function **@pnNewPen ()** is called, it does not use or produce any stack objects, but reads the values of the three mentioned member variables.

In the following example,

```
@mesoUpdateLayout () → [@mesoFail]
```

when the function `@mesoUpdateLayout()` is called, it does not use or produce any stack objects, but writes to the member variable `@mesoFail`.

In the following example,

```
@matoUpdateGObjs(pRedraw) [@matoGObjIDList] → success
```

when the function `@matoUpdateGObjs()` is called, it expects one argument on the object stack (represented by *pRedraw*), and returns a stack object (represented by *success*). The function also reads the value of the member variable `@matoGObjIDList`.

3.4.4 All Components

This is not an actual distinct *component*, but forms the basis of all *components* (similar to a base class in object oriented programming). The following members therefore exist in all *components*.

— Member Variables —

@visClass

```
@visClass := class (?)
```

class Any (typically an integer or string) stack object.

Details

This variable is for use by the ETAC programmer for any desired purpose. For example, the programmer may decide to classify graphics objects into various classes. `@visClass` of each graphics object *component* instance can be assigned a number depending on the class to which it belongs. The program can then perform an appropriate class-dependent action on that graphics object. VIS does not inspect or use this variable. ♦

— Member Functions —

@visDelete()

```
@visDelete()
```

Details

Deletes the *component* instance (**except for**: project, drawing plane, and all item *component* instances).

When a *component* instance is deleted, all child *component* instances of the deleted instance are automatically deleted as well.

This function is disabled for project, drawing plane, and all item *component* instances. A project *component* instance can be deleted via the `vis_DeleteProject()` function. A drawing plane *component* instance is automatically deleted when all of its child VIEW *component* instances have been deleted. All item *component* instances are automatically deleted when their parent graphics object *component* instance is deleted.

Other Information

`vis_DeleteProject()` ♦

@visGetChildCIDs()**@visGetChildCIDs ()** → *cid-seq**cid-seq* An integer sequence.**Details**

Gets a sequence of *crucial identifiers* (*cid-seq*) of the immediate children *component* instances of the *component* instance.

cid-seq is a TAC sequence containing the *crucial identifiers* of the immediate children *component* instances of this *component* instance. The sequence is empty if this *component* instance has no children *component* instances. ♦

@visGetCompID()**@visGetCompID ()** → *cid**cid* An integer stack object.**Details**

Gets the *crucial identifier* (*cid*) of the *component* instance.

When a new *component* instance is created, VIS automatically assigns it a new unique *CID* which remains with that instance throughout the *ETAC session*, and which is never used again even after the instance is deleted. The *CID* is used internally by VIS (and also by the ETAC application program) to identify the *component* instance.

Once allocated, the *CID* of the *component* instance cannot be altered.

Other Information

@visGetName () ▪ **@visGetParCID ()** ♦

@visGetDPAncCID()**@visGetDPAncCID ()** → *anc-cid* | ?*anc-cid* An integer stack object.**Details**

Gets the *crucial identifier* (*anc-cid*) of the ancestor drawing plane *component* instance of the *component* instance.

anc-cid is the *crucial identifier* of the ancestor *drawing plane*, or a NULL stack object (?) is returned if this *component* instance does not have an ancestor *drawing plane*. ♦

@visGetName()**@visGetName ()** → *name-ID**name-ID* A string stack object.**Details**

Gets the *name identifier* (*name-ID*) of the *component* instance.

When a new *component* instance is created with an empty string, VIS automatically assigns it a new unique *name identifier* which can later be altered via the `@visRename()` function. The *name identifier* is unique for the *current* project. *name-ID* can be used by the ETAC application program to identify the *component* instance.

Other Information

`@visGetCompID()` ▪ `@visGetParCID()` ▪ `@visRename()` ♦

@visGetParCID()

`@visGetParCID()` ⚙️ → *par-cid*

par-cid An integer stack object.

Details

Gets the *crucial identifier* (*par-cid*) of the parent *component* instance.

Other Information

`@visGetCompID()` ▪ `@visGetName()` ♦

@visGetType()

`@visGetType()` ⚙️ → *type*

type An integer stack object.

Details

Gets the type of the *component* instance. *type* is one of the values listed in [Component Types](#).

Additional Information

[Component Types](#) ♦

@visRename()

`@visRename(pName)` → *success*

name A string stack object.

success An integer stack object containing a logical boolean value.

Details

Renames the *active component*'s *name identifier* with *pName* if possible, otherwise *success* will be false if not.

pName is the desired new *name identifier* for this *component* instance, and can be an empty string to set a default name.

Each *component* instance belonging the same project *component* instance necessarily has a unique *name identifier*. If *pName* already exists for any *component* instance (at any level) of the *current* project *component* instance, then *success* will be false after calling this function, and this *component* instance's *name identifier* will remain unaltered. The function `vis_CompExists()` can be called to determine whether a *component* instance of the same name as *pName* already exists for the same project *component* instance.

The default project *component* instance, named "@VIS@", cannot be renamed.

Other Information

vis_CompExists() ♦

3.4.5 Project

This is the definition of a project *component*. A VIS ETAC program can have more than one project *component* instance, each of which is a top-level *component* instance. A default project *component* instance, named “@VIS@”, always exists and cannot be deleted.

Information

- Initials: prj
- Type: **visC_Project**
- Parent: VIS
- Creator: **vis_Project()**
- Deletion: **vis_DeleteProject()** (not for default project, “@VIS@”)
- Make Current: **vis_SetCurProject()** ▀ user activating a *VIEW*
- Access Current: **vis_Project(?)**

— Member Functions —

@prjSetHelpFnt()

@prjSetHelpFnt (*pHandler* | ?) [⚠] → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the **F1** help handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing help handler function is unregistered (and returned in *prev-handler*), leaving no registered help handler function.

The help event handler function, *pHandler*, must be defined as follows:

```

fnt:(pCompID[*int*] pVIEWCID[*int*] pMenuEvt[*bool*] pMItemGOID[*int*]
      pXPos[*dec*] pYPos[*dec*]) {...}

```

where,

pCompID is an integer containing the *CID* of the project *component* instance by which the handler function was registered.

pVIEWCID is an integer containing the *CID* of the *active VIEW*.

pMenuEvt is an integer stack object containing a logical boolean value. The value is **true** if the **F1** button was tapped while a menu item was highlighted (pMItemGOID will contain the identification of the menu item); if a menu item was not highlighted, the value is **false**.

pMItemGOID is an integer stack object containing a menu identification number (if pMenuEvt is **true**), or a graphics object *component* instance *CID* (if the mouse cursor was over a graphics object), otherwise it will contain the number zero.

pXPos is an integer stack object containing the X-coordinate of the mouse cursor position, in *base units*, mapped to the *drawing plane*.

pYPos is an integer stack object containing the Y-coordinate of the mouse cursor position, in *base units*, mapped to the *drawing plane*.

The help event handler function is called by VIS when the **F1** key is tapped.

Other Information

[2.15.2_Setting and Creating a Handler Function](#) ♦

3.4.6 Drawing Plane

This is the definition of a drawing plane *component*. A *drawing plane* is a “canvas” for which graphics objects are defined. It uses a Cartesian coordinate system (positive X to the right, positive Y upwards) in decimal *base units*. One *base unit* represents one *logical pica* (DPI/6) on the screen under default conditions. A drawing plane *component* instance cannot be deleted explicitly.

Information

- Initials: dp
- Type: **visC_DPlane**
- Parent: [Project](#)
- Creator: **vis_DPlane()**
- Deletion: Automatically when all the *VIEWS* of the drawing plane *component* instance have been deleted.
- Access: **@VIS()** ▪ **@VISX()**
- Make Current: **@VIS()** ▪ **@VISX()** ▪ via **vis_SetCurProject()** ▪ user activating a *VIEW*
- Access Current: **vis_DPlane(?)**

— Member Variables —

@dpBkColr

@dpBkColr := *bk-colr* (?)

bk-colr An integer stack object.

Details

Sets the background colour (*bk-colr*) for the *VIEWS* of the *active* drawing plane *component*.

bk-colr is an RGB colour which can be obtained from the **vis_RGB()** function. If a NULL stack object (?) is used instead of *bk-colr* then the background colour is the default off-white colour (RGB: 255 251 247).

The background colour can be altered at any time via the **@dpSetBkColr()** function, which uses this variable, after this drawing plane *component* instance has been created.

Related Information

@dpSetBkColr()

Other Information

[Standard Colours](#) ▪ [Common Colours](#) ▪ **vis_RGB()** ♦

@dpFTBkColr

@dpFTBkColr := *bk-colr* (?)

bk-colr An integer stack object.

Details

Sets the background colour (*bk-colr*) for formatted text items that will be created for the *active* drawing plane *component*.

bk-colr is an RGB colour which can be obtained from the **vis_RGB()** function. If a NULL stack object (?) is used instead of *bk-colr* then the background colour for a new formatted text item *component* instance is the value of **@dpBkColr**; if the value of **@dpBkColr** is a NULL stack object (?), then the background colour for a new formatted text item is the same as the *current drawing plane* background colour.

The value contained in **@dpFTBkColr** is the default background colour used by the **vis_FormText()** and **vis_FixFormText()** functions when creating a new formatted text item *component* instance.

Other Information

[Standard Colours](#) ▪ [Common Colours](#) ▪ [1.3.10_Formatted Text Item Component](#) ▪ [3.4.12_Formatted Text Item](#) ▪ [vis_RGB\(\)](#) ♦

@dpDFrameBrush

@dpDFrameBrush := *brush-attr*

brush-attr A brush attributes *component* instance (read-only).

Details

Contains the brush attributes *component* instance (*brush-attr*) for the *drawing frame*. Note that *brush-attr* is pre-assigned by VIS and must not be reassigned by any ETAC program. An ETAC program may only change the features of *brush-attr* (at any time).

The *brush-attr* attributes *component* instance is used to automatically fill in the *drawing frame* rectangle, making the *drawing frame* background visible. The default style is empty (**@brStyle := :#BS_NULL;**); other features are the same as for a default brush attributes *component* instance.

Example

The following illustration shows how the **@dpDFrameBrush** variable can be used.

```
vis_DPlane("") [* New drawing plane component instance. *]
{
    @dpDFrameBrush. [* Pre-existing brush attributes component instance. *]
    {
        void @brRepBkMode(@brBkMode := :#OPAQUE:);
        @brNewBrush(@brColr := :#RGB_Blue: @brStyle := :#BS_SOLID:);
    };
};
```

In the example, a new drawing plane *component* instance is created, with the *drawing frame* rectangle showing a solid blue background.

Note that `void @brRepBkMode(@brBkMode := :#OPAQUE:);` is mandatory because the default is for a transparent brush. Also, `@brStyle` needs to be assigned (via `@brNewBrush()`) because the default is a null brush style. If neither of those two variables is changed, then the *drawing frame* rectangle fill will not be visible.

Other Information

[@brRepBkMode\(\)](#) ▪ [@brNewBrush\(\)](#) ▪ [3.4.16_Brush Attributes](#) ♦

@dpDFramePen

`@dpDFramePen := pen-attr`

pen-attr A pen attributes *component* (read-only).

Details

Contains the pen attributes *component* instance (*pen-attr*) for the *drawing frame*. Note that *pen-attr* is pre-assigned by VIS and must not be reassigned by any ETAC program. An ETAC program may only change the features of *pen-attr* (at any time).

The *pen-attr* attributes *component* instance is used to automatically outline the *drawing frame* rectangle, making the *drawing frame* outline visible. The default style is empty (`@pnStyle:= :#PS_NULL:`); other features are the same as for a default pen attributes *component* instance.

Example

The following illustration shows how the `@dpDFramePen` variable can be used.

```
vis_DPlane("") [* New drawing plane component instance. *]
{
    @dpDFramePen. [* Pre-existing pen attributes component instance. *]
    {
        void @pnRepBkMode(@pnBkMode := :#OPAQUE:);
        @pnNewPen(@pnColr := :#RGB_Blue: @pnWidth := 0.1 @pnStyle := :#PS_SOLID:);
    };
};
```

In the example, a new drawing plane *component* instance is created, with the *drawing frame* rectangle showing a solid blue thick outline.

Note that `void @pnRepBkMode(@pnBkMode := :#OPAQUE:);` is mandatory because the default is for a transparent pen. Also, `@pnStyle` needs to be assigned (via `@pnNewPen()`) because the default is a null pen style. If neither of those two variables is changed, then the *drawing frame* outline will not be visible.

Other Information

[@pnRepBkMode\(\)](#) ▪ [@pnNewPen\(\)](#) ▪ [3.4.15_Pen Attributes](#) ♦

@dpOGTrackAttr

`@dpOGTrackAttr := grph-cntxt`

grph-cntxt A graphics context *component* (read-only).

Details

Contains the graphics context *component* instance (*grph-cntxt*) for the object group tracker. Note that *grph-cntxt* is pre-assigned by VIS and must not be reassigned by any ETAC program. An ETAC program may only change the features of *grph-cntxt* (at any time).

The *grph-cntxt component* instance is used for automatically highlighting the object group tracker when selected. The default object group tracker style has a white background, dashed border, and inside boxes.

Example

The following illustration shows how the `@dpOGTrackAttr` variable can be used.

```
vis_DPlane("") [* New drawing plane component instance. *]
{
  [* New brush attributes for the background colour of the object group tracker. *]
  vis_Brush("ObjTrk") {@brColr := :#RGB_PowderBlue:};

  @dpOGTrackAttr. [* Pre-existing graphics context component instance. *]
  {
    [* Indicate that the object tracker and background are to be altered from the
       default. *]
    void @gcRepOHFlags((:!OH_TRACKER_HILITE: &or :!OH_BKGRND_HILITE:) true);

    [* Change the object tracker details as desired. *]
    @gcNewObjTracker(@gcOTStyles := (:!OT_HATCH_BORD: &or :!OT_BOXES_OUT:));

    [* Change the brush attributes to the new attributes for the background. *]
    void @gcRepBrush(@gcBrushID := "ObjTrk");
  };
};
```

In the example, the features of the object group tracker are to be changed as follows: background colour is powder blue (`:#RGB_PowderBlue:`), the tracker border is hatched (`:!OT_HATCH_BORD:`), and the tracker sizing boxes are to be on the outside (`:!OT_BOXES_OUT:`).

Related Information

[3.4.14_Graphics Context](#)

Other Information

[2.12.3_Manipulating Many Graphics Objects Together](#) ▪ [3.4.15_Pen Attributes](#) ▪ [3.4.16_Brush Attributes](#) ♦

— Member Functions —

@dpSetCloseVIEWFnt()

`@dpSetCloseVIEWFnt (pHandler | ?) ✱ → prev-handler | ?`

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the close *VIEW* event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing close *VIEW* handler function is unregistered (and returned in *prev-handler*), leaving no registered close *VIEW* handler function for the *active drawing plane*.

The close *VIEW* handler function, *pHandler*, must be defined as follows:


```
fnt:(pCompID[*int*] pVIEWCID[*int*]) [* => bool|? *] {...}
```

where,

pCompID is an integer containing the *CID* of the drawing plane *component* instance by which the handler function was registered.

pVIEWCID is an integer containing the *CID* of the *active VIEW*.

return: the handler function must return a boolean value (true or false) or a NULL stack object (?). If it returns false, then the *VIEW* is not closed and a beep is produced, if it returns ?, the *VIEW* is not closed (no beep), otherwise it is closed.

The close *VIEW* event handler function is called by VIS when the user closes a *VIEW* belonging to the drawing plane *component* instance by which the handler function was registered. However, if the close *VIEW* handler function that is registered by the @vSetCloseVIEWFnt() function is called, then this handler function is not called for the same *VIEW*.

Other Information

[2.15.2_Setting and Creating a Handler Function](#) - @vSetCloseVIEWFnt() ♦

@dpSetCommandFnt()

```
@dpSetCommandFnt (pHandler | ?) * → prev-handler | ?
```

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the command event handler function, returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing command event handler function is unregistered (and returned in *prev-handler*), leaving no registered command event handler function.

The command event handler function, *pHandler*, must be defined as follows:

```
fnt:(pCompID[*int*] pVIEWCID[*int*] pCmdID[*int*] pEvent[*int*] pCtrl[*int*])
    [* => bool *] {...}
```

where,

pCompID is an integer containing the *CID* of the drawing plane *component* instance by which the handler function was registered.

pVIEWCID is an integer containing the *CID* of the *active VIEW*.

pCmdID is an integer containing the menu item ID if pEvent and pCtrl both contain zero.

pEvent is an integer (currently) containing zero indicating that the command is from a menu item.

pCtrl is an integer (currently) containing zero indicating that the command is from a menu item.

return: the handler function must return a boolean value (true or false). A true return from the handler indicates that the event has been processed, otherwise the operating system will handle the event.

Note that both `pEvent` and `pCtrl` should be checked for a zero value for future compatibility of the handler function.

The command event handler function is called by VIS when a menu item of a *VIEW* belonging to the drawing plane *component* instance by which the handler was registered is selected directly or by a menu shortcut.

Other Information

[2.15.2_Setting and Creating a Handler Function](#) ♦

@dpSetKeyFnt()

@dpSetKeyFnt (*pHandler* | ?) ^{*} → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the key event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing key event handler function is unregistered (and returned in *prev-handler*), leaving no registered key event handler function for the *active drawing plane*.

The key event handler function, *pHandler*, must be defined as follows:

```
fnt: (pCompID[*int*] pVIEWCID[*int*] pKeyCode[*int*] pKeyStates[*int*] pFlags[*int*]
      pTime[*int*]) {...}
```

where,

`pCompID` is an integer containing the *CID* of the drawing plane *component* instance by which the handler function was registered.

`pVIEWCID` is an integer containing the *CID* of the *active VIEW*.

`pKeyCode` is an integer containing the virtual key code ([Virtual Keys](#)) of the key activated (if `!DP_KEY_CHAR` is not set in `pFlags`) or the character code of the key when down (if `!DP_KEY_CHAR` is set in `pFlags`).

`pKeyStates` is an integer containing zero or more modifier key states (`!DP_SHIFT_KEY_DN`, `!DP_CTRL_KEY_DN`, `!DP_ALT_KEY_DN`) and zero or more mouse button states (`!DP_LFT_MSE_DN`, `!DP_MID_MSE_DN`, `!DP_RGT_MSE_DN`). See [Mouse State](#) for details.

`pFlags` is an integer containing a [Key State](#) value.

`pTime` is an integer containing the time of the event, in milliseconds, since the operating system had been started.

The key event handler function is called by VIS when the user activates a keyboard key for a *VIEW* belonging to the drawing plane *component* instance by which the handler function was registered. However, if the key event handler function that is registered by the `@vSetKeyFnt()` function is called and returns `true`, then this handler function is not called for the same *VIEW*.

Additional Information

[Virtual Keys](#) ▪ [Mouse State](#) ▪ [Key State](#)

Other Information**2.15.2 Setting and Creating a Handler Function** ♦**@dpSetMiscFnt()**

@dpSetMiscFnt (*pHandler* | ?) [⚠] → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the miscellaneous event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing miscellaneous event handler function is unregistered (and returned in *prev-handler*), leaving no registered miscellaneous event handler function.

The miscellaneous event handler function, *pHandler*, must be defined as follows:

```
fnt: (pCompID[*int*] pMiscEvtNum[*int*] pArgs[*seq*]) {...}
```

where,

pCompID is an integer containing the *CID* of the drawing plane *component* instance by which the handler function was registered.

pMiscEvtNum is an integer containing the miscellaneous event identification number ([Miscellaneous Events](#)).

pArgs is a TAC sequence containing the various arguments corresponding to the particular value of pMiscEvtNum.

The miscellaneous event handler function is called by VIS for various events as follows:

- Menu highlight event (pMiscEvtNum = :#VIS_MISC_MENU_HILITE:):
This event occurs when a menu item is highlighted.
pArgs indexes:
 - 1 (integer) Menu ID of the highlighted item, or 0 if a popup or separator is highlighted, or -1 if the menu has just been closed.
- User graphics object tracking event (pMiscEvtNum = :#VIS_MISC_USROBJTRACK:):
This event occurs when a tracker rectangle, or one of its handles, is tracking the mouse cursor.
pArgs indexes:
 - 1 (integer) Event indicator values: 0 (tracking event started), 1 (movement of tracker rectangle ended), 2 (resize of tracker rectangle ended), -1 (tracking event was aborted or failed).
 - 2 (integer) Tracking mode values: 1 (objects were moved individually), 2 (objects were moved via the object group tracker).
 - 3 (sequence) List of graphics object *CIDs* (integers) involved in the operation.
 - 4 (decimal) The distance (in *base units*) that the cursor was moved along the X-axis (positive value to the right; negative value to the left). This value is 0.0 if the value at index 1 is either 0 or 2.

- | | | |
|----|-----------|--|
| 5 | (decimal) | The distance (in <i>base units</i>) that the cursor was moved along the Y-axis (positive value upwards; negative value downwards). This value is 0.0 if the value at index 1 is either 0 or 2. |
| 6 | (integer) | The horizontal resize direction: 0 (if the value at index 1 is either 0 or 1),
:#GO_LEFT: (the left border of the tracker rectangle was moved),
:#GO_RIGHT: (the right border of the tracker rectangle was moved). |
| 7 | (integer) | The vertical resize direction: 0 (if the value at index 1 is either 0 or 1),
:#GO_UP: (the top border of the tracker rectangle was moved),
:#GO_DOWN: (the bottom border of the tracker rectangle was moved). |
| 8 | (decimal) | The horizontal resize factor (the new width of the tracker rectangle divided by the original width). This value is 1.0 if the value at index 1 is either 0 or 1. |
| 9 | (decimal) | The vertical resize factor (the new height of the tracker rectangle divided by the original height). This value is 1.0 if the value at index 1 is either 0 or 1. |
| 10 | (decimal) | The mouse cursor X position (in <i>base units</i>) mapped onto the <i>drawing plane</i> when the event occurred. |
| 11 | (decimal) | The mouse cursor Y position (in <i>base units</i>) mapped onto the <i>drawing plane</i> when the event occurred. |
- *Zoom drag* event (pMiscEvtNum = :#VIS_MISC_ZOOMDRAG:):
This event occurs during *zoom dragging*.
pArgs indexes:

1	(integer)	Event indicator values: 0 (<i>zoom dragging</i> event started), 1 (zooming in ended), 2 (zooming out ended), 3 (every 10ms during zooming).
2	(decimal)	<i>VIEW</i> scale.
3	(integer)	Time of the event, in milliseconds, since the operating system had been started.
 - *Image drag* event (pMiscEvtNum = :#VIS_MISC_IMAGEDRAG:):
This event occurs during *image dragging*.
pArgs indexes:

1	(integer)	Event indicator values: 0 (<i>image dragging</i> event started), 1 (<i>image dragging</i> event ended), 3 (every 10ms during <i>image dragging</i>).
2	(integer)	Time of the event, in milliseconds, since the operating system had been started.
 - Drag selection event (pMiscEvtNum = :#VIS_MISC_USRDRAGSEL:):
This event occurs during user drag selection.
pArgs indexes:

1	(integer)	Event indicator values: 0 (selection event started), 1 (selection event ended), -1 (selection event was aborted or failed).
2	(decimal)	The mouse cursor X position (in <i>base units</i>) mapped onto the <i>drawing plane</i> when the event occurred.
3	(decimal)	The mouse cursor Y position (in <i>base units</i>) mapped onto the <i>drawing plane</i> when the event occurred.

- **VIEW** client area hit event (pMiscEvtNum = :#VIS_MISC_HTCLIENT:):

This event occurs when the mouse cursor enters or exits a **VIEW** client area.

pArgs indexes:

- 1 (integer) **VIEW CID**.
- 2 (integer) Hit indicator values: true (cursor entered the client area), false (cursor exited the client area).
- 3 (integer) Time of the event, in milliseconds, since the operating system had been started.

Example

The following illustration shows how the @dpSetMiscFnt() function can be used.

```
vis_DPlane("")
{
    void @dpSetMiscFnt()
        fnt:(pCompID[*int*] pMiscEvtNum[*int*] pArgs[*seq*])
        {
            when pMiscEvtNum =
                :#VIS_MISC_USROBJTRACK: then [* Process user graphics object tracking. *]
            {
                when pArgs%[3, 1] = [* CID of first object being tracked. *]
                    ObjCID1 then
                {
                    [* pArgs%[1] value: 1 => End of 'move' event; 2 => End of
                        'resize' event. *]
                    when pArgs%[1] =
                        1 then {...}
                        2 then {...}
                        -1 then {...} [* Tracking event aborted or failed. *]
                    endwhen;
                }

                ObjCID2 then
                {
                    [* pArgs%[1] value: 1 => End of 'move' event. *]
                    if ( pArgs%[1] = 1 ) then {...} endif;
                }

                ObjCID3 then
                {
                    [* pArgs%[1] value: 2 => End of 'resize' event. *]
                    if ( pArgs%[1] = 2 ) then {...} endif;
                }
            endwhen;

            :#VIS_MISC_MENU_HILITE: then [* Process menu highlighting. *]
            {
                [* pArgs%[1] value: Menu ID of the highlighted item if greater than
                    zero. *]
                ...
            }
        endwhen;
    };
};
```

In the code fragment above, the graphics object tracking (:#VIS_MISC_USROBJTRACK:) and menu highlighting (:#VIS_MISC_MENU_HILITE:) events are processed.

Additional Information

[Miscellaneous Events](#)

Other Information**2.15.2 Setting and Creating a Handler Function ♦****@dpSetMseBtnFnt()**

@dpSetMseBtnFnt (*pHandler* | ?) [⚠] → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the mouse button event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing mouse button event handler function is unregistered (and returned in *prev-handler*), leaving no registered mouse button event handler function for this drawing plane *component* instance.

The mouse button event handler function, *pHandler*, must be defined as follows:

```
fnt: (pCompID[*int*] pChoiceList[*seq*] pMseFlags[*int*] pXPos[*dec*] pYPos[*dec*]
      pTime[*dec*]) {...}
```

where,

pCompID is an integer containing the *CID* of the drawing plane *component* instance by which the handler function was registered.

pChoiceList is a sequence containing a subset of the *CIDs* (integers) of the graphics objects that have been *hit selected* or *micro-hit selected* by the mouse cursor. Which subset of the *hit selected* or *micro-hit selected* graphics objects are included in the sequence depends on which mouse button had been pressed or released, the state of the modifier keys (**SHIFT**, **CTRL**, **ALT**), and the existing selected objects. The algorithm for the subset is complicated when multiple objects have been *hit selected* or *micro-hit selected*, and will not be explained in this document. Note that pChoiceList does not necessarily contain a list of the selected graphics objects; the **@dpGetSelObjIDs()** function gets a sequence of the *CIDs* of the currently selected graphics objects. pChoiceList will be an empty sequence if no graphics object was *hit selected* or *micro-hit selected*.

pMseFlags is an integer containing information on the mouse button events, mouse button states, and the time span in milliseconds since the last mouse button event related to the current mouse button in action. It also contains information on the states of the **SHIFT**, **CTRL**, and **ALT** keys. The various pieces of information can be extracted as described under [Mouse State](#).

pXPos is a decimal containing the mouse cursor X position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pYPos is a decimal containing the mouse cursor Y position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pTime is an integer containing the time of the event, in milliseconds, since the operating system had been started.

The mouse button event handler function is called by VIS when the user activates a mouse button while the mouse cursor is in the client area of a *VIEW* belonging to the drawing plane *component* instance by which the handler function was registered. However, this handler function is not

called if `pChoiceList` would have only one *CID* and a handler function is registered by the `@goSetMseBtnFnt()` function for that graphics object. Note that only the left, middle, and right mouse buttons are recognised by VIS; other mouse buttons are ignored.

Examples

The following code fragments illustrate how to detect various mouse button events within a mouse button event handler function.

```
(1) (@vis_LMBPrs pMseFlags &and to_bool (:!DP_SHIFT_KEY_DN: &and
    pMseFlags))
(2) (@vis_RMBPrs pMseFlags &and not to_bool (:!DP_CTRL_KEY_DN: &and
    pMseFlags))
(3) (@vis_MMBDclk pMseFlags &and ((:!DP_RGT_MSE_MSK: &and pMseFlags)
    = :!DP_RGT_MSE_UP:))
(4) (@vis_LMBRel pMseFlags &and (@vis_MseTimeSpan pMseFlags > 500))
```

Example (1) detects that the left mouse button was pressed while a **SHIFT** key was down.

Example (2) detects that the right mouse button was pressed while both **CTRL** keys were up.

Example (3) detects that the middle mouse button was double-clicked while the right mouse button was up.

Example (4) detects that the left mouse button was released beyond 500 milliseconds after it was pressed.

Additional Information

Mouse State

Other Information

[2.15.2_Setting and Creating a Handler Function](#) ▪ [2.12_Selecting Graphics Objects](#) ▪

[@dpGetSelObjIDs\(\)](#) ▪ [@goSetMseBtnFnt\(\)](#) ♦

@dpSetMseHitFnt()

`@dpSetMseHitFnt(pHandler | ?) ⚠ → prev-handler | ?`

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the object *hit test* event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing object *hit test* event handler function is unregistered (and returned in *prev-handler*), leaving no registered object *hit test* event handler function for this drawing plane *component* instance.

The object *hit test* event handler function, *pHandler*, must be defined as follows:

```
fnt: (pCompID[*int*] pHitList[*seq*] pStateList[*seq*] pMseFlags[*int*] pXPos[*dec*]
    pYPos[*dec*] pTime[*int*]) {...}
```

where,

pCompID is an integer containing the *CID* of the drawing plane *component* instance by which the handler function was registered.

pHitlist is a sequence containing the *CIDs* (integers) of the graphics objects that have been *hit tested* by the mouse cursor. Note that more than one graphics object can be *hit tested* simultaneously.

pStateList is a sequence containing a hit state (boolean integer) for each corresponding *CID* in pHitlist. If a state is true, then the corresponding graphics object indicated in pHitlist has presently been entered, otherwise (the state is false) it has presently been exited by the mouse cursor.

pMseFlags is an integer containing information on the mouse button states and the time span in milliseconds since the last mouse button event. It also contains information on the states of the **SHIFT**, **CTRL**, and **ALT** keys. The various pieces of information can be extracted as described under [Mouse State](#). Note that only the left, middle, and right mouse buttons are recognised by VIS; other mouse buttons are ignored.

pXPos is a decimal containing the mouse cursor X position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pYPos is a decimal containing the mouse cursor Y position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pTime is an integer containing the time of the event, in milliseconds, since the operating system had been started.

The object *hit test* event handler function is called by VIS when the mouse cursor enters or leaves a graphics object while the mouse cursor is in the client area of a *VIEW* belonging to the drawing plane *component* instance by which the handler function was registered. However, this handler function is not called if pHitlist would have only one *CID*, and a handler (via @goSetMseHitFnt()) is registered for that graphics object.

Additional Information

[Mouse State](#)

Other Information

[2.15.2_Setting and Creating a Handler Function](#) - @goSetMseHitFnt() ♦

@dpSetMseMoveFnt()

@dpSetMseMoveFnt(*pHandler* | ?) [⚙] → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the mouse movement event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing mouse movement event handler function is unregistered (and returned in *prev-handler*), leaving no registered mouse movement event handler function.

The mouse movement event handler function, *pHandler*, must be defined as follows:

```
fnt:(pCompID[*int*] pMseFlags[*int*] pXPos[*dec*] pYPos[*dec*] pTime[*int*]) {...}
```

where,

pCompID is an integer containing the *CID* of the drawing plane *component* instance by which the handler function was registered.

pMseFlags is an integer containing information on the mouse button states and the time span in milliseconds since the last mouse button event. It also contains information on the states of the **SHIFT**, **CTRL**, and **ALT** keys. The various pieces of information can be extracted as described under [Mouse State](#).

pXPos is a decimal containing the mouse cursor X position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pYPos is a decimal containing the mouse cursor Y position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pTime is an integer containing the time of the event, in milliseconds, since the operating system had been started.

The mouse movement event handler function is called by VIS when no graphics object has been *hit tested* while the mouse cursor is moving in a *VIEW* belonging to the drawing plane *component* instance by which the handler function was registered.

Additional Information

[Mouse State](#)

Other Information

[2.15.2_Setting and Creating a Handler Function](#) ♦

@dpSetTimerFnt()

```
@dpSetTimerFnt(pHandler | ?) ⚙️ → prev-handler | ?
```

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the timer event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing timer event handler function is unregistered (and returned in *prev-handler*), leaving no registered timer event handler function.

The timer event handler function, *pHandler*, must be defined as follows:

```
fnt:(pCompID[*int*] pVIEWCID[*int*] pTimerID[*int*]) {...}
```

where,

pCompID is an integer containing the *CID* of the drawing plane *component* instance by which the handler function was registered.

pVIEWCID is an integer containing the *CID* of the *VIEW* on which the timer was created. This is the *VIEW* indicated by the *pVIEWIDName* parameter of the **@dpSetTimer()** function.

`pTimerID` is an integer containing the timer identification number by which the timer was created. This is the timer identification indicated by the `pTimerID` parameter of the `@dpSetTimer()` function.

The timer event handler function is continuously called by VIS each time the timer period (indicated by the `pPeriod` parameter of the `@dpSetTimer()` function) has elapsed. The timer can be cancelled by the `@dpCancelTimer()` function.

Related Information

`@dpSetTimer()` ▬ `@dpCancelTimer()`

Other Information

[2.15.2_Setting and Creating a Handler Function](#) ♦

@dpCancelTimer()

`@dpCancelTimer(pVIEWIDName pTimerID)` ✱

`pVIEWIDName` An integer or string stack object.

`pTimerID` An integer stack object.

Details

Cancels the timer identified by `pTimerID` that was created for the specified *VIEW* (`pVIEWIDName`).

`pVIEWIDName` is the *crucial identifier* or *name identifier* of the *VIEW* indicated by the `pVIEWIDName` parameter of the `@dpSetTimer()` function.

`pTimerID` is the timer identification indicated by the `pTimerID` parameter of the `@dpSetTimer()` function.

Once a timer is cancelled, no more timer events occur for that cancelled timer. A new timer can be created with the same identification as a cancelled one.

Related Information

`@dpSetTimer()` ▬ `@dpSetTimerFnt()` ♦

@dpDeselectGObjs()

`@dpDeselectGObjs()`

Details

Deselects all selected graphics objects in the *current drawing plane*.

Other Information

`@dpGetSelObjIDs()` ♦

@dpExtDFrameBord()

@dpExtDFrameBord (*pAmount* *pBordSpec*) 

pAmount A decimal stack object.

pBordSpec An integer stack object.

Details

Extends a *drawing frame* border (*pBordSpec*) by the specified amount (*pAmount*) in *base units*.

pAmount specifies the distance (in *base units*) that the *drawing frame* border is to be extended. A positive value extends the border outwards from the centre of the *drawing frame*; a negative value extends the border inwards.

pBordSpec specifies which border(s) to extend. It must be one of the values indicated in [Drawing Frame Borders](#).

Additional Information

[Drawing Frame Borders](#)

Other Information

[@dpSetDrawFrame\(\)](#) ▪ [@dpGetDrawFramePos\(\)](#) ▪ [@dpGetDrawFrameDim\(\)](#) ▪

[@dpFitDrawFrame\(\)](#) ♦

@dpFitDrawFrame()

@dpFitDrawFrame (*pCalcObjSize*)

pCalcObjSize An integer stack object containing a logical boolean value.

Details

Adjusts the size of the *drawing frame* to enclose all the displayable non-system graphics objects belonging to the *current drawing plane*.

The *window frame* of the *current VIEW* will automatically be made to fit inside the *drawing frame* if necessary, and the *window boundary* will be made identical to the new *drawing frame* size. Invisible graphics objects are included. System graphics objects (marked with the `:!GO_SYSTEM:` flag in `@goFlags`) are excluded.

If *pCalcObjSize* is true, then the internal bounding boxes of the graphics objects will be recalculated before the *drawing frame* is fitted. *pCalcObjSize* is typically set to true.

Other Information

[@dpSetDrawFrame\(\)](#) ▪ [@dpGetDrawFramePos\(\)](#) ▪ [@dpGetDrawFrameDim\(\)](#) ▪

[@dpExtDFrameBord\(\)](#) ♦

@dpGetDispObjIDs()**@dpGetDispObjIDs()** [⚙] → *cid-seq**cid-seq* An integer sequence stack object.**Details**

Gets a sequence (*cid-seq*) of the *crucial identifiers* of all the displayable graphics objects belonging to the *current drawing plane* in the order that they are displayed. Invisible graphics objects are included.

Other Information**@dpGetSelObjIDs()** ♦**@dpGetDrawFrameDim()****@dpGetDrawFrameDim()** [⚙] → *width height**width* A decimal stack object.*height* A decimal stack object.**Details**

Gets the *drawing frame* size (*width*, *height*) of the *current VIEW*.

The *width* and *height* dimensions are in *base units*.

Other Information**@dpGetDrawFramePos()** ♦**@dpGetDrawFramePos()****@dpGetDrawFramePos()** [⚙] → *x-pos y-pos**x-pos* A decimal stack object.*y-pos* A decimal stack object.**Details**

Gets the *drawing frame* position (*x-pos*, *y-pos*) of the *current VIEW*.

x-pos and *y-pos* are in *base units*, and represent the bottom-left vertex point of the *drawing frame* rectangle.

Other Information**@dpGetDrawFrameDim()** ♦**@dpGetDrawPlaneDens()****@dpGetDrawPlaneDens()** [⚙] → *dens**dens* A decimal stack object.**Details**

Gets the *drawing plane* density (*dens*). *dens* is the number of *drawing plane* dots per *base unit*. A distance measurement on the *drawing plane* has an accuracy of 1/*dens base units*.

Other Information**@dpSetDrawFrame()** ♦**@dpGetMaxVHeight()****@dpGetMaxVHeight()** ⚙️ → *height**height* A decimal stack object.**Details**

Gets the nominal height (*height*) of the client area of a maximised *VIEW*. A maximised *VIEW* includes a title bar but no scroll bars.

height is the height of the mapped client area in *base units*.

Other Information**@dpGetMaxVWidth()** ♦**@dpGetMaxVWidth()****@dpGetMaxVWidth()** ⚙️ → *width**width* A decimal stack object.**Details**

Gets the nominal width (*width*) of the client area of a maximised *VIEW* mapped to the *window frame*. A maximised *VIEW* includes a title bar but no scroll bars.

width is the width of the mapped client area in *base units*.

Other Information**@dpGetMaxVHeight()** ♦**@dpGetMaxVXPos()****@dpGetMaxVXPos()** ⚙️ → *x-pos**x-pos* A decimal stack object.**Details**

Gets the nominal X position (*x-pos*) of the client area of a maximised *VIEW* mapped to the *window frame*. A maximised *VIEW* includes a title bar but no scroll bars.

x-pos is the position, in *base units*, of the left edge of the mapped client area. The value is identical to the position of the left border of the *window frame*.

Other Information**@dpGetMaxVYPos()** ♦

@dpGetMaxVYPos()**@dpGetMaxVYPos ()** → *y-pos**y-pos* A decimal stack object.**Details**

Gets the nominal Y position (*y-pos*) of the client area of a maximised *VIEW* mapped to the *window frame*. A maximised *VIEW* includes a title bar but no scroll bars.

y-pos is the position, in *base units*, of the bottom edge of the mapped client area, which is below the top edge position of the *window frame*.

Other Information**@dpGetMaxVXPos ()** ♦**@dpGetObjListBBox()****@dpGetObjListBBox (*pObjIDs*)** → *bound-box**pObjIDs* A mixed integer and string sequence stack object.*bound-box* A decimal sequence stack object.**Details**

Gets the bounding box (*bound-box*) enclosing the specified graphics objects (*pObjIDs*).

pObjIDs contains the *crucial identifiers* or *name identifiers* (can be mixed) of the graphics object *component* instances belonging to the *active drawing plane*.

bound-box contains the smallest rectangle that encloses all the bounding boxes of the graphics objects indicated by *pObjIDs*. The rectangle is a TAC sequence in the form: [*X-min*, *Y-min*, *X-max*, *Y-max*]. Each value of the sequence is in *base units*. ♦

@dpGetSelObjIDs()**@dpGetSelObjIDs ()** → *cid-seq**cid-seq* An integer sequence stack object.**Details**

Gets a sequence (*cid-seq*) of the *crucial identifiers* of the currently selected graphics objects belonging to the *current drawing plane* in the order that they are displayed.

Other Information[2.12_Selecting Graphics Objects](#) ♦**@dpOGTrackSel()****@dpOGTrackSel ()** → *selected**selected* An integer stack object containing a logical boolean value.**Details**

Determines whether or not the object group tracker is selected (*selected*).

selected is true if the object group tracker is currently selected, otherwise it is false.

When selected, the object group tracker is visible, when unselected it is invisible and inactive.

Additional Information

2.12.3_Manipulating Many Graphics Objects Together ♦

@dpSetBkColr()

@dpSetBkColr() ♦ *[@dpBkColr] → bk-colr*

@dpBkColr An integer value.

bk-colr An integer stack object.

Details

Sets the background colour (*@dpBkColr*) for all *VIEWs* belonging to the *active* drawing plane *component*, returning the previous background colour (*bk-colr*).

@dpBkColr is an RGB colour which can be obtained from the **vis_RGB()** function. If a NULL stack object (?) is used instead of *bk-colr* then the background colour is the default off-white colour (RGB: 255 251 247).

bk-colr is the existing background colour before the new colour is set.

Related Information

@dpBkColr

Other Information

vis_RGB() ♦

@dpSetBusyCursor()

@dpSetBusyCursor() ♦ *(pOn) → changed*

pOn An integer stack object containing a logical boolean value.

changed An integer stack object containing a logical boolean value.

Details

Sets or restores (*pOn*) the default cursor for all *VIEWs* of the *current drawing plane* to the busy cursor (hour glass).

pOn determines whether the existing mouse cursor is to be changed to a busy cursor. If *pOn* is true, the mouse cursor is changed; if *pOn* is false, the mouse cursor returns back to the cursor that changed.

changed is true if cursor was changed from a different cursor, otherwise it is false.

This function can be nested with **@dpSetBusyCursor(true)** and **@dpSetBusyCursor(false)** blocks. For each call to **@dpSetBusyCursor(true)**, there must be a corresponding call to **@dpSetBusyCursor(false)**.

While this function is in effect, **@dpSetCursor()** cannot be used to change the mouse cursor.

Other Information

@dpSetCursor() ♦

@dpSetCursor()

@dpSetCursor (*pCursor*) ^{*} → *prev-cursor*

pCursor An integer stack object.

prev-cursor An integer stack object.

Details

Sets the default mouse cursor (*pCursor*) for all *VIEWS* of the *current drawing plane*, returning a handle to the existing one (*prev-cursor*) if appropriate.

pCursor is a handle to a cursor loaded via the **vis_LoadCursor()** function, or one of the mouse cursor identifiers listed in [Mouse Cursor Identifiers](#). It can also be 1 (one) to lock the current default cursor from changes, or -1 to (possibly) unlock the locked cursor to allow changes, or 0 (zero) to unconditionally unlock the cursor and restore the default cursor to the one before it was locked.

prev-cursor is the cursor handle to the existing one if the current cursor was changed by this function. *prev-cursor* is 0 (zero) if the mouse cursor was not changed.

This function can be nested with **@dpSetCursor(1)** and **@dpSetCursor(-1)** blocks. For each call to **@dpSetCursor(1)**, there must be a corresponding call to **@dpSetCursor(-1)**. For each call to **@dpSetCursor(1)**, an internal lock count is incremented. For each call to **@dpSetCursor(-1)**, the internal lock count is decremented. The mouse cursor is locked while the internal lock count is greater than zero. If the cursor is locked, then calls to change it will not succeed. A call to **@dpSetCursor(0)** immediately returns the lock count to zero, unlocking the cursor to allow it to be changed.

While the cursor is locked, **@dpSetBusyCursor()** cannot be used to change the mouse cursor.

Examples

The following illustrations show how the **@dpSetCursor()** function can be used.

```
(1) void @dpSetCursor(:#IDC_CROSS:); void @dpSetCursor(1); ...;
    void @dpSetCursor(-1);

(2) PrevCurs := @dpSetCursor(vis_LoadCursor('MyCursor.cur')); ...;
    void @dpSetCursor(PrevCurs);
```

In example (1), the current default cursor is changed to a cross and locked so that other calls to **@dpSetCursor()** will not change it. At a later time, the cursor is restored back to what it was before it was changed.

In example (2), the current default cursor is changed to the one specified in the file MyCursor.cur, then later restored back to the original cursor.

Other Information

vis_LoadCursor() ▪ **@dpSetBusyCursor()** ♦

@dpSetDrawFrame()**@dpSetDrawFrame** (*pWidth* | ? *pHeight* | ? *pXPos* | ? *pYPos* | ? *pDens* | ?) [☆]*pWidth* A decimal stack object.*pHeight* A decimal stack object.*pXPos* A decimal stack object.*pYPos* A decimal stack object.*pDens* An integer stack object.**Details**

Sets the *drawing frame* size (*pWidth*, *pHeight*), position (*pXPos*, *pYPos*), and *drawing plane* density (*pDens*).

pWidth and *pHeight* are in *base units*, and sets the *drawing frame* size. The values must not be negative.

pXPos and *pYPos* are in *base units*, and set the *drawing frame* position. The position is the bottom-left vertex point of the frame rectangle.

pDens is the density of the *drawing plane*. This is the number of *drawing plane* dots per *base unit*. A distance measurement on the *drawing plane* has an accuracy of $1/\textit{dens}$ *base units*. The default *drawing plane* density is 48 dots per *base unit*. The value for *pDens* must be a reasonable positive number. A value equal to or greater than one sixth of the DPI setting is reasonable. If a NULL stack object (?) is used for *pDens* instead of an integer, then the *drawing plane* density will remain unchanged.

A NULL stack object (?) for any of the first four arguments represents the nominal values for those arguments respectively. The nominal values are the corresponding values for a maximised *VIEW*.

The *drawing plane* density can only be set once for the same drawing plane *component* instance, and can only be set before any *VIEW*s are created, and before the *drawing plane* is populated with graphics objects. Setting the density in other circumstances may produce undesirable consequences.

Examples

The following illustrations show how the **@dpSetDrawFrame()** function can be used.

```
(1) @dpSetDrawFrame(? ? ? ? 100);
(2) @dpSetDrawFrame(@dpGetDrawFrameDim() -150.0 10.0 ?);
(3) @dpSetDrawFrame(100.0 200.0 @dpGetDrawFramePos() 350);
```

In example (1), the *drawing plane* density is changed to 100 dots per *base unit*. This results in a distance measurement accuracy of $1/100$ ($= 0.01$) *base unit*. The *drawing frame* position and size will be their nominal values, which is the case anyway for a new drawing plane *component* instance.

In example (2), the *drawing frame* position is altered, leaving the other values unchanged. Note that the *drawing frame* dimensions are effectively unaltered, because the **@dpGetDrawFrameDim()** function simply returns the existing dimensions, which are set again by the **@dpSetDrawFrame()** function.

In example (3), the *drawing frame* size is altered, leaving the position unchanged. The *drawing plane* density is changed to 350 dots per *base unit*.

Other Information

[@dpGetDrawFramePos\(\)](#) ▪ [@dpGetDrawFrameDim\(\)](#) ♦

@dpSetMinZoomDist()

@dpSetMinZoomDist (*pDist*) [✱]

pDist An integer stack object.

Details

Sets the minimum distance (*pDist*) that the mouse cursor needs to move before a *zoom drag* event is detected.

pDist is a non-negative integer in pixels.

The default distance is that the mouse cursor needs to move before a *zoom drag* event is detected 10 pixels.

Other Information

[@dpSetZoomAngle\(\)](#) ▪ [@dpSetScaleMult\(\)](#) ▪ [2.5.3_Zooming](#) ♦

@dpSetScaleMult()

@dpSetScaleMult (*pMult*) [✱]

pMult A decimal stack object.

Details

Sets the scale multiplier (*pMult*) for multiplying the distance that the mouse cursor has moved for a *zoom drag* event. The default multiplier is 1.0.

Other Information

[@dpSetZoomAngle\(\)](#) ▪ [@dpSetZoomDist\(\)](#) ▪ [2.5.3_Zooming](#) ♦

@dpSetSmooth()

@dpSetSmooth (*pLevel*) [✱] → *prev-level*

pLevel An integer stack object.

prev-level An integer stack object.

Details

Replaces the smoothing level (*pLevel*) of the drawing rendered in the *VIEWS* of the *current drawing plane*, returning the previous smoothing level (*prev-level*). The default smoothing level is 0, which causes default rendering without smoothing.

pLevel must be between 0 and 6 inclusively. The appropriate smoothing level for a *drawing plane* needs to be determined by trial-and-error.

prev-level is the previous smoothing level before replacement.

Additional Information

[2.4.6_Smoothing](#)

Other Information`@dpSmoothShot()` ♦**@dpSetTimer()**`@dpSetTimer (pVIEWIDName pTimerID pPeriod)` ⚠ → *success*

<i>pVIEWIDName</i>	An integer or string stack object.
<i>pTimerID</i>	An integer stack object.
<i>pPeriod</i>	An integer stack object.
<i>success</i>	An integer stack object containing a logical boolean value.

Details

Sets a timer identified by *pTimerID* for a *VIEW* (*pVIEWIDName*). The timer triggers every *pPeriod* milliseconds.

pVIEWIDName is the *crucial identifier* or *name identifier* of the *VIEW* for which the timer is created.

pTimerID is a programmer-defined positive integer that identifies the timer. Each timer must have a unique identifier for a given *VIEW*.

pPeriod is the period, in milliseconds, that the timer is periodically triggered.

success is `true` if the timer was successfully set up, otherwise it is `false`. Timers are a limited operating system global resource; the application program should check the *success* value to verify that a timer has actually been set up.

Timer events are captured by the handler function registered via the `@dpSetTimerFnt()` function.

Example

See [2.19_Animation Emulation](#) for an example on how to set up and use a timer.

Related Information

`@dpSetTimerFnt()` ▪ `@dpCancelTimer()` ♦

@dpSetZoomAngle()`@dpSetZoomAngle (pAngle)` ⚠

<i>pAngle</i>	A decimal stack object.
---------------	-------------------------

Details

Sets the angle (*pAngle*) from the vertical that triggers a *zoom drag* event. *pAngle* is in radians. The default angle is $\pi/4$ (45 degrees).

If *zoom dragging* is in effect, it is triggered when the left mouse button is dragged upwards or downwards while within the specified angle (*pAngle*) from the vertical line relative to the mouse cursor. A mouse movement upwards zooms into the drawing; a movement downwards zooms out of the drawing.

Other Information

`@dpSetMinZoomDist()` ▪ `@dpSetScaleMult()` ▪ [2.5.3_Zooming](#) ♦

@dpSmoothShot()

@dpSmoothShot (*pEffect*) [✱]

pEffect An integer stack object containing a logical boolean value, or an integer stack object.

Details

Sets the smooth shot value (*pEffect*) of the *VIEW*s of the *current drawing plane*. By default, the smooth shot value is *false*, allowing the specified smoothing (via **@dpSetSmooth()**) to occur during the rendering of a drawing in a *VIEW*.

pEffect must be one of the following values:

- 1 Do not do smoothing on next render; afterwards allow the specified smoothing (via **@dpSetSmooth()**) thereafter. This value is used when the normal rendering is desired to be smooth, but a default rendering (no smoothing) is required occasionally.
- 2 Do double smooth shot on the next render only. This causes two successive renderings to be performed when a drawing is about to be rendered. The first rendering is the default rendering (no smoothing). The second rendering is with the specified smoothing (via **@dpSetSmooth()**).
- true Do the specified smoothing (via **@dpSetSmooth()**) on the next render only, then no smoothing thereafter. This value is used when the normal rendering is desired to be the default (no smoothing), but a smooth rendering is required occasionally.
- false Turn smooth shot off, allowing the specified smoothing (via **@dpSetSmooth()**) to occur for each rendering.

This function is typically called from within an event handler function. The drawing is rendered at the earliest convenience after the handler function has ended. At that point, the smooth shot value set by this function takes effect as described above until the value is changed. To make this function take effect immediately, the **@vRefresh()** function is called before the handler function ends but after this function is called.

The reason for having this function is that rendering a drawing smoothly is extremely slow compared with no smoothing (the default rendering). As a result, there would be a noticeable delay before the drawing appears after rendering has begun. This function helps to disguise the delay at certain times by doing a temporary default rendering immediately before the smooth one. For example, if **@dpSetSmooth(2)** is called within an even handler function, the next render after the handler function ends will be the default render (no smoothing) which is very fast, immediately followed by the specified smooth render on top of the first rendering. Because the first render is fast, the user will see an immediate update of the drawing (although not smooth). After a delay, the second rendering then smooths the drawing as previously specified via **@dpSetSmooth()**. Subsequent renderings will continue to be as specified via **@dpSetSmooth()** until this function is called again.

VIS internally uses various smooth shot settings to produce an immediate update of the drawing in a *VIEW* during scrolling and zooming.

Additional Information

[2.4.6_Smoothing](#)

Other Information

@dpSetSmooth() ■ **@vRefresh()** ♦

@dpUpdateVIEWs()

@dpUpdateVIEWs()

Details

Updates (refreshes) the drawing of all *co-VIEWs* of the *current drawing plane* except for the *current VIEW*. If the *drawing plane* has only one *VIEW*, then this function is ineffective.

The @vRefresh() function can be called to update the drawing in a specified *VIEW* if needed.

Note that VIS updates *VIEWs* automatically when changes to graphics objects are made. However, there are occasions when automatic updating is not viable. This function can be called for a *drawing plane* with *co-VIEWs* when automatic updating does not occur.

Other Information

@vRefresh() ♦

3.4.7 VIEW

This is the definition of a *VIEW component*. Each drawing plane *component* instance must have at least one *VIEW component* instance, otherwise the drawing plane instance will be automatically deleted before the vis_DPlane() function returns. Each *VIEW component* instance corresponds to its own single *VIEW*.

Information

- Initials: v
- Type: visC_VIEW
- Parent: Drawing Plane
- Creator: vis_VIEW()
- Deletion: @visDelete() ▪ user deleting corresponding *VIEW*
- Access: @VIS() ▪ @VISX()
- Make Current: @VIS() ▪ @VISX() ▪ user activating corresponding *VIEW*
- Access Current: vis_VIEW(?)

— Member Variables —

@vExStyle

@vExStyle := *ex-styles* (?)

ex-styles An integer stack object containing binary flags.

Details

Sets the operating system extended window style (*ex-styles*) for a new *VIEW*. No default extended style is set.

ex-styles is zero or more of the values listed in [Extended Window Styles](#). Once set, the extended style for the *VIEW* cannot be altered.

Additional Information

[Extended Window Styles](#) ♦

@vFlags

@vFlags := *flags* (0)

flags An integer stack object containing binary flags.

Details

Sets the flags (*flags*) for a new *VIEW*.

flags is zero or more of the values listed in [VIEW Flags](#). Once set, the flags cannot be altered.

Additional Information

[VIEW Flags](#) ♦

@vForceRef

@vForceRef := *bool* (false)

bool An integer stack object containing a logical boolean value.

Details

Determines whether to force a full immediate refresh when **@vRefresh()** is called.

bool is true to force a full immediate refresh of the *current VIEW*, child *VIEWs*, and *co-VIEWs* when the **@vRefresh()** function is called, or false to allow the refresh to occur only for the *current VIEW* at the earliest convenience.

Related Information

[@vRefresh\(\)](#) ♦

@vHeight

@vHeight := *height* (?)

height A decimal stack object.

Details

Sets the height (*height*) of a *VIEW*. The default restored height of the *VIEW* is 1/4 the length of the average of the width and height of the full screen. This value turns out to be (screen width + screen height) / 8.

height is in *logical picas*. When the *VIEW* is created, the value of **@vHeight** represents the restored height of the *VIEW*. *height* can only be a non-negative number or a NULL stack object (?). When used with the **@vRepVIEWRect()** function, the value of **@vHeight** is interpreted depending on the value of **@vSizeMode**.

The *VIEW*'s height can be altered at any time via the **@vRepVIEWRect()** function, which uses this variable, after this *VIEW component* instance has been created.

Additional Information

[@vRepVIEWRect\(\)](#)

Related Information

[@vSizeMode](#) ♦

@vMaxWinScale

@vMaxWinScale := *scale* (3.0)

scale A decimal stack object.

Details

Sets the maximum *zoom dragging VIEW* scale (*scale*) for the *active VIEW*.

scale must be a positive value. An unreasonably large value will have unreasonable consequences. A reasonable value for *scale* can be obtained via the @vCalcMaxVIEWScale() function.

The maximum (and minimum) *VIEW* scale can be changed via the @vNewScaleLimits() function which uses the value of @vMaxWinScale.

When *zoom dragging*, the *VIEW* scale does not exceed the value of @vMaxWinScale. However, an ETAC program can change the *VIEW* scale at any time to any reasonable value.

Related Information

@vNewScaleLimits()

Other Information

@vCalcMaxVIEWScale() ♦

@vMenuID

@vMenuID := *menu-id* (?)

menu-id An integer stack object.

Details

Sets the menu bar identifier (*menu-id*) for a new *VIEW*.

menu-id is an operating system menu handle (HMENU) that identifies a menu bar, obtained from the @MENU or @DLGMENU commands, or from the @mGetMenuID() function. The menu-ID returned from @mGetMenuID() must represent a menu bar not a popup menu.

Examples

The following code fragment illustrates how the @vMenuID variable can be used.

```

vis_DPlane("")
{
  vis_VIEW("")
  {
    @vMenuID := @MENU ? [* Create the menu bar. *]
    {
      @SMENU ... [* Create a submenu. *]
      {
        @MITEM ...; [* Create a submenu item. *]
      };
      @MITEM ...; [* Create a menu item. *]
      @MITEM ...; [* Create a menu item. *]
    };
  };
};

```

The example above illustrates how to directly create a menu bar for a *VIEW*.

The following code fragment illustrates an alternative way to create the menu bar.

```

MenuData := vis_NewMenuData()
@MENU ? [* Create the menu bar, and pass its ID to vis_NewMenuData(). *]
{
  @SMENU ... [* Create a submenu. *]
  {
    @MITEM ...; [* Create a submenu item. *]
  };
  @MITEM ...; [* Create a menu item. *]
  @MITEM ...; [* Create a menu item. *]
};

vis_DPlane("")
{
  vis_VIEW("")
  {
    [* Assign the created menu bar to the VIEW. *]
    @vMenuID := MenuData.@mGetMenuID();
  };
};

```

Note that, in the example above, the **@MENU** command returns a menu ID which is the argument to the **vis_NewMenuData()** function. The menu items can later be manipulated via the **MenuData** variable.

Other Information

@MENU ▪ **@DLGMENU** ▪ **@mGetMenuID()** ▪ **@mDestroyMenu()** ▪ **vis_NewMenuData()** ♦

@vMinWinScale

@vMinWinScale := *scale* (1.0)

scale A decimal stack object.

Details

Sets the minimum *zoom dragging VIEW* scale (*scale*) for the *active VIEW*.

scale must be a positive value. An unreasonably small value will have unreasonable consequences. A reasonable value for *scale* can be obtained via the **@vCalcMinVIEWScale()** function.

The minimum (and maximum) *VIEW* scale can be changed via the `@vNewScaleLimits()` function which uses the value of `@vMinWinScale`.

When *zoom dragging*, the *VIEW* scale does go below the value of `@vMinWinScale`. However, an ETAC program can change the *VIEW* scale at any time to any reasonable value.

Related Information

`@vNewScaleLimits()`

Other Information

`@vCalcMinVIEWScale()` ♦

@vParVIEWID

`@vParVIEWID := view-id (?)`

view-id An integer or string stack object.

Details

Sets the *crucial identifier* or *name identifier* (*view-id*) of the parent or owner *VIEW* of the *VIEW* being created if it is to have a parent or owner *VIEW*. Once allocated, the parent or owner of the created *VIEW* cannot be removed or changed.

A *VIEW* may be a top-level *VIEW*, may have an owner *VIEW*, or may be a child *VIEW* to another *VIEW*. These relations are defined by the operating system, and will not be explained in this document.

If the *VIEW* being created is:

- to be a top-level *VIEW*, then `@vParVIEWID` should not be assigned, and the created VIEW instance must not have the `:!WS_CHILD:` style set;
- to have an owner *VIEW*, then the *crucial identifier* or *name identifier* (*view-id*) of the owner VIEW *component* instance is assigned to `@vParVIEWID`, and the created and owner VIEW instances must not have the `:!WS_CHILD:` style set;
- to be a child *VIEW*, then the *crucial identifier* or *name identifier* (*view-id*) of the parent VIEW *component* instance is assigned to `@vParVIEWID`, and the created VIEW instance must have the `:!WS_CHILD:` style set.

Once a parent or owner of a *VIEW* is set, VIS does not allow that particular relation to be altered. Also, once a *VIEW* is created as a top-level *VIEW*, it will remain a top-level *VIEW*. If the parent or owner of a *VIEW* needs to be changed, then new *VIEWs* having the desired parent\owner relationship need to be created with features of the existing ones before the existing ones are deleted.

The parent or owner of a *VIEW* typically belongs to a different *drawing plane* than the *VIEW* itself.

Example

The following code fragment illustrates how the `@vParVIEWID` variable can be used.

```
vis_DPlane("") [* Create a drawing plane. *]
{
    vis_VIEW("Parent") {...}; [* Create a parent VIEW. *]
};

vis_DPlane("") [* Create another drawing plane. *]
{
    [* Create a VIEW that will be a child VIEW of the 'Parent' VIEW. *]
    vis_VIEW("Child") {@vParVIEWID := "Parent"; @vStyle := (&or ... :!WS_CHILD:)};
};
```

The code fragment above creates a child *VIEW* (“Child”) with a parent *VIEW* (“Parent”) from a different *drawing plane*.

Related Information

[@vParVIEWProjID](#)

Other Information

[Normal Window Styles](#) ♦

@vParVIEWProjID

@vParVIEWProjID := *proj-id* (?)

proj-id An integer or string stack object.

Details

Sets the *crucial identifier* or *name identifier* (*proj-id*) of the project *component* instance of the parent or owner VIEW *component* instance (if there is one) if that parent or owner instance does not belong to the *current* project. In other words, if the VIEW *component* instance being created has a parent or owner VIEW *component* instance that does not belong to the *current* project, then @vParVIEWProjID needs to be set to the *crucial identifier* or *name identifier* of the project *component* instance that it does belong to.

Example

The following code fragment illustrates how the @vParVIEWProjID variable can be used.

```

vis_Project("Project1") [* Create a project. *]
{
    vis_DPlane("") [* Create a drawing plane. *]
    {
        vis_VIEW("Parent") {...}; [* Create a parent VIEW. *]
    };
};

vis_Project("Project2") [* Create another project. *]
{
    vis_DPlane("") [* Create a drawing plane. *]
    {
        [* Create a VIEW that will be a child VIEW to the 'Parent' VIEW
           belonging to a different project. *]
        vis_VIEW("Child")
        {
            @vParVIEWID := "Parent"; [* Note: 'Parent' belongs to 'Project1'. *]

            [* This needs to be set because 'Parent' does not belong to 'Project2'. *]
            @vParVIEWProjID := "Project1";

            @vStyle := (&or ... :!WS_CHILD:);
        };
    };
};

```

In the code fragment above, the parent *VIEW* (“Parent”) belongs to a different project than the project (“Project 2”) of the child *VIEW* (“Child”), so `@vParVIEWProjID` must be set to the project (“Project 1”) to which the parent *VIEW* belongs. The same situation applies to owner *VIEW*s as well. If the parent and child (or owned) *VIEW*s belong to the same project, then `@vParVIEWProjID` need not be set.

Related Information

`@vParVIEWID` ♦

@vSizeMode

`@vSizeMode` := *mode* (0)

mode An integer stack object.

Details

Determines the interpretation of `@vWidth` and `@vHeight` when the `@vRepVIEWRect()` function is called. If the *VIEW* is being created, this variable has no effect.

mode can have only one of the following values for the interpretation of `@vWidth` and `@vHeight`:

- 0 specified size;
- 1 minimum user resize;
- 2 maximum user resize;
- ? ignore the size (return value from `@vRepVIEWRect()` only).

Additional Information

`@vRepVIEWRect()`

Other Information

`@vWidth` ▪ `@vHeight` ♦

@vStyle

@vStyle := *styles* (?)

styles An integer stack object containing binary flags.

Details

Sets the operating system window style (*styles*) for a new *VIEW*. The default style is `:!WS_DEFAULTVIEW:`.

styles is zero or more of the values listed in [Normal Window Styles](#). Once set, the style for the *VIEW* cannot be altered.

Additional Information

[Normal Window Styles](#) ♦

@vTitle

@vTitle := *title* (?)

title A string stack object.

Details

Sets the title (*title*) in the title bar of a *VIEW*. The default title is: RunETAC.

title can be a NULL stack object (?).

The *VIEW*'s title can be altered at any time via the `@vRepTitle()` function, which uses this variable, after this *VIEW component* instance has been created.

Related Information

`@vRepTitle()` ♦

@vWidth

@vWidth := *width* (?)

width A decimal stack object.

Details

Sets the width (*width*) of a *VIEW*. The default restored width of the *VIEW* is 1/4 the length of the average of the width and height of the full screen. This value turns out to be (screen width + screen height) / 8.

width is in *logical picas*. When the *VIEW* is created, the value of `@vWidth` represents the restored width of the *VIEW*. *width* can only be a non-negative number or a NULL stack object (?). When used with the `@vRepVIEWRect()` function, the value of `@vWidth` is interpreted depending on the value of `@vSizeMode`.

The *VIEW*'s width can be altered at any time via the `@vRepVIEWRect()` function, which uses this variable, after this *VIEW component* instance has been created.

Additional Information

`@vRepVIEWRect()`

Related Information**@vSizeMode** ♦**@vXPos****@vXPos** := *x-pos* (?)*x-pos* A decimal stack object.**Details**

Sets the *VIEW*'s left edge position (*x-pos*) relative to the screen or parent *VIEW*. The default left edge position is at position zero.

x-pos is in *logical picas*, and is relative to the left edge of the screen if the *VIEW* is not a child *VIEW*, or to the left edge of the client area of its parent *VIEW* if the *VIEW* is a child *VIEW*. *x-pos* can be negative, positive, or zero. A positive value is towards the right. *x-pos* can also be a NULL stack object (?).

The *VIEW*'s left edge position can be altered at any time via the **@vRepVIEWRect()** function, which uses this variable, after this *VIEW component* instance has been created.

Related Information**@vRepVIEWRect()** ♦**@vYPos****@vYPos** := *y-pos* (?)*y-pos* A decimal stack object.**Details**

Sets the *VIEW*'s top edge position (*y-pos*) relative to the screen or parent *VIEW*. The default top edge position is at position zero.

y-pos is in *logical picas*, and is relative to the top edge of the screen if the *VIEW* is not a child *VIEW*, or to the top edge of the client area of its parent *VIEW* if the *VIEW* is a child *VIEW*. *y-pos* can be negative, positive, or zero. A positive value is towards the bottom. *y-pos* can also be a NULL stack object (?).

The *VIEW*'s top edge position can be altered at any time via the **@vRepVIEWRect()** function, which uses this variable, after this *VIEW component* instance has been created.

Related Information**@vRepVIEWRect()** ♦

— Member Functions —

@vSetCloseVIEWFnt()

@vSetCloseVIEWFnt (*pHandler* | ?) [✱] → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the close *VIEW* event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing close *VIEW* handler function is unregistered (and returned in *prev-handler*), leaving no registered close *VIEW* handler event function for this *VIEW*.

The close *VIEW* handler function, *pHandler*, must be defined as follows:

```
fnt:(pCompID[*int*]) [* => bool|? *] {...}
```

where,

pCompID is an integer containing the *CID* of the *VIEW component* instance by which the handler function was registered.

return: the handler function must return a boolean value (true or false) or a NULL stack object (?). If it returns false, then the *VIEW* is not closed and a beep is produced, if it returns ?, the *VIEW* is not closed (no beep), otherwise it is closed.

The close *VIEW* event handler function is called by VIS when the user closes the *VIEW* associated with this *VIEW component* instance. However, if a close *VIEW* event handler function is also registered by the @dpSetCloseVIEWFnt() function, then that handler function is not called if this one is for the same *VIEW*.

Other Information

[2.15.2_Setting and Creating a Handler Function](#) - @dpSetCloseVIEWFnt() ♦

@vSetKeyFnt()

@vSetKeyFnt (*pHandler* | ?) [✱] → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the key event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing key event handler function is unregistered (and returned in *prev-handler*), leaving no registered key event handler function for this *VIEW*.

The key event handler function, *pHandler*, must be defined as follows:

```
fnt:(pCompID[*int*] pKeyCode[*int*] pKeyStates[*int*] pFlags[*int*] pTime[*int*])
[* => bool*] {...}
```

where,

pCompID is an integer containing the *CID* of the VIEW *component* instance by which the handler function was registered.

pKeyCode is an integer containing the virtual key code ([Virtual Keys](#)) of the key activated (if `!DP_KEY_CHAR` is not set in pFlags) or the character code of the key when down (if `!DP_KEY_CHAR` is set in pFlags).

pKeyStates is an integer containing zero or more modifier key states (`!DP_SHIFT_KEY_DN`, `!DP_CTRL_KEY_DN`, `!DP_ALT_KEY_DN`) and zero or more mouse button states (`!DP_LFT_MSE_DN`, `!DP_MID_MSE_DN`, `!DP_RGT_MSE_DN`). See [Mouse State](#) for details.

pFlags is an integer containing a [Key State](#) value.

pTime is an integer containing the time of the event, in milliseconds, since the operating system had been started.

return: the handler function must return a boolean value (true or false). If it returns true, and a key event handler function is also registered by the `@dpSetKeyFnt()` function, then that handler function is not called for the same *VIEW*. If it returns false, and a key event handler function is also registered by the `@dpSetKeyFnt()` function, then that handler function is called next after this one.

The key event handler function is called by VIS when the user activates a keyboard key while the *VIEW* associated with this VIEW *component* instance is *current*.

Additional Information

[Virtual Keys](#) - [Mouse State](#) - [Key State](#)

Other Information

[2.15.2_Setting and Creating a Handler Function](#) - `@dpSetKeyFnt()` ♦

@vSetSysCmdFnt()

`@vSetSysCmdFnt (pHandler | ?) ⚠ → prev-handler | ?`

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the system command event handler function.

Registers the system command event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing system command event handler function is unregistered (and returned in *prev-handler*), leaving no registered system command event handler function for this *VIEW*.

The system command event handler function, *pHandler*, must be defined as follows:

```
fnt:(pCompID[*int*] pSysCmdID[*int*]) {...}
```

where,

pCompID is an integer containing the *CID* of the VIEW *component* instance by which the handler function was registered.

pSysCmdID is an integer containing a value listed in [System Command Notification Codes](#).

The system command *VIEW* event handler function is called by VIS when the user releases the left mouse button after performing certain operations on the *current VIEW*.

Additional Information

[System Command Notification Codes](#)

Other Information

[2.15.2_Setting and Creating a Handler Function](#) ♦

@vSetVIEWActFnt()

```
@vSetVIEWActFnt (pHandler | ?) * → prev-handler | ?
```

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the *VIEW* activation event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing *VIEW* activation event handler function is unregistered (and returned in *prev-handler*), leaving no registered *VIEW* activation event handler function for this *VIEW*.

The *VIEW* activation event handler function, *pHandler*, must be defined as follows:

```
fnt:(pCompID[*int*] pActive[*bool*]) {...}
```

where,

pCompID is an integer containing the *CID* of the VIEW *component* instance by which the handler function was registered.

pActive is an integer stack object containing a logical boolean value. If the *VIEW* is not a child *VIEW*, the value is true when the *VIEW* is being activated, and false when it is being deactivated. If the *VIEW* is a child *VIEW*, the value is true when the user clicks the *VIEW*'s title bar, or when the *VIEW* is activated, moved, or sized; the value is never false for a child *VIEW*.

Note that the operating system deactivates a *VIEW* before activating another one.

The *VIEW* activation event handler function is called by VIS when the *VIEW* is being activated or deactivated (if it is not a child *VIEW*); or when the user clicks the *VIEW*'s title bar, or when the *VIEW* is activated, moved, or sized (if it is a child *VIEW*). A *VIEW* becomes activated when it changes from an inactive state to an active state, and vice versa for when the *VIEW* becomes deactivated. Only a top-level *VIEW* can be active. When the user is interacting with a child *VIEW*, the operating system activates the top-level parent *VIEW* related to that child *VIEW*.

Other Information[2.15.2_Setting and Creating a Handler Function](#) ♦**@vSetVIEWSizeFnt()**

```
@vSetVIEWSizeFnt(pHandler | ?) * → prev-handler | ?
```

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the *VIEW* size change event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing *VIEW* size change event handler function is unregistered (and returned in *prev-handler*), leaving no registered *VIEW* size change event handler function for this *VIEW*.

The *VIEW* size change event handler function, *pHandler*, must be defined as follows:

```
fnt: (pCompID[*int*] pMode[*int*] pWidth[*dec*] pHeight[*dec*]) {...}
```

where,

pCompID is an integer containing the *CID* of the *VIEW component* instance by which the handler function was registered.

pMode is an integer containing a value listed in [VIEW Sizing Modes](#).

pWidth is a decimal containing the width (in *base units*) of the *VIEW*'s client area mapped onto the *window frame* width.

pHeight is a decimal containing the height (in *base units*) of the *VIEW*'s client area mapped onto the *window frame* height.

The *VIEW* size change event handler function is called by VIS when a *VIEW* changes its size as a result of user or programmed action.

Additional Information[VIEW Sizing Modes](#)**Other Information**[2.15.2_Setting and Creating a Handler Function](#) - [@vGetWinClientDim\(\)](#) ♦**@vCalcMaxVIEWScale()**

```
@vCalcMaxVIEWScale() * → scale
```

scale A decimal stack object.

Details

Calculates the maximum *VIEW* scale (*scale*) so that one *drawing plane* dot maps to one screen pixel.

scale contains the calculated scale which is typically assigned to **@vMaxWinScale** when the *VIEW* is being created.

This function should be called, after the `@vCalibrateScale()` function has been called, to get the new maximum *VIEW* scale. The *VIEW* scale can be set higher than *scale*, but the translation accuracy of the mouse cursor position on the screen to the *drawing plane* may be very much reduced if so.

Note that a higher *drawing plane* density will result in a higher value of *scale*.

Other Information

`@vMaxWinScale` ▪ `@vCalibrateScale()` ♦

@vCalcMinVIEWScale()

`@vCalcMinVIEWScale()` ⚙️ → *scale*

scale A decimal stack object.

Details

Calculates the minimum *VIEW* scale (*scale*) to retain the *VIEW*'s client area within the *window boundary*.

scale contains the calculated scale which is typically assigned to `@vMinWinScale` when the *VIEW* is being created.

This function should be called, after the `@vSetWinBound()` function has been called, to get the new minimum *VIEW* scale. The *VIEW* scale can be set lower than *scale*, but the *window boundary* will automatically enlarge to keep the *VIEW*'s client area within it if so.

For correct results, the *client rectangle* area should be able to fit within the nominal *window boundary* area before this function is called. This can be achieved, if necessary, by temporarily setting the maximum *VIEW* scale (via the `@vCalcMaxVIEWScale()` function) before calling this function, then setting the scale back. However, if this function is called immediately after the `@vSetWinBound()` function is called (or called from within the *initialiser procedure* of a new *VIEW component* instance), then it will not be necessary to temporarily change the *VIEW* scale.

Other Information

`@vMinWinScale` ▪ `@vSetWinBound()` ▪ `@vCalcMaxVIEWScale()` ♦

@vCalibrateScale()

`@vCalibrateScale(pWidth pHeight pXPos pYPos)`

pWidth A decimal stack object.

pHeight A decimal stack object.

pXPos A decimal stack object.

pYPos A decimal stack object.

Details

Sets the specified *window frame* size (*pWidth*, *pHeight*) and position (*pXPos*, *pYPos*) for a *VIEW* scale of 1.0 (100% scale).

For a new *VIEW component* instance, the *window frame* size is the maximum possible width of a *VIEW*'s client area on the screen; the *reference frame* size is the same as the *window frame* size. This results in a default *VIEW* scale of 1.0. (The *VIEW* scale is defined as the ratio of the *reference frame* width to the *window frame* width.)

This function is called to calibrate the *VIEW* scale to 1.0 for a larger or smaller drawing on the screen by changing the *window frame* and *reference frame* sizes (they will both still be the same size as each other). Changing the *window frame* size implies that a different portion of the drawing will appear under the *window frame*, resulting in a different size of the drawing appearing in the *VIEW*.

pWidth is the new *window frame* and *reference frame* width in *base units*. The value must be a positive number.

pHeight is the new *window frame* and *reference frame* height in *base units*. The value must be a positive number. The heights of the *window frame* and *reference frame* are not currently used in the calculation of the *VIEW* scale.

pXPos is the new position, in *base units*, of the left border of the *window frame*.

pYPos is the new position, in *base units*, of the bottom border of the *window frame*.

Note that, for future compatibility, the *pHeight* value should be calculated so that the ratio of *pHeight* to *pWidth* is the same as the ratio of the nominal *window frame* height to the nominal *window frame* width.

After calling this function, the *VIEW* scale will become 1.0, but, because the width of the *window frame* has changed to *pWidth*, the size of the drawing presented in the *VIEW* would be different than it was previously with a *VIEW* scale of 1.0; the *VIEW* scale is now calibrated to the new drawing size presented in the *VIEW*.

If the `@vCalcMaxVIEWScale()` function is being used in a program, then that function should be called after `@vCalibrateScale()` is called.

Examples

The following illustrations show how the `@vCalibrateScale()` function can be used.

```
[* Allocate width and height variables. *]
WFWidth :- ?; WFHeight :- ?;

vis_DPlane("")
{
    vis_VIEW("MyVIEW") {...};
};

@VIS("MyVIEW").
{
    [* Get the current window frame width and height. *]
    WFHeight := WFWidth := @vGetWinFrameDim();

    [* Calibrate the scale to the new window frame width and height for a VIEW scale
        of 100%. *]
    @vCalibrateScale((2.0 * WFWidth) (2.0 * WFHeight) @vGetWinFramePos());
};
```

In the example above, a normal *VIEW* scale of 0.5 is desired to be calibrated to a scale of 1.0. This is achieved by setting the *window frame* and *reference frame* sizes to be double their original size via the `@vCalibrateScale()` function. With the double size of the *window frame*, twice the extent of the drawing appears in the *VIEW*, but the *VIEW* scale remains 100% because the two frames are still the same size relative to each other. Notice that the *window frame* position is unchanged.

In the following example, the programmer desires that the same coverage of the same drawing appear on different screens with different resolutions at the same *VIEW* scale.


```

[* Allocate width and height variables. *]
WFWidth :- ?; WFHeight :- ?;

vis_DPlane("")
{
    vis_VIEW("MyVIEW") {...};
};

@VIS("MyVIEW").
{
    [* Get the current window frame width and height. *]
    WFHeight := WFWidth := @vGetWinFrameDim();

    [* Calibrate the scale to the new window frame width and height for a VIEW scale
        of 100%. *]
    @vCalibrateScale(120.0 (120.0 * (WFHeight / WFWidth)) @vGetWinFramePos());
};

```

In the code above, the programmer decided that an extent of 120.0 *base units* of a drawing will now correspond to a *VIEW* scale of 100%, even if the drawing appears on different screens with different resolutions for the same *VIEW* scale. For example, if a rectangular graphics object is 120.0 *base units* wide, the whole rectangle width will appear on any screen with any resolution at 100% scale. Normally (without scale calibration), only half the rectangle width will appear on a screen with half the horizontal screen resolution of a different screen at the same *VIEW* scale for both screens. Note that the *pHeight* argument is calculated so that the height-to-width ratio remains the same as the original (for future compatibility).

Other Information

@vCalcMaxVIEWScale() ♦

@vDPToVIEWPos()

@vDPToVIEWPos (*pXPos* *pYPos*) → *x-pos* *y-pos*

pXPos A decimal stack object.

pYPos A decimal stack object.

x-pos A decimal stack object.

y-pos A decimal stack object.

Details

Converts a *drawing plane* position (*pXPos*, *pYPos*) to a corresponding *VIEW* client position (*x-pos*, *y-pos*).

pXPos is the horizontal coordinate of a position (in *base units*) on the *drawing plane*.

pYPos is the vertical coordinate of a position (in *base units*) on the *drawing plane*.

x-pos is the horizontal coordinate of a position (in *logical picas*) on the client area of the *VIEW* relative to the left edge of the client area. A positive value is towards the right.

y-pos is the vertical coordinate of a position (in *logical picas*) on the client area of the *VIEW* relative to the top edge of the client area. A positive value is towards the bottom.

This function is useful to map the mouse cursor position (received in *base units*) to the corresponding point on the *VIEW*'s client area for use with such functions as @vShowPopup() which requires a position in *logical picas*. The function is also useful to create a borderless *VIEW* containing a message that appears near a desired graphics object (for example, to emulate a tool tip); a *VIEW* requires *logical picas* for its position. ♦

@vDrawMBar()

@vDrawMBar() → *success*

success An integer stack object containing a logical boolean value.

Details

Draws the menu bar associated with the *VIEW*.

success is true if the menu bar was successfully drawn, otherwise it is false.

The function assumes that the *VIEW* has a menu bar associated with it. Menu bars are normally redrawn automatically by the operating system, but there may be rare occasions where they are not drawn properly.

Other Information

@vMenuID ♦

@vEnable()

@vEnable(*pEnable* | ?) → *prev-state*

pEnable An integer stack object containing a logical boolean value, or a mark stack object.

prev-state An integer stack object containing a logical boolean value.

Details

Sets and returns the current enableity state of the *VIEW*.

pEnable is true to enable the *VIEW*, or false to disable it. The value can also be a mark stack object (*<!x>*, where *x* is 1 to 7) to restore previous enableity state. If a NULL stack object (?) is used instead of *pEnable* then the *VIEW*'s enableity state remains unchanged and its current enableity state is returned (*prev-state*).

prev-state is true if the *VIEW* was previously enabled, otherwise it is false.

Examples

The following illustrations show how the **@vEnable()** function can be used.

```

(1) @VIS("MyVIEW").{PrevState := @vEnable(false)}; ...;
    @VIS("MyVIEW").{void @vEnable(PrevState)};

(2) @VIS("MyVIEW").{void @vEnable(false)}; ...; void @vEnable(!3)};

(3) CurState := @VIS("MyVIEW").@vEnable(?);

```

In example (1), the *VIEW* identified by MyVIEW is disabled, and the *VIEW*'s previous enableity state (which could be either enabled or disabled) is saved in PrevState. Later, the previous enableity state is restored from PrevState. Saving the enableity state in this way allows a *VIEW* to be disabled and enabled in a nested fashion.

In example (2), the *VIEW* identified by MyVIEW is disabled. However, the previous enableity state is not saved. Later, the enableity state is restored (via the call to **@vEnable(!3)**) to what it was before it was disabled. Changing the enableity state in this way does not allow a *VIEW* to be disabled and enabled in a nested fashion. However, this method avoids using a variable to save the *VIEW*'s previous enableity state (the previous state is saved internally to VIS). Note that any mark stack object from 1 to 7 can be used.

In example (3), the current enableity state of the *VIEW* identified by `<MyVIEW>` is obtained without changing the *VIEW*'s enableity state.

Other Information

`@vShow()` ♦

@vGetWinBoundDim()

`@vGetWinBoundDim()` → *width height*

width A decimal stack object.

height A decimal stack object.

Details

Gets the *window boundary* dimensions (*width*, *height*) of the *VIEW*.

width is the width (in *base units*) of the *window boundary* associated with the *current VIEW*.

height is the height (in *base units*) of the *window boundary* associated with the *current VIEW*.

Additional Information

[1.5.4_Window Boundary](#)

Other Information

`@vGetWinBoundPos()` ▪ `@vSetWinBound()` ♦

@vGetWinBoundPos()

`@vGetWinBoundPos()` → *x-pos y-pos*

x-pos A decimal stack object.

y-pos A decimal stack object.

Details

Gets the *window boundary* position (*x-pos*, *y-pos*) of the *VIEW*.

x-pos is the position of the left border (in *base units*) of the *window boundary* associated with the *current VIEW*.

y-pos is the position of the bottom border (in *base units*) of the *window boundary* associated with the *current VIEW*.

The *window boundary* position is the bottom-left point of the *window boundary*.

Additional Information

[1.5.4_Window Boundary](#)

Other Information

`@vGetWinBoundDim()` ▪ `@vSetWinBound()` ♦

@vGetWinClientDim()**@vGetWinClientDim()** → *width height**width* A decimal stack object.*height* A decimal stack object.**Details**

Gets the *VIEW*'s client area dimensions (*width*, *height*) mapped onto the *window frame*.

width is the width (in *base units*) of the *client rectangle* associated with the *current VIEW*.

height is the height (in *base units*) of the *client rectangle* associated with the *current VIEW*.

Additional Information

[1.5.3_Client Rectangle](#)

Other Information

@vGetWinClientPos() ♦

@vGetWinClientPos()**@vGetWinClientPos()** → *x-pos y-pos**x-pos* A decimal stack object.*y-pos* A decimal stack object.**Details**

Gets the *VIEW*'s client area position mapped onto the *window frame*.

x-pos is the position of the left border (in *base units*) of the *client rectangle* associated with the *current VIEW*.

y-pos is the position of the bottom border (in *base units*) of the *client rectangle* associated with the *current VIEW*.

The *client rectangle* position is the bottom-left point of the *client rectangle*.

Additional Information

[1.5.3_Client Rectangle](#)

Other Information

@vGetWinClientDim() ♦

@vGetWinFrameDim()**@vGetWinFrameDim()** → *width height**width* A decimal stack object.*height* A decimal stack object.**Details**

Gets the *window frame* dimensions (*width*, *height*) of the *VIEW*.

width is the width (in *base units*) of the *window frame* associated with the *current VIEW*.

height is the height (in *base units*) of the *window frame* associated with the *current VIEW*.

Additional Information

1.5.2 Window Frame

Other Information

@vGetWinFramePos() ♦

@vGetWinFramePos()

@vGetWinFramePos() → *x-pos y-pos*

x-pos A decimal stack object.

y-pos A decimal stack object.

Details

Gets the *window frame* position (*x-pos*, *y-pos*) of the *VIEW*.

x-pos is the position of the left side (in *base units*) of the *window frame* associated with the *current VIEW*.

y-pos is the position of the bottom side (in *base units*) of the *window frame* associated with the *current VIEW*.

The *window frame* position is the bottom-left point of the *window frame*.

Additional Information

1.5.2 Window Frame

Other Information

@vGetWinFrameDim() ♦

@vMoveWinFrame()

@vMoveWinFrame(*pXPos pYPos*)

pXPos A decimal stack object.

pYPos A decimal stack object.

Details

Moves the *window frame* of the *VIEW* to the desired position (*pXPos*, *pYPos*).

pXPos is the new position of the left side (in *base units*) of the *window frame* associated with the *current VIEW*.

pYPos is the new position of the bottom side (in *base units*) of the *window frame* associated with the *current VIEW*.

The *window frame* position is the bottom-left point of the *window frame*. The *window frame* may not end up in the desired position, but as close as possible to it. This is because VIS does not allow the *window frame* to exist outside the *window boundary* rectangle. A program can call the @vGetWinFramePos() function after calling this function to determine whether the new location of the *window frame* ended up at the desired position.

Moving the *window frame* scrolls the drawing in the *VIEW*.

Additional Information

1.5.2 Window Frame

Other Information

@vGetWinFramePos() ♦

@vNewSBar()	
@vNewSBar(<i>pStyle</i> <i>pExStyle</i> <i>pTSCNID</i>) → <i>sbar-data</i>	
<i>pStyle</i>	An integer stack object.
<i>pExStyle</i>	An integer stack object.
<i>pTSCNID</i>	An integer or string stack object.
<i>sbar-data</i>	An ETAC data object.

Details

Creates a status bar in the *VIEW*'s client area.

pStyle is zero or more of the values listed in [Status Bar Styles](#) and [Normal Window Styles](#). Once set, the style for the status bar cannot be altered.

pExStyle is zero or more of the values listed in [Extended Window Styles](#). Once set, the extended style for the status bar cannot be altered.

pTSCNID is the *crucial identifier* or *name identifier* of a text style *component* instance for text in the status bar, or 0 (zero) to indicate the default text style.

sbar-data is a [Status Bar](#) ETAC data object by means of which the status bar can be manipulated.

A *VIEW* can have no more than one status bar. If the *VIEW* already has a status bar, then this function will destroy it before creating a new one.

Note that, if a *VIEW* has a horizontal scroll bar, the operating system will place a bottom status bar above the scroll bar within the *VIEW*'s client area.

Example

The following illustration shows how the @vNewSBar() function can be used.

```
[* Allocate a status bar variable to hold the status bar ETAC data object. *]
StatBar :- ?;

vis_DPlane("")
{
    vis_VIEW("MyVIEW") {...};

    [* Create a text style component instance for the status bar text. *]
    vis_TextStyle("MyTxStyle") {@tsItalic := true; @tsColr := :#RGB_Tomato:};

    @VIS("MyVIEW").
    {
        [* Create a status bar for the VIEW. *]
        StatBar := @vNewSBar((: !SBARS_SIZEGRIP: &or : !WS_CLIPSIBLINGS:) 0 "MyTxStyle");
    };
};
```

In the example above, a status bar is created for the VIEW *component* instance named MyVIEW. The status bar will have a sizing grip by which the user can size the *VIEW*, and will have the text written in italic with a “tomato” colour. The : !WS_CLIPSIBLINGS: style is set for the status bar because otherwise sibling *VIEW*s may “dirty” the status bar on some occasions.

Additional Information

[Status Bar Styles](#) ▪ [Normal Window Styles](#) ▪ [Extended Window Styles](#) ▪ [2.5.5_Status Bar](#) ▪ [Status Bar](#)

Other Information

[vis_NewSBarData\(\)](#) ♦

@vNewScaleLimits()

@vNewScaleLimits() [[@vMinWinScale](#) [@vMaxWinScale](#)]

[@vMinWinScale](#) A decimal value.

[@vMaxWinScale](#) A decimal value.

Details

Replaces the minimum and maximum *zoom dragging VIEW* scales with the specified ones ([@vMinWinScale](#), [@vMaxWinScale](#)).

[@vMinWinScale](#) contains the minimum *VIEW* scale for *zoom dragging*. The value must be a positive number.

[@vMaxWinScale](#) contains the maximum *VIEW* scale for *zoom dragging*. The value must be a positive number.

This function replaces the minimum and maximum *VIEW* scale limits for *zoom dragging* to the values specified in [@vMinWinScale](#) and [@vMaxWinScale](#), respectively, for the *active VIEW*.

Examples

The following illustrations show how the **@vNewScaleLimits()** function can be used.

```
(1) @vNewScaleLimits(@vMaxWinScale := @vCalcMaxVIEWScale());
(2) @vNewScaleLimits(@vMinWinScale := 0.5 @vMaxWinScale := 10.0);
```

In example (1), the maximum *VIEW* scale is calculated via the **@vCalcMaxVIEWScale()** function, and set to the member variable **@vMaxWinScale** before the **@vNewScaleLimits()** function is called. The existing value in **@vMinWinScale** is used as the minimum *VIEW* scale.

In example (2), both the minimum and maximum *VIEW* scale values are explicitly set to the member variables first before the **@vNewScaleLimits()** function is called.

Additional Information

[@vMinWinScale](#) ▪ [@vMaxWinScale](#)

Other Information

[@vCalcMinVIEWScale\(\)](#) ▪ [@vCalcMaxVIEWScale\(\)](#) ♦

@vNoBoundChk()

@vNoBoundChk() ([pOn](#))

[pOn](#) An integer stack object containing a logical boolean value.

Details

Specifies whether to check ([pOn](#)) that the right and bottom borders of the *VIEW*'s client area mapped onto the *window frame* are within the *window boundary*.

pOn is true to disable *boundary checking* for the right and bottom borders of the *client rectangle*, or false to enable checking. By default, checking is done.

With *boundary checking* (the default for a *VIEW*), no part of the *client rectangle* can ever exist outside the *window boundary*. However, this function allows the right and bottom borders of the *client rectangle* to exist beyond the right and bottom borders, respectively, of the *window boundary* when the *client rectangle* dimension becomes larger than the *window boundary* dimension in the respective direction. As a consequence, the *window boundary* does not expand to accommodate the *client rectangle* as it does with *boundary checking*. If a *client rectangle* border exists beyond the respective *window boundary* border, then the corresponding scroll bar will be invisible.

Other Information

1.5.4 Window Boundary ♦

@vRefresh()

@vRefresh () [*@vForceRef*]

Details

Refreshes the *current VIEW* and all of its child *VIEWs* immediately.

@vForceRef contains true to force a full immediate refresh of the *current VIEW*, child *VIEWs*, and *co-VIEWs*. If this variable contains false, then only the *current VIEW* is refreshed at the earliest convenience.

VIS automatically refreshes appropriate parts of the *VIEW* when program control is returned back to the operating system after an ETAC program makes certain changes to various *component* instances. However, in some cases, such as when certain attribute *component* instances are changed, VIS does not refresh the *VIEW*. In those cases, the programmer can call this function instead to refresh the *VIEW*.

Note that this function refreshes all of the visible client area of the *VIEW*. For large client areas with smoothing, refreshing a *VIEW* uses considerable resources, so this function should be used sparingly.

Note that this function will always set *@vForceRef* to false before it returns.

Additional Information

@vForceRef

Other Information

@dpUpdateVIEWs () ▪ *@goSetRedraw ()* ♦

@vRepMBar()

@vRepMBar (*pMenuID* | ?) → *menu-id*

pMenuID An integer stack object.

menu-id An integer stack object.

Details

Replaces the menu bar assigned to the *active VIEW* with the specified menu (*pMenuID*), or removes the existing menu bar, and returns the existing menu bar identifier (*menu-id*).

pMenuID is the menu bar identifier of the menu bar to replace the existing one (if any). A menu bar identifier is an operating system menu handle (HMENU), obtained from the `@MENU` or `@DLGMENU` commands, or from the `@mGetMenuID()` function. The menu ID returned from `@mGetMenuID()` must represent a menu bar not a popup menu. *pMenuID* can also be 0 (zero) to remove (but not delete) the existing menu bar. If a NULL stack object (?) is used instead of *pMenuID*, then only the existing menu bar identifier is returned (*menu-id*).

menu-id is the menu bar identifier of the existing menu bar (HMENU), or is 0 (zero) if the *VIEW* does not have a menu bar.

This function does not destroy the existing menu bar. The `@mDestroyMenu()` function destroys a menu bar (or any menu).

Note that a *VIEW* cannot have more than one menu bar, and a child *VIEW* cannot have any.

This function is useful if an application program needs to change the menu bar of a *VIEW* on different occasions during the program.

Examples

The following illustrations show how the `@vRepMBar()` function can be used.

```
(1) MenuData := vis_NewMenuData(@VIS("MyVIEW").@vRepMBar(?));
(2) void vis_NewMenuData(@VIS("MyVIEW").@vRepMBar(0)).@mDestroyMenu();
```

In example (1), the menu bar of the *VIEW* identified by *MyVIEW* is obtained to create a menu data object which is assigned to *MenuData*. The menu items can later be manipulated via the *MenuData* variable. Note, that in this case, the menu bar of *MyVIEW* is not replaced.

Example (2) shows how to destroy a menu bar. Note that `@vRepMBar(0)` does not itself destroy the menu bar; the menu bar is destroyed by the `@mDestroyMenu()` function. The return value of `@mDestroyMenu()` is discarded in this example.

Other Information

`@MENU` ▪ `@DLGMENU` ▪ `@mGetMenuID()` ▪ `@mDestroyMenu()` ▪ `@vMenuID` ▪
`vis_NewMenuData()` ♦

@vRepScale()	
<code>@vRepScale(pScale ?) → prev-scale</code>	
<i>pScale</i>	A decimal stack object.
<i>prev-scale</i>	A decimal stack object.

Details

Replaces the existing *VIEW* scale of the *current VIEW* with the specified one (*pScale*), and returns the existing scale (*prev-scale*).

pScale is a non-negative number for the new *VIEW* scale. If *pScale* is 0.0 (zero), then the existing *VIEW* scale is re-established (this has the same effect as calling this function with the existing *VIEW* scale). If a NULL stack object (?) is used instead of *pScale*, then only the existing *VIEW* scale is returned (*prev-scale*).

prev-scale is the existing *VIEW* scale before replacement.

The *VIEW* scale is defined as the ratio of the *reference frame* width to the *window frame* width. The default *VIEW* scale is 100%.

Other Information

@vZoomToSel () ▀ @vZoomToAll () ◆

@vRepTitle()	
@vRepTitle ()	[@vTitle] → prev-title
@vTitle	A string value.
prev-title	A string stack object.

Details

Replaces the existing title in the title bar of the *active VIEW* with the specified one (*@vTitle*), and returns the existing *VIEW*'s title (*prev-title*).

@vTitle contains the new *VIEW*'s title. If *@vTitle* contains a NULL stack object (?), then only the existing title is returned (*prev-title*).

prev-title is the existing *VIEW*'s title before replacement.

Additional Information

@vTitle ◆

@vRepVIEWRect()	
@vRepVIEWRect ()	[@vXPos @vYPos @vWidth @vHeight @vSizeMode] → pos-size
@vXPos	A decimal value.
@vYPos	A decimal value.
@vWidth	A decimal value.
@vHeight	A decimal value.
@vSizeMode	An integer value.
pos-size	A decimal sequence stack object.

Details

Replaces the existing *active VIEW*'s position (*@vXPos*, *@vYPos*) and size (*@vWidth*, *@vHeight*) on the screen, and returns the existing position and size (*pos-size*).

@vXPos contains the *VIEW*'s left edge position in *logical picas*. If *@vXPos* contains a NULL stack object (?), then the *VIEW*'s existing left edge position is assumed.

@vYPos contains the *VIEW*'s top edge position in *logical picas*. If *@vXPos* contains a NULL stack object (?), then the *VIEW*'s existing top edge position is assumed.

@vWidth contains the *VIEW*'s width in *logical picas*. If *@vWidth* contains a NULL stack object (?), then the *VIEW*'s existing width is assumed.

@vHeight contains the *VIEW*'s height in *logical picas*. If *@vHeight* contains a NULL stack object (?), then the *VIEW*'s existing height is assumed.

@vSizeMode contains a value used to interpret the meaning of the values in *@vWidth* and *@vHeight*. *@vSizeMode* must contain one of the following values:

- 0 Replaces the existing *VIEW* rectangle as specified by *@vXPos*, *@vYPos*, *@vWidth*, and *@vHeight* (this is the default);

- 1 Only sets the minimum user resize width and/or height of the *VIEW* rectangle as specified by *@vWidth* and *@vHeight* (the values contained in *@vXPos* and *@vYPos* are ignored; if *@vWidth* or *@vHeight* contains 0.0, then that value is ignored);
- 2 Only sets the maximum user resize width and/or height of the *VIEW* rectangle as specified by *@vWidth* and *@vHeight* (the values contained in *@vXPos* and *@vYPos* are ignored; if *@vWidth* or *@vHeight* contains 0.0, then that value is ignored);
- ? Ignores the values in *@vXPos*, *@vYPos*, *@vWidth*, and *@vHeight*, and returns only the existing position and size of the *VIEW* and its client area (*pos-size*).

pos-size is a sequence containing the existing position and size of the *VIEW* and its client area. The elements of the sequence are in *logical picas* as follows:

[*x-pos-v*, *y-pos-v*, *width-v*, *height-v*, *x-pos-c*, *y-pos-c*, *width-c*, *height-c*],

where

- 1 *x-pos-v* is the *VIEW*'s left edge position relative to the left edge of the screen or (for a child *VIEW*) the left edge of the parent *VIEW*'s client area (a positive value is towards the right),
- 2 *y-pos-v* is the *VIEW*'s top edge position relative to the top edge of the screen or (for a child *VIEW*) the top edge of the parent *VIEW*'s client area (a positive value is towards the bottom),
- 3 *width-v* is the *VIEW*'s width,
- 4 *height-v* is the *VIEW*'s height,
- 5 *x-pos-c* is the *VIEW*'s client area left edge position relative to the *VIEW*'s left edge (a positive value is towards the right),
- 6 *y-pos-c* is the *VIEW*'s client area top edge position relative to the *VIEW*'s top edge (a positive value is towards the bottom),
- 7 *width-c* is the *VIEW*'s client area width,
- 8 *height-c* is the *VIEW*'s client area height.

The values contained in *@vXPos* and *@vYPos* are relative to the respective left and top edges of the screen if the *active VIEW* is not a child *VIEW*, or relative to the respective left and top edges of the parent *VIEW*'s client area if the *active VIEW* is a child *VIEW*.

The minimum and maximum user resize values limit the respective minimum and maximum size of the *VIEW* when the user tries to resize it.

Changing the position or size of a maximised *VIEW* with *@vSizeMode* = 0 does not affect the restored position and size of the *VIEW*. To change the restored position or size of the *VIEW*, the function *@vShow()* with an argument of *:#SW_RESTORE:* needs to be called before this function is called.

Note that this function will always set *@vSizeMode* to 0 (zero) before it returns.

Examples

The following illustrations show how the *@vRepVIEWRect()* function can be used.

```

(1) @VIS("MyVIEW").{void @vRepVIEWRect(@vXPos := ? @vYPos := 10.0
    @vWidth := ? @vHeight := ?);};
(2) @VIS("MyVIEW").
    {@vUpdateVIEWRect(); @vYPos := 10.0; void @vRepVIEWRect();};
(3) @VIS("MyVIEW").
    {void @vRepVIEWRect(@vWidth := 30.0 @vHeight := ? @vSizeMode := 1);};
(4) @VIS("MyVIEW").
    {void @vRepVIEWRect(@vWidth := 30.0 @vHeight := 0.0
        @vSizeMode := 1);};
(5) PosSize := @VIS("MyVIEW").@vRepVIEWRect(@vSizeMode := ?);
    LeftBordWidth := PosSize%[5];
(6) PosSize := @VIS("MyVIEW").@vRepVIEWRect(@vSizeMode := ?);
    RightSBWidth := (- PosSize%[3] PosSize%[7] (2.0 * PosSize%[5]));

```

In example (1), the top edge of the *VIEW* identified by MyVIEW is moved to be 10.0 *logical picas* from the top of the screen, assuming that the *VIEW* is not a child *VIEW*, or from the top of the parent *VIEW*'s client area if the *VIEW* is a child *VIEW*. The *VIEW*'s width, height, and horizontal position remain unaltered. The value contained in @vSizeMode will be 0 (zero) if not explicitly altered. The value returned from @vRepVIEWRect() is discarded.

Example (2) has the same effect as for example (1), except that the values contained in @vXPos, @vYPos, @vWidth, and @vHeight contain the final position and size of the *VIEW*. Note that the @vUpdateVIEWRect() function updates the values contained in @vXPos, @vYPos, @vWidth, and @vHeight to contain the current position and size of the *VIEW*.

In example (3), the *VIEW* identified by MyVIEW is set so that the user will not be able to resize the width of the *VIEW* to be smaller than 30.0 *logical picas*, and the height of the *VIEW* to be smaller than the current height. The current position and size of the *VIEW* remains unaltered.

Example (4) has the same effect as for example (3), except that there is no restriction on the user changing the height of the *VIEW*.

Example (5) calculates the left border width of the *VIEW* identified by MyVIEW. The left border thickness is the same value as the left edge of the *VIEW*'s client area (assuming that there is no scroll bar on the left side of the *VIEW*). The current position and size of the *VIEW* remains unaltered. The *VIEW* has the same thickness for all its borders.

Example (6) calculates the right scroll bar width of the *VIEW* identified by MyVIEW (assuming that there is a visible scroll bar on the right side of the *VIEW*, otherwise the calculation result will be zero). The calculation is the difference between the *VIEW*'s width and its client area width, minus twice the thickness of the *VIEW*'s border. The current position and size of the *VIEW* remains unaltered.

Additional Information

@vXPos ▪ @vYPos ▪ @vWidth ▪ @vHeight ▪ @vSizeMode

Other Information

@vUpdateVIEWRect() ♦

@vSetKBFocus()**@vSetKBFocus()****Details**

Sets the keyboard focus to the *active VIEW*, activating it or its parent *VIEW* (if the *VIEW* is a child *VIEW*). ♦

@vSetWinBound()**@vSetWinBound(pLeft pTop pRight pBottom)***pLeft* A decimal stack object.*pTop* A decimal stack object.*pRight* A decimal stack object.*pBottom* A decimal stack object.**Details**

Sets the *window boundary* rectangle relative to the *drawing frame*.

pLeft is the distance of the left border of the *window boundary* relative to the left border of the *drawing frame*. A positive value is towards the left. The value is in *base units*.

pTop is the distance of the top border of the *window boundary* relative to the top border of the *drawing frame*. A positive value is towards the top. The value is in *base units*.

pRight is the distance of the right border of the *window boundary* relative to the right border of the *drawing frame*. A positive value is towards the right. The value is in *base units*.

pBottom is the distance of the bottom border of the *window boundary* relative to the bottom border of the *drawing frame*. A positive value is towards the bottom. The value is in *base units*.

Additional Information

[1.5.4 Window Boundary](#)

Other Information

@vGetWinBoundPos() ▪ **@vGetWinBoundDim()** ♦

@vShow()**@vShow(pShow | ?) → prev-state***pShow* An integer stack object.*prev-state* An integer stack object.**Details**

Shows the *active VIEW* in the specified visibility state (*pShow*), and returns its previous visibility state (*prev-state*).

pShow is one of the values listed in [Window Visibility](#). If a NULL stack object (?) is used instead of *pShow*, then only the existing visibility state of the *VIEW* is returned (*prev-state*).

prev-state is the existing visibility state of the *VIEW* before replacement. The value will only be one of `:#SW_HIDE:`, `:#SW_SHOWNORMAL:`, `:#SW_SHOWMINIMIZED:`, `:#SW_SHOWMAXIMIZED:`, indicating that the *VIEW* is invisible, restored, minimised, or maximised, respectively.

Note that, in some cases, the *VIEW* needs to be hidden before it is correctly shown as specified under [Window Visibility](#). For example, if the *VIEW* is already visible but not activated, then `@vShow(:#SW_SHOW:)` will have no effect; the *VIEW* will not become activated as specified (the specifications are based on the operating system documentation). The code fragment to activate the *VIEW* is: `void @vShow(:#SW_HIDE:); void @vShow(:#SW_SHOW:);`.

Additional Information

[Window Visibility](#)

Other Information

`@vEnable()` ▪ `@vSetKBFocus()` ♦

@vShowPopup()

`@vShowPopup (pMenuID pTPMflags pHorLoc pVerLoc pRectSeq | ?) → id-bool`

<i>pMenuID</i>	An integer stack object.
<i>pTPMflags</i>	An integer stack object containing binary flags.
<i>pHorLoc</i>	A decimal stack object.
<i>pVerLoc</i>	A decimal stack object.
<i>pRectSeq</i>	A decimal sequence stack object.
<i>id-bool</i>	An integer stack object containing a logical boolean value, or an integer stack object.

Details

Displays a pop-up menu anywhere on the screen relative to the *active VIEW*'s client area.

pMenuID is the menu identifier of the pop-up menu to show. A menu identifier is an operating system menu handle (HMENU), obtained from the `@POPUP` command, or from the `@mGetMenuID()` function, or from `<@mGetMItem(...) % [3]>`. The menu-ID returned from `@mGetMenuID()` must represent a pop-up or submenu, not a menu bar.

pTPMflags is zero or more of the values listed in [Popup Menu](#).

pHorLoc is the position of the left edge of the pop-up menu relative to the left edge of the *VIEW*'s client area. A positive value is towards the right. The value is in *logical picas*.

pVerLoc is the position of the top edge of the pop-up menu relative to the top edge of the *VIEW*'s client area. A positive value is towards the bottom. The value is in *logical picas*.

pRectSeq is a sequence specifying an area of the screen that the pop-up menu should not overlap. The elements of the sequence are in *logical picas* relative to the *VIEW*'s client area as follows: `[min-X, min-Y, max-X, max-Y]`. If a NULL stack object (?) is used instead of *pRectSeq*, then the pop-up menu is not restricted from any area on the screen.

id-bool is the menu-item identifier of the user-selected item if *pTPMflags* contains `:!TPM_RETURNCMD:`; if the user cancels the pop-up menu without making a selection, or if an error occurs, then the value is `false` instead. Otherwise, the value is `true` if the function succeeds and `false` otherwise.

Note that, if *pTPMflags* does not contain `:!TPM_RETURNCMD:`, then the user-selected item identifier will be received by the event handler function registered by the `@dpSetCommandFnt()` function (if one was registered).

Examples

The following illustrations show how the `@vShowPopup()` function can be used.

The code fragment below creates a pop-up menu to be displayed when any mouse button is pressed or released on a graphics object.

```
gMenuData := vis_NewMenuData()
@POPUP ? [* Create the pop-up menu, and pass its ID to vis_NewMenuData(). *]
{
    @SMENU ...; [* Create a submenu. *]
    @MITEM "Copy" 1001 ...; [* Create a menu item. *]
    @SMENU ...; [* Create a submenu. *]
};

vis_DPlane("")
{
    vis_VIEW("MyVIEW") {};

    vis_GObject("")
    {
        @goFlags := :!GO_HIT_TEST_ON;;

        [* Register the mouse button event handler for the graphics object. *]
        void @goSetMseBtnFnt()
        fnt:(pCompID[*int*] pMseFlags[*int*] pXPos[*dec*] pYPos[*dec*] pTime[*int*])
        {
            ItemID :- ?;

            [* Event handler code. *]
            @VIS("MyVIEW").
            {
                ItemID := @vShowPopup(gMenuData.@mGetMenuID() :!TPM_RETURNCMD:
                                     @vDPToVIEWPos(pXPos pYPos) ?);

                [* Process the selected menu items. *]
                when ItemID is
                ...
                1001 then {...} [* Process the 'Copy' menu item. *]
                ...
                endwhen;
            };
        };

        [* Create the item component instances. *]
        ...
    };
};
```

When the mouse button event handler is called (as a result of a mouse button press or release on the graphics object), the pop-up menu is displayed, and returns the identification (1001) of the user-selected menu item because of the `:!TPM_RETURNCMD:` flag. The menu item selection is then processed. Normally, the pop-up is made to display when the mouse button is released.

The following code fragment is the same as the one above except that a submenu of the menu bar is used for the pop-up menu which is processed as for the menu bar.

```

gMenuData := vis_NewMenuData()
@MENU ? [* Create the menu bar, and pass its ID to vis_NewMenuData(). *]
{
    @SMENU ...; [* Create a submenu. *]
    @SMENU "Edit" 1000 ... [* Create a submenu. *]
    {
        @MITEM "Copy" 1001 ...; [* Create a submenu item. *]
    };
    @SMENU ...; [* Create a submenu. *]
};

vis_DPlane("")
{
    vis_VIEW("MyVIEW")
    {
        [* Assign the created menu bar to the VIEW. *]
        @vMenuID := gMenuData.@mGetMenuID();
    };

    [* Create and register the command event handler function. *]
    void @dpSetCommandFnt()
        fnt:(pCompID[*int*] pVIEWCID[*int*] pCmdID[*int*] pEvent[*int*]
            pCtrl[*int*]) [* => bool *]
    {
        if ( = pEvent pCtrl 0 ) then [* A menu event occurred. *]
        {
            [* Process the selected menu items (pCmdID). *]
            when pCmdID is
                ...
                1001 then {...} [* Process the 'Copy' menu item. *]
                ...
            endwhen;
        }
        endif;

        true; [*RETURN*]
    };
};

vis_GObject("")
{
    @goFlags := :!GO_HIT_TEST_ON;;

    [* Register the mouse button event handler for the graphics object. *]
    void @goSetMseBtnFnt()
        fnt:(pCompID[*int*] pMseFlags[*int*] pXPos[*dec*] pYPos[*dec*] pTime[*int*])
    {
        [* Event handler code. *]
        @VIS("MyVIEW").
        {
            void @vShowPopup(gMenuData.@mGetMItem(1000 true)%[3] 0
                @vDPToVIEWPos(pXPos pYPos) ?);
        };
    };

    [* Create the item component instances. *]
    ...
};
};

```

The command handler function processes all the items of the menu bar's submenus. When the mouse button event handler is called (as a result of a mouse button press or release on the graphics object), the pop-up menu is displayed, and the user-selected menu item is processed by the command handler function (instead of being processed directly) because of the absence of the `:!TPM_RETURNCMD:` flag. Notice that the pop-up menu is an actual submenu of the menu bar.

Additional Information[Popup Menu](#)**Other Information**[2.5.4_Menu](#) ▪ [@dpSetCommandFnt\(\)](#) ▪ [@POPUP](#) ▪ [@mGetMenuID\(\)](#) ▪ [@mGetMItem\(\)](#) ♦**@vUpdateVIEWRect()****@vUpdateVIEWRect()** → [[@vXPos](#) [@vYPos](#) [@vWidth](#) [@vHeight](#)][@vXPos](#) A decimal value.[@vYPos](#) A decimal value.[@vWidth](#) A decimal value.[@vHeight](#) A decimal value.**Details**

Updates the member variables [@vXPos](#), [@vYPos](#), [@vWidth](#), [@vHeight](#) of the *active VIEW* with the position and size of the *VIEW* rectangle on the screen.

[@vXPos](#) contains the *VIEW*'s left edge position in *logical picas*.

[@vYPos](#) contains the *VIEW*'s top edge position in *logical picas*.

[@vWidth](#) contains the *VIEW*'s width in *logical picas*.

[@vHeight](#) contains the *VIEW*'s height in *logical picas*.

The values contained in [@vXPos](#) and [@vYPos](#) are relative to the respective left and top edges of the screen if the *active VIEW* is not a child *VIEW*, or relative to the respective left and top edges of the parent *VIEW*'s client area if the *active VIEW* is a child *VIEW*.

Additional Information[@vXPos](#) ▪ [@vYPos](#) ▪ [@vWidth](#) ▪ [@vHeight](#)**Other Information**[@vRepVIEWRect\(\)](#) ♦**@vZoomToAll()****@vZoomToAll()****Details**

Zooms to all the visible non-system graphics objects in the *current drawing plane*.

The *VIEW* scale of the *current VIEW* is modified to be as large as practical to fit all of the visible non-system graphics objects (those not marked with the `!GO_SYSTEM:` flag in [@goFlags](#)). If the *current drawing plane* contains no graphics objects, then this function has no effect.

Other Information[@vZoomToSel\(\)](#) ▪ [@vRepScale\(\)](#) ♦

@vZoomToSel()

@vZoomToSel (*pObjID* | ?)

pObjID An integer or string stack object.

Details

Zooms to selected graphics objects in the *current drawing plane*.

pObjID is the *crucial identifier* or *name identifier* of the object to zoom to if there are no selected objects.

The *VIEW* scale of the *current VIEW* is modified to be as large as practical to fit all of the selected graphics objects. If the *current drawing plane* contains no graphics objects, and a NULL stack object (?) is used instead of *pObjID*, then this function has no effect.

Examples

The following illustrations show how the **@vZoomToSel ()** function can be used.

```
(1) @vZoomToSel (?) ;
(2) @vZoomToSel ("MyObj") ;
```

In example (1), the *current VIEW* will zoom into all the selected graphics objects. If no graphics objects are selected, then nothing will happen.

In example (2), the graphics object identified by MyObj is to be used as the default graphics object to zoom to if no graphics objects are selected. Otherwise, the *current VIEW* will zoom into all the selected graphics objects.

Other Information

@vZoomToAll () ▪ **@vRepScale ()** ▪ **@visGetCompID ()** ♦

3.4.8 Graphics Object

This is the definition of a graphics object *component*. Each drawing plane *component* instance can have zero or more graphics object *component* instances. A graphics object *component* instance is the parent of other graphics object *component* instances or item *component* instances.

A graphics object has an internally calculated bounding box which is what is highlighted when the graphics object is selected by the user or program.

Each *drawing plane* is associated with an object storage list, and an object display list. If a graphics object *component* instance is on the object storage list, it is not displayed and does not take up space in the *drawing plane*; if it is on the display list, it can potentially be displayed and does take up space in the *drawing plane*.

The *current drawing plane* must be the same as the *drawing plane* within which the graphics object *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: go
- Type: **visC_GObject**
- Parent: **Drawing Plane** ▪ Graphics Object
- Creator: **vis_GObject ()**
- Deletion: **@visDelete ()**
- Access: **@VIS ()** ▪ **@VISX ()**

— Member Variables —

@goContextID**@goContextID** := *ctx-id* (?)*ctx-id* An integer or string stack object.**Details**

Sets the identifier (*ctx-id*) of the graphics context *component* instance for this graphics object *component* instance when being created if one is required.

ctx-id is the *crucial identifier* or *name identifier* of a previously created graphics context *component* instance. If a NULL stack object (?) is used instead of *ctx-id*, then the graphics object will not have a graphics context.

The graphics context *component* instance is used to highlight the graphics object if the context is set for highlighting.

The graphics context can be altered at any time via the **@goRepContext()** function, which uses this variable, after this graphics object *component* instance has been created.

Related Information

[3.4.14 _Graphics Context](#) ▪ **@goRepContext()** ♦

@goDisplay**@goDisplay** := *bool* (**true**)*bool* An integer stack object containing a logical boolean value.**Details**

Determines whether the graphics object is displayed or not when it is being created.

bool is **true** to display the graphics object after it is created, or **false** to create the graphics object hidden and without taking up space in the *drawing plane*.

If **@goDisplay** is set to **false**, the graphics object *component* instance is created on the object storage list rather than the display list.

Each drawing plane *component* instance has an internal object storage list to contain graphics object *component* instances. An ETAC program can relocate graphics object *component* instances to and from the object storage list via the **@goRelocateTo()** and **@goReplaceWith()** functions. The graphics of a graphics object *component* instance existing on the object storage list will not be displayed or take up space in the *drawing plane*.

Other Information

@goRelocateTo() ▪ **@goReplaceWith()** ♦

@goFlags

@goFlags := *flags* (0)

flags An integer stack object containing binary flags.

Details

Sets the flags (*flags*) for the graphics object *component* instance.

flags is zero or more of the values listed under “**General**” in [Graphics Object Values](#).

The flags can be altered at any time via the **@goRepFlags ()** function, which modifies this variable, after this graphics object *component* instance has been created.

Additional Information

[Graphics Object Values](#)

Related Information

@goRepFlags () ♦

@goGrpFlags

@goGrpFlags := *flags* (0)

flags An integer stack object containing binary flags.

Details

Sets the flags (*flags*) for the group graphics object.

flags is zero or more of the values listed under “**Group Object**” in [Graphics Object Values](#).

The flags can be altered at any time via the **@goRepGrpFlags ()** function, which modifies this variable, after this graphics object *component* instance has been created.

Additional Information

[Graphics Object Values](#)

Related Information

@goRepGrpFlags () ♦

@goOriginX

@goOriginX := *x-pos* (0.0)

x-pos A decimal stack object.

Details

Sets the graphics object’s X position (*x-pos*).

x-pos is the horizontal position of the graphics object in *base units*. This is the X-coordinate of the graphics object’s origin.

Related Information

@goRepOrigin ()

Other Information

@goOriginY ▪ @goGetOrigin() ♦

@goOriginY

@goOriginY := *y-pos* (0.0)

y-pos A decimal stack object.

Details

Sets the graphics object's Y position (*y-pos*).

y-pos is the vertical position of the graphics object in *base units*. This is the Y-coordinate of the graphics object's origin.

Related Information

@goRepOrigin()

Other Information

@goOriginX ▪ @goGetOrigin() ♦

— Member Functions —

@goSetMseBtnFnt()

@goSetMseBtnFnt (*pHandler* | ?) [⚠] → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the mouse button event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing mouse button event handler function is unregistered (and returned in *prev-handler*), leaving no registered mouse button event handler function for this graphics object *component* instance.

The mouse button event handler function, *pHandler*, must be defined as follows:

fnt: (pCompID[*int*] pMseFlags[*int*] pXPos[*dec*] pYPos[*dec*] pTime[*dec*]) {...}

where,

pCompID is an integer containing the *CID* of the graphics object *component* instance by which the handler function was registered.

pMseFlags is an integer containing information on the mouse button events, mouse button states, and the time span in milliseconds since the last mouse button event related to the current mouse button in action. It also contains information on the states of the **SHIFT**, **CTRL**, and **ALT** keys. The various pieces of information can be extracted as described under [Mouse State](#).

pXPos is a decimal containing the mouse cursor X position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pYPos is a decimal containing the mouse cursor Y position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pTime is an integer containing the time of the event, in milliseconds, since the operating system had been started.

The mouse button event handler function is called by VIS when the user activates a mouse button, choosing only this graphics object (*pCompID*), while the mouse cursor is in the client area of a *VIEW* belonging to the drawing plane *component* instance by which the handler function was registered. However, if a handler function is also registered by the @dpSetMseBtnFnt () function, then that handler function is not called if this one is for the same graphics object. This handler function is also called for each graphics object if more than one was chosen, and no handler function is registered by the @dpSetMseBtnFnt () function for the parent drawing plane *component* instance. Note that only the left, middle, and right mouse buttons are recognised by VIS; other mouse buttons are ignored.

Additional Information

Mouse State

Other Information

2.15.2_Setting and Creating a Handler Function - @dpSetMseBtnFnt () ♦

@goSetMseHitFnt()

@goSetMseHitFnt (*pHandler* | ?) [⚠] → *prev-handler* | ?

pHandler An ETAC function.

prev-handler An ETAC function.

Details

Registers the object *hit test* event handler function (*pHandler*), returning the previously registered handler function (*prev-handler*) if one was registered, or returns a NULL stack object (?) if one was not.

If the argument to this function is a NULL stack object (?), then any existing object *hit test* event handler function is unregistered (and returned in *prev-handler*), leaving no registered object *hit test* event handler function for this graphics object *component* instance.

The object *hit test* event handler function, *pHandler*, must be defined as follows:

```
fnt: (pCompID[*int*] pState[*bool*] pMseFlags[*int*] pXPos[*dec*] pYPos[*dec*]
      pTime[*int*]) {...}
```

pCompID is an integer containing the *CID* of the graphics object *component* instance by which the handler function was registered.

pState is the hit state (boolean integer) for this graphics object (identified by the *CID* in pCompID). If pState contains true, then this graphics object has presently been entered, otherwise (pState contains false) it has presently been exited by the mouse cursor.

pMseFlags is an integer containing information on the mouse button states and the time span in milliseconds since the last mouse button event. It also contains information on the states of the **SHIFT**, **CTRL**, and **ALT** keys. The various pieces of information can be extracted as described under Mouse State. Note that only the left, middle, and right mouse buttons are recognised by VIS; other mouse buttons are ignored.

pXPos is a decimal containing the mouse cursor X position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pYPos is a decimal containing the mouse cursor Y position (in *base units*) mapped onto the *drawing plane* when the event occurred.

pTime is an integer containing the time of the event, in milliseconds, since the operating system had been started.

The object *hit test* event handler function is called by VIS when the mouse cursor enters or leaves only this graphics object (*pCompID*) while the mouse cursor is in the client area of a *VIEW* belonging to the drawing plane *component* instance by which the handler function was registered. However, if a handler function is also registered by the `@dpSetMseHitFnt()` function, then that handler function is not called if this one is for the same graphics object. This handler function is also called for each graphics object if more than one was *hit*, and no handler function is registered by the `@dpSetMseHitFnt()` function for the parent drawing plane *component* instance.

Additional Information

Mouse State

Other Information

2.15.2 Setting and Creating a Handler Function - `@dpSetMseHitFnt()` ♦

@goGetBoundingBox()

`@goGetBoundingBox()` → *bound-box*

bound-box A decimal sequence stack object.

Details

Gets the actual bounding box (*bound-box*) of the graphics potentially displayed by this graphics object *component* instance.

bound-box is a sequence containing the existing bounding box of the graphics object. The elements of the sequence are in *base units* as follows.

$[x-min, y-min, x-max, y-max]$,

where

- 1 *x-min* is the left border position of the graphics object's bounding box,
- 2 *y-min* is the bottom border position of the graphics object's bounding box,
- 3 *x-max* is the right border position of the graphics object's bounding box,
- 4 *y-max* is the top border position of the graphics object's bounding box.

The bounding box of a graphics object is the smallest rectangle that encloses all of the potentially visible graphics of the unhighlighted graphics object. If the graphics object *component* instance contains curve item *component* instances with the `:!GCI_FULL_BBOX:` flag set, then all of the defining points (including the control points) of the item's shape are included in the bounding box calculation; without the `:!GCI_FULL_BBOX:` flag, only the item's shape itself is included. For line and curve items of the graphics object, the pen thickness is taken into account. For text and formatted text items of the graphics object, the corresponding actual bounding boxes of those items are taken into account, which may be larger than the actual text area.

If the graphics object is a group object, then the bounding box is the smallest rectangle that encloses the bounding boxes of all the items and sub-objects of the graphics object.

Note that a graphics object's bounding box cannot be set by an ETAC program; it is internally calculated by VIS whenever required. ♦

@goGetOrigin()

@goGetOrigin() → *x-pos y-pos*

x-pos A decimal stack object.

y-pos A decimal stack object.

Details

Gets the position (*x-pos*, *y-pos*) of this graphics object *component* instance.

x-pos is the horizontal position of the graphics object in *base units*. This is the X-coordinate of the graphics object's origin.

y-pos is the vertical position of the graphics object in *base units*. This is the Y-coordinate of the graphics object's origin.

Other Information

@goRepOrigin() ♦

@goGetSubObjIDs()

@goGetSubObjIDs() → *cid-seq*

cid-seq An integer sequence stack object.

Details

Gets a sequence (*cid-seq*) of the *crucial identifiers* of the sub-object *component* instances of this graphics object *component* instance in the order that they are displayed. ♦

@goRelocateTo()

@goRelocateTo(*pDstID* | ?) → *success*

pDstID An integer or string stack object.

success An integer stack object containing a logical boolean value.

Details

Relocates this graphics object *component* instance as a sub-object of a group graphics object *component* instance, or to the front of the object display list, or to the object storage list.

pDstID is the *crucial identifier* or *name identifier* of a group graphics object (which must exist in the *current drawing plane*) to which this graphics object is to be relocated if this value is not zero; this graphics object is located to the front of the sub-objects of the group object. If this value is zero, then this graphics object is moved to the front of the object display list of the *current drawing plane* (for displaying). If a NULL stack object (?) is used instead of *pDstID*, then this graphics object is relocated to the object storage list of the *current drawing plane*.

success is true if this graphics object was successfully relocated, otherwise the value is false.

If this graphics object is successfully relocated to a group graphics object, then that group object will be the new parent of this graphics object. If this graphics object is relocated to the object storage or display list, then the *current drawing plane* will be the new parent.

If *pDstID* is the *CID* of a non-group graphics object, then no action occurs and *success* will be false.

The graphics of a graphics object *component* instance existing on the object storage list will not be displayed or take up space in the *drawing plane*.

Note that this function removes the selection and highlighting of this graphics object.

Examples

The following illustrations show how the `@goRelocateTo()` function can be used.

```
(1) void @VIS("MyObj").@goRelocateTo("MyGroupObj");
(2) void @VIS("MyObj").@goRelocateTo(?);
```

In example (1), the graphics object identified by *MyObj* is relocated to the group graphics object identified by *MyGroupObj*, whether *MyObj* existed on the object display list, the object storage list, or as sub-object of another group graphics object. If *MyObj* is already a sub-object of *MyGroupObj*, then it will be moved to the front of the sub-objects, and `@goRelocateTo(?)` will return true.

In example (2), the graphics object identified by *MyObj* is relocated to the object storage list, whether *MyObj* existed on the object display list or as sub-object of a group graphics object. If *MyObj* is already on the object storage list, then it will stay there, and `@goRelocateTo(?)` will return true.

Other Information

`@goReplaceWith()` ♦

@goReorder()

`@goReorder(pPos)`

pPos An integer stack object.

Details

Reorders the depth position (*pPos*) of this graphics object *component* instance in relation to its sibling graphics object *component* instances.

pPos indicates the new depth position of this graphics object, as follows:

- 0 this graphics object is moved to the back position (the first object to be displayed relative to its sibling objects),
- 1 this graphics object is moved one position backward relative to its current position,
- 1 this graphics object is moved one position forward relative to its current position,
- 2 this graphics object is moved to the front position (the last object to be displayed relative to its sibling objects)

If this graphics object is a sub-object of a group graphics object, then the sibling graphics objects are the sub-objects of the group object. Otherwise, the sibling graphics objects are the displayable graphics objects of the *current drawing plane*.

If this graphics object is on the object storage list, then no action occurs.

Note that this function removes the selection and highlighting of this and all the sibling graphics objects if the graphics object was not already in the specified position.

Other Information

[goGetSubObjIDs \(\)](#) ▪ [@dpGetDispObjIDs \(\)](#) ♦

@goRepContext()

@goRepContext () [[@goContextID](#)] → [prev-cid](#) | ?

[@goContextID](#) An integer or string value, or a NULL stack object.

[prev-cid](#) An integer stack object.

Details

Replaces the existing graphics context *component* instance of this graphics object *component* instance with the specified one ([@goContextID](#)), and returns the *crucial identifier* of the existing graphics context *component* instance.

[@goContextID](#) contains the *crucial identifier* or *name identifier* of a graphics context *component* instance of the *current drawing plane*. If [@goContextID](#) contains a NULL stack object (?), then only the *CID* of the existing graphics context (if any) is returned ([prev-cid](#)).

[prev-cid](#) is the *crucial identifier* of the existing graphics context of this graphics object before replacement, or a NULL stack object (?) is returned if this graphics object did not have a graphics context.

Additional Information

[@goContextID](#) ♦

@goRepFlags()

@goRepFlags ([pFlags](#) [pAdd](#)) → [prev-flags](#) [[@goFlags](#)]

[pFlags](#) An integer stack object containing binary flags.

[pAdd](#) An integer stack object containing a logical boolean value.

[prev-flags](#) An integer stack object containing binary flags.

[@goFlags](#) An integer value containing binary flags.

Details

Replaces the existing flags of this graphics object *component* instance with the specified ones ([pFlags](#)), and returns the existing flags ([prev-flags](#)) before replacement.

[pFlags](#) is zero or more of the values listed under “**General**” in [Graphics Object Values](#). If this value is 0 (zero), then no flags are replaced, but the existing flags ([prev-flags](#)) are returned by the function.

[pAdd](#) is true to add the flags in [pFlags](#) to the existing ones in this graphics object, or false to remove the flags in [pFlags](#) from the existing ones in this graphics object.

[prev-flags](#) are the existing flags before replacement.

[@goFlags](#) contains the flags of this graphics object after replacement.

Additional Information

[Graphics Object Values](#) ▪ [@goFlags](#)

Other Information

[@goRepGrpFlags \(\)](#) ♦

@goRepGrpFlags()

@goRepGrpFlags (*pFlags* *pAdd*) → *prev-flags* [*@goGrpFlags*]

pFlags An integer stack object containing binary flags.

pAdd An integer stack object containing a logical boolean value.

prev-flags An integer stack object containing binary flags.

@goGrpFlags An integer stack object containing binary flags.

Details

Replaces the existing flags of this group graphics object *component* instance (if it is a group object) with the specified ones (*pFlags*), and returns the existing flags (*prev-flags*).

pFlags is zero or more of the values listed under “**Group Object**” in [Graphics Object Values](#). If this value is 0 (zero), then no flags are replaced, but the existing flags (*prev-flags*) are returned by the function.

pAdd is `true` to add the flags in *pFlags* to the existing ones in this group graphics object, or `false` to remove the flags in *pFlags* from the existing ones in this group graphics object.

prev-flags are the existing flags of this group graphics object before replacement.

@goGrpFlags contains the flags of this group graphics object after replacement.

This graphics object must be a group object, not a regular one, otherwise an error event will occur. If the result of the following code snip is `true`, then this graphics object is a group object: `to_bool (:!GO_GROUP_OBJ: &and @goRepFlags(0 true))`.

Additional Information

[Graphics Object Values](#) ▪ [@goGrpFlags](#)

Other Information

[@goRepFlags\(\)](#) ♦

@goReplaceWith()

@goReplaceWith (*pSrcID*) → *success*

pSrcID An integer or string stack object.

success An integer stack object containing a logical boolean value.

Details

Replaces this graphics object *component* instance, which must be on the object display list of the *current drawing plane*, with a top-level one (*pSrcID*) on the object storage list.

pSrcID is the *crucial identifier* or *name identifier* of a graphics object *component* instance of the *current drawing plane* existing on the object storage list. The graphics object must not be a sub-object.

success is `true` if the replacement occurred, otherwise it is `false`.

The function swaps this graphics object with the specified one (*pSrcID*) existing on the object storage list; this graphics object ends up being on the object storage list, and the specified one ends up as a displayable object (in front of all other graphics objects) with the same selection and

highlight state as this graphics object. This graphics object can be a sub-object, but will end up on the object storage list as a top-level object.

If this graphics object is not on the object display list, or if the specified graphics object (*pSrcID*) is not on the object storage list, then *success* will be false.

The graphics of a graphics object *component* instance existing on the object storage list will not be displayed or take up space in the *drawing plane*.

Example

The following illustration shows how the `@goReplaceWith()` function can be used.

```
vis_DPlane(...)
{
    vis_VIEW(...) {}

    vis_GObject("GObj1") [* Create an object (circle) on the object storage list. *]
    {
        [* The object is created invisibly on the object storage list. *]
        @goDisplay := false;

        vis_GContext("") {@gcOHFlags := :!OH_BORD_HILITE:}; [* Allow highlighting. *]
        vis_CurveItem("") [* Create a circle item. *]
        {
            @ciFlags := :!GCI_OUTLINED_SHAPE;
            @ciPoints := @SPF{spf_Circle(@spfRadius := 5.0)};
        }
    };

    vis_GObject("GObj2") [* Create an object (triangle) on the object display list. *]
    {
        vis_GContext("") {@gcOHFlags := :!OH_BORD_HILITE:}; [* Allow highlighting. *]
        vis_CurveItem("") [* Create a triangle item. *]
        {
            @ciFlags := :!GCI_OUTLINED_SHAPE;
            @ciPoints := @SPF{spf_EquTriangle(@spfWidth := 5.0)};
        }
    };

    [* Select and highlight the triangle (GObj2). *]
    void @VIS("GObj2").@goRepSelect(:#GO_HILITE_ON);
};

[* Swap the two objects. The circle (GObj1) will become visible and highlighted. *]
@VIS("GObj2").{void @goReplaceWith("GObj1")}; void @VIS("GObj1").@goShow(true);
```

In example above, a graphics object (GObj1) is created on the object storage list (in the invisible state) because of `@goDisplay := false;`. The object is not on the object display list, therefore not originally displayed. A second graphics object (GObj2) is created on the object display list (the default). Both graphics objects allow highlighting, and the second object (GObj2) is actually selected and highlighted. After both objects have been created, the one on the object storage list (GObj1) is swapped with the one on the object display list (GObj2) via the `@goReplaceWith()` function. The first object (GObj1) is now on the display list, ready to be displayed highlighted (because the highlight and selection state of the second object is transferred to the first object during the swap), and the second object (GObj2) is now on the object storage list and not displayable. Because the first object (GObj1) was created invisibly, it needs to be explicitly made visible after having been swapped with the second object (GObj2).

Other Information

`@goRelocateTo()` ♦

@goRepOrigin()

@goRepOrigin() [*@goOriginX* *@goOriginY*] → *x-pos* *y-pos*

@goOriginX A decimal value.

@goOriginY A decimal value.

x-pos A decimal stack object.

y-pos A decimal stack object.

Details

Replaces the existing position of this graphics object *component* instance with the specified one (*@goOriginX*, *@goOriginY*), and returns the existing position (*x-dec*, *x-dec*).

@goOriginX contains the horizontal position of the graphics object in *base units*. This is the X-coordinate of the graphics object's origin.

@goOriginY contains the vertical position of the graphics object in *base units*. This is the Y-coordinate of the graphics object's origin.

x-pos is the existing horizontal position of the graphics object in *base units* before replacement.

y-pos is the existing vertical position of the graphics object in *base units* before replacement.

The graphics object must belong to the *current drawing plane* when calling this function.

Additional Information

@goOriginX - *@goOriginY*

Other Information

@goGetOrigin() ♦

@goRepSelect()

@goRepSelect() (*pState*) → *prev-state*

pState An integer stack object.

prev-state An integer stack object.

Details

Replaces the existing selection state of this graphics object *component* instance, and returns the existing selection state.

pState is one of the values listed under “**Highlight State**” in [Graphics Object Values](#) or 0 (zero). If the value is zero, then only the existing selection state is returned (*prev-state*).

prev-state is the existing selection state of the graphics object before replacement. This value is either : **cGO_HILITE_ON**: if the graphics object was highlighted, or : **cGO_HILITE_OFF**: otherwise.

Note that this function both selects and highlights, or both deselects and unhighlights, the graphics object. The graphics object must allow highlighting for it to be actually highlighted. This can be achieved via the object's graphics context.

The graphics object must belong to the *current drawing plane* when calling this function.

Additional Information[Graphics Object Values](#)**Other Information**[3.4.14 _Graphics Context](#) ▪ [@dpGetSelObjIDs\(\)](#) ▪ [@dpDeselectGObjs\(\)](#) ♦**@goSetRedraw()****@goSetRedraw()****Details**

Sets this graphics object *component* instance to be redrawn by VIS at the earliest convenience.

The graphics object must belong to the *current drawing plane* when calling this function.

Other Information[@vRefresh\(\)](#) ♦**@goShow()****@goShow(*pShow* | ?) → *prev-state****pShow* An integer stack object containing a logical boolean value.*prev-state* An integer stack object containing a logical boolean value.**Details**

Shows or hides (*pShow*) this graphics object *component* instance, or just returns its existing visibility state (*prev-state*).

pShow is true to make the graphics object visible, or false to make it invisible. If a NULL stack object (?) is used instead of *pShow*, then only the existing visibility state is returned (*prev-state*).

prev-state is the existing visibility state of the graphics object before the visibility state change. If the value is true, then the graphics object was visible before the state change, otherwise it was invisible.

The graphics object must belong to the *current drawing plane* when calling this function.

Note: A graphics object can be effectively disabled by removing the `:!GO_HIT_TEST_ON:` flag via the [@goRepFlags\(\)](#) function. ♦

3.4.9 Line Item

A line item *component* defines a poly-line as a sequence of X and Y coordinates to be displayed in a *VIEW*. A line item *component* instance can belong only to one graphics object *component* instance.

The poly-line created by a line item can be filled or outlined, or both, or neither (in this case, the poly-line will be invisible).

For fixed line item *components*, measurements are in *logical picas* rather than *base units*.

The *current drawing plane* must be the same as the *drawing plane* within which the line item *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: `li`
- Type: `visC_LineItem`
- Parent: [Graphics Object](#)
- Creator: `vis_LineItem()` ▪ `vis_FixLineItem()`
- Deletion: `@visDelete()`
- Access: `@VIS()` ▪ `@VISX()`

— Member Variables —

@liBrushID

`@liBrushID := brush-id (?)`

brush-id An integer or string stack object.

Details

Sets the identifier (*brush-id*) of the brush attributes *component* instance for this line item *component* instance when being created if one is required.

brush-id is the *crucial identifier* or *name identifier* of a previously created brush attributes *component* instance. If a NULL stack object (?) is used instead of *brush-id*, then the line item will assume a default background (solid white colour) when the line item is being created.

The brush attributes *component* instance is used to fill this line item's shape, but only if `:!GLI_FILLED_SHAPE:` is set in `@liFlags`.

The brush attributes can be altered at any time via the `@liRepBrush()` function, which uses this variable, after this line item *component* instance has been created.

Related Information

[3.4.16_Brush Attributes](#) ▪ `@liRepBrush()`

Other Information

`@liFlags` ♦

@liFixed

`@liFixed := fixed (false)`

fixed An integer stack object containing a logical boolean value.

Details

If `@liFixed` is `true`, then this line item *component* instance is a fixed item, otherwise it is not. This value is set only by VIS when a fixed line item *component* instance is created. The program should not set this value.

Related Information

`vis_FixLineItem()` ♦

@liFlags

```
@liFlags := flags (0)
```

flags An integer stack object containing binary flags.

Details

Sets the flags (*flags*) for the line item *component* instance.

flags is zero or more of the values listed in [Line Item Creation](#).

The flags can be altered at any time via the `@liRepFlags()` function, which modifies this variable, after this line item *component* instance has been created.

Additional Information

[Line Item Creation](#)

Related Information

`@liRepFlags()` ♦

@liLowerScale

```
@liLowerScale := scale (0.0)
```

scale A decimal stack object.

Details

Sets the lower *VIEW* scale (*scale*) below which this line item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is below this value, this line item will be invisible but will still take up space in the *drawing plane*. Otherwise, the line item is displayed as usual.

The lower *VIEW* scale can be altered at any time via the `@liRepLwrScale()` function, which uses this variable, after this line item *component* instance has been created.

Related Information

`@liRepLwrScale()` ♦

@liPenID

```
@liPenID := pen-id (?)
```

pen-id An integer or string stack object.

Details

Sets the identifier (*pen-id*) of the pen attributes *component* instance for this line item *component* instance when being created if one is required.

pen-id is the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance. If a NULL stack object (?) is used instead of *pen-id*, then the line item will assume a default outline (solid black colour) when the line item is being created.

The pen attributes *component* instance is used to outline this line item's shape, but only if `: !GLI_OUTLINED_SHAPE:` is set in `@liFlags`.

The pen attributes can be altered at any time via the `@liRepPen()` function, which uses this variable, after this line item *component* instance has been created.

Related Information

[3.4.15_Pen Attributes](#) - `@liRepPen()`

Other Information

`@liFlags` ♦

@liPoints

`@liPoints := points (?)`

points A decimal sequence stack object.

Details

Sets the sequence of points (*points*) of a poly-line for the line item *component* instance.

points is a sequence of points in the form $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point of the poly-line in *base units* relative to the origin point of the line item's parent graphics object *component* instance. *points* can be an empty sequence, and can also be a NULL stack object but only when the line item is being created. For a fixed line item, the units are *logical picas* not *base units*.

If *points* is an empty sequence, then the poly-line of the line item is effectively non-existent and will not take up space in the *drawing plane*.

The sequence of points can be altered at any time via the `@liNewPoints()` function, which uses this variable, after this line item *component* instance has been created.

Examples

The following illustrations show how the `@liPoints` function can be used.

(1) `@liPoints := [0.0, 0.0, 2.0, 0.0, 2.0, 2.0, 0.0, 2.0, 0.0, 0.0];`

(2) `@liPoints := [-1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0, 1.0, -1.0, -1.0];`

In example (1), the sequence of points defines a rectangle of size 2.0 *base units* with the bottom-left corner at the origin point of the parent graphics object. The poly-line points are: (0, 0), (2, 0), (2, 2), (0, 2), (0, 0). Note that the point (0, 0) in the poly-line is always at the parent graphics object's origin.

In example (2), the sequence of points defines a rectangle of size 2.0 *base units* with the centre of the rectangle at the origin point of the parent graphics object. The poly-line points are: (-1, -1), (1, -1), (1, 1), (-1, 1), (-1, -1).

Related Information

`@liNewPoints()` ♦

@liUpperScale

@liUpperScale := *scale* (:#+max_dec:)

scale A decimal stack object.

Details

Sets the upper *VIEW* scale (*scale*) above or equal to which this line item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is equal to or above this value, this line item will be invisible but will still take up space in the *drawing plane*. Otherwise, the line item is displayed as usual.

The upper *VIEW* scale can be altered at any time via the @liRepUpScale() function, which uses this variable, after this line item *component* instance has been created.

Related Information

@liRepUpScale() ♦

— Member Functions —

@liNewPoints()

@liNewPoints() [*@liPoints*]

@liPoints A decimal sequence.

Details

Replaces the existing poly-line for the line item *component* instance.

@liPoints contains a sequence of points, in *base units*, defining the poly-line relative to the origin point of the line item's parent graphics object *component* instance.

Additional Information

@liPoints ♦

@liRepBrush()

@liRepBrush() [*@liBrushID*] → *prev-cid* | ?

@liBrushID An integer or string value, or a NULL stack object.

prev-cid An integer stack object.

Details

Replaces the existing brush attributes *component* instance of this line item *component* instance with the specified one (*@liBrushID*), and returns the *crucial identifier* of the existing brush attributes *component* instance.

@liBrushID contains the *crucial identifier* or *name identifier* of the brush attributes *component* instance of the *current drawing plane*. If *@liBrushID* contains a NULL stack object (?), then only the *CID* of the existing brush attributes (if any) is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing brush attributes of this line item before replacement, or a NULL stack object (?) is returned if this line item did not have brush attributes.

Additional Information

@liBrushID ♦

@liRepFlags()	
@liRepFlags (<i>pFlags</i> <i>pAdd</i>) → <i>prev-flags</i> [<i>@liFlags</i>]	
<i>pFlags</i>	An integer stack object containing binary flags.
<i>pAdd</i>	An integer stack object containing a logical boolean value.
<i>prev-flags</i>	An integer stack object containing binary flags.
<i>@liFlags</i>	An integer value containing binary flags.

Details

Replaces the existing flags of this line item *component* instance with the specified ones (*pFlags*), and returns the existing flags (*prev-flags*).

pFlags is zero or more of the values listed in [Line Item Creation](#). If this value is 0 (zero), then no flags are replaced, but the existing flags (*prev-flags*) are returned by the function.

pAdd is `true` to add the flags in *pFlags* to the existing ones in this line item, or `false` to remove the flags in *pFlags* from the existing ones in this line item.

prev-flags are the existing flags before replacement.

@liFlags contains the flags of this line item after replacement.

Additional Information

[Line Item Creation](#) ▪ @liFlags ♦

@liRepLwrScale()	
@liRepLwrScale () [<i>@liLowerScale</i>] → <i>prev-scale</i>	
<i>@liLowerScale</i>	A decimal value.
<i>prev-scale</i>	A decimal stack object.

Details

Replaces the existing lower *VIEW* scale of this line item *component* instance with the specified one (*@liLowerScale*), and returns the existing scale (*prev-scale*).

@liLowerScale contains a non-negative number.

prev-scale is the existing lower *VIEW* scale of this line item before replacement.

Additional Information

@liLowerScale ♦

@liRepPen()

@liRepPen() [*@liPenID*] → *prev-cid* | ?

@liPenID An integer or string value, or a NULL stack object.

prev-cid An integer stack object.

Details

Replaces the existing pen attributes *component* instance of this line item *component* instance with the specified one (*@liPenID*), and returns the *crucial identifier* of the existing pen attributes *component* instance.

@liPenID contains the *crucial identifier* or *name identifier* of a pen attributes *component* instance of the *current drawing plane*. If *@liPenID* contains a NULL stack object (?), then only the *CID* of the existing pen attributes (if any) is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing pen attributes of this line item before replacement, or a NULL stack object (?) is returned if this line item did not have pen attributes.

Additional Information

@liPenID ♦

@liRepUpScale()

@liRepUpScale() [*@liUpperScale*] → *prev-scale*

@liUpperScale A decimal value.

prev-scale A decimal stack object.

Details

Replaces the existing upper *VIEW* scale of this line item *component* instance with the specified one (*@liUpperScale*), and returns the existing scale (*prev-scale*).

@liUpperScale contains a non-negative number.

prev-scale is the existing upper *VIEW* scale of this line item before replacement.

Additional Information

@liUpperScale ♦

3.4.10 Curve Item

A curve item *component* defines a poly-curve as a Bezier sequence of X and Y coordinates to be displayed in a *VIEW*. A curve item *component* instance can belong only to one graphics object *component* instance.

The poly-curve created by a curve item can be filled or outlined, or both, or neither (in this case, the poly-curve will be invisible).

For fixed curve item *components*, measurements are in *logical picas* rather than *base units*.

The *current drawing plane* must be the same as the *drawing plane* within which the curve item *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: `ci`
- Type: `visC_CurveItem`
- Parent: [Graphics Object](#)
- Creator: `vis_CurveItem()` ▪ `vis_FixCurveItem()`
- Deletion: `@visDelete()`
- Access: `@VIS()` ▪ `@VISX()`

— Member Variables —

@ciBrushID

`@ciBrushID := brush-id (?)`

brush-id An integer or string stack object.

Details

Sets the identifier (*brush-id*) of the brush attributes *component* instance for this curve item *component* instance when being created if one is required.

brush-id is the *crucial identifier* or *name identifier* of a previously created brush attributes *component* instance. If a NULL stack object (?) is used instead of *brush-id*, then the curve item will assume a default background (solid white colour) when the curve item is being created.

The brush attributes *component* instance is used to fill this curve item's shape, but only if `:!GCI_FILLED_SHAPE:` is set in `@liFlags`.

The brush attributes can be altered at any time via the `@ciRepBrush()` function, which uses this variable, after this curve item *component* instance has been created.

Related Information

[3.4.16_Brush Attributes](#) ▪ `@ciRepBrush()` ♦

@ciFixed

`@ciFixed := fixed (false)`

fixed An integer stack object containing a logical boolean value.

Details

If `@ciFixed` is true, then this curve item *component* instance is a fixed item, otherwise it is not. This value is set only by VIS when a fixed curve item *component* instance is created. The program should not set this value.

Other Information

`vis_FixCurveItem()` ♦

@ciFlags

@ciFlags := *flags* (0)

flags An integer stack object containing binary flags.

Details

Sets the flags (*flags*) for the curve item *component* instance.

flags is zero or more of the values listed in [Curve Item Creation](#).

The flags can be altered at any time via the **@ciRepFlags()** function, which modifies this variable, after this curve item *component* instance has been created.

Additional Information

[Curve Item Creation](#)

Related Information

@ciRepFlags() ♦

@ciLowerScale

@ciLowerScale := *scale* (0.0)

scale A decimal stack object.

Details

Sets the lower *VIEW* scale (*scale*) below which this curve item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is below this value, this curve item will be invisible but will still take up space in the *drawing plane*. Otherwise, the curve item is displayed as usual.

The lower *VIEW* scale can be altered at any time via the **@ciRepLwrScale()** function, which uses this variable, after this curve item *component* instance has been created.

Related Information

@ciRepLwrScale() ♦

@ciPenID

@ciPenID := *pen-id* (?)

pen-id An integer or string stack object.

Details

Sets the identifier (*pen-id*) of the pen attributes *component* instance for this curve item *component* instance when being created if one is required.

pen-id is the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance. If a NULL stack object (?) is used instead of *pen-id*, then the curve item will assume a default outline (solid black colour) when the curve item is being created.

The pen attributes *component* instance is used to outline this curve item's shape, but only if `:!GCI_OUTLINED_SHAPE:` is set in `@ciFlags`.

The pen attributes can be altered at any time via the `@ciRepPen()` function, which uses this variable, after this curve item *component* instance has been created.

Related Information

[3.4.15_Pen Attributes](#) ▪ `@ciRepPen()`

Other Information

`@ciFlags` ♦

@ciPoints

`@ciPoints := points (?)`

points A decimal sequence stack object.

Details

Sets the sequence of points (*points*) of a poly-curve for the curve item *component* instance.

points is a sequence of points in the form $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point of the poly-curve in *base units* relative to the origin point of the curve item's parent graphics object *component* instance. *points* can be an empty sequence, and can also be a NULL stack object but only when the curve item is being created. For a fixed curve item, the units are *logical picas* not *base units*.

If *points* is an empty sequence, then the poly-curve of the curve item is effectively non-existent and will not take up space in the *drawing plane*.

A poly-curve is a path composed of joined Bezier curve segments. Other public documentation needs to be consulted for the definition of a Bezier curve segment.

The sequence of points can be altered at any time via the `@ciNewPoints()` function, which uses this variable, after this curve item *component* instance has been created.

Examples

The following illustrations show how the `@ciPoints` function can be used.

```
(1) @ciPoints := @SPF{spf_Circle(@spfRadius := 2.0)};;
(2) @ciPoints := @SPF{spf_Rectangle(@spfOrigin := [1.0, 1.0])};;
```

In example (1), an SPF function (`spf_Circle()`) is used to create a circle with a radius of two *base units*. The centre of the circle is at the parent graphics object's origin by default of the function.

In example (2), an SPF function (`spf_Rectangle()`) is used to create a rectangle of the default size of one *base unit* whose centre is at point (1.0, 1.0) relative to the parent graphics object's origin.

Related Information

`@ciNewPoints()` ♦

@ciUpperScale

@ciUpperScale := *scale* (:#+max_dec:)

scale A decimal stack object.

Details

Sets the upper *VIEW* scale (*scale*) above or equal to which this curve item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is equal to or above this value, this curve item will be invisible but will still take up space in the *drawing plane*. Otherwise, the curve item is displayed as usual.

The upper *VIEW* scale can be altered at any time via the **@ciRepUpScale()** function, which uses this variable, after this curve item *component* instance has been created.

Related Information

@ciRepUpScale() ♦

— Member Functions —

@ciNewPoints()

@ciNewPoints() [*@ciPoints*]

@ciPoints A decimal sequence.

Details

Replaces the existing poly-curve for the curve item *component* instance.

@ciPoints contains a sequence of points, in *base units*, defining the poly-curve relative to the origin point of the curve item's parent graphics object *component* instance.

Additional Information

@ciPoints ♦

@ciRepBrush()

@ciRepBrush() [*@ciBrushID*] → *prev-cid* | ?

@ciBrushID An integer or string value, or a NULL stack object.

prev-cid An integer stack object.

Details

Replaces the existing brush attributes *component* instance of this curve item *component* instance with the specified one (*@ciBrushID*), and returns the *crucial identifier* of the existing brush attributes *component* instance.

@ciBrushID contains the *crucial identifier* or *name identifier* of the brush attributes *component* instance of the *current drawing plane*. If *@ciBrushID* contains a NULL stack object (?), then only the *CID* of the existing brush attributes (if any) is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing brush attributes of this curve item before replacement, or a NULL stack object (?) is returned if this curve item did not have brush attributes.

Additional Information

@ciBrushID ♦

@ciRepFlags()

@ciRepFlags (*pFlags* *pAdd*) → *prev-flags* [*@ciFlags*]

<i>pFlags</i>	An integer stack object containing binary flags.
<i>pAdd</i>	An integer stack object containing a logical boolean value.
<i>prev-flags</i>	An integer stack object containing binary flags.
<i>@ciFlags</i>	An integer value containing binary flags.

Details

Replaces the existing flags of this curve item *component* instance with the specified ones (*pFlags*), and returns the existing flags (*prev-flags*).

pFlags is zero or more of the values listed in [Curve Item Creation](#). If this value is 0 (zero), then no flags are replaced, but the existing flags (*prev-flags*) are returned by the function.

pAdd is true to add the flags in *pFlags* to the existing ones in this curve item, or false to remove the flags in *pFlags* from the existing ones in this curve item.

prev-flags are the existing flags before replacement.

@ciFlags contains the flags of this curve item after replacement.

Additional Information

[Curve Item Creation](#) ▪ @ciFlags ♦

@ciRepLwrScale()

@ciRepLwrScale () [*@liLowerScale*] → *prev-scale*

<i>@liLowerScale</i>	A decimal value.
<i>prev-scale</i>	A decimal stack object.

Details

Replaces the existing lower *VIEW* scale of this curve item *component* instance with the specified one (*@liLowerScale*), and returns the existing scale (*prev-scale*).

@liLowerScale contains a non-negative number.

prev-scale is the existing lower *VIEW* scale of this curve item before replacement.

Additional Information

@ciLowerScale ♦

@ciRepPen()**@ciRepPen()** [*@ciPenID*] → *prev-cid* | ?*@ciPenID* An integer or string value, or a NULL stack object.*prev-cid* An integer stack object.**Details**

Replaces the existing pen attributes *component* instance of this curve item *component* instance with the specified one (*@ciPenID*), and returns the *crucial identifier* of the existing pen attributes *component* instance.

@ciPenID contains the *crucial identifier* or *name identifier* of a pen attributes *component* instance of the *current drawing plane*. If *@ciPenID* contains a NULL stack object (?), then only the *CID* of the existing pen attributes (if any) is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing pen attributes of this curve item before replacement, or a NULL stack object (?) is returned if this curve item did not have pen attributes.

Additional Information**@ciPenID** ♦**@ciRepUpScale()****@ciRepUpScale()** [*@ciUpperScale*] → *prev-scale**@ciUpperScale* A decimal value.*prev-scale* A decimal stack object.**Details**

Replaces the existing upper *VIEW* scale of this curve item *component* instance with the specified one (*@ciUpperScale*), and returns the existing scale (*prev-scale*).

@ciUpperScale contains a non-negative number.

prev-scale is the existing upper *VIEW* scale of this curve item before replacement.

Other Information**@ciUpperScale** ♦**3.4.11 Text Item**

A text item *component* defines a plain text string of a desired font with desired characteristics.

For fixed text item *components*, measurements are in *logical picas* rather than *base units*.

The *current drawing plane* must be the same as the *drawing plane* within which the text item *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: `ti`
- Type: `visC_TextItem`
- Parent: `Graphics Object`
- Creator: `vis_TextItem()` ▪ `vis_FixTextItem()`
- Deletion: `@visDelete()`

- Access: **@VIS ()** ▪ **@VISX ()**

— Member Variables —

@tiAutoFitBound

@tiAutoFitBound := *bool* (**true**)

bool An integer stack object containing a logical boolean value.

Details

Sets whether or not (*bool*) the internal bounding box is fitted around the text of the text item *component* instance.

bool is **true** to allow VIS to automatically fit the internal bounding box around the text specified by **@tiText**, otherwise the internal bounding box will be as specified by **@tiBoundingBox**.

VIS automatically fits the bounding box around the text by calling the **@tiFitBoundingBox ()** function. This occurs when the text item is being created, and by the **@tiRepText ()** function as long as **@tiAutoFitBound** contains a **true** value.

@tiAutoFitBound can be set at any time.

Related Information

@tiRepText ()

Other Information

@tiBoundingBox ▪ **@tiFitBoundingBox ()** ♦

@tiBoundingBox

@tiBoundingBox := *bound-box* ([0.0, 0.0, 0.0, 0.0])

bound-box A decimal sequence stack object.

Details

Sets the bounding box (*bound-box*) into which the text of this text item *component* instance is aligned.

bound-box is a sequence containing the bounding box of the text item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed text item, the units are *logical picas* not *base units*.

[*x-min*, *y-min*, *x-max*, *y-max*],

where

- 1 *x-min* is the left border position of the text item's bounding box,
- 2 *y-min* is the bottom border position of the text item's bounding box,
- 3 *x-max* is the right border position of the text item's bounding box,
- 4 *y-max* is the top border position of the text item's bounding box.

The text of the text item is aligned within the bounding box as specified in **@tiFlags**. However, text falling outside the bounding box is clipped if the bounding box is not automatically fitted (via the **@tiFitBoundingBox ()** function).

The bounding box can be altered at any time via the `@tiRepBoundingBox()` function, which uses this variable, after this text item *component* instance has been created.

Related Information

`@tiRepBoundingBox()`

Other Information

`@tiFlags` ▪ `@tiAutoFitBound` ▪ `@tiFitBoundingBox()` ♦

@tiFixed

`@tiFixed := fixed (false)`

fixed An integer stack object containing a logical boolean value.

Details

If `@liFixed` is true, then this text item *component* instance is a fixed item, otherwise it is not. This value is set only by VIS when a fixed text item *component* instance is created. The program should not set this value.

Related Information

`vis_FixTextItem()` ♦

@tiFlags

`@tiFlags := flags (0)`

flags An integer stack object containing binary flags.

Details

Sets the flags (*flags*) for the text item *component* instance.

flags is zero or more of the values listed in [Text Item Creation](#).

The flags can be altered at any time via the `@tiRepFlags()` function, which modifies this variable, after this text item *component* instance has been created.

Additional Information

[Text Item Creation](#)

Related Information

`@tiRepFlags()` ♦

@tiLowerScale

`@tiLowerScale := scale (0.0)`

scale A decimal stack object.

Details

Sets the lower *VIEW* scale (*scale*) below which this text item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is below this value, this text item will be invisible but will still take up space in the *drawing plane*. Otherwise, the text item is displayed as usual.

The lower *VIEW* scale can be altered at any time via the `@tiRepLwrScale()` function, which uses this variable, after this text item *component* instance has been created.

Related Information

`@tiRepLwrScale()` ♦

@tiText

```
@tiText := text ("")
```

text A string stack object.

Details

Sets the plain text (*text*) for this text item *component* instance.

The formatting of the text is specified in `@tiFlags`, and the attributes are specified in the associated text style attributes *component*.

The text can be altered at any time via the `@tiRepText()` function, which uses this variable, after this text item *component* instance has been created.

Related Information

`@tiRepText()`

Other Information

`@tiFlags` ▪ [3.4.18_Text Style Attributes](#) ♦

@tiTextStyleID

```
@tiTextStyleID := style-id (?)
```

style-id An integer or string stack object.

Details

Sets the identifier (*style-id*) of the text style attributes *component* instance for this text item *component* instance when being created if one is required.

style-id is the *crucial identifier* or *name identifier* of a previously created text style attributes *component* instance. If a NULL stack object (?) is used instead of *style-id* when this text item is being created, then the text item will assume the default style.

The text style attributes can be altered at any time via the `@tiRepTextStyle()` function, which uses this variable, after this text item *component* instance has been created.

Related Information

[3.4.18_Text Style Attributes](#) ▪ `@tiRepTextStyle()` ♦

@tiUpperScale

```
@tiUpperScale := scale (:#+max_dec:)
```

scale A decimal stack object.

Details

Sets the upper *VIEW* scale (*scale*) above or equal to which this text item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is equal to or above this value, this text item will be invisible but will still take up space in the *drawing plane*. Otherwise, the text item is displayed as usual.

The upper *VIEW* scale can be altered at any time via the `@tiRepUpScale()` function, which uses this variable, after this text item *component* instance has been created.

Related Information

`@tiRepUpScale()` ♦

— Member Functions —

`@tiFitBoundingBox()`

`@tiFitBoundingBox()` → *success*

success An integer stack object containing a logical boolean value.

Details

Fits the internal bounding box around the text of this text item *component* instance.

success is `true` if the bounding box was changed to fit around the text, otherwise it is `false`.

This function changes the internal bounding box to fit around the text. The flags that were set by `@tiFlags` are taken into account. If the flags contain `:!GTI_WORD_WRAP:`, the text will wrap to the width of the internal bounding box, otherwise the width of the internal bounding box is modified to the width needed to draw the text. Then the height of the internal bounding box is modified to the height needed to draw the text. The fitted bounding box is then aligned to the original bounding box (before it was fitted). See [Text Item Creation](#) for flag settings that are used by this function.

Additional Information

[Text Item Creation](#)

Other Information

`@tiFlags` ▪ `@tiAutoFitBound` ▪ `@tiGetBoundingBox()` ♦

`@tiGetBoundingBox()`

`@tiGetBoundingBox()` → *bound-box*

bound-box A decimal sequence stack object.

Details

Gets the internal bounding box of this text item *component* instance.

bound-box is a sequence containing the internal bounding box of the text item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed text item, the units are *logical picas* not *base units*.

$[x-min, y-min, x-max, y-max]$,

where

- 1 *x-min* is the left border position of the text item's bounding box,
- 2 *y-min* is the bottom border position of the text item's bounding box,

- 3 *x-max* is the right border position of the text item’s bounding box,
- 4 *y-max* is the top border position of the text item’s bounding box.

The internal bounding box of the text item may not be the same as the one set by the `@tiBoundingBox` variable if the `@tiAutoFitBound` variable is `true` when the text is changed, or the `@tiFitBoundingBox()` function was called.

Other Information

`@tiBoundingBox` ▪ `@tiAutoFitBound` ▪ `@tiFitBoundingBox()` ▪ `@tiRepBoundingBox()` ♦

@tiRepBoundingBox()	
@tiRepBoundingBox()	[@tiBoundingBox] → bound-box
@tiBoundingBox	A decimal sequence.
bound-box	A decimal sequence stack object.

Details

Replaces the existing internal bounding box of this text item *component* instance with the specified one (`@tiBoundingBox`), and returns the existing internal bounding box (`bound-box`).
`@tiBoundingBox` contains the new bounding box relative to the parent graphics object *component* instance, as described below at `bound-box`.

`bound-box` is a TAC sequence containing the bounding box of the text item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed text item, the units are *logical picas* not *base units*.

[*x-min*, *y-min*, *x-max*, *y-max*],
where

- 1 *x-min* is the left border position of the text item’s bounding box,
- 2 *y-min* is the bottom border position of the text item’s bounding box,
- 3 *x-max* is the right border position of the text item’s bounding box,
- 4 *y-max* is the top border position of the text item’s bounding box.

The internal bounding box of the text item may become different from the one set by the `@tiBoundingBox` variable if the `@tiAutoFitBound` variable is `true` when the text is changed, or the `@tiFitBoundingBox()` function is called.

Additional Information

`@tiBoundingBox`

Other Information

`@tiGetBoundingBox()` ♦

@tiRepFlags()**@tiRepFlags** (*pFlags* *pAdd*) → *prev-flags* [*@tiFlags*]

<i>pFlags</i>	An integer containing binary flags.
<i>pAdd</i>	An integer containing a logical boolean value.
<i>prev-flags</i>	An integer containing binary flags.
<i>@tiFlags</i>	An integer value containing binary flags.

Details

Replaces the existing flags of this text item *component* instance with the specified ones (*pFlags*), and returns the existing flags (*prev-flags*).

pFlags is zero or more of the values listed in [Text Item Creation](#). If this value is 0 (zero), then no flags are replaced, but the existing flags (*prev-flags*) are returned by the function.

pAdd is `true` to add the flags in *pFlags* to the existing ones in this text item, or `false` to remove the flags in *pFlags* from the existing ones in this text item.

prev-flags are the existing flags before replacement.

@tiFlags contains the flags of this text item after replacement.

Additional Information

[Text Item Creation](#) ▪ **@tiFlags** ♦

@tiRepLwrScale()**@tiRepLwrScale** () [*@tiLowerScale*] → *prev-scale*

<i>@tiLowerScale</i>	A decimal value.
<i>prev-scale</i>	A decimal stack object.

Details

Replaces the existing lower *VIEW* scale of this text item *component* instance with the specified one (*@tiLowerScale*), and returns the existing scale (*prev-scale*).

@tiLowerScale contains a non-negative number.

prev-scale is the existing lower *VIEW* scale of this text item before replacement.

Additional Information

@liLowerScale ♦

@tiRepText()

@tiRepText() [*@tiText @tiAutoFitBound*] → *prev-text*

@tiText A string value.

@tiAutoFitBound An integer containing a logical boolean value.

prev-text A string stack object.

Details

Replaces the existing text of this text item *component* instance with the specified one (*@tiText*), and returns the existing text (*prev-text*).

@tiText contains the new text to replace the existing text.

@tiAutoFitBound contains `true` to automatically fit the internal bounding box (via **@tiFitBoundingBox()**) around the replaced text, otherwise the bounding box is not fitted.

prev-text is the existing text of this text item before replacement.

Additional Information

@tiText ▪ **@tiAutoFitBound**

Other Information

@tiFitBoundingBox() ♦

@tiRepTextStyle()

@tiRepTextStyle() [*@tiTextStyleID*] → *prev-cid*

@tiTextStyleID An integer or string value.

prev-cid An integer stack object.

Details

Replaces the existing text style attributes *component* instance of this text item *component* instance with the specified one (*@tiTextStyleID*), and returns the *crucial identifier* of the existing text style attributes *component* instance.

@tiTextStyleID contains the *crucial identifier* or *name identifier* of a text style attributes *component* instance of the *current drawing plane*. If *@tiTextStyleID* contains the value 0 (zero), then the default text style attributes is assumed. If *@tiTextStyleID* contains a NULL stack object (?), then only the *CID* of the existing text style attributes is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing text style attributes of this text item before replacement, or the value 0 (zero) is returned if this text item had the default text style attributes.

Additional Information

@tiTextStyleID ♦

@tiRepUprScale()

@tiRepUprScale() [*@tiUpperScale*] → *prev-scale*

@tiUpperScale A decimal value.

prev-scale A decimal stack object.

Details

Replaces the existing upper *VIEW* scale of this text item *component* instance with the specified one (*@tiUpperScale*), and returns the existing scale (*prev-scale*).

@tiUpperScale contains a non-negative number.

prev-scale is the existing upper *VIEW* scale of this text item before replacement.

Additional Information

@tiUpperScale ♦

3.4.12 Formatted Text Item

A formatted text item *component* defines an RTF (Rich Text Format) text string containing the characteristics of the text within the RTF codes.

For fixed formatted text item *components*, measurements are in *logical picas* rather than *base units*.

The *current drawing plane* must be the same as the *drawing plane* within which the formatted text item *component* instance was created to avoid failure when using the *component*'s members.

Note that the background colour of a formatted text item does not change with the background colour of the *drawing plane*.

Information

- Initials: `ft`
- Type: **visC_FormText**
- Parent: [Graphics Object](#)
- Creator: **vis_FormText()** ▪ **vis_FixFormText()**
- Deletion: **@visDelete()**
- Access: **@VIS()** ▪ **@VISX()**

— Member Variables —

@ftAutoFitBound

@ftAutoFitBound := *bool* (*true*)

bool An integer stack object containing a logical boolean value.

Details

Sets whether or not (*bool*) the internal bounding box is fitted around the text of the formatted text item *component* instance.

bool is `true` to allow VIS to automatically fit the height of the internal bounding box to the height needed to draw the text specified by `@ftText`, otherwise the internal bounding box will be as specified by `@ftBoundingBox`.

VIS automatically fits the bounding box to the text by calling the `@ftFitBoundingBox()` function. This occurs when the formatted text item is being created, and by the `@ftRepText()` function as long as `@ftAutoFitBound` contains a `true` value.

`@ftAutoFitBound` can be set at any time.

Related Information

`@ftRepText()`

Other Information

`@ftBoundingBox` - `@ftFitBoundingBox()` ♦

`@ftBkColr`

`@ftBkColr` := *bk-colr* (?)

bk-colr An integer stack object.

Details

Sets the background colour (*bk-colr*) of the text specified for this formatted text item *component* instance.

bk-colr is an RGB colour which can be obtained from the `vis_RGB()` function. If a NULL stack object (?) is used instead of *bk-colr* then the background colour is determined by the value of `@dpFTBkColr` of the *active* drawing plane *component* instance when the formatted text item is being created.

The background colour can be altered at any time via the `@ftRepBkColr()` function, which uses this variable, after this formatted text item *component* instance has been created.

Additional Information

`@dpFTBkColr`

Related Information

`@ftRepBkColr()` ♦

Other Information

[Standard Colours](#) - [Common Colours](#) - `vis_RGB()`

`@ftBoundingBox`

`@ftBoundingBox` := *bound-box* ([0.0, 0.0, 0.0, 0.0])

bound-box A decimal sequence stack object.

Details

Sets the bounding box (*bound-box*) into which the RTF text of the formatted text item *component* instance is drawn.

bound-box is a sequence containing the bounding box of the formatted text item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed formatted text item, the units are *logical picas* not *base units*.

$[x\text{-}min, y\text{-}min, x\text{-}max, y\text{-}max]$,

where

- 1 $x\text{-}min$ is the left border position of the formatted text item's bounding box,
- 2 $y\text{-}min$ is the bottom border position of the formatted text item's bounding box,
- 3 $x\text{-}max$ is the right border position of the formatted text item's bounding box,
- 4 $y\text{-}max$ is the top border position of the formatted text item's bounding box.

The text of the formatted text item is aligned to the top of the bounding box and wrapped to the bounding box width. However, text falling outside the bounding box is clipped if the bounding box is not automatically fitted (via the `@ftFitBoundingBox()` function).

It is **important** to set a value with an appropriate width for this variable; the default value will clip all of the text because the width of the bounding box is zero.

The bounding box can be altered at any time via the `@ftRepBoundingBox()` function, which uses this variable, after this formatted text item *component* instance has been created.

Related Information

`@ftRepBoundingBox()`

Other Information

`@ftAutoFitBound` - `@ftFitBoundingBox()` ♦

@ftFixed

`@ftFixed := fixed (false)`

fixed An integer stack object containing a logical boolean value.

Details

If true, this is a fixed item.

If `@ftFixed` is true, then this text item *component* instance is a fixed item, otherwise it is not. This value is set only by VIS when a fixed text item *component* instance is created. The program should not set this value.

Related Information

`vis_FixFormText()` ♦

@ftLowerScale

`@ftLowerScale := scale (0.0)`

scale A decimal stack object.

Details

Sets the lower *VIEW* scale (*scale*) below which this formatted text item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is below this value, this formatted text item will be invisible but will still take up space in the *drawing plane*. Otherwise, the formatted text item is displayed as usual.

The lower *VIEW* scale can be altered at any time via the `@ftRepLwrScale()` function, which uses this variable, after this formatted text item *component* instance has been created.

Related Information

`@ftRepLwrScale()` ♦

@ftText

```
@ftText := text ("")
```

text A string stack object.

Details

Sets the RTF text (*text*), containing formatting RTF codes, for this text item *component* instance.

text contains the text with embedded formatting. RTF (**R**ich **T**ext **F**ormat) is an industry standard. Note that only a very small subset of the RTF codes, as determined by the operating system, is recognised by VIS.

The text can be altered at any time via the `@ftRepText()` function, which uses this variable, after this formatted text item *component* instance has been created.

Related Information

`@ftRepText()` ♦

@ftUpperScale

```
@ftUpperScale := scale (:#+max_dec:)
```

scale A decimal stack object.

Details

Sets the upper *VIEW* scale (*scale*) above or equal to which this formatted text item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is equal to or above this value, this formatted text item will be invisible but will still take up space in the *drawing plane*. Otherwise, the formatted text item is displayed as usual.

The upper *VIEW* scale can be altered at any time via the `@ftRepUprScale()` function, which uses this variable, after this formatted text item *component* instance has been created.

Related Information

`@ftRepUprScale()` ♦

— Member Functions —

@ftFitBoundingBox()

```
@ftFitBoundingBox() → success
```

success An integer stack object containing a logical boolean value.

Details

Fits the internal bounding box around the text of this formatted text item *component* instance.

success is true if the bounding box was changed to fit around the text, otherwise it is false.

This function changes the internal bounding box to fit its height to the height needed to draw the text. The width of the bounding box is not altered.

The `@ftGetBoundingBox()` function can be called to get the internal bounding of the formatted text item after having been fitted.

Other Information

`@ftAutoFitBound` - `@ftGetBoundingBox()` ♦

@ftGetBoundingBox()

`@ftGetBoundingBox()` → *bound-box*

bound-box A decimal sequence stack object.

Details

Gets the internal bounding box of this formatted text item *component* instance.

bound-box is a sequence containing the internal bounding box of the formatted text item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed formatted text item, the units are *logical picas* not *base units*.

$[x-min, y-min, x-max, y-max]$,

where

- 1 *x-min* is the left border position of the formatted text item's bounding box,
- 2 *y-min* is the bottom border position of the formatted text item's bounding box,
- 3 *x-max* is the right border position of the formatted text item's bounding box,
- 4 *y-max* is the top border position of the formatted text item's bounding box.

The internal bounding box of the formatted text item may not be the same as the one set by the `@ftBoundingBox` variable if the `@ftAutoFitBound` variable is true when the formatted text is changed, or the `@ftFitBoundingBox()` function was called.

Other Information

`@ftBoundingBox` - `@ftAutoFitBound` - `@ftFitBoundingBox()` - `@ftRepBoundingBox()` ♦

@ftRepBkColr()

`@ftRepBkColr()` [*@ftBkColr*] → *prev-colr*

@ftBkColr An integer value.

bk-colr An integer stack object.

Details

Replaces the existing background colour of the text of this formatted text *component* instance with the specified one (*@ftBkColr*), and returns the existing colour (*prev-colr*).

@ftBkColr contains the background colour for this formatted text. If *@ftBkColr* contains a NULL stack object (?), then only the existing background colour is returned (*prev-colr*).

prev-colr is the existing background colour before the new colour is set.

Additional Information**@ftBkColr** ♦**@ftRepBoundBox()****@ftRepBoundBox()** [*@ftBoundBox*] → *bound-box**@ftBoundBox* A decimal sequence.*bound-box* A decimal sequence stack object.**Details**

Replaces and returns the actual bounding box of this item.

Replaces the existing internal bounding box of this formatted text item *component* instance with the specified one (*@ftBoundBox*), and returns the existing internal bounding box (*bound-box*).

@ftBoundBox contains the new bounding box relative to the parent graphics object *component* instance, as described below at *bound-box*.

bound-box is a sequence containing the bounding box of the formatted text item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed formatted text item, the units are *logical picas* not *base units*.

[*x-min*, *y-min*, *x-max*, *y-max*],

where

- 1 *x-min* is the left border position of the formatted text item's bounding box,
- 2 *y-min* is the bottom border position of the formatted text item's bounding box,
- 3 *x-max* is the right border position of the formatted text item's bounding box,
- 4 *y-max* is the top border position of the formatted text item's bounding box.

The internal bounding box of the formatted text item may become different from the one set by the **@ftBoundBox** variable if the **@ftAutoFitBound** variable is **true** when the formatted text is changed, or the **@ftFitBoundBox()** function is called.

Additional Information**@ftBoundBox****Other Information****@ftGetBoundBox()** ♦**@ftRepLwrScale()****@ftRepLwrScale()** [*@ftLowerScale*] → *prev-scale**@ftLowerScale* A decimal value.*prev-scale* A decimal stack object.**Details**

Replaces the existing lower *VIEW* scale of this formatted text item *component* instance with the specified one (*@ftLowerScale*), and returns the existing scale (*prev-scale*).

@ftLowerScale contains a non-negative number.

prev-scale is the existing lower *VIEW* scale of this formatted text item before replacement.

Additional Information**@ftLowerScale** ♦**@ftRepText()****@ftRepText()** [*@ftText* *@ftAutoFitBound*] → *prev-text**@ftText* A string value.*@ftAutoFitBound* An integer value containing a logical boolean value.*prev-text* A string stack object.**Details**

Replaces the existing RTF text of this formatted text item *component* instance with the specified one (*@ftText*), and returns the existing text (*prev-text*).

@ftText v the new RTF text to replace the existing text.

@ftAutoFitBound contains `true` to automatically fit the internal bounding box (via **@ftFitBoundingBox()**) around the replaced text, otherwise the bounding box is not fitted.

prev-text is the existing text of this formatted text item before replacement.

Additional Information**@ftText** ▪ **@ftAutoFitBound****Other Information****@ftFitBoundingBox()** ♦**@ftRepUpScale()****@ftRepUpScale()** [*@ftUpperScale*] → *prev-scale**@ftUpperScale* A decimal value.*prev-scale* A decimal stack object.**Details**

Replaces and returns the upper *VIEW* scale.

Replaces the existing upper *VIEW* scale of this formatted text item *component* instance with the specified one (*@ftUpperScale*), and returns the existing scale (*prev-scale*).

@ftUpperScale contains a non-negative number.

prev-scale is the existing upper *VIEW* scale of this formatted text item before replacement.

Additional Information**@ftUpperScale** ♦**3.4.13 Image Item**

An image item *component* defines a bitmap image. Only BMP image formats are supported.

For fixed image item *components*, measurements are in *logical picas* rather than *base units*.

The *current drawing plane* must be the same as the *drawing plane* within which the image item *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: `ii`
- Type: `visC_ImageItem`
- Parent: [Graphics Object](#)
- Creator: `vis_ImageItem()` ▪ `vis_FixImageItem()`
- Deletion: `@visDelete()`
- Access: `@VIS()` ▪ `@VISX()`

— Member Variables —

@iiBitmapID

`@iiBitmapID := bitmap-id (?)`

bitmap-id An integer or string stack object.

Details

Sets the identifier (*bitmap-id*) of the bitmap attributes *component* instance for this image item *component* instance when being created.

bitmap-id is the *crucial identifier* or *name identifier* of a previously created bitmap attributes *component* instance. If a NULL stack object (?) is used instead of *bitmap-id* when this image item is being created, then no image will appear and the image item will not take up space in the *drawing plane*.

The bitmap attributes can be altered at any time via the `@iiRepBitmap()` function, which uses this variable, after this image item *component* instance has been created.

Related Information

[3.4.17_Bitmap Attributes](#) ▪ `@iiRepBitmap()` ♦

@iiBoundingBox

`@iiBoundingBox := bound-box (?)`

bound-box A decimal sequence stack object.

Details

Sets the bounding (*bound-box*) into which the image of this image item *component* instance is drawn.

bound-box is a sequence containing the bounding box of the image item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed image item, the units are *logical picas* not *base units*.

$[x-min, y-min, x-max, y-max]$,

where

- 1 *x-min* is the left border position of the image item's bounding box,
- 2 *y-min* is the bottom border position of the image item's bounding box,
- 3 *x-max* is the right border position of the image item's bounding box,
- 4 *y-max* is the top border position of the image item's bounding box.

If a NULL stack object (?) is used instead of *bound-box*, then the bounding box is automatically fitted (via `@iiFitBoundingBox()`) around the natural size of the image of this image item.

The entire image of the image item is drawn within the bounding box; the image is stretched or shrunk as necessary, but not cropped.

The bounding box can be altered at any time via the `@iiRepBoundingBox()` function, which uses this variable, after this image item *component* instance has been created.

- Related Information**
- `@iiRepBoundingBox()`
- Other Information**
- `@iiFitBoundingBox()` ♦

@iiFixed	
<code>@iiFixed := <i>fixed</i> (false)</code>	
<i>fixed</i>	An integer stack object containing a logical boolean value.

Details

If true, this is a fixed item.

If `@iiFixed` is true, then this image item *component* instance is a fixed item, otherwise it is not. This value is set only by VIS when a fixed image item *component* instance is created. The program should not set this value.

- Other Information**
- `vis_FixImageItem()` ♦

@iiLowerScale	
<code>@iiLowerScale := <i>scale</i> (0.0)</code>	
<i>scale</i>	A decimal stack object.

Details

Sets the lower *VIEW* scale (*scale*) below which this image item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is below this value, this image item will be invisible but will still take up space in the *drawing plane*. Otherwise, the image item is displayed as usual.

The lower *VIEW* scale can be altered at any time via the `@iiRepLwrScale()` function, which uses this variable, after this image item *component* instance has been created.

- Related Information**
- `@iiRepLwrScale()` ♦

@iiUpperScale

@iiUpperScale := *scale* (:#+max_dec:)

scale A decimal stack object.

Details

The highest *VIEW* scale above or equal to which this item becomes invisible.

Sets the upper *VIEW* scale (*scale*) above or equal to which this image item *component* instance becomes invisible.

scale is a non-negative number. Whenever the *VIEW* scale is equal to or above this value, this image item will be invisible but will still take up space in the *drawing plane*. Otherwise, the image item is displayed as usual.

The upper *VIEW* scale can be altered at any time via the **@iiRepUpScale()** function, which uses this variable, after this image item *component* instance has been created.

Related Information

@iiRepUpScale() ♦

— Member Functions —

@iiFitBoundingBox()

@iiFitBoundingBox() → *success*

success An integer stack object containing a logical boolean value.

Details

Fits the bounding box around the natural size of the image of this image item *component* instance.

success is true if the bounding box was fitted around the image, otherwise it is false.

If the image item is not associated with a bitmap attributes *component* instance, then *success* will be false.

Other Information

@iiGetBoundingBox() ♦

@iiGetBoundingBox()

@iiGetBoundingBox() → *bound-box*

bound-box A decimal sequence stack object.

Details

Gets the bounding box of this image item *component* instance.

bound-box is a sequence containing the internal bounding box of the image item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed image item, the units are *logical picas* not *base units*.

[*x-min*, *y-min*, *x-max*, *y-max*],

where

- 1 $x-min$ is the left border position of the image item's bounding box,
- 2 $y-min$ is the bottom border position of the image item's bounding box,
- 3 $x-max$ is the right border position of the image item's bounding box,
- 4 $y-max$ is the top border position of the image item's bounding box.

The bounding box of the image item may not be the same as the one set by the `@iiBoundingBox` variable if the `@iiFitBoundingBox()` function was called.

Other Information

`@iiBoundingBox` ▪ `@iiFitBoundingBox()` ▪ `@iiRepBoundingBox()` ♦

@iiRepBitmap()

`@iiRepBitmap()` [`@iiBitmapID`] → *prev-cid* | ?

@iiBitmapID An integer or string value, or a NULL stack object.

bitmap-cid An integer stack object.

Details

Replaces the existing bitmap attributes *component* instance of this image text item *component* instance with the specified one (*@iiBitmapID*), and returns the *crucial identifier* of the existing bitmap attributes *component* instance.

@iiBitmapID contains the *crucial identifier* or *name identifier* of a bitmap attributes *component* instance of the *current drawing plane*. If *@iiBitmapID* contains a NULL stack object (?), then only the *CID* of the existing bitmap attributes (if any) is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing bitmap attributes of this image item before replacement, or a NULL stack object (?) is returned if this image item had no bitmap attributes.

Additional Information

`@iiBitmapID` ♦

@iiRepBoundingBox()

`@iiRepBoundingBox()` [`@iiBoundingBox`] → *bound-box*

@iiBoundingBox A decimal sequence.

bound-box A decimal sequence stack object.

Details

Replaces the existing bounding box of this image item *component* instance with the specified one (*@iiBoundingBox*), and returns the existing internal bounding box (*bound-box*).

@iiBoundingBox contains the new bounding box relative to the parent graphics object *component* instance, as described below at *bound-box*.

bound-box is a sequence containing the bounding box of the image item. The elements of the sequence are in *base units* relative to the origin of the parent graphics object as follows. For a fixed image item, the units are *logical picas* not *base units*.

[$x-min$, $y-min$, $x-max$, $y-max$],

where

- 1 *x-min* is the left border position of the image item's bounding box,
- 2 *y-min* is the bottom border position of the image item's bounding box,
- 3 *x-max* is the right border position of the image item's bounding box,
- 4 *y-max* is the top border position of the image item's bounding box.

The bounding box of the image item may become different from the one set by the `@iiBoundingBox` variable if the `@iiFitBoundingBox()` function is called.

Additional Information

`@iiBoundingBox`

Other Information

`@iiGetBoundingBox()` ♦

`@iiRepLwrScale()`

`@iiRepLwrScale()` [`@iiLowerScale`] → *prev-scale*

@iiLowerScale A decimal value.

prev-scale A decimal stack object.

Details

Replaces the existing lower *VIEW* scale of this image item *component* instance with the specified one (*@iiLowerScale*), and returns the existing scale (*prev-scale*).

@iiLowerScale contains a non-negative number.

prev-scale is the existing lower *VIEW* scale of this image item before replacement.

Additional Information

`@iiLowerScale` ♦

`@iiRepUprScale()`

`@iiRepUprScale()` [`@iiUpperScale`] → *prev-scale*

@iiUpperScale A decimal value.

prev-scale A decimal stack object.

Details

Replaces the existing upper *VIEW* scale of this image item *component* instance with the specified one (*@iiUpperScale*), and returns the existing scale (*prev-scale*).

@iiUpperScale contains a non-negative number.

prev-scale is the existing upper *VIEW* scale of this image item before replacement.

Additional Information

`@iiUpperScale` ♦

3.4.14 Graphics Context

A graphics context *component* defines the graphics context for use in graphics object *component* instances. Currently, the graphics context provides only highlighting features for the graphics

object. Only one graphics context *component* instance can be associated with a graphics object *component* instance.

The *current drawing plane* must be the same as the *drawing plane* within which the graphics context *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: gc
- Type: **visC_GContext**
- Parent: [Drawing Plane](#)
- Creation Context: [Graphics Object](#)
- Creator: **vis_GContext()**
- Deletion: **@visDelete()**
- Access: **@VIS()** ▪ **@VISX()**

— Member Variables —

@gcBrushID

@gcBrushID := *brush-id* (?)

brush-id An integer or string stack object.

Details

Sets the identifier (*brush-id*) of the brush attributes *component* instance for this graphics context *component* instance for background highlighting if specified.

brush-id is the *crucial identifier* or *name identifier* of a previously created brush attributes *component* instance. If a NULL stack object (?) is used instead of *brush-id*, then the graphics context will assume a default colour (solid grey colour RGB: 200 200 200) for the background highlighting of the associated graphics object *component* instance when the graphics context is being created.

The brush attributes *component* instance is used to fill the background highlighting of the associated graphics object *component* instance, but only if **: !OH_BKGRND_HILITE:** is set in **@gcOHFlags**; the background highlighting covers the whole bounding box of the graphics object.

The brush attributes can be altered at any time via the **@gcRepBrush()** function, which uses this variable, after this graphics context *component* instance has been created.

Related Information

[3.4.16_Brush Attributes](#) ▪ **@gcRepBrush()**

Other Information

@gcOHFlags ♦

@gcOHFlags

@gcOHFlags := *oh-flags* (0)

oh-flags An integer stack object containing binary flags.

Details

Sets the highlight flags (*oh-flags*) for this graphics context *component* instance.

oh-flags is zero or more of the values listed in [Object Highlight Type](#). If this value is 0 (zero), then the associated graphics object *component* instance will not be highlighted.

The flags can be altered at any time via the `@gcRepOHFlags()` function, which modifies this variable, after this graphics context *component* instance has been created.

Additional Information

[Object Highlight Type](#)

Related Information

`@gcRepOHFlags()` ♦

@gcOTBordWidth

`@gcOTBordWidth := width (0.5)`

width A decimal stack object.

Details

Sets the object tracker border and resize boxes width (*width*) for this graphics context *component* instance for tracker rectangle highlighting if specified.

width is in *screen picas*.

This variable has effect only if `:!OH_TRACKER_HILITE:` is set in `@gcOHFlags`.

The width can be altered at any time via the `@gcNewObjTracker()` function, which uses this variable, after this graphics context *component* instance has been created.

Related Information

`@gcNewObjTracker()`

Other Information

`@gcOHFlags` ♦

@gcOTStyles

`@gcOTStyles := ot-styles (:!OT_HATCH_BORD:)`

ot-styles An integer stack object containing binary flags.

Details

Sets the object tracker styles (*ot-styles*) for this graphics context *component* instance for tracker rectangle highlighting if specified.

ot-styles is one or more of the values listed in [Object Tracker Styles](#). A value of 0 (zero) is the same as `:!OT_CLEAR:`.

This variable has effect only if `:!OH_TRACKER_HILITE:` is set in `@gcOHFlags`.

The width can be altered at any time via the `@gcNewObjTracker()` function, which uses this variable, after this graphics context *component* instance has been created.

Additional Information

[Object Tracker Styles](#)

Related Information[@gcNewObjTracker\(\)](#)**Other Information**[@gcOHFlags](#) ♦**@gcPenID****@gcPenID** := *pen-id* (?)*pen-id* An integer or string stack object.**Details**

Sets the identifier (*pen-id*) of the pen attributes *component* instance for this graphics context *component* instance for foreground highlighting if specified.

pen-id is the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance. If a NULL stack object (?) is used instead of *pen-id*, then the graphics context will assume a default outline (dotted one-pixel black outline) for the foreground highlighting of the associated graphics object *component* instance when the graphics context is being created.

The pen attributes *component* instance is used for the foreground highlighting of the associated graphics object *component* instance, but only if : !OH_BORD_HILITE: is set in [@gcOHFlags](#); the foreground highlighting is a rectangle surrounding the whole bounding box of the graphics object.

The pen attributes can be altered at any time via the [@gcRepPen\(\)](#) function, which uses this variable, after this graphics context *component* instance has been created.

Related Information[3.4.15_Pen Attributes](#) - [@gcRepPen\(\)](#)**Other Information**[@gcOHFlags](#) ♦**— Member Functions —****@gcNewObjTracker()****@gcNewObjTracker()** [[@gcOTStyles](#) [@gcOTBordWidth](#)][@gcOTStyles](#) An integer value containing binary flags.[@gcOTBordWidth](#) A decimal value.**Details**

Replaces the object tracker details with the specified ones ([@gcOTStyles](#), [@gcOTBordWidth](#)).

[@gcOTStyles](#) contains the styles used with the object tracker.

[@gcOTBordWidth](#) contains the border width of the object tracker in *screen picas*.

Note that the existing styles and border width of this object tracker cannot be obtained from VIS; they can only be set.

Examples

The following illustrations show how the [@gcNewObjTracker\(\)](#) function can be used.

```
(1) @gcNewObjTracker (@gcOTStyles := (:!OT_HATCH_BORD: &or :!OT_BOXES_OUT:)
    @gcOTBordWidth := :!OT_NULL:);
(2) @gcNewObjTracker (@gcOTBordWidth := 0.75);
```

In example (1), the object tracker is set to have a hatched border with the control handles on the outside of the bounding box of the associated graphics object *component* instance. The border width of the object tracker is unaltered. Note that the code snip `@gcOTBordWidth := :!OT_NULL:` discards the information about the border width, which cannot thereafter be obtained from VIS.

In example (2), the object tracker border width is set to 0.75 *screen picas* (equivalent to 12 pixels, regardless of the DPI setting of the screen). The object tracker styles are also set to whatever value is contained in `@gcOTStyles`.

Additional Information

`@gcOTStyles` - `@gcOTBordWidth` ♦

@gcRepBrush()

`@gcRepBrush ()` [`@gcBrushID`] → *prev-cid* | ?

@gcBrushID An integer or string value, or a NULL stack object.

prev-cid An integer stack object.

Details

Replaces the existing brush attributes *component* instance of this graphics context *component* instance with the specified one (*@gcBrushID*), and returns the *crucial identifier* of the existing brush attributes *component* instance.

@gcBrushID contains the *crucial identifier* or *name identifier* of the brush attributes *component* instance of the *current drawing plane*. If *@gcBrushID* contains a NULL stack object (?), then only the *CID* of the existing brush attributes (if any) is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing brush attributes of this graphics context before replacement, or a NULL stack object (?) is returned if this graphics context did not have brush attributes.

Additional Information

`@gcBrushID` ♦

@gcRepOHFlags()

`@gcRepOHFlags (pFlags pAdd)` → *prev-flags* [`@gcOHFlags`]

pFlags An integer stack object containing binary flags.

pAdd An integer stack object containing a logical boolean value.

prev-flags An integer stack object.

@gcOHFlags An integer value containing binary flags.

Details

Replaces the existing highlight flags of this graphics context *component* instance with the specified ones (*pFlags*), and returns the existing flags (*prev-flags*).

pFlags is one or more of the values listed in [Object Highlight Type](#). If this value is 0 (zero), then no flags are replaced, but the existing flags (*prev-flags*) are returned by the function.

pAdd is true to add the flags in *pFlags* to the existing ones in this graphics context, or false to remove the flags in *pFlags* from the existing ones in this graphics context.

prev-flags are the existing flags before replacement.

@gcOHFlags contains the highlight flags of this graphics context after replacement.

Additional Information

[Object Highlight Type](#) ▪ [@gcOHFlags](#) ♦

@gcRepPen()

@gcRepPen() [*@gcPenID*] → *prev-cid* | ?

@gcPenID An integer or string value, or a NULL stack object.

prev-cid An integer stack object.

Details

Replaces the existing pen attributes *component* instance of this graphics context *component* instance with the specified one (*@gcPenID*), and returns the *crucial identifier* of the existing pen attributes *component* instance.

@gcPenID contains the *crucial identifier* or *name identifier* of a pen attributes *component* instance of the *current drawing plane*. If *@gcPenID* contains a NULL stack object (?), then only the *CID* of the existing pen attributes (if any) is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing pen attributes of this graphics context before replacement, or a NULL stack object (?) is returned if this graphics context did not have pen attributes.

Additional Information

[@gcPenID](#) ♦

3.4.15 Pen Attributes

A pen attributes *component* defines the pen attributes for use in various *component* instances. Only one pen attributes *component* instance can be associated with a *component* instance.

The *current drawing plane* must be the same as the *drawing plane* within which the pen attributes *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: pn
- Type: **visC_Pen**
- Parent: [Drawing Plane](#)
- Creation Context: [Line Item](#) ▪ [Curve Item](#) ▪ [Graphics Context](#)
- Creator: **vis_Pen()**
- Deletion: **@visDelete()**
- Access: **@VIS()** ▪ **@VISX()**

— Member Variables —

@pnBkColr**@pnBkColr** := *bk-colr* (?)*bk-colr* An integer stack object.**Details**

Sets the pen background colour (*bk-colr*) of this pen attributes *component* instance.

bk-colr is an RGB colour which can be obtained from the **vis_RGB()** function. If a NULL stack object (?) is used instead of *bk-colr* then the background colour is the default white colour when the pen attributes instance is being created.

The background colour is used to fill the gaps in styled lines (depends on the **@pnBkMode** setting).

The background colour can be altered at any time via the **@pnRepBkColr()** function, which uses this variable, after this pen attributes *component* instance has been created.

Related Information**@pnRepBkColr()****Other Information**
[Standard Colours](#) ▪ [Common Colours](#) ▪ **vis_RGB()** ▪ **@pnBkMode** ♦
@pnBkMode**@pnBkMode** := *bk-mode* (?)*bk-mode* An integer stack object.**Details**

Sets the background mode (*bk-mode*) of this pen attributes *component* instance.

bk-mode is one of the values listed in [Background Modes](#). If a NULL stack object (?) is used instead of *bk-mode*, then the default value **:#TRANSPARENT:** will be assumed for the background mode when this pen attributes instance is being created.

The background mode can be altered at any time via the **@pnRepBkMode()** function, which uses this variable, after this pen attributes *component* instance has been created.

Additional Information[Background Modes](#)**Related Information****@pnRepBkMode()** ♦

@pnColr

@pnColr := *colr* (:#RGB_Black:)

colr An integer stack object.

Details

Sets the pen foreground colour (*colr*) of this pen attributes *component* instance.

colr is an RGB colour which can be obtained from the **vis_RGB()** function.

The foreground colour can be altered at any time via the **@pnNewPen()** function, which uses this variable, after this pen attributes *component* instance has been created.

Related Information

@pnNewPen()

Other Information

[Standard Colours](#) - [Common Colours](#) - **vis_RGB()** ♦

@pnDrawMode

@pnDrawMode := *draw-mode* (?)

draw-mode An integer stack object.

Details

Sets the drawing mode (*draw-mode*) of this pen attributes *component* instance.

draw-mode is one of the values listed in [Drawing Modes](#). If a NULL stack object (?) is used instead of *draw-mode*, then the default value **:#R2_COPYPEN:** will be assumed for the drawing mode when this pen attributes instance is being created.

The drawing mode can be altered at any time via the **@pnRepDrawMode()** function, which uses this variable, after this pen attributes *component* instance has been created.

Additional Information

[Drawing Modes](#)

Related Information

@pnRepDrawMode() ♦

@pnStyle

@pnStyle := *style* (:#PS_SOLID:)

style An integer stack object.

Details

Sets the pen style (*style*) of this pen attributes *component* instance.

style is one of the values listed in [Pen Styles](#).

The pen style can be altered at any time via the **@pnNewPen()** function, which uses this variable, after this pen attributes *component* instance has been created.

Additional InformationPen Styles**Related Information****@pnNewPen ()** ♦**@pnWidth****@pnWidth** := *width* (0.0)*width* A decimal stack object.**Details**

Sets the pen width (*width*) of this pen attributes *component* instance.

width is the width of the rendered pen in *base units*. A value of 0.0 (zero) renders the pen one pixel wide for all *VIEW* scales.

The pen width can be altered at any time via the **@pnNewPen ()** function, which uses this variable, after this pen attributes *component* instance has been created.

Related Information**@pnNewPen ()** ♦**— Member Functions —****@pnNewPen()****@pnNewPen ()** [*@pnColr* *@pnStyle* *@pnWidth*]*@pnColr* An integer value.*@pnWidth* A decimal value.*@pnStyle* An integer value.**Details**

Sets the pen details (*@pnColr*, *@pnStyle*, *@pnWidth*) of this pen attributes *component* instance.

@pnColr contains the pen foreground colour.

@pnStyle contains the pen style.

@pnWidth contains the pen width in *base units*.

Note that the existing foreground colour, style, and width of this pen attributes instance cannot be obtained from VIS; they can only be set.

Additional Information**@pnColr** ▪ **@pnStyle** ▪ **@pnWidth** ♦

@pnRepBkColr()**@pnRepBkColr ()** [*@pnBkColr*] → *prev-colr**@pnBkColr* An integer value, or a NULL stack object.*prev-colr* An integer stack object.**Details**

Replaces the existing background colour of this pen attributes *component* instance with the specified one (*@pnBkColr*), and returns the existing colour (*prev-colr*).

@pnBkColr contains the background colour of this pen attributes instance. If *@pnBkColr* contains a NULL stack object (?), then only the existing background colour is returned (*prev-colr*).

prev-colr is the existing background colour before the new colour is set.

Additional Information**@pnBkColr** ♦**@pnRepBkMode()****@pnRepBkMode ()** [*@pnBkMode*] → *prev-mode**@pnBkMode* An integer value, or a NULL stack object.*prev-mode* An integer stack object.**Details**

Replaces the existing background mode of this pen attributes *component* instance with the specified one (*@pnBkMode*), and returns the existing mode (*prev-mode*).

@pnBkMode contains the background mode of this pen attributes instance. If *@pnBkMode* contains a NULL stack object (?), then only the existing background mode is returned (*prev-mode*).

prev-mode is the existing background mode before the new mode is set.

Additional Information**@pnBkMode** ♦**@pnRepDrawMode()****@pnRepDrawMode ()** [*@pnDrawMode*] → *prev-mode**@pnDrawMode* An integer value, or a NULL stack object.*prev-mode* An integer stack object.**Details**

Replaces the existing drawing mode of this pen attributes *component* instance with the specified one (*@pnDrawMode*), and returns the existing mode (*prev-mode*).

@pnDrawMode contains the drawing mode of this pen attributes instance. If *@pnDrawMode* contains a NULL stack object (?), then only the existing drawing mode is returned (*prev-mode*).

prev-mode is the existing drawing mode before the new mode is set.

Additional Information[@pnDrawMode](#) ♦

3.4.16 Brush Attributes

A brush attributes *component* defines the brush attributes for use in various *component* instances. Only one brush attributes *component* instance can be associated with a *component* instance.

The *current drawing plane* must be the same as the *drawing plane* within which the brush attributes *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: br
- Type: **visC_Brush**
- Parent: [Drawing Plane](#)
- Creation Context: [Line Item](#) ▪ [Curve Item](#) ▪ [Graphics Context](#)
- Creator: **vis_Brush()**
- Deletion: **@visDelete()**
- Access: **@VIS()** ▪ **@VISX()**

— Member Variables —

@brBitmapID

@brBitmapID := *bitmap-id* (?)*bitmap-id* An integer or string stack object.Details

Sets the identifier (*bitmap-id*) of the bitmap attributes *component* instance for this brush attributes *component* instance when being created. A bitmap attributes instance is required only if **@brStyle** contains the value **:#BS_PATTERN:**.

bitmap-id is the *crucial identifier* or *name identifier* of a previously created bitmap attributes *component* instance. If a NULL stack object (?) is used instead of *bitmap-id* and **@brStyle** contains the value **:#BS_PATTERN:** and bitmap attributes have not already been set (via the **vis_Bitmap()** function) when this brush attributes instance is being created, then a fatal error will occur. If **@brStyle** does not contain the value **:#BS_PATTERN:**, then the value contained in **@brBitmapID** is ignored.

The bitmap attributes can be altered at any time via the **@brRepBitmap()** and **@brNewBrush()** functions, which use this variable, after this image item *component* instance has been created.

Additional Information[@brStyle](#)Related Information[3.4.17_Bitmap Attributes](#) ▪ [@brRepBitmap\(\)](#) ▪ [@brNewBrush\(\)](#)Other Information[vis_Bitmap\(\)](#) ♦

@brBkColr

@brBkColr := *bk-colr* (?)

bk-colr An integer stack object.

Details

Sets the brush background colour (*bk-colr*) of this brush attributes *component* instance.

bk-colr is an RGB colour which can be obtained from the **vis_RGB()** function. If a NULL stack object (?) is used instead of *bk-colr* then the background colour is the default white colour when the brush attributes instance is being created.

The background colour is used to fill the gaps between hatched lines (depends on the **@brBkMode** setting).

The background colour can be altered at any time via the **@brRepBkColr()** function, which uses this variable, after this brush attributes *component* instance has been created.

Related Information

@brRepBkColr()

Other Information

[Standard Colours](#) ▪ [Common Colours](#) ▪ **vis_RGB()** ▪ **@brBkMode** ♦

@brBkMode

@brBkMode := *bk-mode* (?)

bk-mode An integer stack object.

Details

Sets the background mode (*bk-mode*) of this brush attributes *component* instance.

bk-mode is one of the values listed in [Background Modes](#). If a NULL stack object (?) is used instead of *bk-mode*, then the default value **:#TRANSPARENT:** will be assumed for the background mode when this brush attributes instance is being created.

The background mode can be altered at any time via the **@brRepBkMode()** function, which uses this variable, after this brush attributes *component* instance has been created.

Additional Information

[Background Modes](#)

Related Information

@brRepBkMode() ♦

@brColr

@brColr := *colr* (:#RGB_MEDIUM_GREY:)

colr An integer stack object.

Details

Sets the brush foreground colour (*colr*) of this brush attributes *component* instance.

color is an RGB colour which can be obtained from the **vis_RGB()** function.

The foreground colour can be altered at any time via the **@brNewBrush()** function, which uses this variable, after this brush attributes *component* instance has been created.

Related Information

@brNewBrush()

Other Information

[Standard Colours](#) - [Common Colours](#) - **vis_RGB()** ♦

@brDrawMode

@brDrawMode := *draw-mode* (?)

draw-mode An integer stack object.

Details

Sets the drawing mode (*draw-mode*) of this brush attributes *component* instance.

draw-mode is one of the values listed in [Drawing Modes](#). If a NULL stack object (?) is used instead of *draw-mode*, then the default value **:#R2_COPYPEN:** will be assumed for the drawing mode when this brush attributes instance is being created.

The drawing mode can be altered at any time via the **@brRepDrawMode()** function, which uses this variable, after this brush attributes *component* instance has been created.

Additional Information

[Drawing Modes](#)

Other Information

@brRepDrawMode() ♦

@brHatch

@brHatch := *style* (:#HS_DIAGCROSS:)

style An integer stack object.

Details

Sets the hatch style (*style*) of this brush attributes *component* instance. A hatch style is required only if **@brStyle** contains the value **:#BS_HATCHED:**.

style is one of the values listed in [Hatch Styles](#).

The hatch style can be altered at any time via the **@brNewBrush()** function, which uses this variable, after this brush attributes *component* instance has been created.

Additional Information

[Hatch Styles](#)

Other Information

@brNewBrush() ♦

@brStyle	
@brStyle	:= style (:#BS_SOLID:)
style	An integer stack object.

Details

Sets the brush style (*style*) of this brush attributes *component* instance.

style is one of the values listed in [Brush Styles](#).

The brush style can be altered at any time via the @brNewBrush() function, which uses this variable, after this brush attributes *component* instance has been created.

Additional Information

[Brush Styles](#)

Other Information

@pnNewPen () ♦

— Member Functions —

@brNewBrush()	
@brNewBrush ()	[@brColr @brStyle @brHatch @brBitmapID]
@brColr	An integer value.
@brStyle	An integer value.
@brHatch	An integer value.
@brBitmapID	An integer or string value.

Details

Sets the brush details (@brColr, @brStyle, @brHatch, @brBitmapID) of this brush attributes *component* instance.

@brColr contains the brush foreground colour.

@brStyle contains the brush style.

@brHatch contains the brush hatch style.

@brBitmap contains the identifier of the brush bitmap attributes. If @brBitmap contains a NULL stack object (?), then the existing brush bitmap attributes instance for this brush attributes remains unaltered.

Note that the existing foreground colour, brush style, and hatch style of this brush attributes instance cannot be obtained from VIS; they can only be set.

Additional Information

@brColr ▪ @brStyle ▪ @brHatch ▪ @brBitmap ♦

@brRepBitmap()**@brRepBitmap()** [*@brBitmapID*] → *prev-cid* | ?*@brBitmapID* An integer or string value, or a NULL stack object.*prev-cid* An integer stack object.**Details**

Replaces and returns the bitmap attributes.

Replaces the existing bitmap attributes *component* instance of this brush attributes *component* instance with the specified one (*@brBitmapID*), and returns the *crucial identifier* of the existing bitmap attributes *component* instance.

@brBitmapID contains the *crucial identifier* or *name identifier* of a bitmap attributes *component* instance of the *current drawing plane*. If *@brBitmapID* contains a NULL stack object (?), then only the *CID* of the existing bitmap attributes (if any) is returned (*prev-cid*).

prev-cid is the *crucial identifier* of the existing bitmap attributes of this brush attributes instance before replacement, or a NULL stack object (?) is returned if this brush attributes instance had no bitmap attributes.

Additional Information**@brBitmapID** ♦**@brRepBkColr()****@brRepBkColr()** [*@brBkColr*] → *prev-colr**@brBkColr* An integer value, or a NULL stack object.*prev-colr* An integer stack object.**Details**

Replaces the existing background colour of this brush attributes *component* instance with the specified one (*@brBkColr*), and returns the existing colour (*prev-colr*).

@brBkColr contains the background colour of this brush attributes instance. If *@brBkColr* contains a NULL stack object (?), then only the existing background colour is returned (*prev-colr*).

prev-colr is the existing background colour before the new colour is set.

Additional Information**@brBkColr** ♦

@brRepBkMode()

@brRepBkMode () [[@brBkMode](#)] → *prev-mode*

[@brBkMode](#) An integer value, or a NULL stack object.

prev-mode An integer stack object.

Details

Replaces the existing background mode of this brush attributes *component* instance with the specified one ([@brBkMode](#)), and returns the existing mode (*prev-mode*).

[@brBkMode](#) contains the background mode of this brush attributes instance. If [@brBkMode](#) contains a NULL stack object (?), then only the existing background mode is returned (*prev-mode*)

prev-mode is the existing background mode before the new mode is set.

Additional Information

[@brBkMode](#) ♦

@brRepDrawMode()

@brRepDrawMode () [[@brDrawMode](#)] → *prev-mode*

[@brDrawMode](#) An integer value, or a NULL stack object.

prev-mode An integer stack object.

Details

Replaces and returns the drawing mode of the brush.

Replaces the existing drawing mode of this brush attributes *component* instance with the specified one ([@brDrawMode](#)), and returns the existing mode (*prev-mode*).

[@brDrawMode](#) contains the drawing mode of this brush attributes instance. If [@brDrawMode](#) contains a NULL stack object (?), then only the existing drawing mode is returned (*prev-mode*)

prev-mode is the existing drawing mode before the new mode is set.

Additional Information

[@brDrawMode](#) ♦

3.4.17 Bitmap Attributes

A bitmap attributes *component* defines the bitmap attributes for use in various *component* instances. Only one bitmap attributes *component* instance can be associated with a *component* instance.

The *current drawing plane* must be the same as the *drawing plane* within which the bitmap attributes *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: `bm`
- Type: **visC_Bitmap**
- Parent: [Drawing Plane](#)
- Creation Context: [Image Item](#) ▪ [Brush Attributes](#)

- Creator: `vis_Bitmap()`
- Deletion: `@visDelete()`
- Access: `@VIS()` ▪ `@VISX()`

— Member Variables —

@bmImage

`@bmImage := image (?)`

image A memory or string stack object.

Details

Sets the BMP file data or file path of a BMP file for this bitmap attributes *component* instance when being created.

image is bitmap image data in the BMP format (as a TAC memory object), or a file path to a file containing bitmap image data in the BMP format. If a NULL stack object (?) is used instead of *image*, then no image is loaded into this bitmap attributes instance, and the other attributes are ignored.

Note that the bitmap image data must be in BMP format; other formats, such PNG, JPEG, GIF, TIFF, etc, are not currently recognised. If a file path is specified, it must be to a BMP file. VIS does not currently provide facilities to convert bitmap files in other formats to the BMP format; there is no shortage of publically available facilities for such conversion.

The bitmap data can be altered at any time via the `@bmNewBitmap()` function, which uses this variable, after this bitmap attributes *component* instance has been created.

Examples

The following illustrations show how the `@bmImage` variable can be used.

```
(1) @bmImage := 'MyFolder\MyPicture.bmp';
(2) ImageData := read_file :!RF_NO_DATA_CHK: 'MyFolder\MyPicture.bmp';
    @bmImage := ImageData;
```

In example (1), a BMP file is directly specified for the bitmap image.

In example (2), the bitmap data of a BMP file is read into the variable `ImageData`, which can be used elsewhere to set the same data, then set to the `@bmImage` variable. `ImageData` contains a TAC memory object containing the contents of the BMP file.

Related Information

`@bmNewBitmap()` ♦

@bmROPMode

`@bmROPMode := rop-mode (: !SRCCOPY:)`

rop-mode An integer stack object.

Details

Sets the ROP (Raster Operation) mode (*rop-mode*) of this bitmap attributes *component* instance.

rop-mode is one of the values listed in [ROP Modes](#).

The ROP mode can be altered at any time via the `@bmNewBitmap()` function, which uses this variable, after this bitmap attributes *component* instance has been created.

Additional Information

[ROP Modes](#)

Related Information

`@bmNewBitmap()` ♦

@bmStretchMode	
@bmStretchMode	:= <i>str-mode</i> (:#STRETCH_DELETESCANS:)
<i>str-mode</i>	An integer stack object.

Details

Sets the stretch mode (*str-mode*) of this bitmap attributes *component* instance.

str-mode is one of the values listed in [Stretch Modes](#).

The stretch mode can be altered at any time via the `@bmNewBitmap()` function, which uses this variable, after this bitmap attributes *component* instance has been created.

Additional Information

[Stretch Modes](#)

Related Information

`@bmNewBitmap()` ♦

— Member Functions —

@bmNewBitmap()	
@bmNewBitmap()	[<i>@bmImage @bmROPMode @bmStretchMode</i>]
<i>@bmImage</i>	A memory or string value, or a NULL stack object.
<i>@bmROPMode</i>	An integer value.
<i>@bmStretchMode</i>	An integer value.

Details

Sets the bitmap details (*@bmImage*, *@bmROPMode*, *@bmStretchMode*) of this bitmap attributes *component* instance.

@bmImage contains the bitmap image data.

@bmROPMode contains the bitmap ROP mode.

@bmStretchMode contains the bitmap stretch mode.

Note that the existing image data, ROP mode, and stretch mode of this bitmap attributes instance cannot be obtained from VIS; they can only be set.

Additional Information

`@bmImage` ▪ `@bROPMode` ▪ `@bmStretchMode` ♦

3.4.18 Text Style Attributes

A text style attributes *component* defines the text style attributes for use in the text item *component* instance. Only one text style attributes *component* instance can be associated with a text item *component* instance.

The *current drawing plane* must be the same as the *drawing plane* within which the text style attributes *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: `ts`
- Type: `visC_TextStyle`
- Parent: [Drawing Plane](#)
- Creation Context: [Text Item](#)
- Creator: `vis_TextStyle()`
- Deletion: `@visDelete()`
- Access: `@VIS()` ▪ `@VISX()`

— Member Variables —

@tsBkColr

`@tsBkColr := bk-colr (?)`

bk-colr An integer stack object.

Details

Sets the text background colour (*bk-colr*) of this text style attributes *component* instance.

bk-colr is an RGB colour which can be obtained from the `vis_RGB()` function. If a NULL stack object (?) is used instead of *bk-colr* then the background colour is the default white colour when the text style attributes instance is being created.

The background colour is used to fill the background in character cells (depends on the `@tsBkMode` setting).

The background colour can be altered at any time via the `@tsRepBkColr()` and `@tsPutValues()` functions, which use this variable, after this text style attributes *component* instance has been created.

Related Information

`@tsRepBkColr()` ▪ `@tsPutValues()`

Other Information

[Standard Colours](#) ▪ [Common Colours](#) ▪ `vis_RGB()` ▪ `@tsBkMode` ♦

@tsBkMode

`@tsBkMode := bk-mode (?)`

bk-mode An integer stack object.

Details

Sets the text background mode (*bk-mode*) of this text style attributes *component* instance.

bk-mode is one of the values listed in [Background Modes](#). If a NULL stack object (?) is used instead of *bk-mode*, then the default value **:#TRANSPARENT:** will be assumed for the background mode when this text style attributes instance is being created.

The text background mode can be altered at any time via the **@tsRepBkMode()** and **@tsPutValues()** functions, which use this variable, after this text style attributes *component* instance has been created.

Additional Information

[Background Modes](#)

Related Information

@tsRepBkClr() ▪ **@tsPutValues()** ♦

@tsCharSet

@tsCharSet := *char-set* (:#ANSI_CHARSET:)

char-set An integer stack object.

Details

Sets the character set (*char-set*) of this text style attributes *component* instance.

char-set is one of the values listed in [Character Sets](#).

The character set can be altered at any time via the **@tsNewFont()** and **@tsPutValues()** functions, which use this variable, after this text style attributes *component* instance has been created.

Additional Information

[Character Sets](#)

Related Information

@tsNewFont() ▪ **@tsPutValues()** ♦

@tsClr

@tsClr := *clr* (?)

clr An integer stack object.

Details

Sets the text foreground colour (*clr*) of this text style attributes *component* instance.

clr is an RGB colour which can be obtained from the **vis_RGB()** function.

The foreground colour can be altered at any time via the **@tsRepClr()** and **@tsPutValues()** functions, which use this variable, after this text style attributes *component* instance has been created.

Related Information

@tsRepClr() ▪ **@tsPutValues()**

Other Information

[Standard Colours](#) ▪ [Common Colours](#) ▪ **vis_RGB()** ♦

@tsFaceName

```
@tsFaceName := face-name ("")
```

face-name A string stack object.

Details

Sets the font face name (*face-name*) of this text style attributes *component* instance.

face-name is the face name of a desired font for the text associated with this text style attributes instance. The length of this string must not exceed 31 characters.

The face name can be altered at any time via the @tsNewFont() and @tsPutValues() functions, which use this variable, after this text style attributes *component* instance has been created.

Related Informaion

@tsNewFont() ▪ @tsPutValues() ♦

@tsHeight

```
@tsHeight := height (-1.0)
```

height A decimal stack object.

Details

Sets the font height (*height*) of this text style attributes *component* instance.

height is the cell height (if positive), or character height (if negative) of the text associated with this text style attributes instance. The value is in *base units*. A value of 0.0 (zero) indicates a fixed size height determined by the operating system.

The height can be altered at any time via the @tsNewFont() and @tsPutValues() functions, which use this variable, after this text style attributes *component* instance has been created.

Related Information

@tsNewFont() ▪ @tsPutValues() ♦

@tsItalic

```
@tsItalic := italic (false)
```

italic An integer stack object containing a logical boolean value.

Details

Sets whether or not (*italic*) the text associated with this text style attributes *component* instance is italic.

italic is true if the text is italic, otherwise the text is not italic. (This is *italic* text.)

The italic value can be altered at any time via the @tsNewFont() and @tsPutValues() functions, which use this variable, after this text style attributes *component* instance has been created.

Related Information**@tsNewFont()** ▀ **@tsPutValues()** ♦**@tsPitchAndFamily****@tsPitchAndFamily** := *pitch-fam* (:#VARIABLE_PITCH:)*pitch-fam* An integer stack object.**Details**Sets the pitch and family (*pitch-fam*) of this text style attributes *component* instance.*pitch-fam* is a combined value (one for pitch and one for family) listed in [Pitch and Family](#).The pitch and family can be altered at any time via the **@tsNewFont()** and **@tsPutValues()** functions, which use this variable, after this text style attributes *component* instance has been created.**Additional Information**[Pitch and Family](#)**Related Information****@tsNewFont()** ▀ **@tsPutValues()** ♦**@tsStrikeThru****@tsStrikeThru** := *strike* (false)*strike* An integer stack object containing a logical boolean value.**Details**Sets whether or not (*strike*) the text associated with this text style attributes *component* instance is struck through.*strike* is true if the text is struck through, otherwise the text is not struck through. (This is ~~struck through~~ text.)The strike-through value can be altered at any time via the **@tsNewFont()** and **@tsPutValues()** functions, which use this variable, after this text style attributes *component* instance has been created.**Related Information****@tsNewFont()** ▀ **@tsPutValues()** ♦**@tsUnderline****@tsUnderline** := *underline* (false)*underline* An integer stack object containing a logical boolean value.**Details**Sets whether or not (*underline*) the text associated with this text style attributes *component* instance is underlined.*underline* is true if the text is underlined, otherwise the text is not underlined. (This is underlined text.)

The underline value can be altered at any time via the `@tsNewFont()` and `@tsPutValues()` functions, which use this variable, after this text style attributes *component* instance has been created.

Related Information

`@tsNewFont()` ▪ `@tsPutValues()` ♦

@tsWeight

`@tsWeight := weight (:#FW_NORMAL:)`

weight An integer stack object.

Details

Sets the font weight (*weight*) of this text style attributes *component* instance.

weight is one of the values listed in [Font Weights](#). (This is **bold** text.)

The font weight can be altered at any time via the `@tsNewFont()` and `@tsPutValues()` functions, which use this variable, after this text style attributes *component* instance has been created.

Additional Information

[Font Weights](#)

Related Information

`@tsNewFont()` ▪ `@tsPutValues()` ♦

@tsWidth

`@tsWidth := width (0.0)`

width A decimal stack object.

Details

Sets the font width (*width*) of this text style attributes *component* instance.

width is the average width of the text characters associated with this text style attributes instance. The value is in *base units*. A value of 0.0 (zero) indicates a width determined by the operating system.

The font width can be altered at any time via the `@tsNewFont()` and `@tsPutValues()` functions, which use this variable, after this text style attributes *component* instance has been created.

Related Information

`@tsNewFont()` ▪ `@tsPutValues()` ♦

— Member Functions —

@tsGetValues()

```
@tsGetValues() [@tsHeight @tsWidth @tsWeight @tsItalic @tsUnderline @tsStrikeThru
                 @tsCharSet @tsPitchAndFamily @tsFaceName @tsColr @tsBkColr
                 @tsBkMode] → attr-seq
```

attr-seq A sequence stack object.

Details

Places all the text style attributes of this text style attributes *component* instance into a TAC sequence.

attr-seq is a sequence containing the values of the variables of this text style *component* instance as follows.

[*height, width, weight, italic, underline, strike, char-set, pitch-fam, face-name, colr, bk-colr, bk-mode*],

where

- 1 *height* is the font height (*@tsHeight*),
- 2 *width* is the font width (*@tsWidth*),
- 3 *weight* is the font weight (*@tsWeight*),
- 4 *italic* is the italic value of the font text (*@tsItalic*),
- 5 *underline* is the underline value of the font text (*@tsUnderline*),
- 6 *strike* is the strike-through value of the font text (*@tsStrikeThru*),
- 7 *char-set* is the character set of the font (*@tsCharSet*),
- 8 *pitch-fam* is the pitch and family of the font (*@tsPitchAndFamily*),
- 9 *face-name* is the face name of the font (*@tsFaceName*),
- 10 *colr* is the text foreground colour (*@tsColr*),
- 11 *bk-colr* is the text background colour (*@tsBkColr*),
- 12 *bk-mode* is the text background mode (*@tsBkMode*).

The values of the variables of this text style *component* instance are written unaltered into the sequence (*attr-seq*).

Additional Information

@tsHeight ▪ *@tsWidth* ▪ *@tsWeight* ▪ *@tsItalic* ▪ *@tsUnderline* ▪ *@tsStrikeThru* ▪
@tsCharSet ▪ *@tsPitchAndFamily* ▪ *@tsFaceName* ▪ *@tsColr* ▪ *@tsBkColr* ▪ *@tsBkMode*

Other Information

@tsNewFont() ♦

@tsNewFont()

@tsNewFont() [*@tsHeight* *@tsWidth* *@tsWeight* *@tsItalic* *@tsUnderline* *@tsStrikeThru* *@tsCharSet* *@tsPitchAndFamily* *@tsFaceName*]

<i>@tsHeight</i>	A decimal value.
<i>@tsWidth</i>	A decimal value.
<i>@tsWeight</i>	An integer value.
<i>@tsItalic</i>	An integer value containing a logical boolean value.
<i>@tsUnderline</i>	An integer value containing a logical boolean value.
<i>@tsStrikeThru</i>	An integer value containing a logical boolean value.
<i>@tsCharSet</i>	An integer value.
<i>@tsPitchAndFamily</i>	An integer value.
<i>@tsFaceName</i>	A string value.

Details

Sets new font details of this text style attributes *component* instance.

@tsHeight contains the font height.

@tsWidth contains the font width.

@tsWeight contains the font weight.

@tsItalic contains the italic value of the font text.

@tsUnderline contains the underline value of the font text.

@tsStrikeThru contains the strike-through value of the font text.

@tsCharSet contains the character set of the font.

@tsPitchAndFamily contains the pitch and family of the font.

@tsFaceName contains the face name of the font.

Note that the existing attributes (mentioned above) of this text style instance cannot be obtained from VIS; they can only be set.

Additional Information

@tsHeight ▪ *@tsWidth* ▪ *@tsWeight* ▪ *@tsItalic* ▪ *@tsUnderline* ▪ *@tsStrikeThru* ▪
@tsCharSet ▪ *@tsPitchAndFamily* ▪ *@tsFaceName*

Other Information

@tsGetValues() ▪ *@tsPutValues()* ♦

@tsPutValues()

@tsPutValues (*pAttrSeq* | ?) → [*@tsHeight* *@tsWidth* *@tsWeight* *@tsItalic* *@tsUnderline* *@tsStrikeThru* *@tsCharSet* *@tsPitchAndFamily* *@tsFaceName* *@tsColr* *@tsBkColr* *@tsBkMode*]

pAttrSeq A sequence stack object.

Details

Assigns text style attributes in a TAC sequence (*pAttrSeq*) to this text style attributes *component* instance.

pAttrSeq is a TAC sequence containing the values of the variables of this text style *component* instance as follows.

[*height*, *width*, *weight*, *italic*, *underline*, *strike*, *char-set*, *pitch-fam*, *face-name*, *colr*, *bk-colr*, *bk-mode*],

where

- 1 *height* is the font height (*@tsHeight*),
- 2 *width* is the font width (*@tsWidth*),
- 3 *weight* is the font weight (*@tsWeight*),
- 4 *italic* is the italic value of the font text (*@tsItalic*),
- 5 *underline* is the underline value of the font text (*@tsUnderline*),
- 6 *strike* is the strike-through value of the font text (*@tsStrikeThru*),
- 7 *char-set* is the character set of the font (*@tsCharSet*),
- 8 *pitch-fam* is the pitch and family of the font (*@tsPitchAndFamily*),
- 9 *face-name* is the face name of the font (*@tsFaceName*),
- 10 *colr* is the text foreground colour (*@tsColr*),
- 11 *bk-colr* is the text background colour (*@tsBkColr*),
- 12 *bk-mode* is the text background mode (*@tsBkMode*).

If a NULL stack object (?) is used instead of *pAttrSeq*, then no action occurs. If any value of the sequence is a NULL stack object then the corresponding variable is not assigned. The sequence may contain only an initial segment of the values; the rest of the values are treated as being NULL stack objects.

The values of the sequence (*pAttrSeq*) are written unaltered into the variables of this text style *component* instance.

Examples

The following illustrations show how the **@tsPutValues()** function can be used.

```
(1) @tsPutValues([2.5, ?, ?, true]);
(2) AttrSeq := @VIS("MyTextStyle").@tsGetValues(); vis_TextStyle("NewTS")
    { @tsPutValues(AttrSeq); @tsNewFont(); };
```

In example (1), **@tsHeight** is assigned the value 2.5, and **@tsItalic** is assigned true. The rest of the variables remain unaltered.

In example (2), the values of the member variables of an existing text style attributes *component* instance (MyTextStyle) are assigned to a new text style attributes *component* instance (NewTS). This effectively clones MyTextStyle to NewTS.

Additional Information

@tsHeight ▪ @tsWidth ▪ @tsWeight ▪ @tsItalic ▪ @tsUnderline ▪ @tsStrikeThru ▪
@tsCharSet ▪ @tsPitchAndFamily ▪ @tsFaceName ▪ @tsColr ▪ @tsBkColr ▪ @tsBkMode

Other Information

@tsGetValues () ▪ vis_FontDlg () ♦

@tsRepBkColr()

@tsRepBkColr () [@tsBkColr] → prev-colr

@tsBkColr An integer value, or a NULL stack object (prev-colr).

prev-colr An integer stack object.

Details

Replaces the existing background colour of this text style attributes *component* instance with the specified one (@tsBkColr), and returns the existing background colour (prev-colr).

@tsBkColr contains the background colour for this text style attributes instance. If @tsBkColr contains a NULL stack object (?), then only the existing background colour is returned (prev-colr).

prev-colr is the existing background colour before the new colour is set.

Additional Information

@tsBkColr ♦

@tsRepBkMode()

@tsRepBkMode () [@tsBkMode] → prev-mode

@tsBkMode An integer value, or a NULL stack object.

prev-mode An integer stack object.

Details

Replaces the existing background mode of this text style attributes *component* instance with the specified one (@tsBkMode), and returns the existing mode (prev-mode).

@tsBkMode contains the background mode of this text style attributes instance. If @tsBkMode contains a NULL stack object (?), then only the existing background mode is returned (prev-mode)

prev-mode is the existing background mode before the new mode is set.

Additional Information

@tsBkMode ♦

@tsRepColr()

@tsRepColr() [*@tsColr*] → *prev-colr*

@tsColr An integer value, or a NULL stack object.

prev-colr An integer stack object.

Details

Replaces the existing foreground colour of this text style attributes *component* instance with the specified one (*@tsColr*), and returns the existing foreground colour (*prev-colr*).

@tsColr contains the foreground colour for this text style attributes instance. If *@tsColr* contains a NULL stack object (?), then only the existing foreground colour is returned (*prev-colr*).

prev-colr is the existing foreground colour before the new colour is set.

Additional Information

@tsColr ♦

3.4.19 Matrix Object

A matrix object *component* maintains the list of specified graphics objects to be laid out in relation to each other.

A matrix object *component* instance inherits the properties of a graphics object *component*.

See [2.13_Matrix Objects](#) and [1.3.16_Matrix Object Component](#) for further information about matrix object *components*.

The *current drawing plane* must be the same as the *drawing plane* within which the matrix object *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: `mato`
- Type: **visC_MatObject**
- Derived from: `visGObject`
- Parent: [Drawing Plane](#) ▪ [Graphics Object](#)
- Creator: **vis_MatObject()**
- Deletion: **@visDelete()**
- Access: **@VIS()** ▪ **@VISX()**

— Member Variables —

@matoGObjIDList

@matoGObjIDList := *id-seq* ([1])

id-seq A sequence stack object.

Details

Sets the TAC sequence of graphics object identifiers (*id-seq*) of this matrix object *component* instance.

id-seq is a TAC sequence containing the *crucial identifiers* or *name identifiers* (can be mixed) of the graphics object *component* instances for this matrix object.

The number of elements in *id-seq* must not be less than the number of dynamic constrainers of this matrix object when being created. If *id-seq* contains more elements than the number of dynamic constrainers, then the extra graphics objects will be displayed in their specified position and not partake in the matrix object's layout. The number of dynamic constrainers can be obtained from the `@matoNumDynCons()` function.

The list of graphics object identifiers can be altered at any time via the `@matoUpdateGObjs()` function, which uses this variable, after this matrix object *component* instance has been created.

Related Information

`@matoUpdateGObjs()`

Other Information

`@matoNumDynCons()` ♦

@matoMatDefID

`@matoMatDefID := matdef-id (?)`

matdef-id An integer or string stack object.

Details

Sets the identifier (*matdef-id*) of a matrix object definition *component* instance for this matrix object *component* instance.

matdef-id is the *crucial identifier* or *name identifier* of a previously created matrix object definition *component* instance. A NULL stack object (?) must not be used for *matdef-id*, otherwise an error event will occur.

The associated matrix object definition *component* instance specifies the relative layout of the graphics objects associated with this matrix object.

The matrix object definition can be altered at any time via the `@matoAttachDef()` function, which uses this variable, after this matrix object *component* instance has been created.

Note that the existing matrix object definition associated with this matrix object instance cannot be obtained from VIS; it can only be set.

Related Information

[3.4.20_Matrix Object Definition](#) - `@matoAttachDef()` ♦

— Member Functions —

@matoAttachDef()

`@matoAttachDef() [@matoMatDefID]`

@matoMatDefID An integer or string value.

Details

Attaches the specified (*@matoMatDefID*) matrix object definition *component* instance to this matrix object *component* instance, detaching the existing one.

`@matoMatDefID` contains the *crucial identifier* or *name identifier* of a matrix object definition *component* instance of the *current drawing plane*.

The existing matrix object definition is destroyed if this matrix object is its only owner.

Additional Information

`@matoMatDefID` ♦

`@matoNumDynCons()`

`@matoNumDynCons ()` → *num-cons*

num-cons An integer stack object.

Details

Returns the number of dynamic constrainers specified for this matrix object *component* instance.

num-cons is the number of dynamic constrainers calculated directly from the matrix object definition *component* instance specified in the `@matoMatDefID` variable, rather than from the actual internally attached matrix object definition. The number of constrainers is specified in the `@matdConsTypes` variable of the matrix object definition.

Additional Information

`@matdConsTypes`

Related Information

`@matoMatDefID` ♦

`@matoPurgeGObjs()`

`@matoPurgeGObjs ()`

Details

Purges all graphics sub-objects of this matrix object *component* instance to the *drawing plane* display list.

A matrix object is essentially a group graphics object. This function relocates the matrix object's sub-objects to the *drawing plane*, where they are displayed as usual. This matrix object will end up having no graphics sub-objects after this function is called. The `@matoGObjIDList` variable remains unaltered.

This function can be called to retain this matrix object's sub-objects before the matrix object itself is deleted, otherwise the sub-objects will be deleted along with the matrix object.

This function can also be used to ensure that only the graphics objects indicated in the `@matoGObjIDList` variable will be the sub-objects of this matrix object by calling the `@matoUpdateGObjs ()` function immediately after calling this function.

Other Information

`@matoGObjIDList` ▪ `@matoUpdateGObjs ()` ♦

@matoUpdateGObjs()

@matoUpdateGObjs (*pRedraw*) [*@matoGObjIDList*] → *success*

pRedraw An integer stack object containing a logical boolean value.

@matoGObjIDList A sequence value.

success An integer stack object containing a logical boolean value.

Details

Updates the internal graphics sub-object list of this matrix object *component* instance.

pRedraw is true to automatically update the layout of the sub-objects as specified in the associated matrix object definition *component* instance. If the value is false, the sub-object positions and sizes will not be updated to comply with the specified layout. The update is performed via the **@matoUpdateLayout()** function.

@matoGObjIDList contains the identifiers of the graphics object *component* instances used for the update.

success is true if the internal sub-object list was successfully updated, otherwise it is false. Note that *success* can be false if the internal sub-object list was only partially updated because of an error occurring during the update process.

If the number of internal sub-objects is greater than the number indicated by the **@matoGObjIDList** variable, then this function will replace only the initial sequence of internal sub-objects, leaving the remaining sub-objects on the internal list of sub-objects. For example, if the **@matoGObjIDList** variable indicates graphics objects a_1 , a_2 , and a_3 , and the internal sub-object list contains graphics objects b_1 , b_2 , b_3 , b_4 , and b_5 , then, after calling this function, the internal sub-object list will contain graphics objects a_1 , a_2 , a_3 , b_4 , and b_5 .

Additional Information

@matoGObjIDList

Other Information

@matoUpdateLayout() ♦

@matoUpdateLayout()

@matoUpdateLayout ()

Details

Updates the layout of the sub-objects of this matrix object *component* instance as specified in the associated matrix object definition *component* instance.

This function changes the position and size of the graphics sub-objects according to the layout specified in the associated matrix object definition. If the number of sub-objects of this matrix object exceeds the number of dynamic constrainers, then those extra sub-objects will not be affected by this function. If the number of sub-objects is less than the number of dynamic constrainers, then the consequence is undefined.

Other Information

@matoGObjIDList ▪ **@matoUpdateGObjs()** ♦

3.4.20 Matrix Object Definition

A matrix object definition *component* defines the inter-relational layout of a list of specified graphics objects associated with a matrix object *component* instance.

See [2.13_Matrix Objects](#) and [1.3.17_Matrix Object Definition Component](#) for further information about matrix object definition *components*.

The *current drawing plane* must be the same as the *drawing plane* within which the matrix object definition *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: `matd`
- Type: `visC_MatObjDef`
- Parent: [Drawing Plane](#)
- Creation Context: [Matrix Object](#)
- Creator: `vis_MatObjDef()`
- Deletion: `@visDelete()`
- Access: `@VIS()` ▪ `@VISX()`

— Member Variables —

@matdConsTypes	
@matdConsTypes	:= cons-types ([])
cons-types	An integer sequence stack object.

Details

Sets the constrainer type of each constrainer in this matrix object definition *component* instance.

cons-types is a TAC sequence containing the constrainer type of each constrainer. Each element of the sequence is one of the values listed in [Matrix Object \(Constrainer Types\)](#).

The number of elements in *cons-types* is logically the same as the number of constrainers, which is the number of rows times the number of columns (`@matdNumRows` × `@matdNumCols`) specified in the associated matrix object definition. If the number of elements is less than mentioned, then the remaining elements are assumed to have the value `:!MOD_NULL_CONS:`. Elements of the sequence having type `:!MOD_DYNAMIC_CONS:` are associated with the next graphics object listed in the `@matoGObjIDList` variable of the associated matrix object *component* instance. Elements having type `:!MOD_NULL_CONS:` are not associated with a graphics object, but partake in the layout specified by this matrix object definition.

Once set, the constrainer types cannot be altered.

Example

The following illustration shows how the `@matdConsTypes` variable can be used.

```
vis_MatObject("MyMatObj")
{
  @matdNumRows := 2; @matdNumCols := 3;
  @matoGObjIDList := ["Obj1", "Obj2", "Obj3"];

  vis_MatObjDef("MyMatObjDef")
  {
    @matdConstypes := [ :!MOD_DYNAMIC_CONS:, :!MOD_NULL_CONS:, :!MOD_DYNAMIC_CONS:,
                        :!MOD_DYNAMIC_CONS: ];
  };
};
```

In the code fragment above, the matrix object (MyMatObj) is set to display graphics objects (Obj1, Obj2, and Obj3) in a matrix of two rows and three columns. However, there are only three graphics objects, not six. The other constrainers (cells) of the matrix are not associated with a graphics object, but may still be used in the positioning and size of the three graphics objects. The matrix object definition (MyMatObjDef) specifies which constrainers are associated with the three graphics objects. The first element of the sequence assigned to **@matdConstypes** is associated with Obj1; the second element is not associated with a graphics object; the third element is associated with Obj2; and the fourth associated with Obj3. The remaining two constrainers (there are six altogether) have a default value of **:!MOD_NULL_CONS:**, and are therefore not associated with any graphics objects.

When the matrix object is displayed in a *VIEW*, the three graphics objects are positioned and sized in relation to all the constrainers (including the ones not associated with a graphics object) in accordance with other information (not shown in the example) in the matrix object definition (MyMatObjDef).

Additional Information

Matrix Object (Constrainer Types)

Other Information

@matdNumRows ▪ **@matdNumCols** ▪ **@matoGObjIDList** ♦

@matdHorShift

@matdHorShift := *hor-shift* (?)

hor-shift A sequence of subsequences.

Details

Sets the horizontal shift specifications of each constrainer in this matrix object definition *component* instance.

The horizontal shift specification concerns only the horizontal aspect of the relations of the constrainers among each other. There are potentially two parts to a specification, the “Left” part and an optional “Right” part (these names have nothing to do with the concept of “left” and “right”; they are named as mentioned for convenience). The Left specification part contains information about the “Target” border and the “Reference” border. The Right specification part (which is optional) contains information only about the Target border because the Reference border is always the opposite border specified for the Left specification part. VIS processes each constrainer in the order specified by the **@matdShiftOrd** variable. The current constrainer being processed is the Target, and a different constrainer (as specified in the *flags* of *hor-shift*) is the Reference.

hor-shift is a TAC sequence of subsequences containing the horizontal shift specifications of each constrainer in the matrix object definition, corresponding to each element of the sequence

contained in the `@matdConstTypes` variable. There must be at least as many subsequences in *hor-shift* as there are constrainers (`@matdNumRows` × `@matdNumCols`) in the matrix object definition. Each one of the subsequences has the following format.

[*flags*, *left-dist*, *left-cons-num*, *right-dist*, *right-cons-num*],

where

- 1 *flags* is an integer containing one or more of the values listed in [Matrix Object \(Constrainer Information\)](#).
- 2 *left-dist* is a decimal number containing the distance, in *base units*, of the Target border specified in *flags* from the Reference border specified in *flags* for the Left specification part. The Target constrainer (and therefore possibly its container, i.e. its graphics object if any) is moved by VIS to maintain that distance whenever the matrix object is updated.
- 3 *left-cons-num* is an integer containing the constrainer number of the Target constrainer specified in *flags* for the Left specification part. The constrainer number must be between zero and one less than the total number of constrainers.
- 4 *right-dist* is a decimal number containing the distance, in *base units*, of the opposite border of the Target border specified in *flags* for the Left specification part from the Reference border specified in *flags* for the Right specification part. The Target border of the Target constrainer is moved by VIS to maintain that distance whenever the matrix object is updated. This may require that the Target constrainer (and therefore possibly its container, i.e. its graphics object if any) is stretched or shrunk to maintain the *left-dist* and *right-dist* border distances simultaneously.
- 5 *right-cons-num* is an integer containing the constrainer number of the Target constrainer specified in *flags* for the Right specification part. The constrainer number must be between zero and one less than the total number of constrainers.

A subsequence of *hor-shift* could be specified as `:#MCI_DFLT_SHIFT:`, which has default values. If a NULL stack object (?) is used instead of *hor-shift*, then each constrainer will be assumed to have the default shift specification (`:#MCI_DFLT_SHIFT:`).

If *flags* does not have information about the Right specification part, then the values of *right-dist* and *right-cons-num* are ignored (the values should both be zero in this case), and the constrainer is moved without changing its size in the relevant dimension.

If a graphics sub-object of the associated matrix object has the `:!GO_HIDE_MAT_CONT:` flag set in `@goFlags`, then that sub-object will be made invisible and treated as though it had zero width and height.

Once set, the horizontal shift specifications cannot be altered.

Examples

The following illustrations show how the `@matdHorShift` variable can be used.

```
(1) @matdHorShift :=
[
  [:!MCI_WRAP:, 0.0, 0, 0.0, 0],
  [(&or :!MCI_STRETCH: :!MCI_TARGET_BORD1_L: :!MCI_REF_BORD1_L:
    :!MCI_REF_BORD3_R:), 0.0, 0, 0.0, 0],
  [(&or :!MCI_TARGET_BORD1_L: :!MCI_REF_BORD1_L:), 0.1, 0, 0.0, 0],
  [(&or :!MCI_TARGET_BORD3_L: :!MCI_REF_BORD3_L:), -0.1, 0, 0.0, 0],
  [(&or :!MCI_STRETCH: :!MCI_TARGET_BORD1_L: :!MCI_REF_BORD3_L:
    :!MCI_REF_BORD1_R:), 0.3, 2, -0.3, 3]
];
```

Example (1) illustrates the horizontal shift specification for five constrainers (each containing a graphics object) arranged in a particular way within a matrix object. Each subsequence corresponds to a constraider numbered from 0 to 4 with subsequence indexes 1 to 5 respectively. That is to say, that the first subsequence (index 1) corresponds to constraider 0, the second subsequence (index 2) corresponds to constraider 1, and so on. The matrix is assumed to be one row by five columns (not shown in the illustration), and the constrainers processed in the default order from 0 to 4.

Whenever the matrix object is updated, the arrangement is maintained as follows.

Constraider 0 (the first subsequence) is wrapped (: !MCI_WRAP:) horizontally around its container (ie: its corresponding graphics object), resulting in both the constraider and its container having identical left and right borders.

Constraider 1 (which is the Target of the second subsequence) will have its left border (: !MCI_TARGET_BORD1_L:) zero distance (the first 0.0) away from the left border (: !MCI_REF_BORD1_L:) of constraider 0, the Reference constraider (indicated by the first 0). In this specification, the right border of the Target (the opposite border of the one already specified) will also be zero distance (the second 0.0) away from the right border (: !MCI_REF_BORD3_R:) of the Reference constraider 0 (indicated by the second 0). The Target container (with its graphics object) is specified to be stretched (: !MCI_STRETCH:) horizontally as both vertical borders of the Target constraider will have been moved.

Constraider 2 (which is the Target of the third subsequence) will have its left border (: !MCI_TARGET_BORD1_L:) 0.1 *base units* distance to the right (the 0.1) from the left border (: !MCI_REF_BORD1_L:) of constraider 0, the Reference constraider (indicated by the first 0). In this specification, the right border of the Target (the opposite border of the one already specified) will remain the same distance from its left border, meaning that the width of the container remains unaltered because there is no Right specification part.

Constraider 3 (which is the Target of the fourth subsequence) will have its right border (: !MCI_TARGET_BORD3_L:) 0.1 *base units* distance to the left (the -0.1) from the right border (: !MCI_REF_BORD3_L:) of constraider 0, the Reference constraider (indicated by the first 0). In this specification, the left border of the Target (the opposite border of the one already specified) will remain the same distance from its right border, meaning that the width of the container remains unaltered because there is no Right specification part.

Constraider 4 (which is the Target of the fifth subsequence) will have its left border (: !MCI_TARGET_BORD1_L:) 0.3 *base unit* distance to the right (the 0.3) from the right border (: !MCI_REF_BORD3_L:) of constraider 2, the Reference constraider (indicated by the 2). In this specification, the right border of the Target (the opposite border of the one already specified) will be 0.3 *base unit* distance to the left (the -0.3) away from the left border (: !MCI_REF_BORD1_R:) of the Reference constraider 3 (indicated by the 3). The Target

container (with its graphics object) is specified to be stretched (: !MCI_STRETCH:) horizontally since both vertical borders of the Target constrainer will have been moved.

```
(2) @matdHorShift :=
[
    :#MCI_DFLT_SHIFT:, [...], :#MCI_DFLT_SHIFT:,
    [...], [...], [...],
    :#MCI_DFLT_SHIFT:, [...], :#MCI_DFLT_SHIFT:
];
```

In example (2), the matrix object is assumed to be three rows by three columns (not shown in the illustration). Only five graphics objects are used, even though there are nine constrainers. The four constrainers that do not correspond to graphics objects have the default shift specification (: #MCI_DFLT_SHIFT:) in this example. (The shift specifications corresponding to graphics objects are represented by ellipses just for this illustration.) Note that, in a shift specification, all constrainers need to be taken into account.

Additional Information

[Matrix Object \(Constrainer Information\)](#)

Related Information

[@matdVerShift](#) ♦

@matdHorSpaces

@matdHorSpaces := *hor-spaces* (?)

hor-spaces A decimal sequence stack object.

Details

Sets the nominal horizontal column and inter-column spacing of this matrix object definition *component* instance.

hor-spaces is a TAC sequence containing the decimal width of each column in the matrix object, and the distances between the columns. Columns are numbered from left to right. The distances are in *base units*. The first element of the sequence is the distance of the beginning of the first column from the parent graphics object's origin point; the second element is the width of the first column; the third element is the distance between the end of the first column and the beginning of the second column; the fourth element is the width of the second column; and so on. The last element of the sequence is the distance after the last column. Non-width distances can be negative; negative values for widths are treated as zero. The size of the sequence is $2c + 1$, where c is the number of columns (@matdNumCols) in the matrix object. If a NULL stack object (?) is used instead of *hor-spaces*, then each distance is assumed to be 0.0 (zero). See the diagram under [2.13_Matrix Objects](#) for further clarification.

Note that the values of *hor-spaces* are nominal; the values may not have effect depend on the details of the horizontal shift specifications set by the @matdHorShift variable.

Once set, the horizontal spacing cannot be altered.

Related Information

[2.13_Matrix Objects](#) ▪ [@matdVerSpaces](#)

Other Information

[@matdNumCols](#) ▪ [@matdHorShift](#) ♦

@matdNumCols

@matdNumCols := *num-cols* (?)

num-cols An integer stack object.

Details

Sets the number of columns for this matrix object definition *component* instance.

num-cols is a positive integer specifying the number of columns of the matrix object.

Once set, the number of columns cannot be altered.

Related Information

[@matdNumRows](#) ♦

@matdNumRows

@matdNumRows := *num-rows* (?)

num-rows An integer stack object.

Details

Sets the number of rows for this matrix object definition *component* instance.

num-rows is a positive integer specifying the number of rows of the matrix object.

Once set, the number of rows cannot be altered.

Related Information

[@matdNumCols](#) ♦

@matdShiftOrd

@matdShiftOrd := *shift-ord* (?)

shift-ord An integer sequence stack object.

Details

Sets the shift order of each constrainer of this matrix object definition *component* instance.

shift-ord is a TAC sequence containing integer elements representing a unique constrainer number. The sequence size is the same as, or less than, the number of constrainers in the matrix object. The default values for the full sequence are consecutive numbers beginning with 0 (zero). If a NULL stack object (?) is used instead of *shift-ord*, then the full default sequence is assumed.

The number of constrainers in the matrix object is **@matdNumRows** × **@matdNumCols**. When the matrix object is updated via the **@matoUpdateLayout()** function, the constrainers are processed in the order specified in *shift-ord*.

Note that, because the values of *shift-ord* are unique, every constrainer number will effectively be in the sequence. The consequence is unpredictable if *shift-ord* does not effectively contain all of the constrainer numbers uniquely.

Once set, the shift order cannot be altered.

Examples

The following illustrations show how the `@matdShiftOrd` variable can be used.

```
(1) @matdShiftOrd := [1, 3, 0, 2];
(2) @matdShiftOrd := [3, 2, 4]; [* This is invalid. *]
```

In example (1), the number of constrainers is assumed to be seven (not shown in the illustration), but only four of the constrainer numbers are specified. The remaining constrainer numbers are the remaining default numbers. Therefore, the sequence is effectively [1, 3, 0, 2, 4, 5, 6]. Note that the numbers 4, 5, and 6 are the three remaining default values of the sequence; the sequence effectively contains all of the constrainer numbers uniquely.

In example (2), the number of constrainers is assumed to be five (not shown in the illustration), but only three of the constrainer numbers are specified. The remaining constrainer numbers are the remaining default numbers. Therefore, the sequence is effectively [3, 2, 4, 3, 4], which is invalid because not all of the constrainer numbers are specified and also some are repeated.

Other Information

`@matdNumRows` ▪ `@matdNumCols` ▪ `@matoUpdateLayout()` ♦

@matdVerShift

`@matdVerShift` := *ver-shift* (?)

ver-shift A sequence of sequences.

Details

The vertical shift specifications of each constrainer in a matrix object.

Sets the vertical shift specifications of each constrainer in this matrix object definition *component* instance.

The vertical shift specification concerns only the vertical aspect of the relations of the constrainers among each other. There are potentially two parts to a specification, the “Left” part and an optional “Right” part (these names have nothing to do with the concept of “left” and “right”; they are named as mentioned for convenience). The Left specification part contains information about the “Target” border and the “Reference” border. The Right specification part (which is optional) contains information only about the Target border because the Reference border is always the opposite border specified for the Left specification part. VIS processes each constrainer in the order specified by the `@matdShiftOrd` variable. The current constrainer being processed is the Target, and a different constrainer (as specified in the *flags* of *ver-shift*) is the Reference.

ver-shift is a TAC sequence of subsequences containing the vertical shift specifications of each constrainer in the matrix object definition, corresponding to each element of the sequence contained in the `@matdConTypes` variable. There must be at least as many subsequences in *ver-shift* as there are constrainers (`@matdNumRows` × `@matdNumCols`) in the matrix object definition. Each one of the subsequences has the following format.

[*flags*, *left-dist*, *left-cons-num*, *right-dist*, *right-cons-num*],

where

- 1 *flags* is an integer containing one or more of the values listed in [Matrix Object \(Constrainer Information\)](#),
- 2 *left-dist* is a decimal number containing the distance, in *base units*, of the Target border specified in *flags* from the Reference border specified in *flags* for the Left specification

- part. The Target constrainer (and therefore possibly its container, i.e. its graphics object if any) is moved by VIS to maintain that distance whenever the matrix object is updated.
- 3 *left-cons-num* is an integer containing the constrainer number of the Target constrainer specified in *flags* for the Left specification part. The constrainer number must be between zero and one less than the total number of constrainers.
 - 4 *right-dist* is a decimal number containing the distance, in *base units*, of the opposite border of the Target border specified in *flags* for the Left specification part from the Reference border specified in *flags* for the Right specification part. The Target border of the Target constrainer is moved by VIS to maintain that distance whenever the matrix object is updated. This may require that the Target constrainer (and therefore possibly its container, i.e. its graphics object if any) is stretched or shrunk to maintain the *left-dist* and *right-dist* border distances simultaneously.
 - 5 *right-cons-num* is an integer containing the constrainer number of the Target constrainer specified in *flags* for the Right specification part. The constrainer number must be between zero and one less than the total number of constrainers.

A subsequence of *ver-shift* could be specified as `:#MCI_DFLT_SHIFT:`, which has default values. If a NULL stack object (?) is used instead of *ver-shift*, then each constrainer will be assumed to have the default shift specification (`:#MCI_DFLT_SHIFT:`).

If *flags* does not have information about the Right specification part, then the values of *right-dist* and *right-cons-num* are ignored (the values should both be zero in this case), and the constrainer is moved without changing its size in the relevant dimension.

If a graphics sub-object of the associated matrix object has the `:!GO_HIDE_MAT_CONT:` flag set in `@goFlags`, then that sub-object will be made invisible and treated as though it had zero width and height.

Once set, the vertical shift specifications cannot be altered.

Examples

The following illustrations show how the `@matdVerShift` variable can be used.

```
(1) @matdVerShift :=
[
  [ :!MCI_WRAP: , 0.0, 0, 0.0, 0 ],
  [ (&or :!MCI_WRAP: :!MCI_TARGET_BORD3_L: :!MCI_REF_BORD3_L: ), 0.0,
    0, 0.0, 0 ],
  [ (&or :!MCI_STRETCH: :!MCI_WRAP: :!MCI_TARGET_BORD3_L:
        :!MCI_REF_BORD1_L: :!MCI_REF_BORD1_R: ), -0.1, 1, 0.0, 0 ],
  [ (&or :!MCI_STRETCH: :!MCI_WRAP: :!MCI_TARGET_BORD3_L:
        :!MCI_REF_BORD1_L: :!MCI_REF_BORD1_R: ), -0.1, 1, 0.0, 0 ],
  [ (&or :!MCI_STRETCH: :!MCI_WRAP: :!MCI_TARGET_BORD3_L:
        :!MCI_REF_BORD1_L: :!MCI_REF_BORD1_R: ), -0.6, 0, 0.3, 0 ]
];
```

Example (1) illustrates the vertical shift specification for five constrainers (each containing a graphics object) arranged in a particular way within a matrix object. Each subsequence corresponds to a constrainer numbered from 0 to 4 with subsequence indexes 1 to 5 respectively. That is to say, that the first subsequence (index 1) corresponds to constrainer 0, the second subsequence (index 2) corresponds to constrainer 1, and so on. The matrix is assumed to be one row by five columns (not shown in the illustration), and the constrainers processed in the default order from 0 to 4.

Whenever the matrix object is updated, the arrangement is maintained as follows.

Constrainer 0 (the first subsequence) is wrapped (`: !MCI_WRAP:`) vertically around its container (ie: its corresponding graphics object), resulting in both the constrainer and its container having identical top and bottom borders respectively.

Constrainer 1 (which is the Target of the second subsequence) will have its top border (`: !MCI_TARGET_BORD3_L:`) zero distance (the first `0.0`) away from the top border (`: !MCI_REF_BORD3_L:`) of constrainer 0, the Reference constrainer (indicated by the first 0). However, the Target constrainer is wrapped (`: !MCI_WRAP:`) vertically around its container (ie: its corresponding graphics object) first, resulting in both the constrainer and its container having identical top and bottom borders, respectively, before the top border is moved. In this specification, the bottom border of the Target (the opposite border of the one already specified) will remain the same distance from its top border, meaning that the height of the container remains unaltered because there is no Right specification part.

Constrainer 2 (which is the Target of the third subsequence) will have its top border (`: !MCI_TARGET_BORD3_L:`) `0.1 base units` distance below (the `-0.1`) the bottom border (`: !MCI_REF_BORD1_L:`) of constrainer 1, the Reference constrainer (indicated by the 1). However, the Target constrainer is wrapped (`: !MCI_WRAP:`) vertically around its container (ie: its corresponding graphics object) first. In this specification, the bottom border of the Target (the opposite border of the one already specified) will be zero distance (the `0.0`) away from the bottom border (`: !MCI_REF_BORD1_R:`) of the Reference constrainer 0 (indicated by the 0). The Target container (with its graphics object) is specified to be stretched (`: !MCI_STRETCH:`) vertically as both horizontal borders of the Target constrainer will have been moved.

Constrainer 3 (which is the Target of the fourth subsequence) will have the same alignment as for constrainer 2 because the shift specifications are the same.

Constrainer 4 (which is the Target of the fifth subsequence) will have its top border (`: !MCI_TARGET_BORD3_L:`) `0.6 base unit` distance below (the `-0.6`) the bottom border (`: !MCI_REF_BORD1_L:`) of constrainer 0, the Reference constrainer (indicated by first the 0). However, the Target constrainer is wrapped (`: !MCI_WRAP:`) vertically around its container (ie: its corresponding graphics object) first. In this specification, the bottom border of the Target (the opposite border of the one already specified) will be `0.3 base unit` distance above (the `0.3`) the bottom border (`: !MCI_REF_BORD1_R:`) of the Reference constrainer 0 (indicated by the second 0). The Target container (with its graphics object) is specified to be stretched vertically (`: !MCI_STRETCH:`) since both horizontal borders of the Target constrainer will have been moved.

```
(2) @matdVerShift := ?;
```

In example (2), the matrix object is assumed to be three rows by two columns (not shown in the illustration). The default vertical shift specification (`: #MCI_DFLT_SHIFT:`) is assumed for the six constrainers. This results in having no gaps between the three rows, and the height of each row is the same as the highest container (ie: graphics object) in the respective row. No vertical shifting of constrainers occurs.

Additional Information

[Matrix Object \(Constrainer Information\)](#)

Other Information

@matdHorShift ♦

@matdVerSpaces

@matdVerSpaces := *ver-spaces* (?)

ver-spaces A decimal sequence stack object.

Details

Sets the nominal vertical row and inter-row spacing of this matrix object definition *component* instance.

ver-spaces is a TAC sequence containing the decimal height of each row in the matrix object, and the distances between the rows. Columns are numbered from bottom to top. The distances are in *base units*. The first element of the sequence is the distance of the bottom of the first row from the parent graphics object's origin point; the second element is the height of the first row; the third element is the distance between the top of the first row and the bottom of the second row; the fourth element is the height of the second row; and so on. The last element of the sequence is the distance after the last row. Non-height distances can be negative; negative values for heights are treated as zero. The size of the sequence is $2r + 1$, where r is the number of rows (**@matdNumRows**) in the matrix object. If a NULL stack object (?) is used instead of *ver-spaces*, then each distance is assumed to be 0.0 (zero). See the diagram under [2.13_Matrix Objects](#) for further clarification.

Note that the values of *ver-spaces* are nominal; the values may not have effect depend on the details of the vertical shift specifications set by the **@matdVerShift** variable.

Once set, the vertical spacing cannot be altered.

Related Information

[2.13_Matrix Objects](#) ▪ **@matdHorSpaces**

Other Information

@matdNumRows ▪ **@matdVerShift** ♦

3.4.21 Measurement Object

A measurement object *component* incorporates a graphics object *component* and represents the image of one of four types of measurements: standard, angle, radial, and right-angle. The actual type of measurement and its properties is defined by an associated measurement object definition *component* instance.

See [2.14_Measurement Objects](#) and [1.3.18_Measurement Object Component](#) for further information about measurement object *components*.

A measurement object *component* instances inherits the properties of a graphics object *component*.

The *current drawing plane* must be the same as the *drawing plane* within which the measurement object *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: meso
- Type: **visC_MesObject**
- Derived from: visGObject
- Parent: [Drawing Plane](#) ▪ [Graphics Object](#)
- Creator: **vis_MesObject()**

- Deletion: `@visDelete()`
- Access: `@VIS()` ▪ `@VISX()`

— Member Variables —

@mesoArrowHdPenID

`@mesoArrowHdPenID := pen-id (?)`

pen-id An integer or string stack object.

Details

Sets the identifier (*pen-id*) of the pen attributes *component* instance for the outline of the arrowheads of this measurement object *component* instance.

pen-id is the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance of the *current drawing plane*. If a NULL stack object (?) is used instead of *pen-id*, then the arrowheads will assume a default outline (solid black colour) when the measurement object is being created.

The pen attributes instance is used to draw the outline of the arrowheads (excluding the arrow lines) for standard, angle, and radial measurements. `@mesoArrowHdPenID` does not apply to right-angle measurements.

The pen attributes can be altered at any time via the `@mesoNewArrowHdAttr()` function, which uses this variable, after this measurement object *component* instance has been created.

Related Information

[3.4.15 Pen Attributes](#) ▪ `@mesoNewArrowHdAttr()` ♦

@mesoArrowHdSize

`@mesoArrowHdSize := size (0.5)`

size A decimal stack object.

Details

Sets the size (*size*) of the arrowheads of all arrow lines of this measurement object *component* instance.

size is the longitudinal length, in *base units*, of the arrowheads for standard, angle, and radial measurements, and must be a decimal number greater than zero. The width of the arrowheads will remain in proportion to the length. `@mesoArrowHdSize` does not apply to right-angle measurements.

The arrowhead size can be altered at any time via the `@mesoNewArrowHdSize()` function, which uses this variable, after this measurement object *component* instance has been created.

Related Information

`@mesoNewArrowHdSize()` ♦

@mesoArrowLnPenID

@mesoArrowLnPenID := *pen-id* (?)

pen-id An integer or string stack object.

Details

Sets the identifier (*pen-id*) of the pen attributes *component* instance for the arrow lines of this measurement object *component* instance.

pen-id is the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance of the *current drawing plane*. If a NULL stack object (?) is used instead of *pen-id*, then the arrow lines will assume a default outline (solid black colour) when the measurement object is being created.

The pen attributes instance is used to draw the arrow lines (excluding the outlines of the arrowheads) for standard, angle, and radial measurements. **@mesoArrowLnPenID** does not apply to right-angle measurements.

The pen attributes can be altered at any time via the **@mesoNewArrowLnAttr()** function, which uses this variable, after this measurement object *component* instance has been created.

Related Information

@mesoNewArrowLnAttr() ♦

@mesoFail

@mesoFail := *fail* (0)

fail An integer stack object.

Details

Indicates a failure code for **@mesoUpdateLayout()**.

This value is set only by VIS as a failure code for a call to the **@mesoUpdateLayout()** function if the **@mesoValidate** variable is true, or if **@mesoPointList** has no elements. The program should not set this value.

@mesoFail has one of the following values after a call to **@mesoUpdateLayout()**:

- 0 Success. The call to the **@mesoUpdateLayout()** function succeeded.
- 1 An invalid measurement type was specified in the **@mesdMeasType** variable of the associated measurement object definition *component* instance.
- 2 An insufficient number of points were specified in **@mesoPointList** for the measurement type.
- 3 Some of the specified points in **@mesoPointList** were invalidly equal for the measurement type.
- 4 Some of the specified points in **@mesoPointList** were invalidly configured.

If **@mesoFail** does not contain 0 after **@mesoUpdateLayout()** is called, then no action will occur.

Related Information[@mesoUpdateLayout\(\)](#) ▀ [@mesoValidate](#)**Other Information**[@mesdMeasType](#) ♦**@mesoLinePenID****@mesoLinePenID** := *pen-id* (?)*pen-id* An integer or string stack object.**Details**

Sets the identifier (*pen-id*) of the pen attributes *component* instance for the measurement lines of this measurement object *component* instance.

pen-id is the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance of the *current drawing plane*. If a NULL stack object (?) is used instead of *pen-id*, then the measurement lines will assume a default outline (solid black colour) when the measurement object is being created.

The pen attributes instance is used to draw the measurement lines for standard and angle measurements, and the right-angle symbol for the right-angle measurement. **@mesoLinePenID** does not apply to radial measurements.

The pen attributes can be altered at any time via the **@mesoNewLineAttr()** function, which uses this variable, after this measurement object *component* instance has been created.

Related Information[@mesoNewLineAttr\(\)](#) ♦**@mesoMesDefID****@mesoMesDefID** := *mesdef-id* (?)*mesdef-id* An integer or string stack object.**Details**

Sets the identifier (*mesdef-id*) of a measurement object definition *component* instance for this measurement object *component* instance when being created.

mesdef-id is the *crucial identifier* or *name identifier* of a previously created measurement object definition *component* instance. A NULL stack object (?) must not be used for *mesdef-id*, otherwise an error event will occur.

If a child measurement object definition *component* instance is defined as a child *component* of this measurement object, then **@mesoMesDefID** will automatically be assigned the *CID* of that child *component* instance. Otherwise, **@mesoMesDefID** needs to be assigned in the *initialiser procedure* when creating this measurement object.

Once set, the measurement object definition *component* instance cannot be altered.

Related Information[3.4.22_Measurement Object Definition](#) ♦

@mesoPointList

@mesoPointList := *pt-list* ([1])

pt-list A decimal sequence stack object.

Details

Sets a sequence (*pt-list*) containing two or four points that determine the measurement value of this measurement object *component* instance.

pt-list is a TAC sequence containing either two points, for radial and right-angle measurements, or four points, for standard and angle measurements. The points are in *base units*, and are relative to the origin point (**@goOriginX**, **@goOriginY**) of the measurement object. The sequence is in the form $[x_1, y_1, x_2, y_2[, x_3, y_3, x_4, y_4]]$.

The points in the **@mesoPointList** determine where the measurement and arrow lines are drawn along with the measurement value. The measurement object will not be updated or presented, or an error event will occur, if **@mesoPointList** does not have the correct number of points.

The sequence of points can be altered at any time via the **@mesoUpdateLayout()** function, which uses this variable, after this measurement object *component* instance has been created.

Related Information

@mesoUpdateLayout() ♦

@mesoSigDigits

@mesoSigDigits := *sig-digs* (3)

sig-digs An integer stack object.

Details

Sets the number of significant digits (*sig-digs*) of the measurement value displayed by this measurement object *component* instance.

sig-digs must be a positive integer. **@mesoSigDigits** does not apply to right-angle measurements.

The number of significant digits can be altered at any time via the **@mesoNewSigDigits()** function, which uses this variable, after this measurement object *component* instance has been created.

Related Information

@mesoNewSigDigits() ♦

@mesoVal

@mesoVal := *val* (?)

val A decimal stack object.

Details

Indicates the value of the measurement after the creation of the measurement object *component* instance.

This value is set only by VIS after the measurement object has been created (via the **vis_MesObject()** function) successfully. If the creation fails, **@mesoVal** will contain a NULL stack object (?).

Related Information

vis_MesObject() ♦

@mesoValidate

@mesoValidate := *bool* (false)

bool An integer stack object containing a logical boolean value.

Details

Determines (*bool*) whether the sequence of points in **@mesoPointList** are valid when **@mesoUpdateLayout()** is called.

bool is true if the points in **@mesoPointList** are to be validated for the type of measurement set in the associated measurement object definition *component* instance. If this value is false, the points are not validated, and **@mesoUpdateLayout()** sets the reason in the **@mesoFail** variable.

The value of this variable can be altered at any time. The **@mesoUpdateLayout()** function uses this variable.

Related Information

@mesoUpdateLayout() ▪ **@mesoFail** ♦

@mesoValueMult

@mesoValueMult := *val-mult* (1.0)

val-mult A decimal stack object.

Details

Sets a decimal measurement value multiplier (*val-mult*) of this measurement object *component* instance.

The value of **@mesoValueMult** is multiplied with the value internally calculated by the measurement object on each update (via **@mesoUpdateLayout()**) only when the measurement value is displayed. **@mesoValueMult** does not apply to right-angle measurements.

The measurement value multiplier can be altered at any time via the **@mesoNewValueMult()** function, which uses this variable, after this measurement object *component* instance has been created.

Related Information

@mesoNewValueMult()

Other Information

@mesoUpdateLayout() ♦

@mesoValueObjID

@mesoValueObjID := *obj-id* (?)

obj-id An integer or string stack object.

Details

Sets the identifier (*obj-id*) of a custom value graphics object *component* instance for this measurement object *component* instance when being created if one is required.

obj-id is the *crucial identifier* or *name identifier* of a previously created graphics object *component* instance of the *current drawing plane*. If a NULL stack object (?) is used instead of *obj-id*, then the normal measurement value will be displayed.

If the first item *component* of the graphics object is a text item or a fixed text item *component*, **@mesoUpdateLayout()** will set the text item to the measurement value if a value is to be displayed. However, if the **:!MSOD_NO_SET_VAL:** flag is set in **@mesdFlags**, then the first text item or fixed text item *component* remains unaltered each time **@mesoUpdateLayout()** is called. This allows any graphics object to be displayed instead of the measurement value.

The custom graphics object can be replaced by a different one by removing the original one from the measurement object (which is internally a group graphics object), then adding the new one to the measurement object as the first sub-object.

Other Information

@mesoUpdateLayout() - **@mesdFlags** - **@goRelocateTo()** ♦

@mesoValueStyleID

@mesoValueStyleID := *style-id* (?)

style-id An integer or string stack object.

Details

Sets the identifier (*style-id*) of a text style attributes *component* instance for the value object of this measurement object *component* instance.

style-id is the *crucial identifier* or *name identifier* of a previously created text style attributes *component* instance of the *current drawing plane*. If a NULL stack object (?) is used instead of *obj-id*, then the displayed measurement value will have a default font with a white background when the measurement object is being created.

The text style is used to draw the measurement value for standard, angle, and radial measurements. **@mesoValueStyleID** does not apply to right-angle measurements. The text style attributes is not required if a custom value object (via **@mesoValueObjID**) was specified for the measurement object because the custom object would then have its own text style.

The text style attributes can be altered at any time via the **@mesoNewValueAttr()** function, which uses this variable, after this measurement object *component* instance has been created.

Related Information

@mesoNewValueAttr()

Other Information

@mesoValueObjID ♦

— Member Functions —

@mesoCalcVal()**@mesoCalcVal** (*pCalc*)*pCalc* An integer stack object containing a logical boolean value.**Details**

Determines whether or not (*pCalc*) to calculate the measurement value rather than using the assigned value for this measurement object *component* instance.

If *pCalc* is true then the measurement value is automatically calculated and displayed (by default) each time the **@mesoUpdateLayout()** function is called, otherwise the value last assigned by the **@mesoSetValue()** function is used as the measurement value. The measurement value is calculated as follows. For a standard and angle measurement the value is the distance, in *base units*, between the third and fourth points in **@mesoPointList**. For a radial measurement the value is the distance, in *base units*, between the first and second points in **@mesoPointList**. This function does not apply to right-angle measurements.

The default is to automatically calculate the measurement value each time the **@mesoUpdateLayout()** function is called.

Related Information**@mesoSetValue()****Other Information****@mesoPointList** ▪ **@mesoUpdateLayout()** ♦**@mesoNewArrowHdAttr()****@mesoNewArrowHdAttr** () [*@mesoArrowHdPenID*]*@mesoArrowHdPenID* An integer or string value, or NULL stack object.**Details**

Replaces the pen attributes *component* instance of this measurement object *component* instance for outlining the arrowheads of the arrow lines.

@mesoArrowHdPenID contains the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance of the *current drawing plane*. If *@mesoArrowHdPenID* contains a NULL stack object (?), then no action occurs.

The new pen attributes will take effect next time the **@mesoUpdateLayout()** function is called.

Additional Information**@mesoArrowHdPenID** ♦

@mesoNewArrowHdSize()

@mesoNewArrowHdSize() [*@mesoArrowHdSize*]

@mesoArrowHdSize A decimal stack object.

Details

Replaces the size of the arrowheads of all arrow lines of this measurement object *component* instance.

@mesoArrowHdSize contains the longitudinal length, in *base units*, of the arrowheads.

The new arrowhead size will take effect next time the @mesoUpdateLayout() function is called.

Additional Information

@mesoArrowHdSize ♦

@mesoNewArrowLnAttr()

@mesoNewArrowLnAttr() [*@mesoArrowLnPenID*]

@mesoArrowLnPenID An integer or string value, or NULL stack object.

Details

Replaces the pen attributes *component* instance of this measurement object *component* instance for outlining the arrow lines.

@mesoArrowLnPenID contains the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance of the *current drawing plane*. If *@mesoArrowLnPenID* contains a NULL stack object (?), then no action occurs.

The new pen attributes will take effect next time the @mesoUpdateLayout() function is called.

Additional Information

@mesoArrowLnPenID ♦

@mesoNewLineAttr()

@mesoNewLineAttr() [*@mesoLinePenID*]

@mesoLinePenID An integer or string value, or NULL stack object.

Details

Replaces the pen attributes *component* instance of this measurement object *component* instance for outlining the measurement lines.

@mesoLinePenID contains the *crucial identifier* or *name identifier* of a previously created pen attributes *component* instance of the *current drawing plane*. If *@mesoLinePenID* contains a NULL stack object (?), then no action occurs.

The new pen attributes will take effect next time the @mesoUpdateLayout() function is called.

Additional Information

@mesoLinePenID ♦

@mesoNewSigDigits()

@mesoNewSigDigits() [*@mesoSigDigits*]

@mesoSigDigits An integer value.

Details

Replaces the significant digits of the displayed measurement value of this measurement object *component* instance.

@mesoSigDigits contains the number of significant digits, which must be a positive integer.

The new significant digits will take effect next time the @mesoUpdateLayout() function is called.

Additional Information

@mesoSigDigits ♦

@mesoNewValueAttr()

@mesoNewValueAttr() [*@mesoValueStyleID*]

@mesoValueStyleID An integer or string value, or NULL stack object.

Details

Replaces the text style attributes *component* instance of this measurement object *component* instance for the displayed measurement value.

@mesoValueStyleID contains the *crucial identifier* or *name identifier* of a previously created text style attributes *component* instance of the *current drawing plane*. If *@mesoValueStyleID* contains a NULL stack object (?), then no action occurs.

The new text style attributes will take effect next time the @mesoUpdateLayout() function is called.

Additional Information

@mesoValueStyleID ♦

@mesoNewValueMult()

@mesoNewValueMult() [*@mesoValueMult*]

@mesoValueMult A decimal stack object.

Details

Replaces the measurement value multiplier of this measurement object *component* instance.

@mesoValueMult contains the decimal measurement value multiplier.

The new value multiplier will take effect next time the @mesoUpdateLayout() function is called.

Additional Information

@mesoValueMult ♦

@mesoSetValue()

@mesoSetValue (*pVal*)*pVal* A decimal stack object.

Details

Sets a measurement value (*pVal*) for the measurement object *component* instance to be used instead of the internally calculated one.

pVal is the decimal value to be displayed (multiplied by the value multiplier set by the @mesoNewValueMult() function) if the @mesoCalcVal() function has last been called with an argument of false, otherwise the value is calculated by the measurement object and displayed (by default) each time the @mesoUpdateLayout() function is called.

This function does not apply to right-angle measurements.

The new measurement value will take effect next time the @mesoUpdateLayout() function is called.

Related Information

@mesoCalcVal()

Other Information

@mesoNewValueMult() ♦

@mesoShowVal()

@mesoShowVal (*pShow*)*pShow* An integer stack object containing a logical boolean value.

Details

Specifies whether or not (*pShow*) the measurement value of this measurement object *component* instance is visible.

If *pShow* is true then the measurement value becomes visible (displayed), otherwise the measurement value becomes invisible (not displayed). The measurement value is visible by default.

This function does not apply to right-angle measurements.

The new visibility will take effect next time the @mesoUpdateLayout() function is called. ♦

@mesoUpdateLayout()

@mesoUpdateLayout () [*@mesoPointList* *@mesoValidate*] → *value* | ? [*@mesoFail*]*@mesoPointList* An decimal sequence.*@mesoValidate* An integer value containing a logical boolean value.*@mesoFail* An integer value.

Details

Updates the measurement image layout of this measurement object *component* instance.

[`@mesoPointList`](#) contains a sequence of two or four points, in *base units*, depending on the measurement type.

[`@mesoValidate`](#) determines whether the sequence of points in `@mesoPointList` are valid before this function updates the measurement object.

value is the raw calculated or assigned measurement value of the measurement object, not multiplied by the value multiplier (`@mesoValueMult`), if the function succeeds. If the `@mesoCalcVal()` function was last called with an argument of `false`, then *value* will be the value assigned by the last call to the `@mesoSetValue()` function. For a right-angle measurement, *value* is meaningless. A NULL stack object (?) is returned if the function fails.

[`@mesoFail`](#) is set by this function if it fails. If the function succeeds this value is zero.

The caller executes this function after altering the points in `@mesoPointList` relating to a measurement, or after calling a member function to change a member variable.

Additional Information

`@mesoPointList` ▪ `@mesoValidate` ▪ `@mesoFail`

Other Information

`@mesoValueMult` ▪ `@mesoCalcVal()` ▪ `@mesoSetValue()` ♦

3.4.22 Measurement Object Definition

A measurement object definition *component* defines a measurement type and its properties for a measurement object *component* instance.

See [2.14_Measurement Objects](#) and [1.3.19_Measurement Object Definition Component](#) for further information about measurement object definition *components*.

The *current drawing plane* must be the same as the *drawing plane* within which the measurement object definition *component* instance was created to avoid failure when using the *component*'s members.

Information

- Initials: mesd
- Type: `visC_MesObjDef`
- Parent: [Drawing Plane](#)
- Creation Context: [Measurement Object](#)
- Creator: `vis_MesObjDef()`
- Deletion: `@visDelete()`
- Access: `@VIS()` ▪ `@VISX()`

— Member Variables —

@mesdAncPoint**@mesdAncPoint** := *anc-point* (?)*anc-point* An integer stack object.**Details**

Sets the anchor point number (*anc-point*) for the arrowheads of the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

anc-point is the anchor point number for the arrowheads of standard, angle, and radial measurements. If a NULL stack object (?) is used instead of *anc-point*, then an appropriate number will be assumed for the arrowhead type (specified by **@mesdArrowType**). If **@mesdArrowType** contains :#MSOD_CUST_ARR:, then the default anchor point number is -1. Valid values for *anc-point* are -1 to 9. **@mesdAncPoint** does not apply to right-angle measurements.

The anchor point number determines at what position relative to the arrowhead's bounding box the end point of the arrow line will be connected. Setting **@mesdAncPoint** is meaningful only for custom arrowheads. See the title **Creating an Arrowhead** in the document VISPointSetFnts.pdf for further details.

Once set, the anchor point number cannot be altered.

Other Information

@mesdArrowType ▪ **@mesdCustAhead** ♦

@mesdArrowBase**@mesdArrowBase** := *arr-base* (?)*arr-base* An integer stack object.**Details**

Sets the arrow base number (*arr-base*) for the arrowheads of the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

arr-base is the arrow base number for the arrowheads of standard, angle, and radial measurements. If a NULL stack object (?) is used instead of *arr-base*, then an appropriate number will be assumed for the arrowhead type (specified by **@mesdArrowType**). If **@mesdArrowType** contains :#MSOD_CUST_ARR:, then the default arrow base number is 6. Valid values for *arr-base* are -1, -2, and 1 to 9. **@mesdArrowBase** does not apply to right-angle measurements.

The arrow base number represents the position point of the arrowhead, relative to the arrowhead's bounding box, that will coincide with the original end point of the arrow line. Setting **@mesdArrowBase** is meaningful only for custom arrowheads. See the title **Creating an Arrowhead** in the document VISPointSetFnts.pdf for further details.

Once set, the arrow base number cannot be altered.

Other Information

@mesdArrowType ▪ **@mesdCustAhead** ♦

@mesdArrowType

@mesdArrowType := *arr-type* (?)

arr-type An integer stack object.

Details

Determines the shape of the predefined arrowheads to be displayed by the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

arr-type is one of the values listed under “**Arrowhead Types**” in [Measurement Object Values](#). **@mesdArrowType** does not apply to right-angle measurements.

If **@mesdArrowType** contains :#MSOD_CUST_ARR:, then the programmer-specified shape specified in the **@mesdCustAhead** variable can be used for the arrowhead type. If a NULL stack object (?) is used instead of *arr-type*, then the default value :#MSOD_STD_ARR: will be assumed for the arrowhead type.

Once set, the arrowhead type cannot be altered.

Additional Information

[Measurement Object Values](#)

Related Information

@mesdCustAhead ♦

@mesdCustAhead

@mesdCustAhead := *arr-points* (?)

arr-points A decimal sequence stack object.

Details

Sets a poly-curve sequence for the definition of a custom arrowhead to be displayed by the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

This variable applies only if the **@mesdArrowType** variable contains :#MSOD_CUST_ARR:. The variables **@mesdArrowBase** and **@mesdAncPoint** may also need to be set for custom arrowheads.

arr-points is a TAC sequence of poly-curve points that define an arrowhead to be used in the measurement object. The arrowhead must be defined horizontally (parallel to the X-axis) and its base (the part of the arrowhead connected to the arrow line) must be on the left side (towards the decreasing numbers on the X-axis). The arrowhead can be any closed shape of any size. **@mesdCustAhead** does not apply to right-angle measurements. If a NULL stack object (?) is used instead of *arr-points*, then the standard (default) arrowhead shape is assumed.

The poly-curve is in the form $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point of the poly-curve in *base units*. The poly-curve consists of joined Bezier curve segments. Other public documentation needs to be consulted for the definition of a Bezier curve segment.

Once set, the arrowhead definition cannot be altered.

Example

The following illustration shows how the `@mesdCustAhead` variable can be used.

```
vis_DPlane("")
{
  /* Create a standard measurement object with custom arrowheads. */
  vis_Pen("Arrowhead Pen") {@pnColr := :#RGB_Coral;; @pnWidth := 0.1;};
  vis_MesObject("")
  {
    @mesoPointList := [/* Four points for a standard measurement object. */];
    @mesoArrowHdSize := 1.5; @mesoArrowHdPenID := "Arrowhead Pen";

    vis_MesObjDef("") /* Create the measurement object definition. */
    {
      @mesdArrowType := :#MSOD_CUST_ARR;;
      @mesdFlags and= not :!MSOD_FILL_AHEAD;;
      @mesdArrowBase := 5; @mesdAncPoint := 4;
      @mesdCustAhead := @SPF{spf_Square();};
    }
  }
};
```

In the example above, a standard measurement object (the default) is created. The custom arrowheads (`@mesdArrowType := :#MSOD_CUST_ARR:`) of the arrow lines will be coral coloured (`@pnColr := :#RGB_Coral:`) squares centred (`@mesdArrowBase := 5` and `@mesdAncPoint := 4`) on the relevant measurement lines. The size of the squares will be 1.5 *base units* (`@mesoArrowHdSize := 1.5`), with each side having a 0.1 *base unit* thickness (`@pnWidth := 0.1`). The squares are not filled (`@mesdFlags and= not :!MSOD_FILL_AHEAD:`). The path of each square is specified by `@mesdCustAhead := @SPF{spf_Square();}`.

Related Information

`@mesdArrowType` ▪ `@mesdArrowBase` ▪ `@mesdAncPoint` ♦

@mesdDist1

`@mesdDist1 := dist (?)`

dist A decimal stack object.

Details

Sets the distance, in *base units*, before the start of the measurement line or the arrow line, or sets the size of the right-angle symbol (⊥) displayed by the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

For standard and angle measurements, *dist* is the maximum distance from the first argument point in `@mesoPointList` to the beginning of the first measurement line, and from the second argument point of in `@mesoPointList` to the beginning of the second measurement line.

For a radial measurement with inner arrow lines, *dist* is the maximum distance between the first argument point in `@mesoPointList` and the beginning of the radial arrow line. `@mesdDist1` does not apply to radial measurements with outer arrow lines.

For a right-angle measurement, the absolute value of *dist* is the size of the right-angle symbol. If *dist* is positive, the right-angle symbol will be on the right side of an imaginary line from the first argument point in `@mesoPointList` to the second. If *dist* is negative, the right-angle symbol will be on the left side of the imaginary line. If *dist* is zero, the right-angle symbol will not appear.

If a NULL stack object (?) is used instead of *dist*, then 0.5 *base units* is assumed for the distance.

Once set, the distance cannot be altered.

Other Information

@mesoPointList ▪ @mesdDist2 ♦

@mesdDist2	
@mesdDist2	:= dist (?)
dist	A decimal stack object.

Details

Sets the length of the measurement lines beyond the arrowheads for the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

For standard and angle measurements *dist* is the distance from the third argument point in @mesoPointList to the end of the first measurement line, and from the fourth argument point in @mesoPointList to the end of the second measurement line. @mesdDist2 does not apply to radial and right-angle measurements. If a NULL stack object (?) is used instead of *dist*, then 0.5 *base units* is assumed for the distance.

Once set, the distance cannot be altered.

Other Information

@mesoPointList ▪ @mesdDist1 ♦

@mesdFlags	
@mesdFlags	:= flags (:!MSOD_@DEFAULT:)
flags	An integer stack object containing binary flags.

Details

Sets the flags for this measurement object definition *component* instance.

flags is zero or more of the values listed under “**Definition Flags**” in [Measurement Object Values](#).

The flags can be altered at any time via the @mesdAddFlags() and @mesdClearFlags() functions, which modify this variable, after this measurement object definition *component* instance has been created.

Additional Information

[Measurement Object Values](#)

Related Information

@mesdAddFlags() ▪ @mesdClearFlags() ♦

@mesdInArrow

@mesdInArrow := *arrow-loc* (?)

arrow-loc An integer stack object containing a logical boolean value.

Details

Sets the arrow locations relative to the measurement lines for the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

arrow-loc is `true` to display the arrow line between the measurement lines, or `false` to display the arrows on the outside of the measurement line. If a NULL stack object (?) is used instead of *arrow-loc*, then `true` is assumed for the arrow locations.

For standard, angle, and radial measurements, if *arrow-loc* is `true` then the measurement object produces a single arrow line inside (between) the measurement lines, or between the first and second argument points in **@mesoPointList** for a radial measurement. If *arrow-loc* is `false` then two arrows are produced by the measurement object on the outside of the measurement lines or the two points in **@mesoPointList**. The outer arrowheads point in the opposite direction as would the inner arrowheads. **@mesdInArrow** does not apply to right-angle measurements.

Once set, the arrow locations cannot be altered.

Other Information

@mesoPointList ♦

@mesdMeasType

@mesdMeasType := *meas-type* (: #MSOD_STD_TYPE:)

meas-type An integer stack object.

Details

Sets the type of measurement to be displayed by the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

meas-type is one of the values listed under “**Measurement Types**” in [Measurement Object Values](#).

Once set, the measurement type cannot be altered.

Additional Information

[Measurement Object Values](#) ♦

@mesdOutALen

@mesdOutALen := *out-len* (?)

out-len A decimal stack object.

Details

Sets the length, in *base units*, of the outside arrows for the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

This variable applies only if the **@mesdInArrow** variable contains `false`.

out-len is the length of the outside arrows for standard, angle, and radial measurements. The measurement is from the connection point of the arrowhead to the base of the arrow line. For angle measurements, it is the arc length. **@mesdOutALen** does not apply to right-angle measurements. A value of zero is invalid, however, a negative value is valid. If a NULL stack object (?) is used instead of *out-len*, then 1.0 *base unit* is assumed for the length.

If *out-len* is a negative value, then the two “outside” arrows will point in the opposite direction, and their length will be the absolute value of *out-len*.

Once set, the outside arrow length cannot be altered.

Related Information

@mesdInArrow ♦

@mesdValPos1

@mesdValPos1 := *val-pos* (?)

val-pos A decimal stack object.

Details

The perpendicular distance of the value object from the centre point of the arrow line for the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

For standard, angle, and radial measurements, *val-pos* is the distance from the centre of the inner arrow line (or where the inner arrow line would be for measurements with outer arrow lines) to the centre of the value object. The distance, in *base units*, is perpendicular to the arrow line (or where it would be) and is on the right-hand side of the arrow line from the first arrowhead connection point to the second. **@mesdValPos1** does not apply to a right-angle measurement. If a NULL stack object (?) is used instead of *val-pos*, then 0.0 *base units* is assumed for the distance.

Once set, the perpendicular distance cannot be altered.

Other Information

@mesdValPos2 ♦

@mesdValPos2

@mesdValPos2 := *val-pos* (?)

val-pos A decimal stack object.

Details

Sets the longitudinal distance of the value object from the centre point of the arrow line for the associated measurement object *component* instance when this measurement object definition *component* instance is being created.

For standard and radial measurements with an inner arrow line, *val-pos* is the distance, in *base units*, from the centre of the arrow line to the centre of the value object. The distance is parallel to the arrow line and is towards the second arrowhead connection point for a positive value, or in the opposite direction for a negative value.

For standard measurements with outer arrow lines, *val-pos* is the parallel distance, in *base units*, from the connection point of the first or second arrowhead to the centre of the value object towards the base of the arrow line; if *val-pos* is positive, the distance is from the second

arrowhead; if *val-pos* is negative, it is from the first arrowhead; if *val-pos* is zero the distance is as described for an inner arrow line.

For a radial measurement with outer arrow lines, *val-pos* is the parallel distance, in *base units*, from the connection point of the second arrowhead to the centre of the value object towards the base of the arrow line for a positive value, or in the opposite direction for a negative value.

For an angle measurement with an inner arrow line, *val-pos* is an angle, in radians, from the centre of the inner arrow line to the centre of the value object in a counter-clockwise direction for a positive value, or a clockwise direction for a negative value.

For an angle measurement with outer arrow lines, *val-pos* is an angle, in radians, from the connection point of the first or second arrowhead to the centre of the value object towards the base of the arrow line; if *val-pos* is positive, the angle is from the second arrowhead; if *val-pos* is negative, it is from the first arrowhead; if *val-pos* is zero, the angle is as described for an inner arrow line for the angle measurement. The apex of the angle is the intersection of the measurement lines.

If a NULL stack object (?) is used instead of *val-pos*, then 0.0 *base units* or radians (as appropriate for the measurement type) is assumed for the distance.

@mesdValPos2 does not apply to right-angle measurements.

Once set, the longitudinal distance cannot be altered.

Other Information

@mesdValPos1 ♦

— Member Functions —

@mesdAddFlags()

@mesdAddFlags (*pFlags*) → [*@mesdFlags*]

pFlags An integer stack object containing binary flags.

@mesdFlags An integer value containing binary flags.

Details

Adds to the existing internal flags of this measurement object definition *component* instance.

pFlags is zero or more of the values listed under “**Definition Flags**” in [Measurement Object Values](#).

@mesdFlags will contain the original flags of @mesdFlags with those in *pFlags* added to them.

Note that the internal flags of this measurement object definition are not necessarily the same as those in @mesdFlags. For example, if the program alters the flags in @mesdFlags without calling this function (or the @mesdClearFlags () function), then the flags in @mesdFlags would not be the same as the internal flags. There is no way to obtain the internal flags, so @mesdFlags should never be altered directly.

Additional Information

[Measurement Object Values](#) ▪ @mesdFlags

Other Information

@mesdClearFlags () ♦

@mesdClearFlags()**@mesdClearFlags** (*pFlags*) → [**@mesdFlags**]*pFlags* An integer stack object containing binary flags.**@mesdFlags** An integer value containing binary flags.**Details**

Clears flags from the existing internal flags of this measurement object definition *component* instance.

pFlags is zero or more of the values listed under “**Definition Flags**” in [Measurement Object Values](#).

@mesdFlags will contain the original flags of **@mesdFlags** with those in *pFlags* removed from them.

Note that the internal flags of this measurement object definition are not necessarily the same as those in **@mesdFlags**. For example, if the program alters the flags in **@mesdFlags** without calling this function (or the **@mesdAddFlags()** function), then the flags in **@mesdFlags** would not be the same as the internal flags. There is no way to obtain the internal flags, so **@mesdFlags** should never be altered directly.

Additional Information

[Measurement Object Values](#) ▪ **@mesdFlags**

Other Information

@mesdAddFlags() ♦

3.4.23 Data Pack

Defines a data pack *component* for programmer’s use. A data pack *component* instance can be created anywhere under a project *component* instance or any of its child *component* instances to any level. If created under a data pack *component* instance, then the parent *component* instance will be that data pack instance, otherwise the parent will be the *active* project.

The only members of a data pack *component* are the ones common to all *components* (see [3.4.4 All Components](#)).

Information

- Initials: dpk
- Type: **visC_DataPack**
- Parent: [Project](#) ▪ Data Pack
- Creation Context: any
- Creator: **vis_DataPack()**
- Deletion: **@visDelete()**
- Access: **@VIS()** ▪ **@VISX()**

No members (members can be added by the application program for its own use)

3.5 VIS Application Programming Interface

This section describes the VIS application programming interface commands and functions.

3.5.1 Variable Naming Conventions

The following variable naming conventions apply to VIS.

- **VIS API (public):** `<vis_...>` and `<@x...>` where `x` is any letter (for general functions and commands); `<@vis_...>` (for specialised functions and commands); `<spf_...>` (for shape path functions); `<@spf...>` (for named parameters of shape path functions); `<pts_...>` (for point set functions).
- **VIS API (private):** `<_vis_...>` (for general functions and commands); `<_spf_...>` (for shape path functions); `<_pts_...>` (for point set functions). Programmers **MUST NOT ACCESS** these variables in their programs; the names can change in any future release without notice.

NOTE: Programmers must not use the above naming conventions for creating variables for their own programs. However, programmers can create public member variable and function names in the form `<x...>`, and private member variable and function names in the form `<_1x...>`, `<_2x...>`, etc, where `x` is the lower-case initials of the data object type name. Global variable and function names should begin with a lower-case ‘g’ to avoid any potential conflict with VIS variable names.

3.5.2 Command and Function Summary

In the following table, **highlighted** entries indicate functions that create *component* instances.

Member	Description
Menu	The Menu data object (vis_NewMenuData()).
@mCheckMenuItem()	Sets or clears a menu item’s check-mark.
@mDestroyMenu()	Destroys the menu.
@mEnableMenuItem()	Enables or disables a menu item.
@mGetCount()	Gets the menu item count.
@mGetDefault()	Gets the menu default item.
@mGetMenuID()	Returns the menu ID of this data object.
@mGetMenuItem()	Gets the details of a menu item.
@mGetPos()	Gets the parent menu and item position for a given item ID.
@mMarkRadio()	Marks a menu item as the only radio item in an item group.
@mPutMenuItem()	Appends, inserts, or modifies a menu item.
@mRemoveMenuItem()	Deletes a menu item, or detaches or deletes a submenu from a menu.
@mRepMInfo()	Replaces the menu information.
Status Bar	The Status Bar data object (vis_NewSBarData()).
@sbDestroyIcon()	Destroys an icon.
@sbDestroySBar()	Destroys the status bar.
@sbGetSBarHndl()	Returns the status bar handle of this data object.
@sbGetSize()	Gets the width and height of the status bar.
@sbLoadIcon()	Loads an icon.
@sbRepBkClr()	Replaces the background colour of the status bar.
@sbRepIcon()	Replaces the icon of a status bar part.
@sbRepMode()	Replaces the status bar display mode to a simple or parts display.
@sbRepParts()	Replaces the part widths of the status bar.
@sbRepText()	Replaces the text of a status bar part.
@sbSetMinHeight()	Sets the minimum height of the status bar.

@sbShowSBar()	Shows or hides the status bar.
Idle-time Job	The Idle-time Job data object (vis_NewIdleJobData()).
@itjEvents :- false	Set by VIS when all the event messages have just been processed.
@itjJobActive :- false	Set by VIS to indicate whether the job is currently being processed.
@itjJobCount :- ?	Increased by VIS for each recurrence of the job.
@itjJobEndFnt :- ?	Possibly set by the program to be called after the job has been completed.
@itjJobFnt :- ?	Set by the program to perform the desired job during idle-time.
@itjJobID :- ?	Set by the program to identify the job.
@itjJobQueued :- false	Set by VIS to indicate whether the job is still waiting in the queue.
@itjMoreTime :- false	Set by the job to indicate whether the same job is to recur in the next idle-time period.
@itjProjectID :- ?	Possibly set by the program if the job is not for the <i>current</i> project.
@itjUrgent :- false	Set by the caller to put the job first in the queue rather than last.
Commands	The API Commands .
@BITMAP	Extracts the bitmap handle (HBITMAP) from a bitmap attributes <i>component</i> instance.
@DLGMENU	Creates a menu bar for a dialog box.
@MENU	Creates and possibly attaches a menu bar for a <i>VIEW</i> .
@MITEM	Creates a menu item.
@POPOP	Creates a pop-up menu.
@SMENU	Creates a submenu item.
@vis_LMBDclk	Indicates whether the left mouse button has just been double-clicked.
@vis_LMBPrs	Indicates whether the left mouse button has just been pressed.
@vis_LMBRel	Indicates whether the left mouse button has just been released.
@vis_MMBDclk	Indicates whether the middle mouse button has just been double-clicked.
@vis_MMBPrs	Indicates whether the middle mouse button has just been pressed.
@vis_MMBRel	Indicates whether the middle mouse button has just been released.
@vis_MseMove	Indicates whether the mouse cursor has just moved.
@vis_MseTimeSpan	Returns the number of milliseconds since the last mouse button event.
@vis_RMBDclk	Indicates whether the right mouse button has just been double-clicked.
@vis_RMBPrs	Indicates whether the right mouse button has just been pressed.
@vis_RMBRel	Indicates whether the right mouse button has just been released.
Functions	The API Functions .
@VIS()	Gets the specified <i>component</i> instance.
@VISX()	Executes a procedure if the specified <i>component</i> instance exists.
vis_Bitmap()	Creates a bitmap attributes <i>component</i> instance.
vis_Brush()	Creates a brush attributes <i>component</i> instance.
vis_ColorDlg()	Gets a colour from a user.
vis_CompExists()	Determines whether a <i>component</i> exists.
vis_CurveItem()	Creates a curve item <i>component</i> instance.
vis_CvtToCID()	Converts a non-project <i>component</i> ID to its corresponding <i>CID</i> .
vis_DataPack()	Creates a data pack <i>component</i> instance.
vis_DeleteIdleJob()	Deletes an idle-time job from the idle-time job queue.

vis_DeleteProject()	Deletes a project <i>component</i> .
vis_DPlane()	Creates a drawing plane <i>component</i> instance.
vis_EndWaitEvents()	Ends user event processing.
vis_FixCurveItem()	Creates a fixed curve item <i>component</i> instance.
vis_FixFormText()	Creates a fixed formatted text item <i>component</i> instance.
vis_FixImageItem()	Creates a fixed image item <i>component</i> instance.
vis_FixLineItem()	Creates a fixed line item <i>component</i> instance.
vis_FixTextItem()	Creates a fixed text item <i>component</i> instance.
vis_FontDlg()	Gets a font from the user.
vis_FormText()	Creates a formatted text item <i>component</i> instance.
vis_GContext()	Creates a graphics context <i>component</i> instance.
vis_GetCNameList()	Gets a list of <i>component</i> names with their parent <i>component</i> names.
vis_GetMenuTpl()	Gets a menu template from resource file data.
vis_GetPicaPix()	Gets the number of pixels in a <i>logical pica</i> (DPI / 6).
vis_GetPNameList()	Gets a list of project <i>component</i> names.
vis_GObject()	Creates a graphics object <i>component</i> instance.
vis_ImageItem()	Creates an image item <i>component</i> instance.
vis_LineItem()	Creates a line item <i>component</i> instance.
vis_LoadCursor()	Loads a cursor from a file.
vis_MatObjDef()	Creates a matrix object definition <i>component</i> instance.
vis_MatObject()	Creates a matrix object <i>component</i> instance.
vis_MesObjDef()	Creates a measurement object definition <i>component</i> instance.
vis_MesObject()	Creates a measurement object <i>component</i> instance.
vis_NewIdleJobData()	Creates a new idle-time job data object.
vis_NewMenuData()	Creates a new menu data object.
vis_NewSBarData()	Creates a new status bar data object.
vis_Pen()	Creates a pen attributes <i>component</i> instance.
vis_PostIdleJob()	Posts an idle-time job for later processing.
vis_Project()	Creates a project <i>component</i> instance.
vis_PurgeIdleJobs()	Purges (deletes) all idle-time jobs from the idle-time job queue.
vis_ReadClipStr()	Reads a string from the clipboard.
vis_RGB()	Creates a colour value as an integer.
vis_SetCurProject()	Sets the specified project as the <i>current</i> one.
vis_TextItem()	Creates a text item <i>component</i> instance.
vis_TextStyle()	Creates a text style attributes <i>component</i> instance.
vis_VIEW()	Creates a VIEW <i>component</i> instance.
vis_WaitForEvents()	Waits for and processes user and other events.
vis_WriteClipStr()	Writes a string to the clipboard.
vis_WriteGObjs()	Writes a sequence of graphics objects to a disk file or the clipboard.

3.5.3 Notational Conventions

The following conventions are used in the description boxes of the following sections.

- For member variables, the top row contains the name of the member variable (in **bold**), followed by an assignment symbol (**:=**), followed by the label representing the required argument (in *blue*), followed by the initial value of the variable (in **bold**) in parentheses.
- For commands, the top row contains the name of the command (in **bold**), followed by the labels representing the required input arguments (in *blue*) before the red arrow (→), and the labels representing the produced output arguments (in *blue*) after the arrow. The red arrow itself is not part of the syntax and is not to be entered.
- For member functions, the top row contains the name of the member function (in **bold**), with the labels representing the required input arguments (in *blue*) within parentheses, followed by a red arrow (→), and the labels representing the produced output arguments (in *blue*) after the arrow. The red arrow itself is not part of the syntax and is not to be entered.
- Input argument labels will not be present if a command or member function does not require input arguments.
- Output argument labels and the red arrow will not be present if a command or member function does not produce output arguments.
- Each argument is a stack object on the object stack.
- The leftmost argument in a series of input argument labels is the topmost stack object on the object stack; the next argument label is the second top stack object, and so on. Likewise for output argument labels.
- Red vertical bars (|) separate one of a number of options to be entered. The bars themselves are not part of the syntax and are not to be entered.
- The description of each argument (if any) exists in the rows following the top row.

In the following example of the top row of a description box,

```
vis_GetMenuTmp1 (pMenuID pResData) → res-data
```

when the function **vis_GetMenuTmp1()** is called, it expects two arguments on the object stack (represented by *pMenuID* and *pResData*), and returns a stack object (represented by *res-data*).

In the following example,

```
vis_GetCNameList() → nid-list
```

when the function **vis_GetCNameList()** is called, it expects no arguments on the object stack, but returns a stack object (represented by *nid-list*).

In the following example,

```
vis_PostIdleJob (pITJob)
```

when the function **vis_PostIdleJob()** is called, it expects one argument on the object stack (represented by *pITJob*), but returns no stack object.

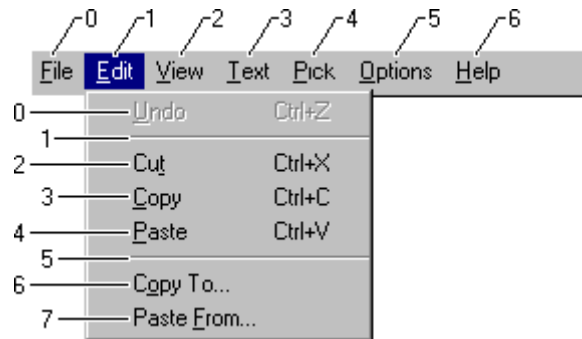
3.5.4 Menu

This is the definition of a menu ETAC data object. A menu data object allows access and manipulation of a menu and its menu items and submenus.

A unique application-defined integer (16-bit value), called a menu-item identifier (item-ID), is associated with each menu item within the same drawing plane *component* instance. When the user chooses a command item from a menu, the item's identifier is received by the event handler registered via the **@dpSetCommandFnt()** function. Menu items that open submenus have menu-

item identifiers just as command items do. However, the event handler does not receive the identifier when such an item is selected from a menu. Instead, the operating system opens the submenu associated with the menu item. Note that, if the application-defined item-ID is not unique for all menu items within the same drawing plane *component* instance, then confusion may reign supreme.

In addition to having a unique identifier, each menu item in a menu bar or menu has a unique position number. The leftmost item in a menu bar, or the top item in a menu, has position zero. The position number increases for subsequent menu items. The operating system assigns a position number to all items in a menu, including separators and submenus. The following illustration shows the position numbers of items in a menu bar and in a menu.



A menu itself also associated with a unique identifier. In this case, the identifier is supplied by the operating system and is known as the menu handle (HMENU).

Incidentally, the submenu of a menu bar is called a “drop-down menu”.

The **@MENU** and **@DLGMENU** commands are used to construct a menu bar. The **@vRepMBar()** function is used to attach a constructed menu bar to a *VIEW*. The **@RepMBar()** function (see ETACProgLang(Official).pdf for details) is used to attach a constructed menu bar, created by **@DLGMENU**, to a dialog box. A menu bar is displayed at the top of a *VIEW* outside the client area. The **@SMENU** command is used to create a submenu. The **@MITEM** command is used to create a menu item (including a separator line) not associated with a submenu (the **@BITMAP** command can be used to display a bitmap with a menu item). The **@POPUP** command is used to create a pop-up menu. A pop-up menu can be displayed anywhere on the screen (but typically within a *VIEW*).

Note that a *VIEW* cannot have more than one menu bar, and a child *VIEW* cannot have any (by order of the operating system).

Information

- Initials: m
- Creator: **vis_NewMenuData()**
- Deletion: **@mDestroyMenu()**

— Member Functions —

@mCheckMItem()

@mCheckMItem(*pLoc* *pCmdLoc* *pCheck* | ?) → *prev-state* | ?

pLoc An integer stack object.

pCmdLoc An integer stack object containing a logical boolean value.

pCheck An integer stack object containing a logical boolean value.

prev-state An integer stack object containing a logical boolean value.

Details

Sets or clears (*pCheck*) the specified (*pLoc*) menu item's check-mark.

pLoc specifies the menu item of this menu to be checked or cleared. It is either an item identifier (item-ID) or a position. If it is an item-ID, it can be the identifier of any recursive submenu of this menu.

pCmdLoc is true if *pLoc* is an item-ID, otherwise *pLoc* is the item position number.

pCheck is true to check the menu item, or false to clear the item check-mark. If a NULL stack object (?) is used instead of *pCheck*, then the menu check-mark is toggled.

prev-state is true if the menu item was previously checked, or false if the menu item was not previously checked, otherwise a NULL stack object (?) is returned on failure.

Other Information

@mMarkRadio() ♦

@mDestroyMenu()

@mDestroyMenu() → *success*

success An integer stack object containing a logical boolean value.

Details

Destroys the menu and all of its submenus.

success is true if the menu was destroyed successfully, otherwise it is false.

This function does not destroy the menu data object itself. However, the data object is unusable (becomes defunct) after the menu is destroyed.

Note that a menu bar that is assigned to a *VIEW* is automatically destroyed when the application closes.

Other Information

@mRemoveMItem() ♦

@mEnableMItem()

@mEnableMItem(*pLoc* *pCmdLoc* *pEnable* | ?) → *prev-state* | ?

pLoc An integer stack object.

pCmdLoc An integer stack object containing a logical boolean value.

prev-state An integer stack object containing a logical boolean value.

Details

Enables or disables (*pEnable*) the specified (*pLoc*) menu item, or just returns its existing enableity state (*prev-state*).

pLoc specifies the menu item of this menu. It is either an item identifier (item-ID) or a position. If it is an item-ID, it can be the identifier of any recursive submenu of this menu.

pCmdLoc is true if *pLoc* is an item-ID, otherwise *pLoc* is the item position number.

pEnable is true to enable the menu item, or false to disable the menu item. If a NULL stack object (?) is used instead of *pEnable*, then only the existing enableity state is returned (*prev-state*).

prev-state is the existing enableity state of the menu item before the enableity state change. If the value is true, then the menu item was enabled before the state change, otherwise it was disabled. A NULL stack object (?) is returned on failure. ♦

@mGetCount()

@mGetCount() → *count*

count An integer stack object.

Details

Gets the number of items (*count*) in this menu.

count is the number of items (including separators and submenus) in this menu. ♦

@mGetDefault()

@mGetDefault() → *item-id* | -1

item-id An integer stack object.

Details

Gets the menu default item (*item-id*). Disabled items are ignored.

item-id is the item identifier of the default item (if there is one). The value -1 (negative one) is returned on failure or if there is no default item. ♦

@mGetMenuID()

@mGetMenuID() → *menu-id* | ?

menu-id An integer stack object.

Details

Returns the menu identifier (*menu-id*) associated with this menu data object.

menu-id is the menu identifier of this data object, or a NULL stack object (?) is returned if this data object does not have a menu identifier.

If this data object does not have a menu identifier, then it is defunct and should be deleted. The data object becomes defunct when the menu is deleted (via @mDestroyMenu()). The value represented by *menu-id* can therefore be used to determine whether this data object was deleted via @mDestroyMenu(). ♦

@mGetMItem()

@mGetMItem(*pLoc* *pCmdLoc*) → *info* | ?

pLoc An integer stack object.

pCmdLoc An integer stack object containing a logical boolean value.

info A sequence stack object.

Details

Gets the details (*info*) of a menu item (*pLoc*).

pLoc specifies the menu item of this menu. It is either an item identifier (item-ID) or a position. If it is an item-ID, it can be the identifier of any recursive submenu of this menu.

pCmdLoc is true if *pLoc* is an item-ID, otherwise *pLoc* is the item position number.

info is a sequence containing details relating to the specified menu item. The contents of the sequence is as follows.

[*item-label*, *item-id*, *menu-id* | 0, *bitmap* | 0, *mft-flags*, *mfs-flags*],

where

- 1 *item-label* is the menu label as specified when the item is created (string).
- 2 *item-id* is the application-defined 16-bit value that identifies the menu item (integer).
- 3 *menu-id* is the handle (HMENU) to the submenu associated with the menu item, or 0 (zero) if the menu item is not associated with a submenu (integer).
- 4 *bitmap* is the handle to the bitmap (HBITMAP) associated with the menu item, or 0 (zero) if the menu item is not associated with a bitmap (integer).
- 5 *mft-flags* is zero or more of the values listed in [Menu Item Types](#) (integer).
- 6 *mfs-flags* is zero or more of the values listed in [Menu Item States](#) (integer).

A NULL stack object (?) is returned if the function fails.

Additional Information

[Menu Item Types](#) - [Menu Item States](#)

Other Information**@mPutMItem()** ♦**@mGetPos()****@mGetPos** (*pItemID*) → *menu-id* | 0 *pos* | -1*pItemID* An integer stack object.*menu-id* An integer stack object.*pos* An integer stack object.**Details**

Gets the parent menu and item position for a given menu item identifier.

pItemID is the item identifier (item-ID) of a menu item of this or any recursive submenu of this menu.

menu-id is the menu identifier (HMENU) of the parent menu of the item specified at *pItemID*. The value 0 (zero) is returned if the parent could not be found.

pos is the position of the menu specified at *pItemID* within its parent menu. The value -1 (negative one) is returned if the menu item could not be found.

Since every menu item necessarily has a parent menu, if the specified item-ID cannot be found, then neither can its parent menu. Therefore, if 0 (zero) is returned, then so is -1 (negative one), and vice versa.

Example

The following illustration shows how the **@mGetPos()** function can be used.

```
(1) void MenuID := MData.@mGetPos(1000);
    NumMItems := vis_GetMenuData(MenuID).@mGetCount();
```

In example (1), the parent menu-ID (MenuID) of a menu item with the specified item-ID (1000) is obtained to determine the number of menu items (NumMItems) in the parent menu (MenuID). The specified menu item (1000) is assumed to be at any level below the menu represented by MData.

Other Information**vis_GetMenuData()** ♦**@mMarkRadio()****@mMarkRadio** (*pMarkLoc pFirstLoc pLastLoc pCmdLoc*) → *success**pMarkLoc* An integer stack object.*pFirstLoc* An integer stack object.*pLastLoc* An integer stack object.*pCmdLoc* An integer stack object containing a logical boolean value.*success* An integer stack object containing a logical boolean value.**Details**

Marks a menu item as the only radio item in an item group.

pMarkLoc specifies the menu item of this menu to be marked with the radio button mark (typically a bullet symbol •). It is either an item identifier (item-ID) or a position as indicated by *pCmdLoc*. The menu item must be inclusively between the items specified by *pFirstLoc* and *pLastLoc*.

pFirstLoc specifies the first menu item of this menu in the item group. It is either an item identifier (item-ID) or a position as indicated by *pCmdLoc*. If it is an item-ID, it can be the identifier of any recursive submenu of this menu.

pLastLoc specifies the last menu item of this menu in the item group. It is either an item identifier (item-ID) or a position as indicated by *pCmdLoc*. It must be an item at the same menu level as specified for *pFirstLoc*.

pCmdLoc is true if *pMarkLoc*, *pFirstLoc*, and *pLastLoc* are an item-ID, otherwise *pMarkLoc*, *pFirstLoc*, and *pLastLoc* are the item position number.

success is true if the function was successful, otherwise it is false.

The function clears the radio button symbols from the menu items inclusively between those specified by *pFirstLoc* and *pLastLoc*, then sets the radio button for the menu item specified by *pMarkLoc*.

The radio button symbol can be cleared from the menu items in the item group (specified by *pFirstLoc* and *pLastLoc*) by setting *pMarkLoc* to an invalid position, although in this case, *success* will be false.

Other Information

@mCheckMItem() ♦

@mPutMItem()

@mPutMItem(*pIns* *pLoc* | -1 *pCmdLoc* *pItemSeq*) → *success*

<i>pIns</i>	An integer stack object containing a logical boolean value.
<i>pLoc</i>	An integer stack object.
<i>pCmdLoc</i>	An integer stack object containing a logical boolean value.
<i>pItemSeq</i>	A sequence stack object.
<i>success</i>	An integer stack object containing a logical boolean value.

Details

Appends, inserts, or modifies a menu item.

pIns is true to append or insert the menu item specified at *pItemSeq*. Otherwise (false) the specified menu item is modified.

pLoc specifies the menu item of this menu. It is either an item identifier (item-ID) or a position. If it is an item-ID, it can be the identifier of any recursive submenu of this menu. To append the specified menu item, -1 (negative one) is used instead of *pLoc*, and *pIns* is set to true (*pCmdLoc* is ignored).

pCmdLoc is true if *pLoc* is an item-ID, otherwise *pLoc* is the item position number. If *pLoc* is -1 (negative one), then this value is ignored.

pItemSeq is a sequence containing details relating to the specified menu item. The contents of the sequence is as follows.

[*item-label*, *item-id*, *menu-id* | 0, *bitmap* | 0, *mft-flags*, *mfs-flags*],

where

- 1 *item-label* is the menu label as specified when the item is created (string).
- 2 *item-id* is the application-defined 16-bit value that identifies the menu item (integer).
- 3 *menu-id* is the handle (HMENU) to the submenu associated with the menu item, or 0 (zero) if the menu item is not associated with a submenu (integer).
- 4 *bitmap* is the handle to the bitmap (HBITMAP) associated with the menu item, or 0 (zero) if the menu item is not associated with a bitmap. This value is obtained via the `@BITMAP` command (integer).
- 5 *mft-flags* is zero or more of the values listed in [Menu Item Types](#) (integer).
- 6 *mfs-flags* is zero or more of the values listed in [Menu Item States](#) (integer).

success is true if the function was successful, otherwise it is false.

Examples

The following illustrations show how the `@mPutMItem()` function can be used.

```
(1) void MData.@mPutMItem(true -1 false [...]);
(2) void MData.@mPutMItem(true 1000 true [...]);
(3) void MData.@mPutMItem(false 1000 true [...]);
```

In example (1), the specified menu item ([...]) is appended to the menu at the top level.

In example (2), the specified menu item ([...]) is inserted after the menu item identified by 1000.

In example (3), the menu item identified by 1000 is modified to the specified menu item ([...]). The function `@mGetMItem()` can be used to get the details of menu item 1000 to modify as desired, then the `@mPutMItem()` function can be called to modify the menu item (not shown in the example).

Additional Information

[Menu Item Types](#) ▪ [Menu Item States](#) ▪ `@BITMAP`

Other Information

`@mGetMItem()` ♦

@mRemoveMItem()

`@mRemoveMItem(pLoc pCmdLoc pDel) → rtn`

<i>pLoc</i>	An integer stack object.
<i>pCmdLoc</i>	An integer stack object containing a logical boolean value.
<i>pDel</i>	An integer stack object containing a logical boolean value.
<i>rtn</i>	An integer stack object containing a logical boolean value, or an integer stack object.

Details

Deletes a menu item, or detaches or deletes a submenu from a menu.

pLoc specifies the menu item of this menu to delete or detach. It is either an item identifier (item-ID) or a position. If it is an item-ID, it can be the identifier of any recursive submenu of this menu.

pCmdLoc is true if *pLoc* is an item-ID, otherwise *pLoc* is the item position number.

pDel is `true` to delete the specified item or submenu and return a boolean value (*rtn*), or `false` to detach the specified submenu (but delete the associated string item) and return its menu-ID, allowing the submenu to be reused.

rtn is the menu-ID (HMENU) of the menu if it is detached (but not deleted), otherwise the value is `true` if the function was successful or `false` if it was not successful.

If *pLoc* specifies a menu item that is not associated with a submenu, then that item is deleted whether or not *pDel* is `true`, and *rtn* will be a boolean value.

Other Information

[@mDestroyMenu\(\)](#) ♦

@mRepMInfo()

@mRepMInfo (*pMNSFlags*) → *prev-flags*

pMNSFlags An integer stack object containing binary flags.

prev-flags An integer stack object containing binary flags.

Details

Replaces the existing menu MNS flags of this menu with the specified ones (*pMNSFlags*), and returns the existing MNS flags (*prev-flags*).

pMNSFlags is zero or more of the values listed in [Menu Styles](#).

prev-flags are the existing MNS flags before replacement.

Additional Information

[Menu Styles](#) ♦

3.5.5 Status Bar

This is the definition of a status bar ETAC data object. A status bar data object allows access and manipulation of a status bar.

The [@vNewSBar\(\)](#) function is used to construct a status bar and attach it to a *VIEW*. A status bar is displayed in the client area of a *VIEW*. A status bar can be created for a dialog box via the [@NewSBar\(\)](#) function (see ETACProgLang(Official).doc for details).

A status bar retains two modes simultaneously, but only displays one of them at a time. The simple mode retains only one item (text and/or icon), and is referred to by the number 0 (zero). The parts mode retains a specified number of items (each having their own text and/or icon). The items of the parts mode are referred to by an integer beginning with the number 1 for the left part, and increasing by one for each subsequent part.

Note that a *VIEW* cannot have more than one status bar.

- Initials: `sb`
- Creator: [vis_NewSBarData\(\)](#)
- Deletion: [@mDestroySBar\(\)](#)

— Member Functions —

@sbDestroyIcon()**@sbDestroyIcon** (*plcon*) → *success**plcon* An integer stack object.*success* An integer stack object containing a logical boolean value.**Details**

Destroys an icon (*plcon*) previously loaded via the **@sbLoadIcon()** function.

plcon is the handle (HICON) of an icon obtained from the **@sbLoadIcon()** function.

success is true if the function was successful, otherwise it is false.

Note that icons are automatically destroyed when the application program closes.

Related Information

@sbLoadIcon()

Other Information

@sbRepIcon() ♦

@sbDestroySBar()**@sbDestroySBar()****Details**

Destroys the status bar (the icons of the status bar are not destroyed).

This function does not destroy any icons used in the status bar. Those icons can be destroyed via the **@sbDestroyIcon()** function.

This function does not destroy the status bar data object itself. However, the data object is unusable (becomes defunct) after the status bar is destroyed.

Note that a status bar that is assigned to a *VIEW* is automatically destroyed when the application closes.

Related Information

@sbDestroyIcon() ♦

@sbGetSBarHndl()**@sbGetSBarHndl()** → *sbar-hndl* | ?*sbar-hndl* An integer stack object.**Details**

Returns the status bar handle (*sbar-hndl*) associated with this status bar data object.

sbar-hndl is the status bar handle of this data object, or a NULL stack object (?) is returned if this data object does not have a status bar handle.

If this data object does not have a status bar handle, then it is defunct and should be deleted. The data object becomes defunct when the status bar is deleted (via `@sbDestroySBar()`). The value represented by *sbar-hndl* can therefore be used to determine whether this data object was deleted via `@sbDestroySBar()`. ♦

@sbGetSize()

`@sbGetSize()` → *size*

size A decimal sequence stack object.

Details

Gets the width and height (*size*) of the status bar.

size is a TAC sequence in the form: [*width*, *height*]. Each value of the sequence is in *logical picas*. ♦

@sbLoadIcon()

`@sbLoadIcon(pPath pSize)` → *icon-id* | ?

pPath A string stack object.

pSize A decimal stack object.

icon-id An integer stack object.

Details

Loads an icon existing in a file (*pPath*), returning its identifier (*icon-id*).

pPath is the full or relative path and filename of the file containing the icon. The path must be to a file containing an icon image, this is typically a file with extension *.ico.

pSize is the desired size of the icon in *logical picas*. If this value is 0 (zero), the size will be the natural size of the icon.

icon-id is the icon identifier (HICON) of the loaded icon, or 0 (zero) if the file content was invalid for an icon. A NULL stack object (?) is returned if the file does not exist or the path is invalid.

A loaded icon should be deleted via the `@sbDestroyIcon()` function after it is no longer needed to reduce unnecessary resources. However, icons are automatically destroyed when the application program closes.

Related Information

`@sbRepIcon()` ▪ `@sbDestroyIcon()` ♦

@sbRepBkColr()

`@sbRepBkColr(pColr | ?)` → *prev-colr*

pColr An integer stack object.

prev-colr An integer stack object.

Details

Replaces the existing background colour of the status bar to the specified one (*pColr*), and returns the existing colour (*prev-colr*).

pColr is the background colour of the status bar. The colour is an RGB colour which can be obtained from the **vis_RGB()** function. If a NULL stack object (?) is used instead of *pColr*, then only the existing background colour is returned (*prev-colr*).

prev-colr is the existing background colour before the new colour is set.

Other Information

[Standard Colours](#) ▪ [Common Colours](#) ▪ **vis_RGB()** ♦

@sbReplIcon()	
@sbReplIcon(<i>pPartNum</i> <i>pIcon</i> ?) → <i>icon-id</i> ?	
<i>pPartNum</i>	An integer stack object.
<i>pIcon</i>	An integer stack object.
<i>icon-id</i>	An integer stack object.

Details

Replaces the existing icon of a status bar part (*pPartNum*) with the specified one (*pIcon*), and returns the existing icon (*icon-id*).

pPartNum is the status bar part number for which the icon is to be replaced (or obtained). A value of 0 (zero) indicates the simple part. A non-zero value indicates the desired part number of a complex part (part numbers begin with 1).

pIcon is the identifier of the icon to replace the existing one in the specified part. If a NULL stack object is used instead of *pIcon*, then only the existing icon identification is returned (*icon-id*).

icon-id is the existing icon identifier (HICON) of the specified part before replacement, or a NULL stack object (?) is returned if this function fails.

An icon identifier can be obtained via the **@sbLoadIcon()** function.

Related Information

@sbLoadIcon() ♦

@sbRepMode()	
@sbRepMode(<i>pMode</i> ?) → <i>prev-mode</i>	
<i>pMode</i>	An integer stack object containing a logical boolean value.
<i>prev-mode</i>	An integer stack object containing a logical boolean value.

Details

Replaces the existing status bar display mode to the specified one (*pMode*), and returns the existing display mode (*prev-mode*).

pMode is **true** to set the display mode to the simple mode, or **false** to set it to the parts mode. If a NULL stack object (?) is used instead of *pMode*, then only the existing display mode is returned (*prev-mode*).

prev-mode is the existing display mode of the status bar before replacement. A value of **true** indicates a simple mode, and a value of **false** indicates a parts mode.

A status bar can display only one mode at a time, even though it retains two separate modes. ♦

@sbRepParts()	
@sbRepParts (<i>pParts</i> ?) → <i>prev-parts</i> ?	
<i>pParts</i>	A decimal sequence stack object.
<i>prev-parts</i>	A decimal sequence stack object.

Details

Replaces the existing part widths of the parts mode of the status bar with the specified widths (*pParts*), returning the existing part widths (*prev-parts*).

pParts is a TAC sequence containing the widths of the parts in *logical picas*. The last width may be `-1.0` to indicate that the width is the rest of the status bar. Each element of the sequence refers to the corresponding part of the status bar. The number of parts cannot be greater than 255. If a NULL stack object is used instead of *pParts*, then only the existing part widths are returned (*prev-parts*).

prev-parts is a TAC sequence containing the existing widths (as described in *pParts*) before replacement, or a NULL stack object (?) is returned if this function fails.

The parts of the parts mode of the status bar are defined by this function. For example, if *pParts* has three elements, then the number of parts of the status bar will be three. Note that the simple mode of a status bar does not have parts.

Examples

The following illustrations show how the `@sbRepParts()` function can be used.

(1) <code>void SBar.@sbRepParts([3.5, 5.0, 4,78, -1]);</code>
(2) <code>Widths := SBar.@sbRepParts(?);</code>

In example (1), the parts mode of the status bar contains four parts, each with the specified widths in the given order. The last part is the remaining section to the end of the status bar.

In example (2), the `@sbRepParts()` function returns the part widths of the parts mode of the status bar.

Other Information

`@sbSetMinHeight()` ♦

@sbRepText()	
@sbRepText (<i>pPartNum</i> <i>pText</i> ? <i>pType</i>) → <i>prev-text</i> ?	
<i>pPartNum</i>	An integer stack object.
<i>pText</i>	A string stack object.
<i>pType</i>	An integer stack object.
<i>prev-text</i>	A string stack object.

Details

Replaces the existing text of the status bar part (*pPartNum*) with the specified text (*pText*), returning the existing text of that part (*prev-text*).

pPartNum is the status bar part number for which the text is to be replaced (or obtained). A value of 0 (zero) indicates the simple part. A non-zero value indicates the desired part number of a complex part (part numbers begin with 1).

pText is the text to replace the existing text. If a NULL stack object is used instead of *pText*, then only the existing text is returned (*prev-text*).

pType is zero or more of the value listed in [Status Bar Text](#).

prev-text is the existing text of the specified part before replacement, or a NULL stack object (?) is returned if this function fails.

Additional Information

[Status Bar Text](#)

Other Information

@sbRepIcon () ♦

@sbSetMinHeight()

@sbSetMinHeight (*pHeight*)

pHeight A decimal stack object.

Details

Sets the minimum height (*pHeight*) of the status bar.

pHeight is the minimum height of the status bar in *logical picas*.

The actual height of the status bar could be larger than the set minimum height if, for example, the status bar text is higher than the minimum height. ♦

@sbShowSBar()

@sbShowSBar (*pShow* | ?) → *prev-state*

pShow An integer stack object containing a logical boolean value.

success An integer stack object containing a logical boolean value.

Details

Shows or hides (*pShow*) the status bar, or just returns the previous visibility state (*prev-state*).

pShow is true to make the status bar visible, or false to make it invisible. If a NULL stack object (?) is used instead of *pShow*, then only the existing visibility state is returned (*prev-state*).

prev-state is the existing visibility state of the status bar before the visibility state change. If the value is true, then the status bar was visible before the state change, otherwise it was invisible. ♦

3.5.6 Idle-time Job

This is the definition of an idle-time job ETAC data object. An idle-time job contains a function that is run at application program at idle-time. Any number of idle-time jobs can exist simultaneously. Idle-time jobs are queued and run only once unless specified to recur by the program.

The **vis_WaitForEvents()** function periodically inspects and processes queued operating system event messages, related to an application program, and performs idle-time job processing between the inspections, even if there are no queued events to process.

Idle-time jobs are processed only if allowed by the **vis_WaitForEvents()** function, which is the default.

- Initials: `itj`
- Creator: **vis_NewIdleJobData()**

— Member Variables —

@itjEvents

@itjEvents := *events* (**false**)

events An integer stack object containing a logical boolean value.

Details

VIS sets **@itjEvents** to `true` immediately after all the current operating system event messages for the application program have just been processed. If there are no queued event messages when VIS inspects the event message queue, VIS sets **@itjEvents** to `false`. The program should not set this value. ♦

@itjJobActive

@itjJobActive := *active* (**false**)

active An integer stack object containing a logical boolean value.

Details

VIS sets **@itjJobActive** to `true` if the idle-time job is currently being processed; otherwise **@itjJobActive** is `false`. The program should not set this value. ♦

@itjJobCount

@itjJobCount := *job-count* (?)

job-count An integer stack object.

Details

VIS increases **@itjJobCount** for each recurrence of the idle-time job; the default value of **@itjJobCount** is a NULL stack object (?). The program should not set this value.

The idle-time job recurs if the program sets the **@itjMoreTime** variable to `true` from within the body of the job function variable, **@itjJobFnt**. If the job does not recur, the value of **@itjJobCount** will be 0 (zero).

Related Information

@itjMoreTime ♦

@itjJobEndFnt

@itjJobEndFnt := *end-fnt* (?)

end-fnt An ETAC function.

Details

Possibly set by the program for VIS to call after the idle-time job has been completed.

end-fnt is an ETAC function which is called by VIS after it last calls the **@itjJobFnt** function variable. The function has no parameters.

After this function variable is called, the idle-time job is removed by VIS. The program need not set this function if it is not required.

Other Information

@itjJobFnt ♦

@itjJobFnt

@itjJobFnt := *job-fnt* (?)

job-fnt An ETAC function.

Details

Set by the program to perform the desired job during idle-time.

job-fnt is an ETAC function, called by VIS, containing the job that is performed during application idle-time. The function has no parameters.

The program should always set this function variable, otherwise idle-time processing would be pointless.

Other Information

@itjJobEndFnt ♦

@itjJobID

@itjJobID := *job-id* (?)

job-id An integer or string stack object.

Details

Set by the program to identify the idle-time job.

job-id is a unique identifier set by the program before the idle-time job is posted.

This variable is used by the program to identify particular idle-time jobs. It is also used by the **vis_DeleteIdleJob()** function to delete a queued idle-time job. This variable must be set if the job is to be deleted while still queued.

Other Information

vis_DeleteIdleJob() ♦

@itjJobQueued

@itjJobQueued := *queued* (false)

queued An integer stack object containing a logical boolean value.

Details

VIS sets **@itjJobQueued** to `true` when the idle-time job is queued waiting to begin. The program should not set this value.

If the job is being processed, then this variable will be `false`. Therefore, the variable will always be `false` from within the **@itjJobFnt** and **@itjJobEndFnt** function variables. The variable will also be `false` if the job has not yet been posted via the **vis_PostIdleJob()** function.

Other Information

vis_PostIdleJob() ♦

@itjMoreTime

@itjMoreTime := *more-time* (false)

more-time An integer stack object containing a logical boolean value.

Details

Set by the program within the **@itjJobFnt** function variable to indicate whether the same idle-time job is to recur in the next idle-time period.

more-time is `true` if the job requires more time to be processed.

If the **@itjJobFnt** function variable sets this variable to `true`, then that function variable is called again by VIS, after processing existing operating system messages without pausing, with an increment to the **@itjJobCount** variable. Normally, the event message processing operation pauses each time before inspecting and processing the message queue. The pause period is set by the **vis_WaitForEvents()** function.

Example

The following illustration shows how the **@itjMoreTime** variable can be used.


```

[* Create and post an idle-time job to process in parts. *]
Job := vis_NewIdleJobData();

Job. [* Idle-time job data. *]
{
  @itjJobID := 10;
  @itjJobFnt := uses_parts static_local [...] fnt:()
  {
    @Parts
    [
      {[* Do part 1. *]; @itjMoreTime := true;},
      {[* Do part 2. *]; @itjMoreTime := true;},
      {[* Do part 3. *]; @itjMoreTime := true;},
      {@itjMoreTime := false;}
    ];
  };
};

vis_PostIdleJob(Job);

```

In the example above, an idle-time job has been created, but it is too time consuming to process all in one call. The job, in this case, can be split up into three parts, each of which is performed by a separate call to the **@itjJobFnt** function variable. The ETAC `uses_parts` command causes the job function to execute each of the procedure elements of the sequence argument to the `@Parts` command in turn for each call of the **@itjJobFnt** function variable.

The process of performing the job is as follows. When VIS first calls the **@itjJobFnt** function variable, only the first procedure (`[* Do part 1. *]`) is executed (via `uses_parts`). Because **@itjMoreTime** is set to `true`, VIS next calls the function variable again, but this time only the second procedure (`[* Do part 2. *]`) is executed, and so on until **@itjMoreTime** is set to `false` in the last procedure, after which the function variable is no longer called, so the job completes.

Typically, to execute a job in parts, the state of the job needs to be saved between parts. The ETAC `static_local` command creates a static local dictionary for the job function, in which the job state can be stored for use in each call to the function.

Other Information

vis_WaitForEvents() ♦

@itjProjectID

@itjProjectID := *proj-id* (?)

proj-id An integer or string stack object.

Details

Possibly set and used by the program if the idle-time job is not for the *current* project.

proj-id is the *crucial identifier* or *name identifier* of the project *component* instance relevant to the idle-time job. VIS sets this to the *current* project *CID* in the **vis_NewIdleJobData()** function.

When the idle-time job is executed, the *current* project may not be the one appropriate for the job. This variable can be used to make the required project *current* before the job is performed.

Example

The following illustration shows how the **@itjProjectID** variable can be used.

```

[* Create and post an idle-time job. *]
Job := vis_NewIdleJobData();

Job. [* Idle-time job data. *]
{
  @itjJobID := 10;
  @itjProjectID := "The Project";
  @itjJobFnt := fnt:()
  {
    SavedProjID :- ?;

    SavedProjID := vis_SetCurProject(@itjProjectID);

    [* Perform the job. *]

    void vis_SetCurProject(SavedProjID);
  };
};

vis_PostIdleJob(Job);

```

In the example above, the idle-time job needs to be executed only for the project *component* instance named “The Project”. The *name identifier* of that project is set to the @itjProjectID variable to be available for use in the job.

Other Information

vis_SetCurProject() ♦

@itjUrgent	
@itjUrgent := urgent (false)	
urgent	An integer stack object containing a logical boolean value.

Details

Set by the program to put the idle-time job first in the job queue rather than last.

urgent is true to put the job first in the job queue when the job is posted so that the job is processed before all other queued jobs. If this value is false, the job is put last in the job queue.

Idle-time jobs are normally processed by VIS in the order that they are posted. ♦

3.5.7 API Commands

The following sections describe the API commands available for VIS.

@BITMAP	
@BITMAP <i>bmattr-id</i> → <i>bitmap-hndl</i>	
<i>bmattr-id</i>	An integer or string stack object.
<i>bitmap-hndl</i>	An integer stack object.

Details

Extracts the bitmap handle (*bitmap-hndl*) from the specified bitmap attributes *component* instance (*bmattr-id*).

bmattr-id is the *crucial identifier* or *name identifier* of a previously created bitmap attributes *component* instance.

bitmap-hndl is the handle (HBITMAP) of the specified bitmap attributes *component* instance.

This command is currently used by the @mPutMItem() function and the @SMENU and @MITEM commands.

Related Information

@mPutMItem() ▪ @SMENU ▪ @MITEM ♦

@DLGMENU

@DLGMENU *body* → *menu-hndl* | ?

body A procedure stack object.

menu-hndl An integer stack object.

Details

Creates a menu bar for a dialog box, returning its handle (*menu-hndl*).

body is a procedure containing the details of the menu bar to construct. The menu bar is constructed via the @SMENU and @MITEM commands.

menu-hndl is the handle (HMENU) of the constructed menu bar, or a NULL stack object (?) is returned if this command fails.

The menu is attached to the dialog box via the @RepMBar() function described in the document ETACProgLang(Official).pdf.

The vis_NewMenuData() function can be used with *menu-hndl* to access and manipulate the menu items and submenus.

Additional Information

@SMENU ▪ @MITEM

Other Information

vis_NewMenuData() ♦

@MENU

@MENU *view-id* | ? *body* → *menu-hndl* | ?

view-id An integer stack object.

body A procedure stack object.

menu-hndl An integer stack object.

Details

Creates and possibly attaches a menu bar for a *VIEW* (*view-id*), returning its handle (*menu-hndl*).

view-id is the *crucial identifier* or *name identifier* of a previously created VIEW *component* instance to which the constructed menu bar is attached. If a NULL stack object (?) is used instead of *view-id*, then the menu bar is not attached.

body is a procedure containing the details of the menu bar to construct. The menu bar is constructed via the @SMENU and @MITEM commands.

menu-hndl is the handle (HMENU) of the constructed menu bar, or a NULL stack object (?) is returned if this command fails.

The `vis_NewMenuData()` function can be used with *menu-hndl* to access and manipulate the menu items and submenus.

See `@vMenuID` for an example of how to construct a menu.

Additional Information

`@SMENU` - `@MITEM`

Other Information

`vis_NewMenuData()` ♦

@MITEM	
@MITEM <i>label</i> ? <i>item-id</i> ? <i>bitmap-hndl</i> ? <i>mft-flags</i> ? <i>mfs-flags</i> ?	
<i>label</i>	A string stack object.
<i>item-id</i>	An integer stack object.
<i>bitmap-hndl</i>	An integer stack object.
<i>mft-flags</i>	An integer stack object containing binary flags.
<i>mfs-flags</i>	An integer stack object containing binary flags.

Details

Creates a menu item as part of the construction of a menu or menu bar.

label is the menu label. It can contain a shortcut character preceded by the ampersand (&) symbol.

item-id is the item identifier (item-ID) for this menu item. The identifier should be unique for all menu items within the same drawing plane *component* instance to avoid unpredictable consequences.

bitmap-hndl is the handle (HBITMAP) obtained from the `@BITMAP` command.

mft-flags is zero or more of the values listed in [Menu Item Types](#).

mfs-flags is zero or more of the values listed in [Menu Item States](#).

If a NULL stack object (?) is used instead of any of the arguments above, then the corresponding property is not set for the menu item.

This command MUST only be called from within the body of a `@MENU`, `@DLGMENU`, or `@POPUP` command.

Examples

The following illustrations show how the `@MITEM` command can be used.

```
(1) @MITEM "&Copy" 1021 ? ? ?;
(2) @MITEM ? ? ? :!MFT_SEPARATOR: ?;
(3) @MITEM "&Default" 1016 @BITMAP "Bitmap" ? ?;
```

In example (1), a regular menu item with only a label and item-ID is created.

In example (2), an item separator is created.

In example (3), an item with a bitmap and label is created. The bitmap exists in a previously created bitmap attributes *component* instance identified by “Bitmap”.

Additional Information

[Menu Item Types](#) ▪ [Menu Item States](#)

Related Information

[@MENU](#) ▪ [@DLGMENU](#) ▪ [@POPUP](#) ▪ [@BITMAP](#)

Other Information

[@mPutMItem\(\)](#) ▪ [@mGetMItem\(\)](#) ♦

@POPUP

@POPUP *mns-flags* | ? *body* → *menu-hndl* | ?

mns-flags An integer stack object containing binary flags.

body A procedure stack object.

menu-hndl An integer stack object.

Details

Creates a pop-up menu, returning its handle (*menu-hndl*).

mns-flags is zero or more of the values listed in [Menu Styles](#). If a NULL stack object (?) is used instead of *mns-flags*, then no properties are set for the menu.

body is a procedure containing the details of the menu to construct. The menu is constructed via the **@SMENU** and **@MITEM** commands.

menu-hndl is the handle (HMENU) of the constructed menu, or a NULL stack object (?) is returned if this command fails.

The returned menu handle (*menu-hndl*) is used with the **@vShowPopup()** function.

Additional Information

[Menu Styles](#) ▪ [@SMENU](#) ▪ [@MITEM](#)

Other Information

[@vShowPopup\(\)](#) ♦

@SMENU

@SMENU *label* | ? *item-id* | ? *bitmap-hndl* | ? *mft-flags* | ? *mfs-flags* | ? *mns-flags* | ? *body*

label A string stack object.

item-id An integer stack object.

bitmap-hndl An integer stack object.

mft-flags An integer stack object containing binary flags.

mfs-flags An integer stack object containing binary flags.

mns-flags An integer stack object containing binary flags.

body A procedure stack object.

Details

Creates a submenu item as part of the construction of a menu or menu bar.

label is the menu label. It can contain a shortcut character preceded by the ampersand (&) symbol.

item-id is the item identifier (item-ID) for this submenu item. The identifier should be unique for all menu items within the same drawing plane *component* instance to avoid unpredictable consequences.

bitmap-hndl is the handle (HBITMAP) obtained from the **@BITMAP** command.

mft-flags is zero or more of the values listed in [Menu Item Types](#).

mfs-flags is zero or more of the values listed in [Menu Item States](#).

mns-flags is zero or more of the values listed in [Menu Styles](#).

If a NULL stack object (?) is used instead of any of the arguments above, then the corresponding property is not set for the submenu item.

body is a procedure containing the details of the submenu to construct. The submenu is constructed via the **@SMENU** and **@MITEM** commands.

This command **MUST** only be called from within the body of a **@MENU**, **@DLGMENU**, or **@POPUP** command.

Additional Information

[Menu Item Types](#) ▪ [Menu Item States](#) ▪ [Menu Styles](#) ▪ **@SMENU** ▪ **@MITEM**

Related Information

@BITMAP ▪ **@MENU** ▪ **@DLGMENU** ▪ **@POPUP**

Other Information

@mPutMItem() ▪ **@mGetMItem()** ♦

@vis_LMBDclk

@vis_LMBDclk *mse-flags* → *bool*

mse-flags An integer stack object containing binary flags.

bool An integer stack object containing a logical boolean value.

Details

Indicates (*bool*) whether the left mouse button has just been double-clicked.

mse-flags is the value of the `pMseFlags` parameter of the event handler functions registered by the **@dpSetMseBtnFnt()**, **@goSetMseBtnFnt()**, **@dpSetMseHitFnt()**, **@goSetMseHitFnt()**, and **@dpSetMseMoveFnt()** functions.

bool is true if *mse-flags* indicates that the left mouse button has just been double-clicked, otherwise it is false.

Related Information

@dpSetMseBtnFnt() ▪ **@goSetMseBtnFnt()** ▪ **@dpSetMseHitFnt()** ▪ **@goSetMseHitFnt()**
▪ **@dpSetMseMoveFnt()** ♦

@vis_LMBPrs**@vis_LMBPrs** *mse-flags* → *bool**mse-flags* An integer stack object containing binary flags.*bool* An integer stack object containing a logical boolean value.**Details**Indicates (*bool*) whether the left mouse button has just been pressed.*mse-flags* is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.*bool* is true if *mse-flags* indicates that the left mouse button has just been pressed, otherwise it is false.**Related Information**

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
 ▪ `@dpSetMseMoveFnt()` ♦

@vis_LMBRel**@vis_LMBRel** *mse-flags* → *bool**mse-flags* An integer stack object containing binary flags.*bool* An integer stack object containing a logical boolean value.**Details**Indicates (*bool*) whether the left mouse button has just been released.*mse-flags* is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.*bool* is true if *mse-flags* indicates that the left mouse button has just been released, otherwise it is false.**Related Information**

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
 ▪ `@dpSetMseMoveFnt()` ♦

@vis_MMBDclk**@vis_MMBDclk** *mse-flags* → *bool**mse-flags* An integer stack object containing binary flags.*bool* An integer stack object containing a logical boolean value.**Details**Indicates (*bool*) whether the middle mouse button has just been double-clicked.

mse-flags is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.

bool is true if *mse-flags* indicates that the middle mouse button has just been double-clicked, otherwise it is false.

Related Information

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
▪ `@dpSetMseMoveFnt()` ♦

@vis_MMBPrs

`@vis_MMBPrs` *mse-flags* → *bool*

mse-flags An integer stack object containing binary flags.

bool An integer stack object containing a logical boolean value.

Details

Indicates (*bool*) whether the middle mouse button has just been pressed.

mse-flags is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.

bool is true if *mse-flags* indicates that the middle mouse button has just been pressed, otherwise it is false.

Related Information

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
▪ `@dpSetMseMoveFnt()` ♦

@vis_MMBRel

`@vis_MMBRel` *mse-flags* → *bool*

mse-flags An integer stack object containing binary flags.

bool An integer stack object containing a logical boolean value.

Details

Indicates (*bool*) whether the middle mouse button has just been released.

mse-flags is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.

bool is true if *mse-flags* indicates that the middle mouse button has just been released, otherwise it is false.

Related Information

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
▪ `@dpSetMseMoveFnt()` ♦

@vis_MseMove

@vis_MseMove *mse-flags* → *bool*

mse-flags An integer stack object containing binary flags.

bool An integer stack object containing a logical boolean value.

Details

Indicates (*bool*) whether the mouse cursor has just moved.

mse-flags is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.

bool is true if *mse-flags* indicates that the mouse cursor has just been moved, otherwise it is false.

This function is useful if *mse-flags* is not tied to any particular handler, for example, if *mse-flags* is passed to a common function from a number of different mouse handlers. That common function can then determine whether a mouse movement event or a mouse button event occurred.

Related Information

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
▪ `@dpSetMseMoveFnt()` ♦

@vis_MseTimeSpan

@vis_MseTimeSpan *mse-flags* → *span*

mse-flags An integer stack object containing binary flags.

span An integer stack object.

Details

Returns the number of milliseconds (*span*) since the last mouse button event.

mse-flags is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.

span is the number of milliseconds since the relevant last mouse button event. For a mouse movement event, the relevant mouse button event is any of the three mouse button events; for a mouse button event, the relevant mouse button event is for that particular button that triggered the event.

Related Information

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
▪ `@dpSetMseMoveFnt()` ♦

@vis_RMBDclk**@vis_RMBDclk** *mse-flags* → *bool**mse-flags* An integer stack object containing binary flags.*bool* An integer stack object containing a logical boolean value.**Details**Indicates (*bool*) whether the right mouse button has just been double-clicked.*mse-flags* is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.*bool* is true if *mse-flags* indicates that the right mouse button has just been double-clicked, otherwise it is false.**Related Information**

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
 ▪ `@dpSetMseMoveFnt()` ♦

@vis_RMBPrs**@vis_RMBPrs** *mse-flags* → *bool**mse-flags* An integer stack object containing binary flags.*bool* An integer stack object containing a logical boolean value.**Details**Indicates (*bool*) whether the right mouse button has just been pressed.*mse-flags* is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.*bool* is true if *mse-flags* indicates that the right mouse button has just been pressed, otherwise it is false.**Related Information**

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
 ▪ `@dpSetMseMoveFnt()` ♦

@vis_RMBRel**@vis_RMBRel** *mse-flags* → *bool**mse-flags* An integer stack object containing binary flags.*bool* An integer stack object containing a logical boolean value.**Details**Indicates (*bool*) whether the right mouse button has just been released.

mse-flags is the value of the `pMseFlags` parameter of the event handler functions registered by the `@dpSetMseBtnFnt()`, `@goSetMseBtnFnt()`, `@dpSetMseHitFnt()`, `@goSetMseHitFnt()`, and `@dpSetMseMoveFnt()` functions.

bool is true if *mse-flags* indicates that the right mouse button has just been released, otherwise it is false.

Related Information

`@dpSetMseBtnFnt()` ▪ `@goSetMseBtnFnt()` ▪ `@dpSetMseHitFnt()` ▪ `@goSetMseHitFnt()`
▪ `@dpSetMseMoveFnt()` ♦

3.5.8 API Functions

The following sections describe the API functions for VIS.

@VIS()

`@VIS(pCompIDName)` → *comp*

pCompIDName An integer or string stack object.

comp An ETAC data object.

Details

Gets the specified (*pCompIDName*) *component* instance (*comp*).

pCompIDName is the *crucial identifier* or *name identifier* of a previously created *component* instance.

comp is the actual *component* instance (an ETAC data object) requested.

If the specified (*pCompIDName*) *component* instance does not exist in the *current* project, then an error event occurs. To avoid the error event, see the `@VISX()` function.

It is not advisable to store the actual *component* instance returned by this function. The typical and recommended way to use this function is via the selection (“dot”) operator directly. For example, `@VIS(id).member` rather than `variable := @VIS(id); variable.member`. The reason that the selection operator should be used directly is that this function does some error checking on the specified *component* instance, especially in DEBUG mode, each time it is called, whereas accessing a member from a variable does not do any checking.

Component instances of the following types must exist on the *current drawing plane* before calling this function with any of those instances: `visC_GObject`, `visC_LineItem`, `visC_CurveItem`, `visC_TextItem`, `visC_ImageItem`, `visC_FormText`, `visC_GContext`, `visC_Pen`, `visC_Brush`, `visC_Bitmap`, `visC_TextStyle`, `visC_MatObject`, `visC_MesObject`. Otherwise, the consequence is unpredictable in non-DEBUG mode, and an error event will occur in DEBUG mode.

Other Information

`@VISX()` ▪ [Component Types](#) ♦

@VISX()

@VISX(*pCompIDName* *pProc*)*pCompIDName* An integer or string stack object.*pProc* A procedure stack object.

Details

Executes a procedure (*pProc*) if the specified (*pCompIDName*) *component* instance exists.

pCompIDName is the *crucial identifier* or *name identifier* of a previously created *component* instance.

pProc is the procedure that is executed if the specified (*pCompIDName*) *component* instance exists. If the instance does not exist, then the procedure is not executed.

If the specified (*pCompIDName*) *component* instance does not exist in the *current* project, then no action will occur. If the specified (*pCompIDName*) *component* instance does exist in the *current* project, then *component* instances of the following types must also exist on the *current drawing plane* before calling this function with any of those instances: **visC_GObject**, **visC_LineItem**, **visC_CurveItem**, **visC_TextItem**, **visC_ImageItem**, **visC_FormText**, **visC_GContext**, **visC_Pen**, **visC_Brush**, **visC_Bitmap**, **visC_TextStyle**, **visC_MatObject**, **visC_MesObject**. Otherwise, the consequence is unpredictable in non-DEBUG mode, and an error event will occur in DEBUG mode.

Examples

The following illustrations show how the **@VISX()** function can be used.

```
(1) @VISX("MyGObj") {void @goShow(false);};
(2) @VISX("MyVIEW") {@visDelete();};
```

In example (1), if the graphics object *component* instance identified by "MyGObj" does not exist in the *current* project, then no action occurs, otherwise the graphics object is made invisible.

In example (2), if the specified *VIEW* (MyVIEW) does not exist with the *current* project (it may have been deleted by the user) then no action occurs, otherwise the *VIEW* is deleted.

Other Information

@VIS() - [Component Types](#) ♦

vis_Bitmap()

vis_Bitmap(*pCompName* *pProc*)*pCompName* A string stack object.*pProc* A procedure stack object.

Details

Creates a bitmap attributes *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the bitmap *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.14_Bitmap Attributes Component](#) - [3.4.17_Bitmap Attributes](#) ♦

vis_Brush()

vis_Brush(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a brush attributes *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the brush attributes *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.13_Brush Attributes Component](#) - [3.4.16_Brush Attributes](#) ♦

vis_ColorDlg()

vis_ColorDlg(*pColr* *pFlags* *pVIEWIDName* | ?) → *colr-ref* | -1 | ?

pColr An integer stack object.

pFlags An integer stack object.

pVIEWIDName An integer or string stack object.

colr-ref An integer stack object.

Details

Gets a colour (*colr-ref*) from the user via a dialog box.

pColr is the default colour selection for the user. The colour is an RGB colour which can be obtained from the **vis_RGB()** function.

pFlags is zero or more of the values listed in [Colour Dialog Box](#).

pVIEWIDName is the *crucial identifier* or *name identifier* of a previously created VIEW *component* instance to be the owner window of the colour dialog box. If a NULL stack object (?) is used instead of *pVIEWIDName*, then the colour dialog box will be owned by VIS.

colr-ref is the RGB colour (COLORREF) that was selected in the colour dialog box. The value -1 (negative one) is returned if the user cancelled the dialog box, or a NULL stack object (?) is returned if an error occurred.

Additional Information[Colour Dialog Box](#)**Other Information**[Standard Colours](#) ▪ [Common Colours](#) ▪ [vis_RGB\(\)](#) ♦**vis_CompExists()****vis_CompExists**(*pCompIDName*) → *bool**pCompIDName* An integer or string stack object.*bool* An integer stack object containing a logical boolean value.**Details**

Determines whether or not (*bool*) the specified (*pCompIDName*) *component* instance exists.

pCompIDName is the *crucial identifier* or *name identifier* of a previously created *component* instance.

bool is true if the specified *component* instance exists with the *current* project, otherwise the value is false. ♦

vis_CurveItem()**vis_CurveItem**(*pCompName* *pProc*)*pCompName* A string stack object.*pProc* A procedure stack object.**Details**

Creates a curve item *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the curve item *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information[1.3.7_Curve Item Component](#) ▪ [3.4.10_Curve Item](#) ♦

vis_CvtToCID()

vis_CvtToCID (*pCompIDName*) → *cid*

pCompIDName An integer or string stack object.

cid An integer stack object.

Details

Converts a non-project *component* instance identifier (*pCompIDName*) to its corresponding *CID* (*cid*).

pCompName the *crucial identifier* or *name identifier* of a previously created *component* instance.

cid is the *crucial identifier* of the specified *component* instance.

If the specified *component* instance (*pCompIDName*) does not exist with the *current* project, then an error event will occur.

Other Information

[vis_CompExists\(\)](#) ♦

vis_DataPack()

vis_DataPack (*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a data pack *component* instance.

pCompName is the unique *name identifier* of the data pack *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.20_Data Pack Component](#) - [3.4.23_Data Pack](#) ♦

vis_DeleteIdleJob()

vis_DeleteIdleJob (*pJobID*)

pJobID An integer or string stack object.

Details

Deletes the specified (*pJobID*) idle-time job from the idle-time job queue.

pJobID is the unique identifier set by the program before the idle-time job was posted.

Idle-time jobs are automatically deleted from the idle-time job queue before VIS processes them. This function can be used to delete a queued idle-time job before it is processed by VIS.

Additional Information

[@itjJobID](#) ♦

vis_DeleteProject()

vis_DeleteProject(*pProjIDName*)

pProjIDName An integer or string stack object.

Details

Deletes the specified (*pProjIDName*) project *component* instance.

pProjIDName is the *crucial identifier* or *name identifier* of a previously created project *component* instance.

The default project *component* instance, named “@VIS@”, cannot be deleted. Neither can the *current* project be deleted.

Other Information

[vis_SetCurProject\(\)](#) ♦

vis_DPlane()

vis_DPlane(*pCompName* | ? *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a drawing plane *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the drawing plane *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance. If a NULL stack object (?) is used instead of *pCompName*, then the *component* instance corresponding to the *current drawing plane* is made temporarily *active* during the execution of the procedure specified by *pProc*.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.3_Drawing Plane Component](#) - [3.4.6_Drawing Plane](#) ♦

vis_EndWaitEvents()

vis_EndWaitEvents(*pVIEWIDName*)

pVIEWIDName An integer or string stack object.

Details

Ends user and system event processing handled by the specified (*pVIEWIDName*) *VIEW*.

pVIEWIDName is the *crucial identifier* or *name identifier* of the *VIEW* indicated by the **vis_WaitForEvents()** function.

The event processing is normally terminated by closing or deleting the specified (*pVIEWIDName*) *VIEW*. However, this function can be used to terminate event processing for the specified *VIEW* without deleting that *VIEW*. When event processing is terminated, the corresponding **vis_WaitForEvents()** function ends. However, if events are being processed by a different *VIEW* while the **vis_WaitForEvents()** function is waiting for events, then the **vis_WaitForEvents()** function will end after event processing is terminated for that different *VIEW*. This function does not actually terminate event processing itself. Rather, it marks the specified *VIEW* so that event processing handled by it should terminate at an appropriate time before it processes the next event. The **vis_WaitForEvents()** function also ends if the timeout value specified to it elapses.

Note that event processing can be passed on to other *VIEWS* within event handlers (via the **vis_WaitForEvents()** function) from a *VIEW* that is currently handling event processing.

Related Information

vis_WaitForEvents() ♦

vis_FixCurveItem()

vis_FixCurveItem(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a fixed curve item *component* instance with the specified name (*pCompName*).

This is the same as for the **vis_CurveItem()** function but the measurements of appropriate member variables are in *logical picas* not *base units*.

A fixed curve item does not change its viewed size when the *VIEW* scale changes.

Additional Information

vis_CurveItem() ♦

vis_FixFormText()

vis_FixFormText(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a fixed formatted text item *component* instance with the specified name (*pCompName*).

This is the same as for the **vis_FormItem()** function but the measurements of appropriate member variables are in *logical picas* not *base units*.

A fixed formatted text item does not change its viewed size when the *VIEW* scale changes.

Additional Information

vis_FormItem() ♦

vis_FixImageItem()

vis_FixImageItem(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a fixed image item *component* instance with the specified name (*pCompName*).

This is the same as for the **vis_ImageItem()** function but the measurements of appropriate member variables are in *logical picas* not *base units*.

A fixed image text item does not change its viewed size when the *VIEW* scale changes.

Additional Information

vis_ImageItem() ♦

vis_FixLineItem()

vis_FixLineItem(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a fixed line item *component* instance with the specified name (*pCompName*).

This is the same as for the **vis_LineItem()** function but the measurements of appropriate member variables are in *logical picas* not *base units*.

A fixed line item does not change its viewed size when the *VIEW* scale changes.

Additional Information

vis_LineItem() ♦

vis_FixTextItem()

vis_FixTextItem(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a fixed text item *component* instance with the specified name (*pCompName*).

This is the same as for the **vis_TextItem()** function but the measurements of appropriate member variables are in *logical picas* not *base units*.

A fixed text item does not change its viewed size when the *VIEW* scale changes.

Additional Information

vis_TextItem() ♦

vis_FontDlg()

vis_FontDlg(*pTSSeq* | ? *pFlags* | ? *pVIEWIDName* | ?) → *font-info* | ?

pTSSeq An integer, string, or sequence stack object.

pFlags An integer stack object.

pVIEWIDName An integer or string stack object.

font-info A mixed integer, string, and decimal sequence stack object.

Details

Gets font information (*font-info*) from the user via a dialog box.

pTSSeq is a TAC sequence containing the values to initialise the font dialog box as follows.

[*height*, *width*, *weight*, *italic*, *underline*, *strike*, *char-set*, *pitch-fam*, *face-name*, *colr*],

where

- 1 *height* is the font height (a decimal value in *base units*; can be a negative value),
- 2 *width* is the font width (a decimal value in *base units*; the width value is not used by the dialog box),
- 3 *weight* is the font weight (an integer value),
- 4 *italic* is the italic value of the font text (an integer value containing a logical boolean value),
- 5 *underline* is the underline value of the font text (an integer value containing a logical boolean value),
- 6 *strike* is the strike-through value of the font text (an integer value containing a logical boolean value),
- 7 *char-set* is the character set of the font (an integer value listed in [Character Sets](#)),
- 8 *pitch-fam* is the pitch and family of the font (an integer value listed in [Pitch and Family](#)),
- 9 *face-name* is the face name of the font (A string value),
- 10 *colr* is the text foreground colour (an integer value containing an RGB colour),

If a NULL stack object (?) is used instead of *pTSSeq*, then default values will be assumed. If any value of the sequence is a NULL stack object then the corresponding value will assume a default value. The sequence may contain only an initial segment of the values; the rest of the values are treated as being NULL stack objects.

pFlags is zero or more of the values listed in [Font Dialog Box](#). If a NULL stack object (?) is used instead of *pFlags*, then `!CF_EFFECTS` will be the default.

pVIEWIDName is the *crucial identifier* or *name identifier* of a previously created VIEW *component* instance to be the owner window of the font dialog box. If a NULL stack object (?) is used instead of *pVIEWIDName*, then the font dialog box will be owned by VIS.

font-info is a TAC sequence containing the values selected by the user, as described at *pTSSeq*, or an empty sequence if the user cancelled the dialog box. A NULL stack object (?) is returned if an error occurred. Note that *font-info* references the same sequence (not a new one) as the one specified at *pTSSeq* unless an empty sequence is returned.

For more information on *pTSSeq*, see the corresponding members of the [Text Style Attributes component](#).

Examples

The following illustrations show how the `vis_FontDlg()` function can be used.

```
(1) MyTextStyle.{@tsPutValues(vis_FontDlg(? ? ?)); @tsNewFont();};
(2) MyFontDet := vis_FontDlg(? ? ?); MyFaceName := MyFontDet%[9];
```

In example (1), some font details are obtained from the user via a font dialog box. The returned sequence is then set to the text style *component* instance named “MyTextStyle”. Notice that there is no need to check the returned sequence because if it is empty, then MyTextStyle will be unmodified; likewise if a NULL stack object is returned from `vis_FontDlg()`. However, this example assumes that the member variables of MyTextStyle match the desired font characteristics if the dialog box is cancelled or fails.

In example (2), only the face name of a font obtained from the user is desired. If the user cancels, then MyFaceName will contain a NULL stack object; if a NULL stack object is returned from `vis_FontDlg()`, then an error event will occur at `MyFontDet%[9]`.

Additional Information

[Font Dialog Box](#) ▪ [Character Sets](#) ▪ [Pitch and Family](#)

Other Information

[Standard Colours](#) ▪ [Common Colours](#) ▪ `vis_RGB()` ▪ `@tsGetValues()` ▪ `@tsPutValues()` ♦

vis_FormText()

vis_FormText (*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a formatted text item *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the formatted text item *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.10_Formatted Text Item Component](#) - [3.4.12_Formatted Text Item](#) ♦

vis_GContext()

vis_GContext (*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a graphics context *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the graphics context *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.11_Graphics Context Component](#) - [3.4.14_Graphics Context](#) ♦

vis_GetCNameList()

vis_GetCNameList() → *nid-seq*

nid-seq A sequence of string subsequences.

Details

Gets a sequence (*nid-seq*) of the names of all *component* instances in the *current* project along with their parent *component* instance names.

nid-seq is a TAC sequence of the form: [[*comp-name*, *parent-comp-name*], ...] of all the *component* instances owned by the *current* project, where *comp-name* is the *name identifier* of the *component* instance, and *parent-comp-name* is the *name identifier* of the parent *component* of *comp-name*; *comp-name* and *parent-comp-name* are string values.

Other Information

[vis_GetPNameList\(\)](#) ♦

vis_GetMenuTmpl()

vis_GetMenuTmpl(*pMenuID* *pResData*) → *res-data*

pMenuID An integer stack object.

pResData A memory stack object.

res-data A memory stack object.

Details

Gets the specified (*pMenuID*) menu resource data (*res-data*) from resource file data (*pResData*).

pMenuID is the menu identifier of the menu resource data contained in *pResData*.

pResData contains raw resource data as would exist in a *.res file.

res-data contains the menu resource data that was contained in *pResData*.

pResData can contain data of a mixture of resources including menu resources. This function extracts the menu resource identified by *pMenuID*, and puts that menu resource into a memory stack object (*res-data*). If the specified menu resource does not exist in *pResData*, then *res-data* will be empty.

The returned menu resource data is typically used with the **vis_NewMenuData()** function.

Examples

The following illustrations show how the **vis_GetMenuTmpl()** function can be used.

```
(1) MenuRes :=
    vis_GetMenuTmpl(100 read_file :!RF_NO_DATA_CHK: 'MyResFile.res');

    if {neq 0 size MenuRes;} then
        {MenuData := vis_NewMenuData();}
    else
        {pop;}
    endif;
```

In example (1), the file MyResFile.res may contain a menu resource with an ID of 100, perhaps with other resource data. The file is read, and the returned memory object is passed to the **vis_GetMenuTmpl()** function which extracts the specified menu resource into a new memory object (MenuRes). If the file did not contain the specified menu resource, then MenuRes will have zero size; if it did contain the specified resource, then that resource is passed to the **vis_NewMenuData()** function. Note that the `size` command leaves the memory object contained in MenuRes on the stack, allowing it to be used as an argument to the **vis_NewMenuData()** function; or the memory object is popped off the stack if it had zero size.

```
(2) MenuRes :=
    vis_GetMenuTmpl(100 read_file :!RF_NO_DATA_CHK: 'MyResFile.res');

    if ( pop swap size MenuRes != 0 ) then
        {MenuData := vis_NewMenuData(MenuRes);}
    endif;
```

In example (2), does the same job as for example (1) using a different method; the memory object left by the `size` command is popped within the condition of the 'if' statement.

Related Information**vis_NewMenuData()** ♦**vis_GetPicaPix()****vis_GetPicaPix()** → *pixels**pixels* An integer stack object.**Details**Gets the number of pixels (*pixels*) in a *logical pica* (DPI / 6).The value represented by *pixels* depends on the DPI (dots per inch) setting of the screen. ♦**vis_GetPNameList()****vis_GetPNameList()** → *nid-seq**nid-seq* A sequence of string subsequences.**Details**Gets a sequence (*nid-seq*) of the names of all project *component* instances.*nid-seq* is a TAC sequence of the form: `[ [comp-name, ""], ... ]` of all the *component* instances currently owned by VIS, where *comp-name* is the *name identifier* of the *component* instance as a string. Note that project *components* do not have a *component* parent since they are at the top of the *component* hierarchy, hence the empty string.**Other Information****vis_GetCNameList()** ♦**vis_GObject()****vis_GObject**(*pCompName* *pProc*)*pCompName* A string stack object.*pProc* A procedure stack object.**Details**Creates a graphics object *component* instance with the specified name (*pCompName*).*pCompName* is the unique *name identifier* of the graphics object *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.*pProc* is the *initialiser procedure* to initialise the *component* instance before it is created.The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.**Additional Information**[2.8_Graphics Objects](#) ▪ [3.4.8_Graphics Object](#) ♦

vis_ImageItem()

vis_ImageItem(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates an image item *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the image item *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Other Information

[1.3.9_Image Item Component](#) - [3.4.13_Image Item](#) ♦

vis_LineItem()

vis_LineItem(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a line item *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the line item *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.6_Line Item Component](#) - [3.4.9_Line Item](#) ♦

vis_LoadCursor()

vis_LoadCursor(*pFilePath*) → *crs-hndl*

pFilePath A string stack object.

crs-hndl An integer stack object.

Details

Loads cursor data from a file (*pFilePath*), and returns its cursor handle (*crs-hndl*).

pFilePath is the path of a file containing cursor data which must be in either cursor (*.cur) or animated cursor (*.ani) format.

crs-hndl is the handle (HCURSOR) of the loaded cursor, or 0 (zero) if the cursor data was not loaded. The cursor is not displayed.

The returned handle is typically used by the @dpSetCursor() function.

Example

The following illustration shows how the **vis_LoadCursor()** function can be used.

```
(1) PrevCurs := @dpSetCursor(vis_LoadCursor('MyCursor.cur'));
```

In example (1), the current default cursor is changed to the one specified in the file MyCursor.cur.

Other Information

@dpSetCursor() ♦

vis_MatObjDef()

vis_MatObjDef(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a matrix object definition *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the matrix object definition *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.17_Matrix Object Definition Component](#) ▪ [3.4.20_Matrix Object Definition](#) ♦

vis_MatObject()

vis_MatObject (*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a matrix object *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the matrix object *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.16_Matrix Object Component](#) - [3.4.19_Matrix Object](#) ♦

vis_MesObjDef()

vis_MesObjDef (*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a measurement object definition *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the measurement object definition *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.19_Measurement Object Definition Component](#) - [3.4.22_Measurement Object Definition](#) ♦

vis_MesObject()

vis_MesObject (*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a measurement object *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the measurement object *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.18_Measurement Object Component](#) - [3.4.22_Measurement Object Definition](#) ♦

vis_NewIdleJobData()

vis_NewIdleJobData () → *job-data*

job-data An ETAC data object.

Details

Creates a new (*job-data*) idle-time job ETAC data object.

job-data is an ETAC data object containing members for setting up and posting an idle-time job.

At least the `@itjJobFnt` member variable needs to be set with the idle-time job function before the idle-time job is posted.

The idle-time job is posted via the `vis_PostIdleJob()` function.

Additional Information

[3.5.6_Idle-time Job](#)

Related Information

[vis_PostIdleJob\(\)](#) ♦

vis_NewMenuData()

vis_NewMenuData (*pMenu*) → *menu-data* | ?

pMenu An integer, memory, or sequence stack object.

menu-data An ETAC data object.

Details

Creates a new menu ETAC data object (*menu-data*) from a menu (*pMenu*).

pMenu is one of three possibilities: (1) a menu or menu bar identifier (integer), (2) menu resource data (TAC memory object), (3) a sequence containing menu information (TAC sequence). These are described below.

menu-data is an ETAC data object containing members for manipulating the specified menu or menu bar, or a NULL stack object (?) is returned if the function fails.

(1) a menu bar identifier is an operating system menu handle (HMENU) obtained from the: `@vMenuID` variable, `@vRepMBar()` function, `@mGetMenuID()` function, `@mGetPos()` function, `@mRemoveMItem()` function, `<@mGetMItem(...) % [3]>` expression, `@MENU` command, `@DLGMENU` command, or `@POPUP` command.

(2) menu resource data is a TAC memory object obtained from the `vis_GetMenuTpl()` function.

(3) the sequence contains menu item information only for a menu bar as follows.

```
[(item-label, item-id, menu-id), ...],
```

where

- 1 *item-label* is the menu label (string),
- 2 *item-id* is the application-defined 16-bit value that identifies the menu item (integer).
- 3 *menu-id* is the handle (HMENU) to the submenu associated with the menu item, or 0 (zero) if the menu item is not associated with a submenu (integer).
- 4 ... and so on for further groups of elements like the first three.

Each group of three elements corresponds to a menu item in the menu bar.

Examples

The following illustrations show how the `vis_NewMenuData()` function can be used.

```
(1) MenuData := vis_NewMenuData(@VIS("MyVIEW").@vRepMBar(?));
(2) MenuData2 := vis_NewMenuData(MenuData1.@mGetMItem(1000 true) %[3]);
(3) MenuData := vis_NewMenuData(["Ne&w", 150, 0, "&Prev", 151, 0, "&Next", 152, 0]);
```

In example (1), the menu bar of the *VIEW* identified by *MyVIEW* is obtained to create a menu data object which is assigned to *MenuData*. The menu items can later be manipulated via the *MenuData* variable.

In example (2), the submenu of a menu item (identifier 1000) in an existing menu data object (*MenuData1*) is obtained and put into a new menu data object (*MenuData2*) for manipulation.

In example (3), a menu bar is created with three items (*New*, *Prev*, and *Next*) that function as simple buttons that can also be activated by the **ALT+W**, **ALT+P**, and **ALT+N** keys, respectively. The menu identifiers of the three items are 150, 151, and 152, respectively.

Additional Information

[3.5.4_Menu](#)

Other Information

`@vMenuID` ▪ `@vRepMBar()` ▪ `@mGetMenuID()` ▪ `@mGetPos()` ▪ `@mRemoveMItem()` ▪
`@mGetMItem()` ▪ `@MENU` ▪ `@DLGMENU` ▪ `@POPUP` ▪ `vis_GetMenuTpl()` ♦

vis_NewSBarData()

vis_NewSBarData (*pSBarHndl*) → *sbar-data*

pSBarHndl An integer stack object.

sbar-data An ETAC data object.

Details

Creates a new status bar ETAC data object (*sbar-data*) from a status bar handle (*pSBarHndl*).

pSBarHndl is a status bar handle.

sbar-data is an ETAC data object containing members for manipulating the specified status bar.

The status bar handle is obtained from the @NewSBar () function (see ETACProgLang(Official).pdf for details) to manipulate the status bar.

This function is mainly for use with dialog boxes containing a status bar. The status bar data object for a *VIEW* is created via the @vNewSBar () function.

Other Information

@vNewSBar () ♦

vis_Pen()

vis_Pen (*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a pen attributes *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the pen attributes *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.12_Pen Attributes Component](#) ▪ [3.4.15_Pen Attributes](#) ♦

vis_PostIdleJob()

vis_PostIdleJob (*pITJob*)

pITJob An ETAC data object.

Details

Posts an idle-time job (*pITJob*) for later processing during idle-time.

pITJob is the idle-time job ETAC data object containing the idle-time job. This value is obtained from the **vis_NewIdleJobData()** function.

Related Information

vis_NewIdleJobData() ♦

vis_Project()

vis_Project(*pCompName* | ? *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a project *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the project *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance. The name must not be “@VIS@”, because that is the name of the default project *component* instance. If a NULL stack object (?) is used instead of *pCompName*, then the *component* instance corresponding to the *current* project is made temporarily *active* during the execution of the procedure specified by *pProc*.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The created *component* instance is owned by VIS.

If the specified name (*pCompName*) already exists, then VIS will alter the name of the created *component* instance to make it unique.

Other Information

[1.3.2_Project Component](#) ▪ [3.4.5_Project](#) ♦

vis_PurgeIdleJobs()

vis_PurgeIdleJobs()

Details

Purges (deletes) all idle-time jobs from the idle-time job queue.

Idle-time jobs being processed will have already been deleted from the idle-time job queue.

Other Information

vis_NewIdleJobData() ♦

vis_ReadClipStr()

vis_ReadClipStr() → *string*

string A string stack object.

Details

Reads a string (*string*) from the clipboard.

string is a plain string that was on the operating system clipboard, or an empty string if there was no such string.

This function removes the string (if any) from the clipboard after reading it.

Other Information

[vis_WriteClipStr\(\)](#) ♦

vis_RGB()

vis_RGB(*pRed pGreen pBlue*) → *colr-ref*

pRed An integer stack object.

pGreen An integer stack object.

pBlue An integer stack object.

colr-ref An integer stack object.

Details

Creates a colour value (*colr-ref*) from red, green, and blue intensity values (*pRed pGreen pBlue*).

pRed is the intensity of the red colour component from 0 to 255.

pGreen is the intensity of the green colour component from 0 to 255.

pBlue is the intensity of the blue colour component from 0 to 255.

colr-ref is an integer (COLORREF) representing the specified colour.

A standard or common colour can be specified as a pre-processor name in place of this function.

Other Information

[Standard Colours](#) - [Common Colours](#) ♦

vis_SetCurProject()

vis_SetCurProject(*pProjIDName* | ?) → *proj-cid*

pProjIDName An integer or string stack object.

proj-cid An integer stack object.

Details

Sets the specified project (*pProjIDName*) as the *current* one, and returns the previous *current* project's *crucial identifier* (*proj-cid*).

pProjIDName is the *crucial identifier* or *name identifier* of a previously created project *component* instance, or the default project name, “@VIS@”. If a NULL stack object (?) is used instead of *pProjIDName*, then only the *CID* of the previous *current* project is returned.

proj-cid is the *CID* of the previous *current* project before the specified one becomes *current*.

Example

The following illustration shows how the **vis_SetCurProject()** function can be used.

```
(1) SavedProj := vis_SetCurProject("OtherProj"); ...;
    void vis_SetCurProject(SavedProj);
```

In example (1), the *current* project is changed to another project (OtherProj), saving the existing *current* project in SavedProj. Some work is done with the *components* of that other project, then the original *current* project is restored. ♦

vis_TextItem()

vis_TextItem(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a text item *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the text item *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Other Information

[1.3.8_Text Item Component](#) - [3.4.11_Text Item](#) ♦

vis_TextStyle()

vis_TextStyle(*pCompName* *pProc*)

pCompName A string stack object.

pProc A procedure stack object.

Details

Creates a text style attributes *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the text style attributes *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Additional Information

[1.3.15_Text Style Attributes Component](#) - [3.4.18_Text Style Attributes](#) ♦

vis_VIEW()**vis_VIEW**(*pCompName* | ? *pProc*)*pCompName* A string stack object.*pProc* A procedure stack object.**Details**

Creates a VIEW *component* instance with the specified name (*pCompName*).

pCompName is the unique *name identifier* of the VIEW *component* instance created, or an empty string to let VIS create a unique *name identifier* for the instance. If a NULL stack object (?) is used instead of *pCompName*, then the *component* instance corresponding to the *current VIEW* is made temporarily *active* during the execution of the procedure specified by *pProc*.

pProc is the *initialiser procedure* to initialise the *component* instance before it is created.

The parent *component* instance for which the created *component* instance is to be a child must be *active*. The created *component* instance is owned by the *current* project.

If the specified name (*pCompName*) already exists with the *current* project, then VIS will alter the name of the created *component* instance to make it unique for the *current* project.

Other Information

[1.3.4_VIEW Component](#) - [3.4.7_VIEW](#) ♦

vis_WaitForEvents()**vis_WaitForEvents**(*pArg* | ?) → *rtn-code**pArg* An integer or string stack object, or an ETAC data object.*rtn-code* An integer stack object.**Details**

Waits for and processes user and certain operating system events.

pArg is one of two possibilities: (1) the *crucial identifier* or *name identifier* of a previously created VIEW *component* instance, or (2) an ETAC data object containing desired member variable allocations. These are described below. If a NULL stack object (?) is used instead of *pArg* then the *current VIEW* is assumed for option (1).

rtn-code is one of the following values.

- 0 The value in the `wfeTimeout` member variable (see below at point (2)) has elapsed.
- 1 The **vis_EndWaitEvents()** function was called to end this function.
- 2 This function ended because the user closed the specified *VIEW*.
- 1 The event messages are already being processed for the specified *VIEW*.
- 2 The specified *VIEW* was pending deletion but has now been successfully deleted.
- 3 The specified *VIEW* was pending deletion but VIS made an unsuccessful attempt to delete it.

(1) Refers to the *VIEW* that processes the events. After this function begins, if the *VIEW* is deleted, either by the user or by the program, the function will end. However, if event processing is being handled by a different *VIEW* while this function is waiting for events, then this function

will end after event processing is terminated for that different *VIEW*. If this option is specified, then the default assumed values (except for `wfeVIEWIDName`) specified at point (2) would take effect.

(2) Refers to an ETAC data object, created by the caller, whose member variables are used as named parameters as follows (the default value of the member variables is a NULL stack object).

<code>wfeVIEWIDName</code>	Contains the <i>crucial identifier</i> or <i>name identifier</i> of a previously created <i>VIEW component</i> instance, as described at point (1). If this variable contains a NULL stack object (?), then the <i>current VIEW</i> is assumed.
<code>wfeTimeout</code>	Contains a timeout value in milliseconds (a positive integer). This function will end no later than the timeout value since the function was called (the function could end earlier). A timeout value of -1 (negative one) indicates an infinite timeout. In this case, the function will need to be ended by some other means. If this variable contains a NULL stack object (?), then -1 (infinite timeout) is assumed.
<code>wfeSleepTime</code>	Contains the sleep time in milliseconds (an integer). If the sleep time is a <u>positive</u> value, then this function will wait for arriving events for no longer than the specified value. However, if an event arrives earlier than the sleep time value, then this function will immediately process all queued events on the event queue. After the sleep time period has elapsed, and/or after all the queued events have been processed, this function will do idle-time processing if specified. If the sleep time is <u>zero</u> or a <u>negative</u> value, then this function will unconditionally sleep for the sleep time period (taken as the positive of the value) instead of waiting for arriving events. The function will then check for queued events after sleeping and process them (if any). After the sleep time period has elapsed and all the queued events have been processed, this function will do idle-time processing if specified. If this variable contains a NULL stack object (?), then 200 milliseconds is assumed for the sleep time value.
<code>wfeDisableVIEWS</code>	Contains a boolean value as follows. If the value is <code>true</code> , then the <i>VIEWs</i> of the <i>current drawing plane</i> are disabled if the <i>current drawing plane</i> is not the parent of the specified <i>VIEW</i> . If the value is <code>false</code> , then the close boxes of the <i>VIEWs</i> of the <i>current drawing plane</i> are disabled if the <i>current drawing plane</i> is not the parent of the specified <i>VIEW</i> . Any disabled <i>VIEWs</i> or close boxes are enabled just before this function ends. If this variable contains a NULL stack object (?), then <code>true</code> is assumed for the value.
<code>wfeDoIdle</code>	Contains a boolean value indicating whether this function performs idle-time processing. If the value is <code>true</code> , then idle-time processing is performed on queued idle-time jobs, otherwise no idle-time processing is performed (any queued idle-time jobs will remain queued until this function is called again with a value of <code>true</code>). If this variable contains a NULL stack object (?), then <code>true</code> is assumed for the value.

The ETAC data object referred to at point (2) is constructed as follows:

`data:{wfeVIEWIDName :- ...; wfeTimeout :- ...; etc}`. Only non-default parameters need be allocated. Note that the parameters, where specified, must be ALLOCATED (using the `<:->` operator) not ASSIGNED.

This function periodically inspects and processes queued operating system event messages, related to an application program, and performs idle-time job processing between the inspections,

even if there are no queued events to process. The events are processed by calling the relevant handler function set by the program, or are processed internally for certain system events (such as scrolling and painting).

This function is ended by: (a) the user closing the specified *VIEW* (subject to program approval), (b) the `vis_EndWaitEvents()` function, (c) the specified *VIEW* being deleted by the program, or (d) the elapse of the timeout period specified by the `wfeTimeout` member variable mentioned at point (2) above.

If this function is never called in an application program, then the application program will end without processing any events (ie: a pointless exercise for a graphics program).

Note that event processing can be passed on to other *VIEW*s within event handlers (via this function) from a *VIEW* that is currently processing events.

Examples

The following illustrations show how the `vis_WaitForEvents()` function can be used.

```
(1) void vis_WaitForEvents(?);  
(2) void vis_WaitForEvents("MyVIEW");  
(3) void vis_WaitForEvents(data:{wfeVIEWIDName :- "MyVIEW";  
                                wfeTimeout :- 0; wfeDoIdle :- false;});
```

In example (1), the *current VIEW* will process the events. The values specified at point (2) will be the default ones. It is not advisable to use a NULL stack object (?) for the *current VIEW* if there is more than one *VIEW* in the application program at the time that this function is called, unless it is certain that the *current VIEW* is the intended *VIEW*.

In example (2), the *VIEW* identified by "MyVIEW" will process the events. The values specified at point (2) will be the default ones.

In example (3), the *VIEW* identified by "MyVIEW" (determined by the value of `wfeVIEWIDName`) will process the events for up to 200 milliseconds (the default value for `wfeSleepTime`), then the `vis_WaitForEvents()` function will end (because `wfeTimeout` is 0) without performing any queued idle-time jobs (because `wfeDoIdle` is false). The rest of the values specified at point (2) will be the default ones.

Example (3) can be used from within an event handler function to temporarily process pending user events. After the handler function ends, the previous waiting `vis_WaitForEvents()` function will regain event processing. The `vis_WaitForEvents()` function can be called recursively (from within event handler functions) any number of times.

In the examples above, if the *current drawing plane* is not the parent of the specified *VIEW*, then the *VIEW*s of that *drawing plane* will be disabled because of the default value (true) of the `wfeDisableVIEWs` parameter.

Related Information

[`vis\_EndWaitEvents\(\)`](#)

Other Information

[3.5.6_Idle-time Job](#) ♦

vis_WriteClipStr()

vis_WriteClipStr(*pString*)

pString A string stack object.

Details

Writes a string (*pString*) to the clipboard.

pString is a plain string is to be written to the operating system clipboard.

This function clears the clipboard before writing the string.

Other Information

vis_ReadClipStr() ♦

vis_WriteGObjs()

vis_WriteGObjs(*pObjIDs* *pFilePath* | ?)

pObjIDs A mixed integer and string sequence stack object.

pFilePath A string stack object.

Details

Writes a sequence of graphics objects (*pObjIDs*) to a disk file (*pFilePath*) or the clipboard.

pObjIDs is a TAC sequence containing the *crucial identifiers* or *name identifiers* (can be mixed) of the graphics object *component* instances belonging to the *active drawing plane*.

pFilePath is the file path that the graphics objects indicated by *pObjIDs* are written to. Only the following file extension are supported: *.emf, *.bmp, *.jpg, *.gif, *.png, or *.tif. If a NULL stack object (?) is used instead of *pFilePath*, then the graphics objects are written to the clipboard as enhanced metafiles; the metafiles can then be pasted into other compatible applications. Note that the application program rendering the metafiles determines whether to apply smoothing or anti-aliasing, and whether or not to render patterned outlines and fills as solid outlines and fills.

The default size of the written graphics objects is proportional to the visible size on the *VIEW*. To get a higher resolution for bitmap images, the *VIEW* scale needs to be increased before this function is called, and conversely for a lower resolution. The width of the collective graphics objects in pixels is calculated by the following formula: $W = S \times \text{DPI} / 6 \times B$, where W is the pixel width of the graphics objects written to disk, S is the *VIEW* scale, DPI is the DPI (dots per inch) setting of the screen, and B is the total width of the graphics objects in *base units*. The height of the graphics objects is the appropriate for the width. The default DPI is 96. The formula is valid only for an uncalibrated *VIEW* scale and a non *elastic window*.

Other Information

vis_WriteClipStr() ♦

Bibliography

The Official ETAC Programming Language copyright © Victor Vella (2025).

MSDN for Visual Studio .NET 2003 copyright 1987-2002 Microsoft Corporation.

Glossary

A

active (refers to a *component* instance)

This is the first *component* instance of a referred *component* type from the top of the TAC dictionary stack.

B

base unit

This is the only distance unit for a *drawing plane* and all of its frames. One *base unit* is a decimal unit that corresponds to one *logical pica* (one-sixth of a logical inch on the screen) under default conditions. A *base unit* also corresponds to 12 logical font points. A *base unit* can represent any physical distance desired by a programmer in a drawing.

When a position is in *base units*, that position is relative to the *drawing plane* absolute origin, unless stated otherwise.

boundary check

This is a process automatically performed by VIS that checks whether the area of the *client rectangle* fits entirely within the area of the *window boundary*.

C

client rectangle

This is the client area portion of a *VIEW* mapped as an imaginary rectangle onto the *window frame* with the top-left position of the rectangle always being mapped to the top-left position of the *window frame*. The mapped rectangle is the *client rectangle*.

component

This is a specialised ETAC data object that is part of the VIS higher-level framework. It is recognised and maintained by VIS but created by the program.

co-VIEW

This is each *VIEW*, if there is more than one, that is associated with the same *drawing plane*.

creation context

This is the *component* instance, other than a parent *component*, that must be in the process of being created when another *component* instance is being created. This implies that the created *component* instance must be created during the execution of the *initialiser procedure* of the *creation context*.

crucial identifier

This is a unique VIS generated numerical identifier for a *component* instance. The identifier is unique throughout all of the *ETAC sessions* of a running RunETAC.exe or AppETAC.dll file.

current (applies to a project, *drawing plane*, and *VIEW*)

VIS maintains internal data structures containing the current project, *drawing plane*, and *VIEW*. These data structures correspond with the project, drawing plane, and *VIEW components*, respectively, and have information referencing other parts of the VIS internal framework. When the programmer accesses member functions of those *components*, the corresponding data structures need to be *current* for proper operation.

D
drawing frame

This is an imaginary rectangular frame on the *drawing plane* that, along with the *window boundary*, defines the limits of the presentation of a drawing in a *VIEW*.

drawing plane

This is a conceptual set of points in a rectangle measuring a little less than $2^{32} \times 2^{32}$ points on which graphics objects are specified. The *drawing plane* represents a Cartesian coordinate system (with positive Y values upward). Measurements in the *drawing plane* are in *base units* not points. The absolute origin (coordinate (0, 0)) is at the centre of the *drawing plane*.

E
elastic window

This is a feature of a created *VIEW* that keeps the same visible drawing in the *VIEW* regardless of the *VIEW*'s client area width (the drawing stretches or shrinks proportionally with a change in the *VIEW*'s width). With a non *elastic window*, the drawing remains the same size with a change in the *VIEW*'s size.

ETAC code

This is *ETAC script* or *TAC binary instructions*. A file containing *ETAC code* typically has an extension of `<etac>`, `<tac>`, `<ptac>`, or `<btac>`.

ETAC packed script

This is *ETAC text script* that has been pre-processed or expanded, and then compressed. A file containing *ETAC packed script* is a binary file, typically having an extension of `<ptac>`.

Note that the term “*ETAC packed script*” is used in the same sense as the word “code”, as in “*ETAC packed script code*”.

ETAC script

This is *ETAC text script* or *ETAC packed script*. A file containing *ETAC script* typically has an extension of `<etac>`, `<tac>`, or `<ptac>`.

Note that the term “*ETAC script*” is used in the same sense as the word “code”, as in “*ETAC script code*”.

ETAC session

This is the period devoted to the processing of *ETAC code* by the ETAC interpreter. New *ETAC sessions* can exist within a given *ETAC session* for different *ETAC code*. Therefore, a given *ETAC session* can produce a new *ETAC session* (relating to different *ETAC code* from the given *ETAC session*) so that when the new *ETAC session* ends, the given *ETAC session* resumes.

ETAC text script

This is ETAC program code that is in human readable and writable text form. A file containing *ETAC text script* typically has an extension of `<etac>`.

Note that the term “*ETAC text script*” is used in the same sense as the word “code”, as in “*ETAC text script code*”.

H

hit selection

This is the selection of a graphics object when the mouse cursor is clicked within the object's bounding box. The graphics object needs to be set to be visible to be selected. See also *micro-hit selection*.

hit testing

This is a software test that determines whether the mouse cursor has just entered or left a graphics object. The graphics object needs to be set to be visible for *hit testing*.

I

image dragging

This is scrolling the drawing in a *VIEW* by pressing and holding down the right mouse button and dragging the mouse first horizontally, then in the desired direction, with all other mouse buttons and all modifier keyboard keys (**SHIFT**, **CTRL**, **ALT**) up.

initialiser procedure

This is a TAC procedure called when a new *component* instance is created, typically to initialise the variable members of the *component* instance.

L

loader script

This is an automatically loaded *ETAC code* file typically used to set up the ETAC environment before the specified *ETAC code* files are executed. The file name for the default *loader script* is RunETAC.btac or RunETAC.etac. The default location of the *loader script* is in the System directory under the installed RunETAC directory.

If an ETAC program is using VIS, the *loader script* also executes visETAC.etac, and if that does not exist, it executes visETAC.btac (or visETAC-D.btac for the debug version) existing in the same directory as the *loader script*.

logical pica

This is a decimal unit proportional to one-sixth of a logical inch. Therefore, one (1.0) *logical pica* is one-sixth of a logical inch. A logical inch is equivalent to the length of the number of pixels specified in the DPI (dots per inch) setting of the screen. The default DPI is 96.

M

micro-hit selection

This is the selection of a graphics object when the mouse cursor is clicked on any one of the object's items, but not selected when the cursor is clicked outside all of those items. For line and curve items, *micro-hit selection* occurs when the mouse cursor is on or near the item's path. The graphics object needs to be set to be visible to be selected.

N

name identifier

A *name identifier* (NID or "name ID") is a unique programmer-defined or VIS generated textual identifier for a *component*. Each project *component* instance has its own unique system of *name identifiers* for the *components* under it. A program assesses a *component* either by its *CID* or *name identifier*.

R

reference frame

This is an imaginary rectangular frame that is used to define the *VIEW* scale (the scale of the drawing as presented in a *VIEW*). The *VIEW* scale is defined as the ratio of the *reference frame* width to the *window frame* width.

S

screen pica

This is a decimal unit proportional to 16 pixels. Therefore, one (1.0) *screen pica* is equivalent to the length of 16 pixels (96 DPI / 6) on the screen. The pixel length of a measurement in *screen picas* does not vary with the screen DPI setting.

supersampling

This is a technique that involves rendering the drawing into a larger bitmap then reducing the size of the bitmap, based on some kind of smoothing algorithm, to the proper size before transferring the bitmap onto the *VIEW*.

T

TAC binary instruction

This is a binary form of ETAC program code existing in a binary file which typically has an extension of `<btac>`.

V

VIEW

Visual Interactive ETAC Window — an operating system window created by an ETAC program, or its representation in VIS (Visual Interaction System). A *VIEW*'s client area is represented by the *client rectangle* in a *drawing plane*.

W

window boundary

This is an imaginary rectangular frame on the *drawing plane*, specified relative to the *drawing frame*, that always retains the *client rectangle* within it.

window frame

This is an imaginary rectangular frame on the *drawing plane* that represents either the maximum possible width of a *VIEW*'s client area on the screen (for a non *elastic window*), or the actual width of the client area at any one time (for an *elastic window*).

Z

zoom dragging

This is zooming in and out of the drawing in a *VIEW* by pressing and holding down the right mouse button and dragging the mouse upwards to zoom in, or downwards to zoom out, with all other mouse buttons and all modifier keyboard keys (**SHIFT**, **CTRL**, **ALT**) up.