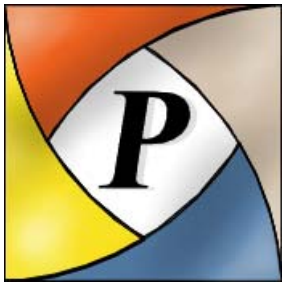
Institute for Energy Technology
OECD Halden Reactor Project



ProcSee

Graphical User Interface Management System

USER'S GUIDE 3.10.1

Institute for Energy Technology
OECD Halden Reactor Project

This document will be subjected to revisions in the future as the development of ProcSee continues. New versions will be issued at new releases of the ProcSee system.

The information in this document is subject to change without notice and should not be construed as a commitment by Institute for Energy Technology.

Institute for Energy Technology, OECD Halden Reactor Project, assumes no responsibility for any errors that may appear in this document.

Published by : Institute for Energy Technology, OECD Halden Reactor Project
Date : June 2016
Revision : 3.10.1

ProcSee Documentation

Table of Contents

Table of Contents v

1 Introduction 9
1.1 Key Points 9
1.2 Conventions 10
1.3 How to Use this Guide 10
1.4 Support 10

2 Related Documents 13

PART 1

3 The ProcSee Concept 17
3.1 General Overview 17
3.2 File Formats 26

4 Getting Started 29
4.1 Starting ProcSee Programs 29
4.2 Creating a Main Resource File 30
4.3 Defining Resources and Application Variables 30
4.4 Creating Classes and Libraries 31
4.5 Creating Pictures 31
4.6 Dynamic Graphics 32

5 The Symbol Table 33
5.1 Hierarchy, not Anarchy 33
5.2 Names and Scopes 35
5.3 Look up Mechanism 36
5.4 Anonymous Objects 39

PART 2

6 Introduction 43
6.1 What Makes GED a Specialized Graphics Editor? 43
6.2 Interface Development in the Graphics Editor 44

7 User Programs in ProcSee 45
7.1 Representation of User Programs 45
7.2 The Application 46
7.3 Editing a Running Application 47
7.4 Loading an Application from File 49
7.5 Creating a New Application 51
7.6 Saving an Application 51
7.7 Creating a Document of an Application 53

8 pTALK, the ProcSee Language 55
8.1 The Heritage 55
8.2 Comments 56
8.3 Identifiers 56
8.4 Expressions 57
8.5 Statements 61
8.6 Local Variables 65
8.7 Functions 66
8.8 Member Expressions 69
8.9 For Loops 69
8.10 The Switch Statement 70
8.11 Bitwise Manipulation 72
8.12 Dynamic Binding 74
8.13 Generating Code at Run-Time 75
8.14 Several Ways to do The Same Thing 77

9 Basic User Interface Components 79
9.1 The Building Blocks of a User Interface 79
9.2 Defining Resources 80
9.3 Defining Colours in GED 83
9.4 Defining Fonts in GED 89
9.5 Defining patterns in GED 95
9.6 Defining gradient patterns in GED 99
9.7 Defining Linestyles in GED 108
9.8 Defining Cursors in Ged 110
9.9 Resource Styles 117
9.10 Defining Windows in GED 120
9.11 Display 127
9.12 Layers 131
9.13 Libraries 133
9.14 Processes 137
9.15 Attributes, Functions and Dialogues 144
9.16 Constructors and Destructors 155

10 Undo 159

10.1 **Undo Settings** . . . 161

11 Active Shapes . . . 163
11.1 Why using active shapes? . . . 164

12 Pictures and Classes . . . 165
12.1 Composite User Interface Components . . . 165
12.2 The Coordinate System . . . 166
12.3 Working with Pictures . . . 168
12.4 Using Classes . . . 173
12.5 Attributes, Functions, and Dialogues . . . 177

13 The Drawing Editor . . . 179
13.1 Introduction . . . 179
13.2 The File Menu . . . 181
13.3 The Edit Menu . . . 185
13.4 The View Menu . . . 192
13.5 The Graphics Menu . . . 211
13.6 The Windows Menu . . . 220
13.7 The Options Menu . . . 220

14 Testing the User Interface . . . 231
14.1 Testing in the Drawing Editor . . . 231
14.2 Testing with User Programs . . . 233

15 Navigation in the main tree . . . 235
15.1 Using the navigation commands . . . 235
15.2 Edit the navigation commands . . . 236

16 Historic Trends . . . 237
16.1 Introduction . . . 237
16.2 The Trend Logger . . . 238
16.3 Trend Configuration Files . . . 244
16.4 Trend Diagram . . . 246
16.5 Advanced Features . . . 258

17 C O M . . . 265
17.1 What is COM? . . . 265
17.2 COM features . . . 266
17.3 COM components supported by ProcSee . . . 267
17.4 COM types . . . 267
17.5 Using COM from GED . . . 268
17.6 COM definitions in the symbol table . . . 289
17.7 Using COM in ProcSee . . . 293
17.8 Predefined COM Data Types in ProcSee . . . 298
17.9 COM Binary files . . . 299
17.10 Create your own COM components . . . 300

18 Drag and Drop . . . 305
18.1 Introduction . . . 305
18.2 Using drag and drop in ProcSee . . . 306
18.3 Drag source . . . 306
18.4 Drop target . . . 308
18.5 Simple example . . . 311
18.6 Advanced example . . . 312
18.7 Drag and drop on X-Windows . . . 316

PART 3

19 Application Code Basics . . . 319
19.1 A Small Example . . . 319
19.2 Compiling and Linking your Program . . . 321
19.3 Handshaking with the Run-Time Manager . . . 321
19.4 The Main Loop . . . 322
19.5 Summary . . . 325

20 Variables . . . 327
20.1 Variables are owned by Processes . . . 327
20.2 Creating a Variable . . . 327
20.3 Changing Variable Values . . . 329
20.4 Record Variables and Types . . . 331
20.5 Changing Variables from pTALK . . . 334

21 Commands and Messages . . . 335
21.1 Motivation . . . 335
21.2 Synchronous versus Asynchronous . . . 336
21.3 Sending Commands to RTM . . . 337
21.4 Sending Commands from the RTM . . . 338
21.5 Broadcasting Messages . . . 341
21.6 Remote Function Call versus Broadcasting . . . 347

22 Extended API features . . . 349
22.1 Communicating with several RTMs . . . 349
22.2 Integrating OSF/Motif . . . 351
22.3 The Concept . . . 351
22.4 Motif integration example . . . 353

APPENDIX A - MyApp.Tdoc . . . 359

APPENDIX B - Trend diagram with ruler . . . 363

APPENDIX C - motif integration example . . . 367

Index . . . 373

1 Introduction

ProcSee is a User Interface Management System (UIMS) for developing Graphical User Interfaces (GUIs). An interactive graphics editor can be used to test a user interface immediately in order to verify that dynamic behaviour works as intended. Moreover, GUI components can be created, modified, and deleted at run-time. The ProcSee system is designed to operate in a network environment. It can run on one workstation while the application programs that it services can be distributed over several workstations.

1.1 Key Points

Important features and characteristics of the ProcSee system include:

- ProcSee is the new name of Picasso-3, and is the third generation of the Picasso / ProcSee project. It is built on more than 20 years of experience in development of graphical user interfaces. Earlier versions of Picasso / ProcSee have been used in more than 40 industrial applications.
- ProcSee is being developed using C++ and object-oriented programming techniques.
- An Interactive Graphics Editor makes development and testing of user interfaces more efficient. This allows ProcSee to be used as a prototyping tool for creating user interfaces independently of actual applications.
- Separation of central algorithms from the graphical presentation makes maintenance less expensive.
- The design of user interfaces can be optimised using the object-oriented class concept offered by the system.

- The system is based on industry standards including X-Windows and the Unix operating system. Any programming language that supports C-callable routines can be integrated with ProcSee.
- It is an open system in the sense that it is designed to coexist with software components developed by other vendors.

1.2 Conventions

The following conventions are used in this guide:

- ProcSee and pTALK, the ProcSee user interface programming language are printed in 12 point Avant Garde.
- Courier bold is used for constants such as **.pctx**
- **$PROCSEE_DIR** refers to the directory where ProcSee resides.
- **$ARCH** refers to the (architecture) type of computer used.
- <*appName*> means that *appName* is a user specified name.

- An **I-beam** icon is used to highlight where more information about a specific topic can be found.

- An explanation mark icon highlights important information in the text.

1.3 How to Use this Guide

The features and capabilities of ProcSee are described in this User's Guide.

- Part 1 is a brief introduction to ProcSee and an overview of the concepts of the system.
- Part 2 is dedicated to the design of user interfaces.
- Part 3 describes the application programmer's development environment.

1.4 Support

The following e-mail addresses are available for ProcSee support

procsee-support@hrp.no
General questions concerning ProcSee should be directed to this address.

procsee-requests@hrp.no
Requests for new or changed functionality of ProcSee should be directed to this address.

procsee-bugs@hrp.no
Bugs and problems found in ProcSee should be directed to this address.

2 Related Documents

This chapter lists the related ProcSee user documentation.

ProcSee Documentation

The ProcSee Tutorial - a brief introduction to the ProcSee system. The tutorial contains detailed examples of how to build a complete application, including a small simulator program.

The ProcSee Reference Manual - a detailed reference to languages, programs, and functions available in the ProcSee system.

The ProcSee Utility Library - a C++ class library that can be used to implement non-trivial data structures in application programs.

Software Bus Documentation

The Software Bus Documentation Reference Manual - a detailed reference to the communication functions provided by the Software Bus.

The Software Bus User's Guide - a description of how the functionality of the Software Bus should be used.

PART 1

Concept and overview

3 The ProcSee Concept

This chapter gives a general overview of the ProcSee system. Interactive design and testing, reusability of components, integration and communication are the key words. The internal file format used by ProcSee is also described.

3.1 General Overview

UIMS

The ProcSee system is a flexible and powerful tool for developing and testing graphical user interfaces. Furthermore it is an environment for executing applications. It is the third generation of *User Interface Management Systems* (UIMS) developed as part of the OECD Halden Reactor Project since 1984.

application

In this User's Guide, the term *application* is used to describe all items that are involved in order to realise a user interface. This includes external programs like simulators and database servers as well as the representation of the user interface in ProcSee. In other words, it is the end-user's view of the complete system, regardless of the number of programs involved in realising it.

process

The term *process* is often used in conjunction with the term application. In ProcSee terminology, a process is a representation of an external program in an application. A *process database* is a collection of variables and user defined data types common to the external pro-

gram and the representation of it in ProcSee. A *default process* in an application is a representation of a virtual external program that is not physically running.

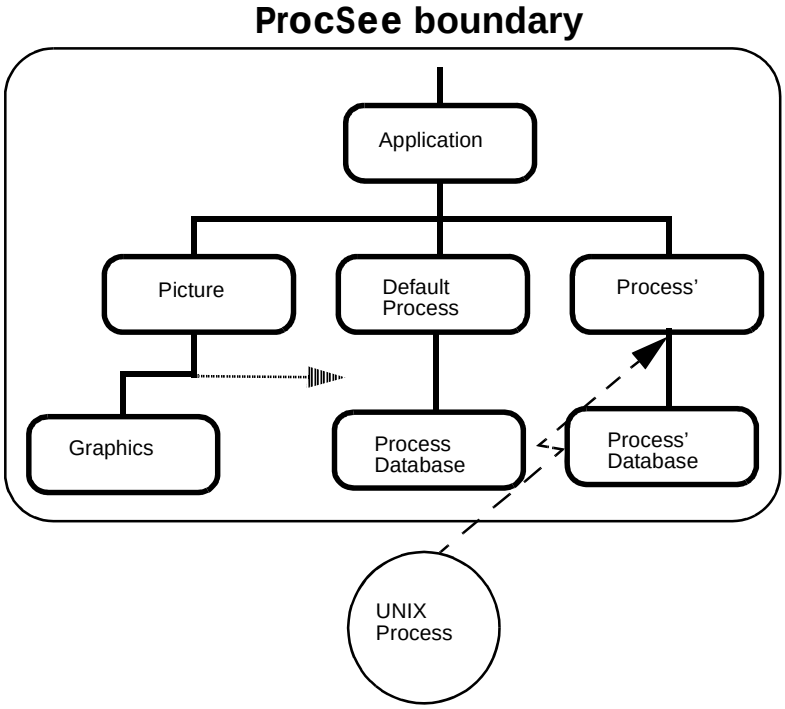


Figure 1

A ProcSee application is the end user's view of the complete system. The figure shows an application with one external UNIX process and one default process. The UNIX process represented in ProcSee is indicated by a '

benefits

By using the ProcSee system you should be able to:

- enhance the quality of graphical user interfaces in applications that are being developed.
- reduce the development and maintenance costs throughout your application's lifecycle.

Graphics Editor

To achieve the first goal, the ProcSee system offers an interactive design / test environment that makes development and testing more efficient. The *Graphics Editor* (GED) is used in this process. GED has two modes of operation; edit-mode is for drawing graphics and and the creation of dynamics and dialogues, and the test mode is for testing dynamics on-line. Switching between the two modes is done from a menu in GED.

More information about the Graphics Editor can be found in Part 2 of this User Guide.

In this way, switching between the design and testing of your interfaces can be done quickly and efficiently. Using this facility, ProcSee can be used as a powerful prototyping tool. Even though savings in this phase of your application's lifecycle can be considerable, the real potential for savings can be found in the maintenance phase. This is when your application is installed and is used for its intended purpose. In this phase, errors are usually found and cor-

rected, but when ProcSee is used, correcting them is not the main activity. In fact, most of your time is likely to be spent improving and enhancing the user interface functionality.

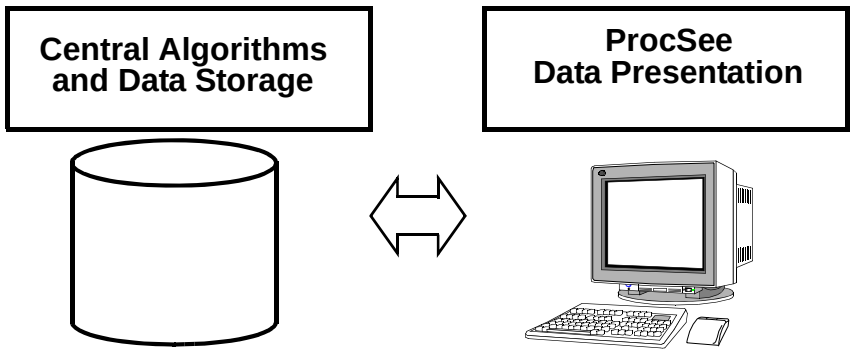
separation

Most changes in functionality involve presenting information that is already stored in a system in new or alternative ways. The best approach to reducing the maintenance costs is therefore to separate the user interface from the application code and information storage. A modification of the interface will then require minimal changes to the application's code.

A major requirement of the ProcSee system is to offer a complete separation of the user interface from the application code.

Figure 2

Most changes in applications are done in the presentation of data that is already stored in your application. The separation of central algorithms from the presentation part is a key issue in reducing



best of breed

The ProcSee system is based on industry standards. It runs under X-Windows on standard UNIX workstations and has been developed using the C++ programming language and object oriented programming techniques. Building software applications today, is much about choosing the right tool for the right job. This means that application developers must pick from the best of the breed and that ProcSee must be able to operate in a distributed computing environment. It must cooperate with other running tasks and support a client-server architecture.

Run Time Manager

The central part of the system is the *Run Time Manager* (RTM), which realises the user interface.

API

Communication is performed through a set of high-level interface routines that are linked with every program. These *Application Programmer's Interface* (API) routines give application programs access to all functionality in the RTM.

ProcSee classes

Building graphical user interfaces is a time consuming process. A major feature of the ProcSee system is that the user can develop reusable high-level user interface components. These components are building blocks that represent real world objects such as valves, tanks and pumps of different kinds. In ProcSee, terminology, we call the representation of these real world objects *classes*. When such

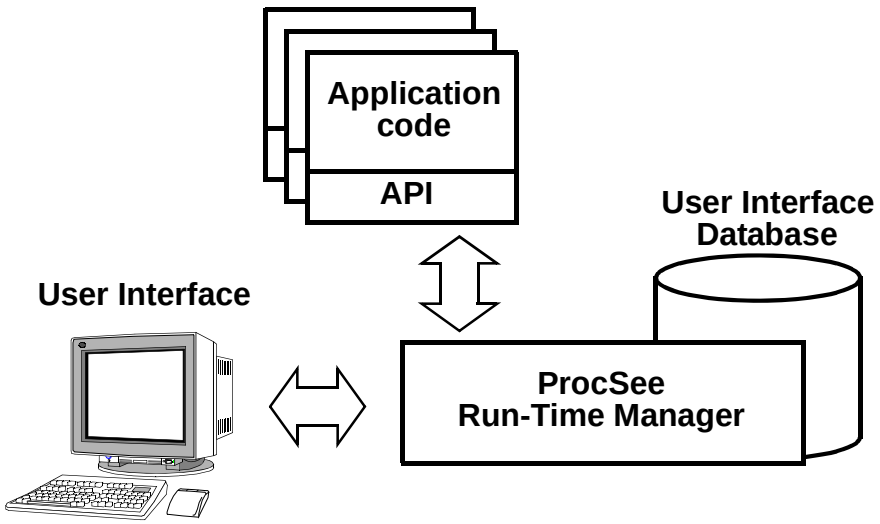


Figure 3 *The end-user interacts with the user interface described in the interface database and realised by the RTM. Application programs, communicating through the API library, can be distributed on several workstations.*

a real world object is used in a picture, we say that the class is *instantiated* in the picture. Instantiation of classes is as easy in ProcSee as the drawing of primitive shapes such as rectangles and circles.

The Graphics Editor is used to create the *look* (appearance) and *feel* (behaviour) of graphical objects. These can be primitive shapes or instances of high-level classes.

Interactive User Interface Design

iterative process

The design and implementation of graphical user interfaces is typically an iterative process. This means that you design and develop the interface, you test it out, and then you redesign it and test again etc. The design tool should therefore support the designer with methods that simplify this iterative process.

The ProcSee Graphics Editor offers general editing functionality similar to that which you would expect to find in a general purpose graphics editor, as well as more specialized functionality for creating dynamic graphics.

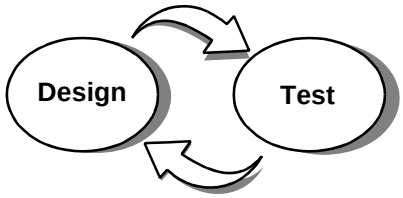
on-line compilation

User interface development is traditionally carried out in two steps. First, to use some off-line tools to specify the interface and somehow compile this specification, and then to run an on-line system that uses the compiled user interface specification. In ProcSee there is no distinction between off-line and on-line. All ProcSee tools operate on-line and therefore no off-line re-compilation and restarting of the system is necessary. The editing functionality is an integrated part of the ProcSee system, and hence editing operations are available at all times. This is utilised in the graphics editor by allowing the user interface designer to easily switch between the design and test modes. In the test mode, the system runs in its real

run-time environment and all parts of the system can be tested. If the actual application is available, it can be connected to the ProcSee system while the user interface is being designed and edited, enabling the designer to test the interface using real application data.

Figure 4

ProcSee offers a development environment where the user interface designer can easily switch between design and test modes, without re-compiling and restarting the system.



A user interface consists of more than static graphics. In order to be able to reflect changes in an application program, it must be possible to connect the interface dynamically to application program data. In ProcSee, the end user can interact with the system through dialogues, describing what actions are to be executed as a response to a user's input.

Dynamic Graphics

pTALK

A key feature of the graphics editor is the functionality for the creation of dynamic graphics, for connecting graphics to application data. *Dynamics* are created using ProcSee's user interface language, pTALK. pTALK is a C/C++ like language with additional constructs for graphics manipulation. The language offers a powerful and flexible way of creating dynamics. The language is compiled by the RTM at run-time, and hence new pTALK code can be added and tested/executed immediately during the design as well as during the execution of an interface.

For more information on pTALK, see chapter 8 *"pTALK, the ProcSee Language"* on page 55 and the pTALK man(ual) pages in the Reference Manual.

dialogues

Another important feature of the ProcSee system is the ability to create powerful and flexible *dialogues*. In ProcSee, a dialogue is defined as the interactive part of a graphic element, and is integrated with the graphic element itself. Dialogues can be defined at different levels in an interface. User interfaces developed in ProcSee consist of three levels: from top to bottom, the application level, the picture level, and the element level. This hierarchical structure allows the user interface designer to create local as well as more global dialogues.

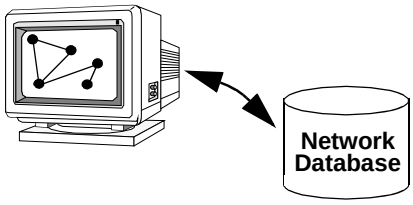
An example of the use of a dialogue attached to a valve object would be to toggle the state of the valve whenever the user clicks on it using the left mouse button. A dialogue consists of two parts: an *event description* and an *action* to be executed whenever the event occurs. Both the event and the action are described using the pTALK language. The action part of a dialogue is a pTALK statement, and can thus contain calls to standard functions as well as user defined functions. Since both the event and the action descriptions are user defined, highly specialized dialogues can be developed easily using the pTALK language.

run-time creation

As edit functionality is an integrated part of the ProcSee system, graphical objects can be manipulated at run-time. This can be used in applications where a dynamic data structure is represented graphically, and hence the structure of the graphics will change as the data structure changes. A typical example application is a network application where the number of nodes and the connectivity between nodes change over time. By using standard functions in pTALK, new graphic network nodes and connection lines could easily be created to reflect the changes in the network application. Another application area where run-time manipulation of graphics is needed, is in editor-like applications. The ProcSee graphics editor is itself an example of an application that uses the run-time editing functionality provided by the ProcSee Run-Time Manager.

Figure 5

Graphics can be created at run-time to reflect a dynamic data structure. A dynamic network described in a database can be presented graphically in ProcSee by creating new graphical elements at run-time, as the network changes.



Reuse of Interface Components

class concept

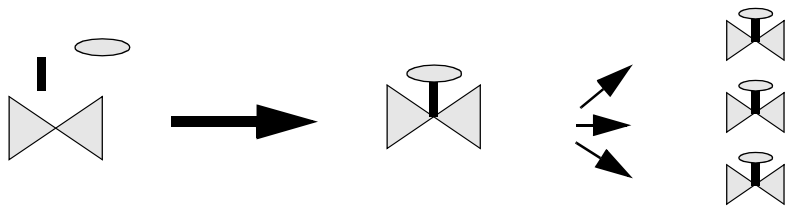
User interfaces are often built to depict real world components. The ProcSee *class* concept allows you to model these components in the interface. The components (valves, pumps, multiplexers, etc.) are often drawn as a combination of simple shapes. A valve can be represented by a drawing composed of a polygon, a line, and an ellipse.

The design and maintenance of user interfaces can be optimised by using the class concept offered in ProcSee, instead of drawing each component from scratch each time. The actual use of a class in a user

interface picture is called an *instance* of the class. Any changes made

Figure 6

Simple shapes are combined into high-level components, classes, that can be instantiated into the user interface.



to the design of a class will be reflected in all instances of it, thus reducing maintenance effort.

class behaviour

A real world component often has state information and a behaviour associated with it. A valve has a state: it is either open or closed. It also has a behaviour: it can be opened or closed. In the ProcSee system, the state information for a class is stored as *attributes* of the class. The component's behaviour is also part of the class, and is implemented as pTALK functions and dialogues. The fact that not only the appearance of the real world components, but also the state and behaviour associated with the components are part of the classes, is one of the object oriented features of the ProcSee system.

A ProcSee class is composed of simple shapes such as rectangles, circles and lines, and of attributes, functions, and dialogues. A class can also contain instances of other classes.

reuse of classes

A class can be designed independently of its surroundings, making it possible to reuse components in different user interfaces, and different parts of a user interface. A class can be viewed as a small user interface in itself, and this is taken advantage of in the ProcSee system; classes are constructed in the same way as ordinary user interface pictures, i.e. using the interactive environment provided by the ProcSee Graphics Editor.

Classes can be stored in *libraries* for reuse. Preferably, classes describing related components should be stored in the same library. To make libraries independent of context, other interface elements can be stored in them, such as colours, fonts, and functions.

Another use of libraries, is to easily change the appearance of interface components in an application. This can be done by switching between libraries containing classes with identical names. This can be useful if an application needs to conform to different standards for the look and feel of components.

An example could be a picture containing instances of two classes called 'Valve' and 'Pump'. These classes could be stored in a library named '**ISO_lib**'. Equivalent components could also be stored in a library called '**ANSI_lib**'. Switching between the two standards can then be done by simply switching libraries.

Integration and Communication

coexistence

ProcSee supports several mechanisms to enable coexistence with other software components in order to create a complete integrated system. Integration mechanisms fall into two categories, namely *integrating by communication*, and *integrating user interfaces*. The first category deals with message passing in a computer network, while the second deals with cooperation in a graphics environment.

communication

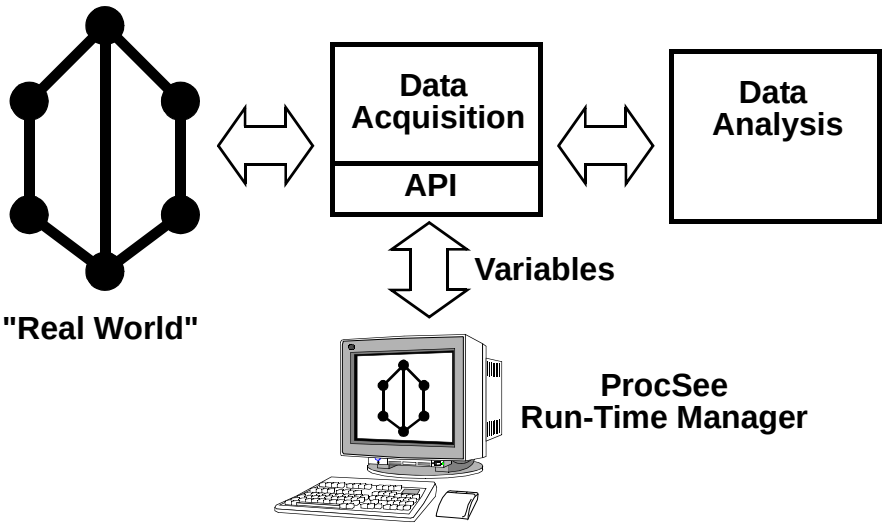
Since ProcSee encourages distributed systems, *communication* is a keyword in the integration process. ProcSee allows other programs to communicate with a user interface through an *Application Programmer's Interface*. The API is a routine library which puts the functionality of ProcSee within reach of any C-callable program.

data sharing

Data sharing is important in a distributed system. Data has a source, which is the program that owns the data and controls it. However, the data is typically needed in other programs. The data generated in an acquisition system must, for example, be available to a separate program for analysis, and to the user interface system for presentation.

Figure 7

This is an example of a data acquisition system which owns some data and a data analysis system that does calculations on the same data. Presentation of the data is performed by the RTM through API communication.



A program containing variables, can make them available to the user interface by calling selected API routines. Variables can be simple floating-point numbers, integers, or complex record structures. After variables have been declared to ProcSee, they can be referenced in any pTALK expression as part of the user interface logic.

variable update

The ProcSee API hides the error-prone process of packaging and interpreting bytes, which all too often becomes a major activity when programming a message based system. When a variable is declared to ProcSee using the appropriate API calls, the API routines take care of sending and interpreting the necessary messages. Whenever a variable is changed by a pTALK expression, the new

value must be sent from the Run-Time Manager to the application program. When a variable is changed by the program itself, the new value must be sent to the Run-Time Manager.

remote procedures

In addition to data sharing, there is often a need to issue commands to other programs. The goal is to make the programs look and feel like one integrated system, even though they are actually separate programs running on different computers. Calling *remote procedures*, i.e. routines in other programs, is one way of achieving this, and ProcSee supports this in both directions.

An application program can call a pTALK function, and actually execute any legal pTALK expression or statement. Since pTALK is an interpretative language[1], the source code can be built at run-time, and even be entered by the end user.

Routines in an application program can be called from within a pTALK expression. This requires that the application program declares the routines to ProcSee prior to a call, in much the same way as with variables. This means that the user interface can easily issue commands to a program, for example to tell a database system to save its data to file:

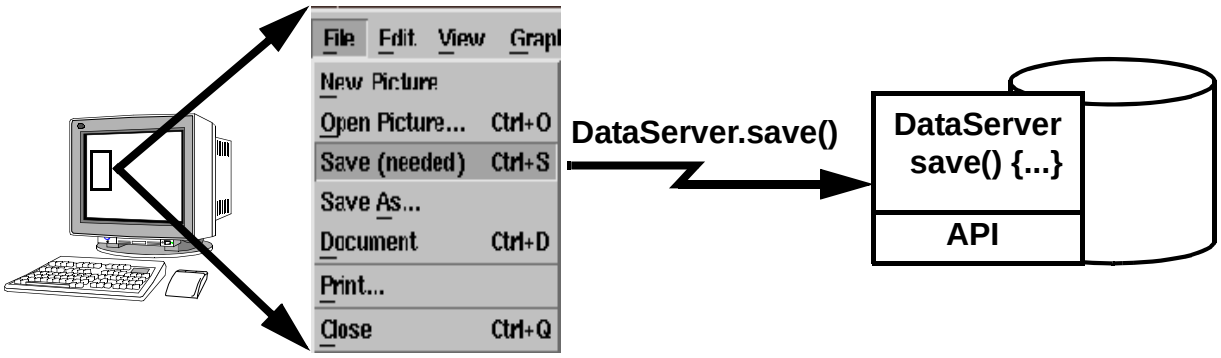


Figure 8 *A ProcSee database application where a dialogue in a menu calls a routine in an application process called 'DataServer'.*

An object-oriented notation is used in this call, as we, for example, tell the DataServer(the object) to save (the message).

filter program

Calling API routines is trivial as long as the application program's source code is available. When we want to integrate with another commercial product, a different strategy is required. Unfortunately there is no accepted standard for high-level inter-process communication within the UNIX world, and a program cannot possibly support the protocols of all other available systems. A solution is therefore to put a filter program between the two systems, which translates between the two protocols. The filter program will typically include both systems' different API libraries. This strategy can

1Although pTALK is interpretative, the source code is still compiled transparently within the Run-Time Manager to increase execution speed.

be used to integrate ProcSee with commercial products such as database management systems, spreadsheets, or *Geographical Information Systems* (GIS).

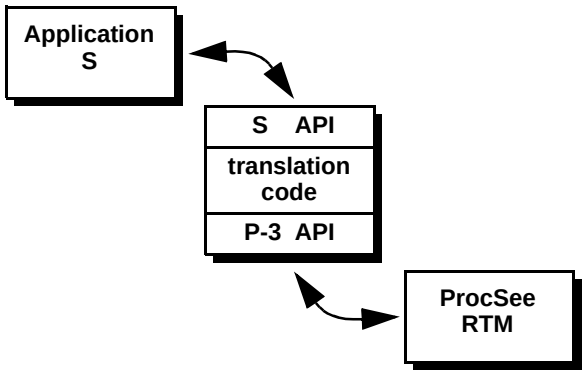


Figure 9

Communication is shown here between two applications with their own protocols. A filter process is used to translate between the protocols. In this example, ProcSee is comunicating with another

ProcSee communicates using TCP/IP, and supports both synchronous and asynchronous message passing. See Part 3 of this User's Guide for more information on communication between processes and the RTM.

3.2 File Formats

binary files

In most software systems there is a need to save and load the whole system or parts of it. This is done in ProcSee using binary files. Application are usually built step by step. That is, starting from scratch gradually building up to a complete application. The process of saving this application, or just parts of it, would require that only those parts that are actually changed during a work session are saved. Instead of saving an application as one big chunk, we have defined four levels of saving:

- global application data, such as resources, global attributes and functions. The extension for these files is **`.pctx`**
- libraries, containing information global to the library as well as graphical primitives and definitions of classes. The extension for library files i **.plib**
- data, known both to the RTM and external processes, is saved in process database files. Extension for data files is **.pdat**
- pictures together with relevant data are stored in files with the extension **.ppic**

Binary files are generated from the RTM by selecting the save choice in the file menu of the Graphics Editor.

Tdoc file format

Reading the binary file format is easy for machines, but not for human beings. Therefore, we also have a human-readable file format in ProcSee. The extension for these files is **.Tdoc**.

favourite text editor The documentation file format is generated from the RTM by selecting the documentation choice in the file menu in the Graphics Editor. **Tdoc** files can also be produced using your favourite text editor like *vi* or *emacs* if you learn the syntax.

More information about the process database file format (.**pdat**) can be found in Part 2 of the Reference Manual. The documentation language (**Tdoc**) format is also described here. The Graphics Editor is described in chapter 13 *"The Drawing Editor"* on page 179 of this Users's Guide.

pcc The human-readable Tdoc file format can be converted into a binary format using a conversion tool called ***pcc***, which is supplied as part of the ProcSee system.

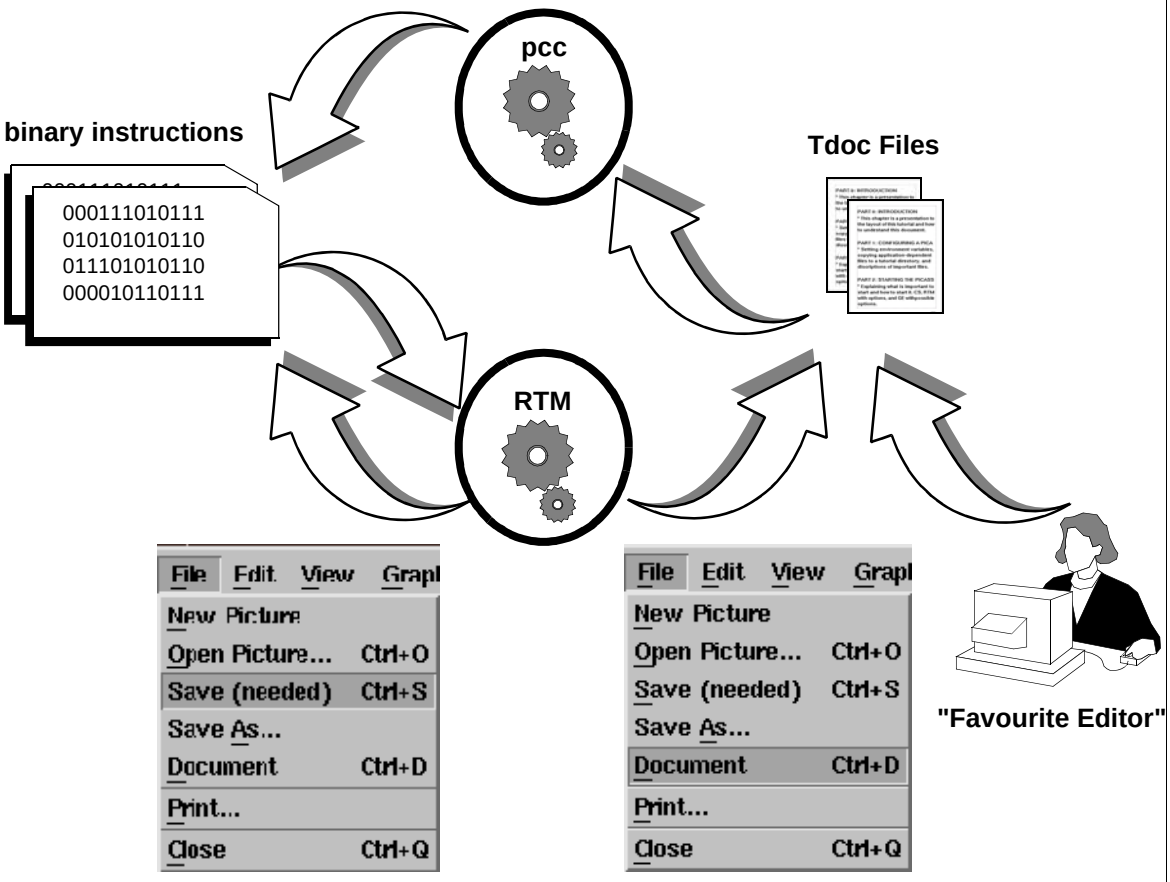


Figure 10 *A **Tdoc** file can be written using your favourite test editor. The file can then be converted into a binary format using the **pcc** program. This binary format can be read by the RTM. From the RTM, binary files can be produced by choosing the save item from GED's file menu. Tdoc files can be produced by choosing the document item.*

In ProcSee, as in real life, there is more than one road leading to Rome! **pcc** and the **Tdoc** format is one example of this. Besides us-

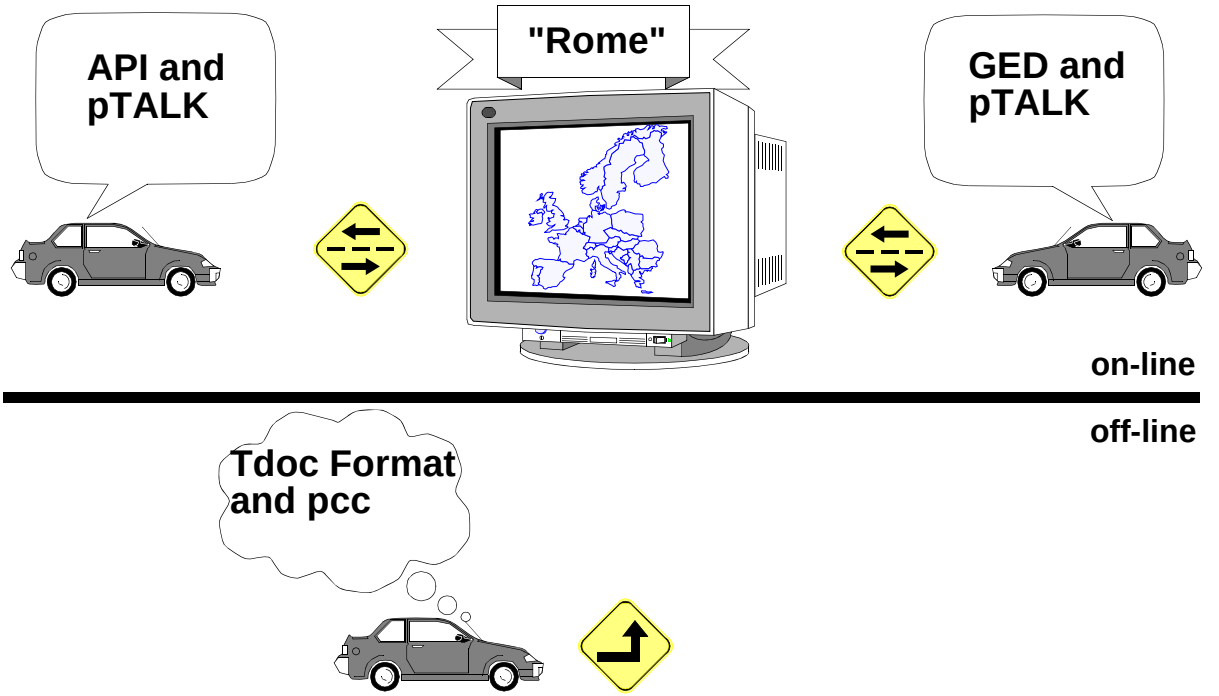


Figure 11 *The figure illustrates how graphical primitives and other objects, represented by the map on the screen, can be added to your application. The cars and traffic signs illustrate the different methods that can be used. Using **GED** and **API** is an interactive way of communicating. Creating objects using the **Tdoc** format and **pcc** is done off-line*

ing **pcc** and the **Tdoc** format, objects can be added to your application using the Graphics Editor, which is the most common method, and by inter-application communication via the Application Programmers Interface.

There is no *good* or *bad* road to follow. Which of the three different alternatives to choose depends on the task to be done.

4 Getting Started

This chapter describes how to start User Interface development using ProcSee. It is assumed that the ProcSee system has been successfully installed.

4.1 Starting ProcSee Programs

The following assumes that ProcSee has been properly installed and that the user environment has been set up as specified in the ProcSee Reference Manual, "Installing ProcSee".

To begin creating user interfaces using ProcSee, the following programs must be started:

control — A name server for the SWBus processes. The communication is not going through this process, but it is needed in order to establish a connection between two processes.

rtm — The ProcSee Run-time Manager; the heart of the ProcSee system that manages the actual creation and execution of user interfaces.

ged — The graphics editor; an OSF / Motif based editor used for interactive creation of user interfaces.

To start the programs type the following at the command line:

```
control &
rtm &
ged &
```

If the `control` program is not started manually, it is automatically started by the RTM.

When started, each ProcSee program prints its name and release number to the standard output, which is usually the terminal window in which it was started. It is important to include this information when contacting the ProcSee mailing groups.

4.2 Creating a Main Resource File

When the GED window appears, ensure that focus is set to the default application **NoName**. An explanation of how to set the focus of an application in GED is provided in section 7.3 *"Editing a Running Application"* on page 47.

The application **NoName** contains predefined colours, fonts, patterns, and linestyles, and can serve as a framework for creating new applications. Save and document the **NoName** application to file, using a name other than **NoName**. The application should not be saved in the ProcSee distribution directory but in a new, or other existing, directory.

A description of how to **Save As...** is provided in section 7.6 *"Saving an Application"* on page 49. A description of how to **Document** is provided in section 7.7 *"Creating a Document of an Application"* on page 49.

After performing the Save As... and the Document operations there should be two files in the working directory; **<appName>.pctx** and **<appName>.Tdoc**. A description of the file formats is given in section 3.2 *"File Formats"* on page 22.

4.3 Defining Resources and Application Variables

The following resources should be defined before actually attempting to create a user interface:

- colours (see section 9.3 *"Defining Colours in GED"*)
- fonts (see section 9.4 *"Defining Fonts in GED"*)
- windows (see section 9.5 *"Defining Windows in GED"*)

Colours, fonts , and windows can be defined interactively using the Graphics Editor (GED). They can also be specified in a document version of the main resource file (**<appName>.Tdoc**). To do this, load the Tdoc file into your favourite text editor and add new entries for colours, windows, and fonts. Existing entries can, of course, be modified or removed.

The components of a user interface are often connected to data in an application program (simulator, process control system etc.). In order to be able to connect components to application data without

having the application running and connected to ProcSee, the desired application variables can be declared in an **<appDB>.pdat** file. The pdat file is referred to in a process entry in the **<appName>.Tdoc** file. A detailed description of the pdat file format is given in the ProcSee Reference Manual, "The ProcSee database file format (.pdat)".

The syntax for the *window*, *font*, and *process* entries is described in detail in the ProcSee Reference Manual, "THE CONFIGURATION LANGUAGE".

A sample resource file is provided in "" on page 359.

When the desired modifications have been made to the <appName>.Tdoc file, compile the document file using the pcc program.

```
> pcc MyApp.Tdoc
```

pcc will produce a new MyApp.pctx file.

The new application file (MyApp.pctx) can now be loaded using GED (see section 7.4 *"Loading an Application from File"* on page 48).

New colours, fonts, and windows can be added and existing once modified using GED. A description on how to use these editors is provided in chapter 9 *"Basic User Interface Components"* on page 79.

4.4 Creating Classes and Libraries

In ProcSee classes are resuable interface components. They are typically stored in libraries (see section 9.7 *"Libraries"* on page 89). It is often useful to define a collection of classes before starting to create pictures (the actual user interface that will be presented to the user). A well defined set of classes will make the creation and maintenance of pictures easier and more efficient.

More information on classes can be found in section 10.4 *"Using Classes"* on page 100.

4.5 Creating Pictures

Pictures constitute the visible part of the user interface, and are displayed in windows.

A description of pictures is provided in section 10.3 *"Working with Pictures"* on page 97.

4.6 Dynamic Graphics

Attributes of user interface components can be connected to application data in order to reflect some state of an application.

Attributes can be assigned to application data in two different ways; regular assignment or dynamic assignment. In the following examples a rectangle's foregroundFillColour attribute is assigned the application variable MyStatus. More information on assignments and dynamic assigments / bindings can be found in chapter 8 *"pTALK, the ProcSee Language"* on page 55, *"Assignment Expressions"* on page 60, and section 8.12 *"Dynamic Binding"* on page 71.

Regular assignment

In a regular assignment the current value of the right operand is assigned to the left operand.

In this case the current value of MyStatus is assigned to foregroundFillColour:

```
foregroundFillColour = MyStatus;
```

Dynamic assignment

In a dynamic assignment the left operand is dynamically connected to the right operand. Whenever the value of the right operand changes, the value of the left operand will also change.

To specify dynamic connection, the right entry should be enclosed in backquotes. For example:

```
foregroundFillColour = `MyStatus`;
```

5 The Symbol Table

Any computer program that involves named objects or some sort of language needs some kind of retrieval mechanism. Such a mechanism is commonly called a symbol table. In this chapter we will explain how the Run-Time Manager's symbol table is structured.

5.1 Hierarchy, not Anarchy

If we examine a typical user interface, we find that it consists of hundreds, maybe even thousands, of objects. Objects can be pictures, windows, colours, fonts, lines, rectangles and so forth.

structuring

In order to be able to work with so many objects, we are forced to structure them in some way to avoid complete chaos. The best known way to do this is to arrange them as objects and sub-objects in a tree structure. We call this hierarchy a *symbol table*.

directories and files

Those who are familiar with operating systems like UNIX, MS-DOS, Windows NT, or VMS, will find that these systems arrange data in much the same way as ProcSee arranges user interface objects. In the same way that a directory or folder contains files, a picture contains graphical shapes. A directory may also contain other directories, like a picture may contain a group that in turn contains shapes.

A hierarchy can be visualized in several ways, as shown in Figure 12. The tree variant and the C style variant will be used extensively throughout this guide.

Figure 12

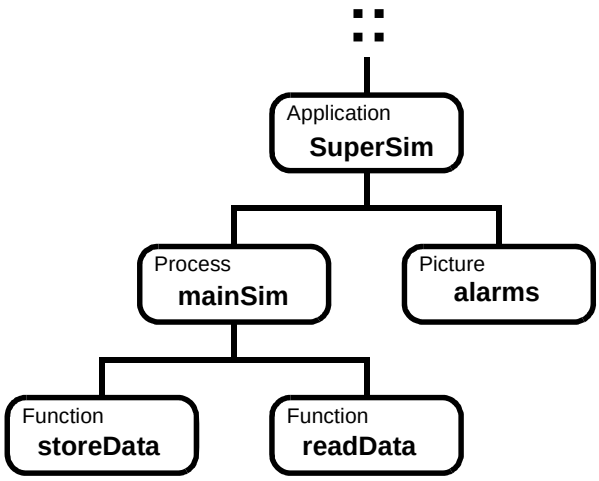
Although the three drawings seem quite different, they actually depict the same information, but in distinctly different ways.

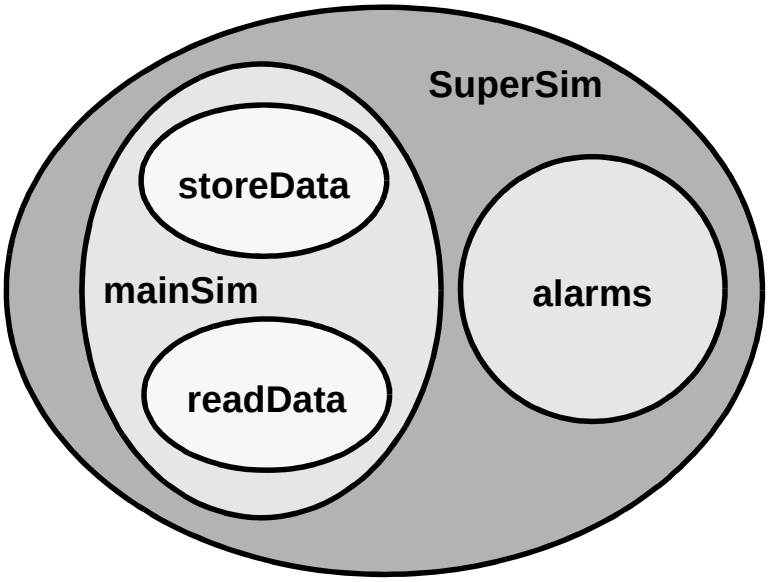
The first drawing shows a symbol table as a tree structure with internal nodes and leaf nodes. The internal node mainSim owns (is the parent of) the two leaf nodes storeData and readData. These are members (children) of mainSim.

The second drawing shows much the same, but here ownership is shown as containment. Objects drawn contained inside another object, are owned by that object. Manoeuvring inwards in this drawing is equal to manoeuvring downwards in the first drawing.

The third drawing shows ownership as nested begin-end clauses, here written in C style. Reading past a bracket (begin clause) means manoeuvring down the tree structure.Reading past a closing bracket (end clause) means to manoeuvre one step upwards in the tree structure.

In other words, all three drawings show ownership and containment, i.e. a hierarchy.






```
application SuperSim
{

   picture alarms
   {
   }

   process mainSim
   {
      function readData

      function storeData
   }

}
```

5.2 Names and Scopes

names

Any object in ProcSee may have a name. Objects of some categories must have names, while others may remain anonymous.

A legal name consists of maximum 31 characters, a character being either an underscore, digit, or an upper- or lower case letter. A name cannot start with a digit. Exceptions to these rules are discussed in chapter 8 "pTALK, the ProcSee Language" on page 55, but should be avoided whenever possible.

scopes

A collection of all objects sharing the same parent constitute what we call a *scope*. Another way of viewing this is to say that the parent itself *is* the scope of its children. A scope can also be understood as a *position* in a hierarchy. The term *scope* will be used extensively throughout this guide.

root scope

The topmost scope is called the *root scope* or the *global scope*. In formal notation we give this scope the special name **::**.

uniqueness

In order to identify an object by name, no other object within the same scope may have the same name. The name of any object must be unique to the scope in which it appears. However, any number of objects in a scope can be anonymous.

Fullnames

If the function object **storeData** is a member of the object **mainSim**, then **storeData** *is in the scope of* **mainSim**. We can express this formally as **mainSim.storeData**. The name of an object in conjunction with the name of all its ancestors, is called its *fullname*. The fullname of **storeData**, in the example in Figure 12, is **::SuperSim.mainSim.storeData**.

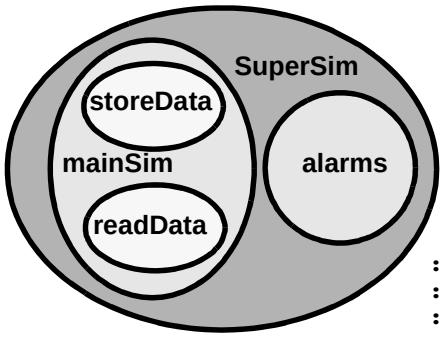
As stated above, the rule of name uniqueness within a scope does not prevent us from giving two objects the same name as long as they belong in different scopes. Since two objects can have the name **storeData**, the name of the object is not sufficient to uniquely identify it. However, an object's fullname does uniquely identify it, as two objects will never have the same fullname.

This leads us to a fourth way of visualizing a hierarchy in addition to those in Figure 12. This is shown in Figure 13.

Figure 13

A hierarchy of objects can also be viewed as a list of fullnames.

The fullnames of all objects in Figure 12 is shown.




```
::SuperSim
::SuperSim.mainSim
::SuperSim.mainSim.storeData
::SuperSim.mainSim.readData
::SuperSim.alarms
```

5.3 Look up Mechanism

In the previous sections we have established that

1) An object's *name* is not always enough to uniquely identify the object.
2) An object's *fullname* always identifies the object.

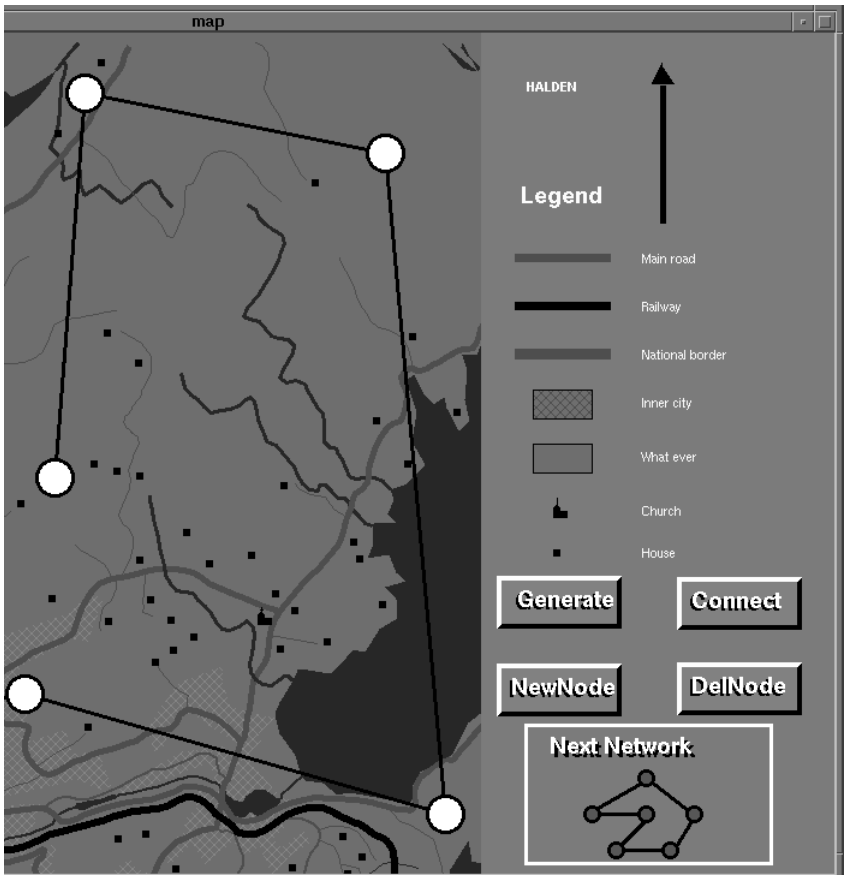
We could, of course, always use the object's fullname to identify it, but since the fullname will often be long, this would be cumbersome. Therefore, we need a look up mechanism.

Figure 14

*This figure shows an application **commNet** with an instance of a simple network class named **net1** displayed in a picture named **map**.*

The white circles indicate nodes, and the black lines between them indicates edges, or connections, in the network.

When the connect button is clicked on, a connectivity analysis of the network is performed.



The look up mechanism can be explained by an example. Let us examine the following features of the application shown in Figure 14:

- The name of the application is ***commNet***.
- The picture is named ***map***.
- An instance of a general network class is named ***net1***.
- A function in this network class called ***connAnalyse()*** carries out a connectivity analysis of the network.
- A dialogue in the picture, which when triggered,calls a function named ***connAnalyse()***.

A simplified view of the symbol table of this application, leaving out all irrelevant information, would look like this:

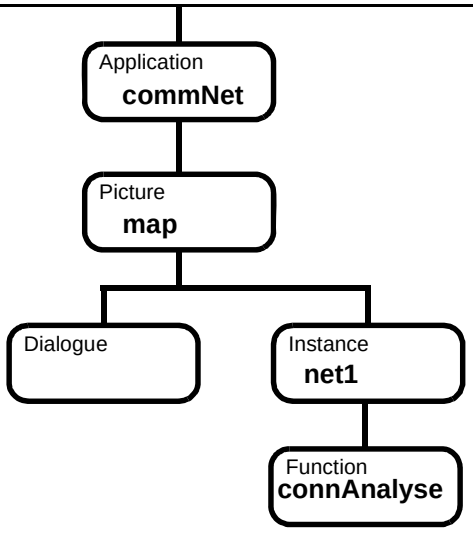


Figure 15

The symbol table hierarchy of the look up mechanism example, viewed as a tree structure.

The search for ***`net1.connAnalyse()`*** *is started within the scope of the dialogue*

When referring to the **`connAnalyse()`** function in the Instance **net1** of picture **`map`**, the action part of the dialogue could use the full name of the function. This would look like:

- **`::comNet.map.net1.connAnalyse();`**

However, knowing the scope of the dialogue, a much neater way of referring to the function is:

- **`net1.connAnalyse();`**

Starting out from the scope of the dialogue, the look up algorithm goes through the following steps:

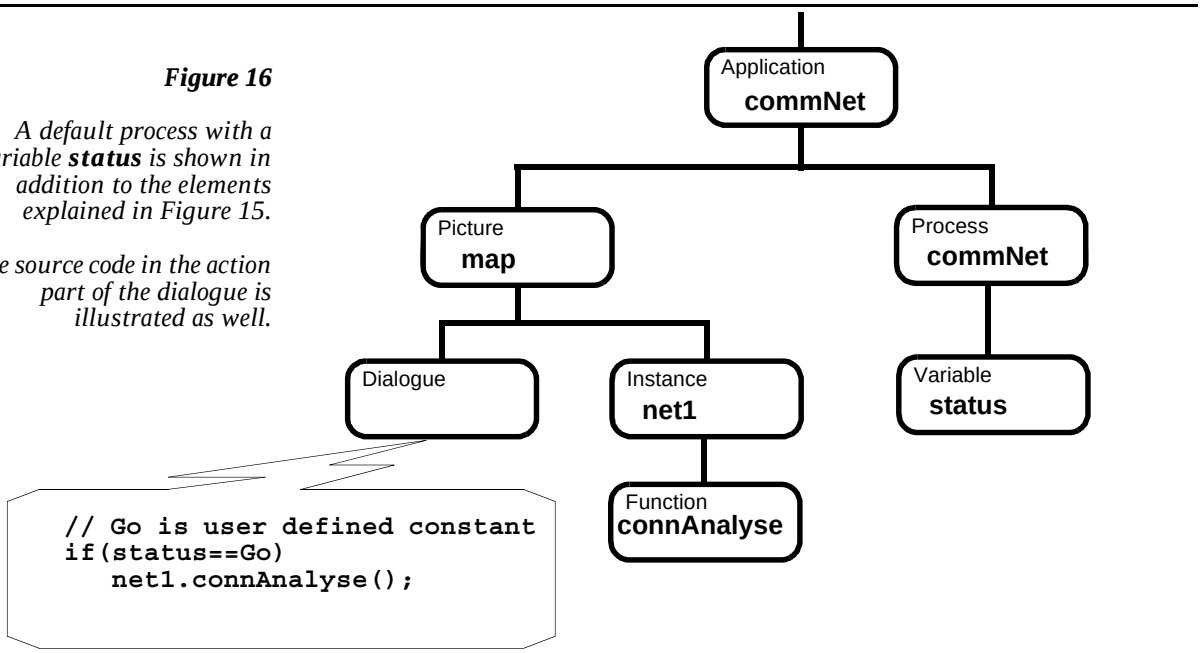
1) Look for **`net1`** in the scope of the dialogue. **`net1`** is not found among the members or children here.
2) Move one step up. The scope of **`map`** is searched and **`net1`** is found.
3) Within this scope, start searching for a function called **`connAnalyse()`**. It is found as a direct successor of **`net1`**.

Process variables and libraries are treated in a special way by the look up mechanism. If an object is referred to and not found when the top scope is reached, then processes and libraries in the context are also searched.

Figure 16

*A default process with a variable **status** is shown in addition to the elements explained in Figure 15.*

The source code in the action part of the dialogue is illustrated as well.



In the example in Figure 16, the action part of the dialogue is connected to a process variable named **status**. When this variable is looked up, it is not found in the scope of the dialogue. The search continues up to the top scope without success. The process scope is then searched and the variable is found.

If, in the top scope, all processes and variables are searched for an object without success, the look up mechanism fails. Note that within a scope, only children directly connected to a parent are searched. If two processes have a variable with the same name, the look up mechanism will always find the first of them. In such cases,a full-name must always be used when referring to the objects.

5.4 Anonymous Objects

We have already explained that some objects may be anonymous, meaning that they do not have names. In order to identify these objects despite this, a mechanism of integer identifiers is used.

32 bit identifiers

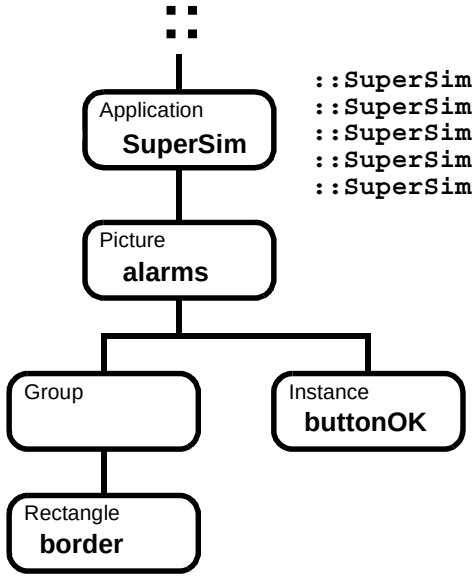
All anonymous objects have a 32 bit integer identifier. Like names, these identifiers are unique within scopes. Although we do not usually refer to an object by its integer identifier, they will frequently appear as part of an anonymous object's fullname. An example is shown in Figure 17.

Figure 17

The picture alarms contains two objects. One of them is anonymous, i.e. it has no name. The anonymous group in turn contains one object named 'border'.

The fullnames of all objects are shown in the listing.

The name component for the anonymous object consists of its 32 bit identifier, shown as a hexadecimal number.




```
::SuperSim
::SuperSim.alarms
::SuperSim.alarms.buttonOK
::SuperSim.alarms.0x0e00004f
::SuperSim.alarms.0x0e00004f.border
```

PART 2

Designing User Interfaces

6 Introduction

In the ProcSee system, all interface components can be defined using the Graphics Editor (GED). In addition to common editing functionality, GED also offers more specialized editing functionality for connecting interface components to dynamic data using the pTALK language.

6.1 What Makes GED a Specialized Graphics Editor?

dynamic graphics Many graphics editors exist in the market place, but a common denominator for many of these is that they can only be used to produce static pictures. In a ProcSee user interface, highly dynamic pictures are needed, so a specialized graphics editor is necessary to support the creation of dynamic graphics. Dynamic graphics are used to encompass connection with user program data as well as to specify the behaviour of an interface through dialogues.

resource editor GED is not only used to create graphic pictures, but is also used to specify other parts of the user interface such as class libraries, attributes, functions, and dialogues.

easy testing A graphical user interface developed using GED is usually highly dynamic, and it is therefore important that it can be tested while it is under development. GED allows the user interface designer to switch easily between development and test modes. No restart and compile procedures are necessary for changes to the interface to take effect.

6.2 Interface Development in the Graphics Editor

The following can be defined from within the editor using the current version of GED :

- Applications with attributes, functions, and dialogues
- Pictures, picture attributes, functions, and dialogues
- Class libraries, class attributes, functions, and dialogues
- Shape dialogues, shape attribute values can also be modified.
- Colours

In the current version of ProcSee there are some user interface resources that cannot be defined using the editor. These resources must be edited using a document version of an application's main resource file (the <*application name*>.Tdoc file) using a text editor such as emacs or vi. The resources that cannot be edited using GED are:

- Patterns, fonts, linestyles, and cursors
- Record types
- Trend Log and Trend variable Configuration File

Recommended Sequence of User Interface Development

When starting the development of a new user interface, it is important to first define the basic resources needed to build the interface. By resources we mean, colours, fonts, patterns, linestyles, cursors, and windows. All of these can, of course, be modified later, but changing them late in an interface development cycle tends to increase the amount of work that is necessary.

Below is a list of user interface development activities, listed in the order in which they should normally be carried out:

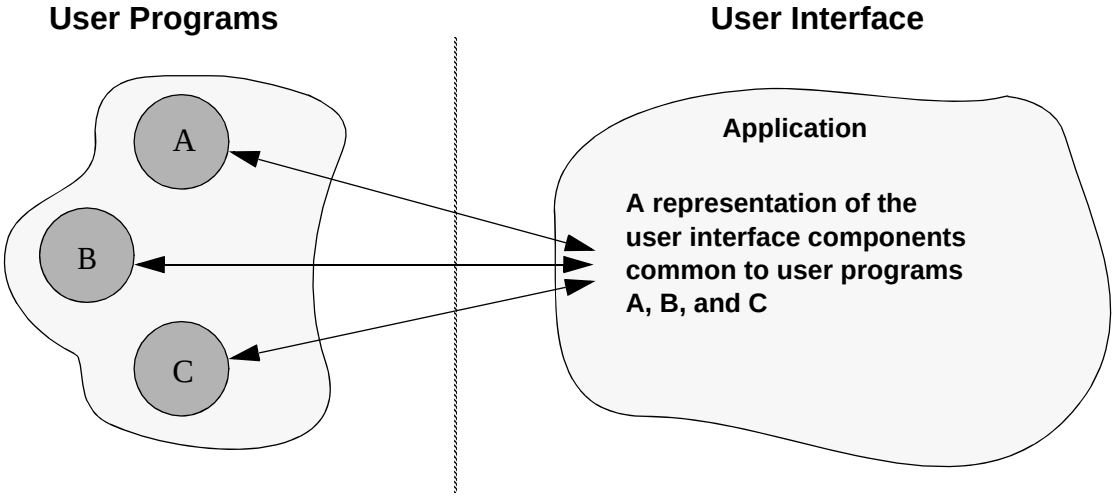
- Confer with the application program developer who is responsible for the user program, and try to get as good an idea about what program information needs to be presented in the user interface.
- If possible, create a process database, containing the user program variables needed in the interface.
- Define the interface resources. i.e. the colours, patterns, fonts, linestyles, cursors, and windows that will be used.
- Determine which class libraries are needed. If necessary, classes can be added to the libraries as the development effort progresses.
- Establish which global attributes and functions are needed.
- Start to create the graphical part of the interface by drawing pictures.

7 User Programs in ProcSee

The ProcSee Run-time Manager (RTM) contains all of the parts of a user interface for one specific set of user programs. An application is a representation of the user programs including all parts of the graphical user interface.

7.1 Representation of User Programs

The interface of an application consists of many parts. There are pictures and resources that describe the look of shapes in pictures, and variables that are used to communicate with the user interface. There is also a "computing" part of an application; the *user programs*. User interface components are contained in an *application*, enabling the user to group together all user interface components for a specific set of user programs.



7.2 The Application

In ProcSee terminology, the word *application* is used to describe a container in the Run-Time Manager that holds a complete representation of a graphical user interface for a set of user programs. The different parts of the user interface are:

- Colours, patterns, fonts, linestyles, cursors, and windows
- Processes (variables and remote functions)
- Record types, attributes, functions, and dialogues
- Libraries
- Pictures

Each part will be described in turn in the chapters that follow.

graphics editor - GED Applications can be edited by starting the graphics editor *GED*. GED will start with an empty application that may be edited. Applications that are already running can also be opened in GED, as can previously created applications that have been stored in files.

Several applications can be loaded into GED simultaneously. You can choose which application to work on by :

Using the **Focus Window** or the **Focus** button, which is accessible from GED's main window.

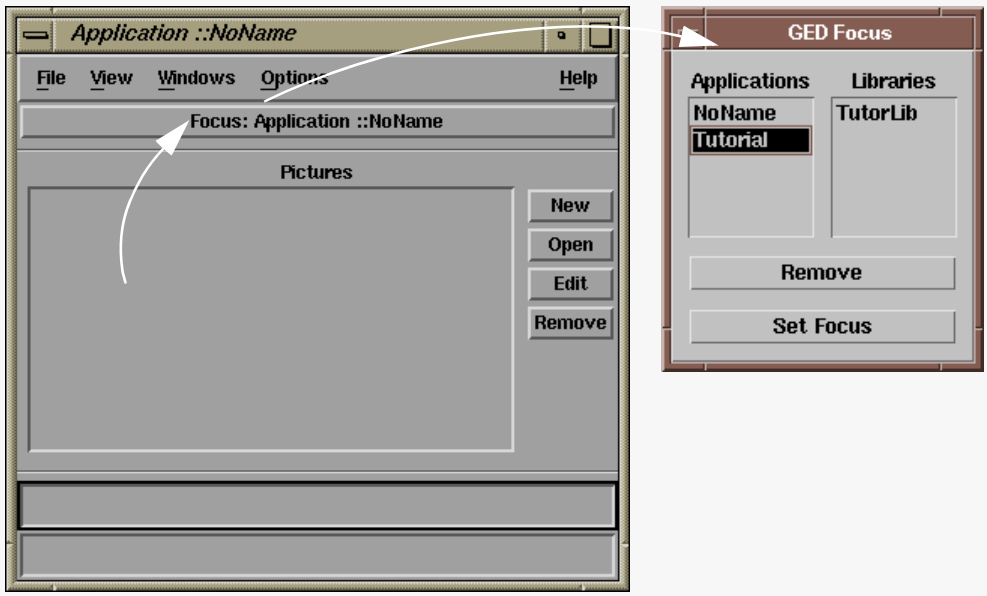
Selecting an application in the tree underneath the application tree-node.

7.3 Editing a Running Application

To work on an already running application, click on the Focus button in the GED main window. A new window entitled GED Focus will be displayed. This window contains lists of loaded applications and libraries. Select an application by clicking on the desired application name. To set the focus to the selected application click on the Set Focus button. The main GED window will then display a list of the pictures defined for the selected application

Figure 18

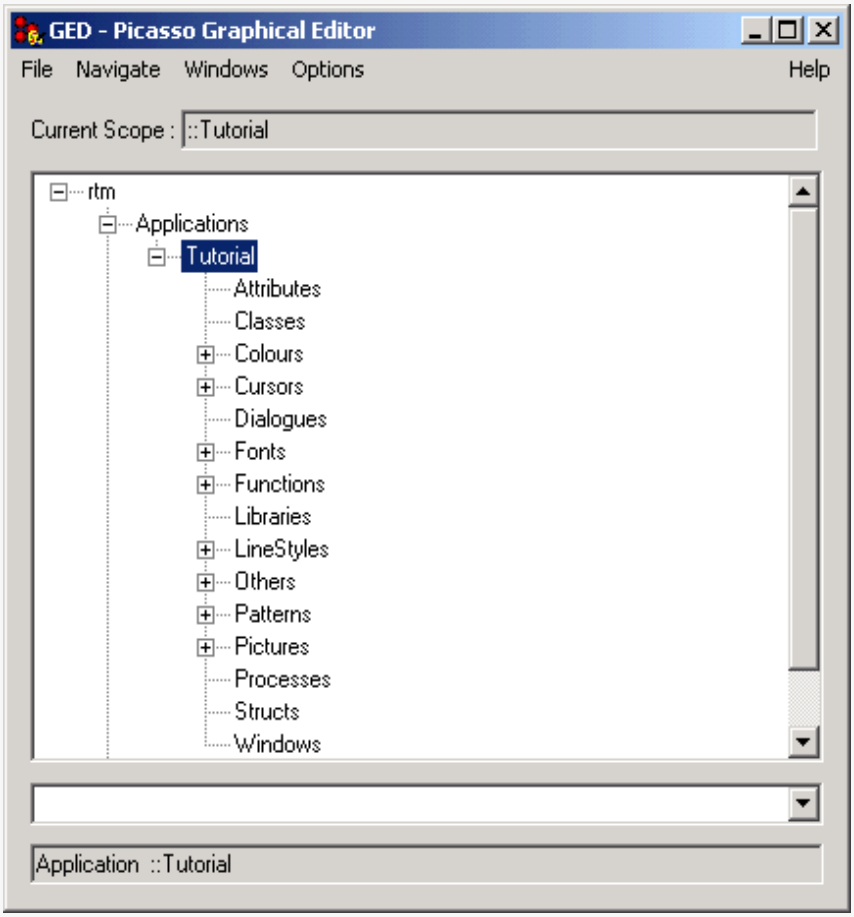
The figure illustrates how a new application already running in RTM can be selected for editing.



To work on an already running application, either double-click on the **`Applications`** node or click on the '+' symbol in the tree. A list of loaded applications will be displayed. Select an application by double-clicking on the desired application name or the '+' symbol. The resources, pictures, libraries etc. defined in this application will be displayed. The tree is organized in a hierarchy and is more or less a mirror of the symbol-table used in the Run-Time Manager.

Figure 19

The figure illustrates how a new application already running in RTM can be selected for editing.

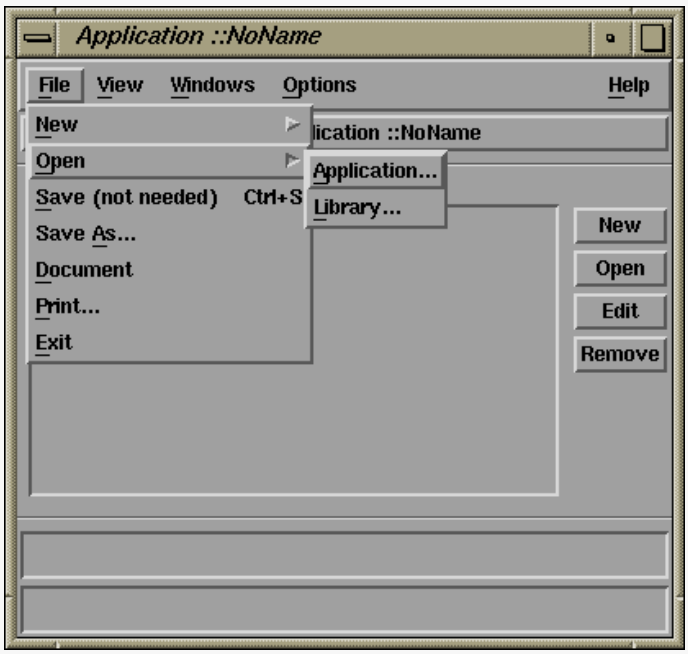


7.4 Loading an Application from File

To load an existing application from file, select the File menu from GED's main window. Then, from the **File** menu, select **Open**, and from the Open menu, select **Application**, as illustrated in Figure 20.

Figure 20

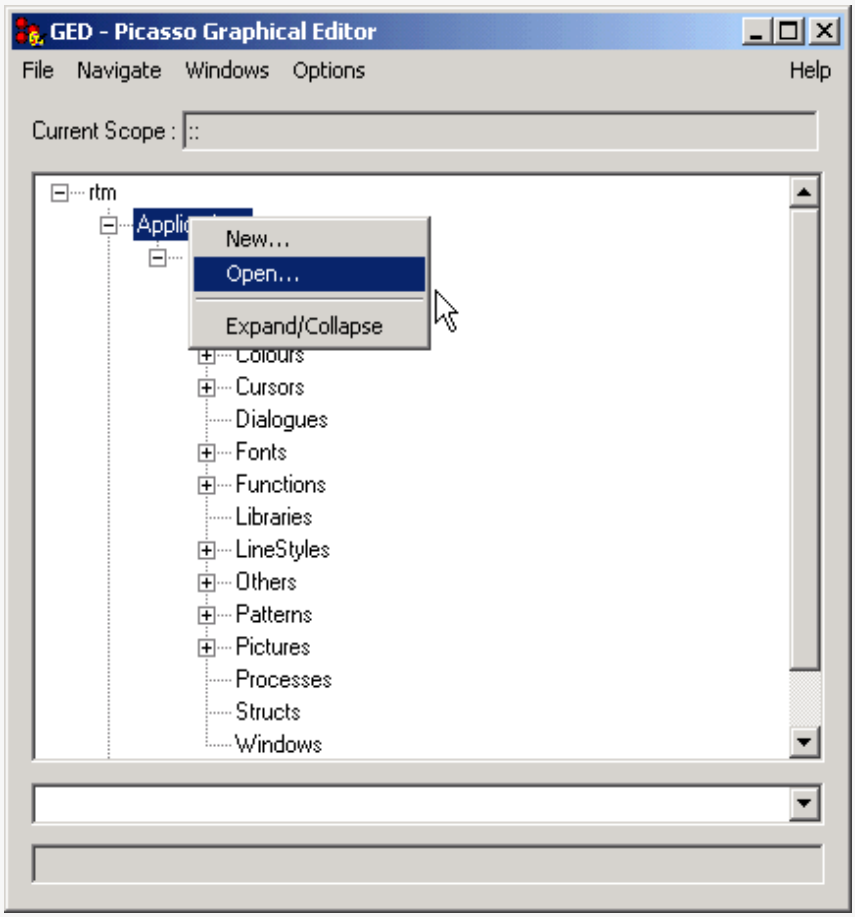
Opening an already existing application stored on file.



To load an existing application from file select the **Application** node in the tree controller and press the right mouse button. A popup menu will appear at the cursor position. From this menu select **Open...**. An alternative is from the **File** menu in the main menu bar to choose **Open...** when the **Application** node is selected..

Figure 21

Opening an already existing application stored on file.



A list of available applications will then be displayed in a File selection window. A ProcSee application file should always have the extension ***.pctx***. An application can be selected from this list by double clicking on the application name, or by clicking on the application name and then clicking on the **OK** button.

7.5 Creating a New Application

You can create a new application by selecting the **File** menu from GED's main menu, then selecting **New**, followed by **Application** from the New menu (see Figure 20).

The new application will be given the name **`NoName`**. NoName will also be displayed in the title line of the GED window.

To create a new application select the **Application** node in the tree controller. Then choose either **New** from the **File** menu in the menu bar or from the popup menu that will appear if you click on the right mouse button. (see Figure 21)

The new application will be given a name like **`NoName1, NoName2, ..., NoNameN`**. It is possible to edit several NoName applications in the Windows version of GED.

The new application should be given a proper name that is different from NoName. This can be done by selecting **Save As...** .

7.6 Saving an Application

To save an application to file, select **Save** from the File menu.

To save an application to file select it in the tree view. Choose **Save** from the **File** menu or from the menu that pops up when the right mouse button is clicked.* indicate that the application needs saving.

When using an external trend logger RTM the application name must be the same as the display RTM application name. If they are located in the same directory they have to be given different file names. How to give them different file names are described below.

Typically, the application will be saved to a file called <*application name*>.pctx. However, if the application has been loaded from a file with a name different to that of the application, this filename will be used.

If you want to save the application under a different name, select **Save As...** from the **File** menu and a Save As window pops up. Here you can enter a new name for your application

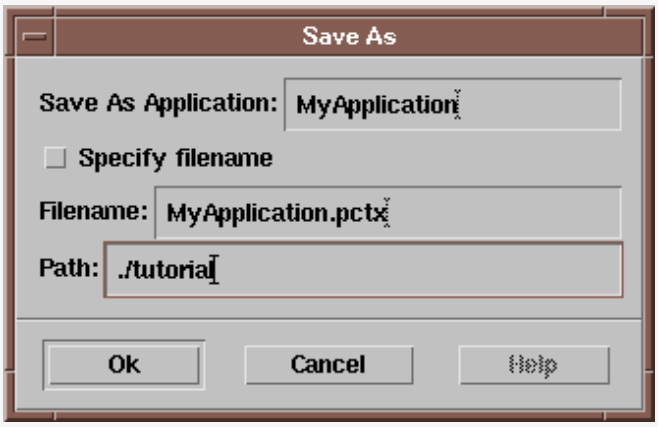
If you want to save the application to a filename different to that of the application, select the **Specify filename** toggle, and enter the filename in the **Filename:** input field, and the directory path to which you want to save the application. The default filename is <*application name*>.pctx. When the desired information is entered, click on the **OK** button and the application is saved to file. Note that you should save your application at regular intervals so that you will loose a minimal amount of work if there is a power cut or if your workstation crashes.

If the application has changed since the last time it was saved, the Save selection on the **File** menu will be labelled **Save (needed)**. If no changes have been made then it will be labelled **Save (not needed)**

.

Figure 22

A Save As... pop up window. An application can be given a new name by specifying the new name and the new directory path in the Save as pop up window. The application can be saved to a filename different to that of the application.



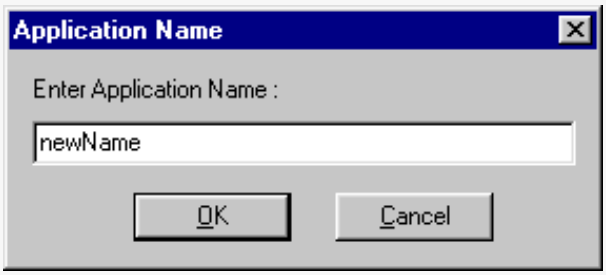
The If you want to save the application under a different name select it in the tree view. Choose **Save As** in the **File** menu or from the menu that pops up when the right mouse button is clicked.

You have to give the application a name in the "*Application Name*" window that is displayed. Press **Ok** and a standard Windows *"Save As"* window appears. From this window you can specify the directory where the application should be saved. You can also give the file a different name than the application.

Note that you should save your application at regular intervals so that you will loose minimal amount of work if the system crashes

Figure 23

An application can be given a new name by specifying the new name in this pop up window. The application can be saved to a filename different to that of the application.



7.7 Creating a Document of an Application

If you require human-readable documentation of an application a Tdoc file can be generated. This is a text file containing a complete description of the application. The document file can be compiled into a corresponding binary file using the *pcc* program,

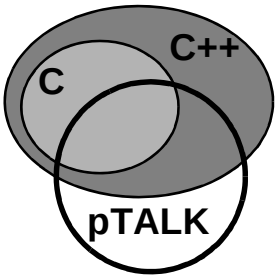
To create a document file select .**Document** in the **File** menu in GED. This will generate a file called <*application name*>.Tdoc.

To create a document file select the application you want to document in the tree controller. Choose either **Document** from the **File** menu in the menu bar or from the popup menu that appear when the right mouse button is clicked. This will generate a file called <*application name*>.Tdoc.

8 pTALK, the ProcSee Language

The ProcSee language is called pTALK. Although ProcSee supports the construction of complete user interfaces with only a minimum of programming, using pTALK is often the most powerful and efficient way to describe some features of a user interface. This chapter introduces pTALK.

8.1 The Heritage



pTALK could have been designed as a completely new language with little in common with other existing programming languages. Such a language could have been highly dedicated and hence extremely well suited to its purpose.

However, a dedicated language would have led to a steep learning curve for users that are already familiar with other programming languages. Since the majority of ProcSee users have some previous experience of languages such as Pascal, Modula, C, C++, etc., it was considered more appropriate to develop a language based on the most commonly used of these languages. pTALK is, therefore, a direct descendent of C and C++, which are amongst the most widely used languages in the computer software development world.

C and C++

pTALK provides most of the constructs of ANSI C and supplements them with useful object-oriented constructs from C++. A small number of entirely new constructs have been introduced to solve tasks that are specific to run-time languages, i.e. languages that are not compiled in the strictest meaning of the word.

All deviations from ANSI C are clearly marked throughout this chapter.

8.2 Comments

It is often difficult to browse through a computer program and try to discover how it works, especially if the program was written by someone else or was written some time ago. It is therefore advisable to annotate code with comments, to make it easier for humans to read.

Comments can be used to summarise the purpose of a function or variable, or to explain what an obscure segment of code does or how an algorithm works. Programmer should use lots of comments, but not so many that they obscure the actual code.

The compiler ignores comments, stripping them out before generating low-level code, so they have no overall affect on the performance of the code when it is executing.

Comments are indicated by 'delimiters', so that the compiler can identify and ignore them.

single-line comments If the compiler finds a double slash, `//`, it will ignore everything to the right of it.

multiline comments If the compiler finds a slash followed by an asterisk, `/*`, it will ignore everything it reads until it finds a `*/`. These are called 'comment pairs'.

nesting comments A comment pair cannot be placed inside another comment pair. However, double slash comments can be nested, even inside a comment pair, and vice versa.

A typical program contains a mixture of both comment forms. Longer, multiline, explanations are usually placed within comment pairs, whereas double slashes are used for single-line and half-line remarks.

Comments can also be used to prevent lines of code from being executed by 'commenting them out'.

Comments are used extensively in the code examples used in this guide.

8.3 Identifiers

An *identifier* is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Identifiers are case-sensitive. Identifiers longer than 31 characters are truncated. Reserved keywords are not legal identifiers.

To make it possible to handle variables with special names, *identifier* can also be a string of any characters surrounded by '$' characters. To include the '$' character itself in an identifier, it must be escaped with a back-slash '\$'. To include a backslash as part of an identifier it must be written as '\\'.

The variable with the special name ’34-XYZ 1290’ has to be written as $34-XYZ 1290$ in the pTALK code because it starts with a digit, and contains a ’-’ and a ’ ’ (space) character. A more extreme case: ‘0-$s\q‘ becomes: $0-\$s\\q$.

Note that the dollar sign, ’$’, should be used in all special names within the ProcSee RTM, but should be omitted in variable names used in parameters of Pf-functions dedicated to variable manipulation.

8.4 Expressions

type and value

An *expression* is anything that can have a *value*. An expression has a *type*, can be either simple or complex, and can include *operators*.

Constant Expressions

Numbers, characters and strings are called *constant expressions*, or, simply, *constants*.

The table below shows examples of constants:

constant	type	comments
1461382 -123	int	integer constant, decimal base
0d1461382 -0D123	int	base can be given explicitly using 0d or 0D prefix[a]
1_461_382 - 1_2_3	int	_ delimiters can be used (but not as first character)[b]
0x46b2 0X78FA	int	hexadecimal numbers start with 0x or 0X
0344 01234567	int	octal numbers start with 0
0o344 0O1234567	int	base can be given explicitly using 0o or 0O prefix[c]
0b11001 0B_1011_1100	int	binary numbers start with 0b or 0B[d]
3.14 .0003	double	floating point constant, double precision
31.4e-1 5E9	double	exponents can be given using e or E
’a’ ’X’	char	character constants are enclosed in single quotes
"Hello World"	char[]	string literals are enclosed in double quotes

a. Decimal prefix is not defined in C/C++
b. Underscore delimiters can be used on all integer constants, and this is not defined in C/C++
c. Octal prefix is not defined in C/C++
d. Binary numbers are not defined in C/C++

Character constants and string literals can contain *escape sequences* to denote special characters or to give special meanings to ordinary characters.

The table below shows examples of escape sequences:

sequence	examples	comments
\'	'\''	single quotes need only be preceeded by a backslash within a character constant
\"	"He said \"Hey\""	double quotes need only be preceded by a backslash within a string literal
\`	`\``	backquotes need only be preceded by a backslash within a backquoted character string[a]
\n	"Hello\nto\nyou"	newline
\b	"Hellu\bo"	backspace
ooo	"This is a space:\040"	the character code is parsed as an octal number and must contain three digits

a. The backquote sequence is not defined in C/C++

Arithmetic Expressions

+ - / * operators

Arithmetic expressions involve operations needed to perform trivial mathematics, i.e. addition, subtraction, division, and multiplication. The pTALK operators are `+`, `-`, `/` and `*`, respectively.

Integer Arithmetic

Division between integers results in an integer. Hence, when performing *integer division*, `7/2` is `3` and not `3.5`. The quotient is truncated if it contains a fractional part. This is important to remember when dealing with integer operands.

The remainder of the integer division `7/2` is `1`. Formally, we say that *7 modulus 2 is 1*. Modulus operations can be performed in pTALK using the modulus (`%`) operator. Hence, the expression `7%2` evaluates to `1`.

Operator Precedence

In order to resolve expressions such as `5+4*3` in a predictable manner, *precedence* rules exist that determine how an expression should be evaluated. These rules determine whether `5+4*3` should be evaluated as `(5+4)*3` or as `5+(4*3)`.

precedence

By associating a precedence value to each operator, an ambiguous situation can be resolved. The rule is that the operator with the highest precedence is evaluated first. In pTALK, the `*` operator has higher precedence than the `+` operator, so the above expression would be evaluated as `5+(4*3)`, which evaluates to the value `17`.

grouping

Expression such as `4*3.14/2` are special because the `*` and `/` operators have the same precedence. In order to determine which subexpression to evaluate first, *grouping* is used. The `*` and `/` operators

group from left to right. This means that in a case of equal precedence, the sub-expressions are evaluated from left to right. Thus `(4*3.14)` is evaluated first, and the result is then divided by `2`. Other pTALK operators group from right to left.

Since pTALK features over 40 different operators, it is beyond the scope of this introductory section to provide a complete precedence and grouping table. Please refer to the precedence table in the Reference Manual.

Logical and Relational Expressions

Logical expressions work with logic, i.e. truth. They are used to form conditions and evaluate whether the conditions are true or false. Unlike in the real world, where facts can be 'partly false' or 'not quite true', facts in pTALK are either 100% true or 100% false. Some languages have special data types to handle logical or boolean values. pTALK takes a simpler approach; the value 0 (zero) is false, and anything else is true. Hence the values 500 and -70 are equally true.

Relational and logical operators are introduced below.

relational operators Relational operators compare operands and return a truth value based on the comparison. There are six relational operators, and examples of each of these are shown below:

EXAMPLE

```
// pTALK

A == B   // returns 1 (true) if A is equal to B, 0 (false) otherwise
A != B   // returns 1 if A is different from B, 0 otherwise

A > B    // returns 1 if A is greater than B, 0 otherwise
A >= B   // returns 1 if A is greater than or equal to B, 0 otherwise
A < B    // returns 1 if A is less than B, 0 otherwise
A <= B   // returns 1 if A is less than or equal to B, 0 otherwise
```

logical operators Logical operators are used to combine other expressions, resulting in a boolean value. There are three logical operators, and examples are shown below:

EXAMPLE

```
// pTALK

A && B  // returns 1 (true) if both A and B is true, 0 (false) otherwise
A || B  // returns 1 if either A or B is true, 0 otherwise

! A     // returns the inverse truth value of A; 1 if A is false, 0 otherwise
```

The *logical AND* (**&&**) operator and the *logical OR* (||) operator both use *short circuit evaluation*. For example, if **A** is false in the expression **A && B**, then the whole expression must evaluate to false regardless of the value of **B**. pTALK will not evaluate **B** at all if **A** is false. If **A** is true in the expression **A || B**, then the whole expression must evaluate to true regardless of the value of **B**. pTALK will skip the evaluation of **B** in this case.

Assignment Expressions

In the examples described so far, expressions have been evaluated to find the a result of a calculation. However, it is often necessary to *change* the value of data objects. This is done using *assignment expressions*. The simplest assignment operator is the *pure assignment* (**=**) operator. The right operand is evaluated, and the resulting value is assigned to the left operand. The result of the expression itself is the left operand.

Here's some examples of assignment expressions:

EXAMPLE

```
// pTALK
A = 56                 // A is assigned (is changed to) the value 56
A = B                  // A is assigned the value of B
A = 34 + B / 3.14     // A is assigned the calculated value of the expression
```

Some requirements exist for the left operand. Clearly, assigning anything to a literal is not permitted, so the expression **3=5** is illegal[1]. The expression **A*B=4** is also illegal because **A** and **B** cannot "share" the value **4** in a meaningful way.

lvalues The rule is that the left side of an assignment operator must be an *lvalue*. The term lvalue is short for *location value*, meaning something that has a location, a position in memory. Variables and attributes all have locations in memory where the values are stored, while literals and expressions do not.

rvalues Literals and expressions are *rvalues*, or *read values*. In the case of expression statements, lvalues can be considered to be *left side values* and rvalues *right side values*.

Assignments group from right to left, so several assignments can be combined in one expression, for example **A=B=5** where **5** is assigned to **B**. The result of **B=5** is **B** itself, whose value in turn is assigned to **A**.

1 Although early versions of FORTRAN happily accepted this, so 3 evaluated to 5 later on in the program...

Special Assignment Operators

A=A+B is an expression that is used extensively in a typical program. In pTALK, the above expression can be written as **A+=B** in order to reduce the amount of typing required, and probably improve readability too. All arithmetic operators (and some others that will be discussed later in this chapter) can be combined in assignments like this.

Special increment (**++**) and decrement (**--**) operators also exist to make coding easier by reducing the amount of typing required and the amount of code to read.

Examples of the special assignment operators are show below:

EXAMPLE

```
// pTALK

aRatherLongVariableName += 5   // add 5 to the variable

B /= 2                         // divide B by two

C %= 5                         // equal to the expression C = C % 5

A *= C/B+4                    // multiply A by the value of the expression

node345numberOfEdges++         // increment the variable by 1

A--                            // decrement the variable by 1
```

8.5 Statements

While an expression has a value and type, a statement has neither. A *statement* can be thought of as a command; an order to do something. The simplest statements are **;** (semicolon) or **{}** (curly braces), both of which do nothing. Neither a semicolon nor a pair of curly brackets are particularly useful on their own, but they illustrate an important rule: A statement is anything that ends with a semicolon or is enclosed in curly braces[1].

1 A computer language expert might react to this rather inaccurate description, but it is never the less a useful rule of thumb...

Declaration Statement

A declaration followed by a semicolon[1] is a *declaration statement*.

EXAMPLE

```
// pTALK
// declaration statement

float floatVariable;
```

Expression Statement

An expression followed by a semicolon is an *expression statement*.

EXAMPLE

```
// pTALK
// expression statement

floatVariable = 3.14 * Y;
```

Compound Statement

Several statements can be grouped together to make a *compound statement*. This is done by enclosing a sequence of statements within curly braces, as shown below:

EXAMPLE

```
// pTALK
// compound statement

{
   floatVariable = 3.14 * Y;
   Y++;
}
```

Note that pTALK is a *free format language*, meaning that spaces and line breaks can be put anywhere between tokens. The example below will give exactly the same result as the example above:

EXAMPLE

```
// pTALK
// peculiar spacing and line breaks....

{ floatVariable = 3.14   *
  Y
; Y
                         ++ ; }
```

1 Note that the semicolon is *not* used as a separator between statements, as in languages like Pascal, but an integral part of a specific kind of statement.

The layout in the example above is not recommended, but it illustrates the freedom to format code freely in order to improve legibility.

Conditional Statements

It is often only necessary to perform an action if one or more preconditions are met. The most common constructs for this are the *if statement* and the *if-else* statement. For example, one might wish to set an alarm so that it is triggered if a liquid level exceeds a predefined limit or is reset otherwise. This can be done using the code below, assuming that the three variables **`level`**, **`limit`** and **`alarm`** have been defined previously.

IF - ELSE

```
// pTALK

if ( level > limit )
   alarm = 1;
else
   alarm = 0;
```

In order to perform more than one operation, a compound statement can be used. To test for several conditions, **`if-else`** constructs can be nested. The example below shows both:

IF - ELSE

```
// pTALK

if ( level > 400 )
{
   alarm = 4;
   popUpAlarmPicture();
}
else if ( level > 300 )
   alarm = 2;
else
   alarm = 0;
```

Important Note about Keywords

In the section above *reserved words* or *keywords* have been introduced. The token **`else`** is a reserved word, as are **`while`**, **`if`**, **`do`**, **`for`**, **`int`**, **`double`**, **`float`**, **`case`**, **`switch`**, **`operator`** and many others[1]. Reserved words are tokens that have special meanings in pTALK, and as such cannot be reused as user-defined identifiers. For example, neither a variable nor anything else is permitted to be given the name **`switch`**.

1 Refer to the pTALK manual pages in The ProcSee Reference Manual for a complete list of reserved words.

Record type names and class names are also reserved, and cannot be use as names for functions, attributes, pictures etc.

Loops

In order to repeat some operation until, or as long as, some condition is satisfied, a *do-while statement* or a *while statement* can be used. The following example calculates the sum of all numbers between *1* and *n*, using both loop constructs:

EXAMPLE

```
// pTALK

int sum = 0, i = 1;         // Alternative I: while statement
while ( i < n )
{
   sum += i;
   i++;
}


int sum = 0, i = 1;         // Alternative II: do-while statement
do
{
   sum += i;
   i++;
}
while ( i < n );
```

The compound statement is repeated as long as the test expression evaluates to true. The only difference between the two constructs lies in *when* the test is evaluated. In the **`while`** statement, the test is evaluated *before* each loop (hence the test condition is put after the while statement at the top of the loop), and in the **`do-while`** statement, the test is evaluated *after* each loop (hence it's placement after the while statement at the bottom of the loop).

Consequently, the **`do-while`** statement will execute at least once, since the test is evaluated *after* each loop. The **`while`** statement will not execute at all if the test condition initially evaluates to false.

Breaking and Continuing a Loop

break

A **`while`** statement or a **`do-while`** statement usually runs until the loop condition indicates that it should stop. It is possible to terminate a loop at any time by applying a *break statement*. The **`break`** statement causes transfer of control to the statement that immediately follows the **`while`** or **`do-while`** statement.

continue

It is also possible to terminate the current iteration of a **`while`** or **`do-while`** loop and start the next iteration immediately by applying a *continue statement*. Control is then transferred to the conditional expression.

Note that where several **while** or **do-while** statements are nested inside each other, the **break** or **continue** statements influence only the most deeply nested statement.

EXAMPLE

```
// pTALK

int sum = 0, i = 1;
while ( i < n )
{
   if ( i % 7 == 0 )
   {
      i++;
      continue;     // skip all multiples of 7
   }
   sum += i;
   if ( i >= 1000 ) break;          // break anyway if i >= 1000
   i++;
}
```

8.6 Local Variables

Sometimes it is necessary to store the result of a calculation temporarily for later recall. Local variables in pTALK are used for this purpose.

Local variables must be declared before they are used, with a name, type[1] and, optionally, an initial value. They can be declared anywhere where a statement is legal. The value that they hold exists only while the program control is within the compound statement that they belong to.

The example below shows how to declare local variables for storing intermediate results. In the example, the variables **a**, **b**, **c**, **pi**, **e**, and **ptr** can only be accessed from statements that appears within the curly brackets that enclose the variable declarations.

EXAMPLE

```
// pTALK

{
      :
   int    a, b, c;
   float  pi = 3.14, e = 2.71;
   Valve*  ptr;
      :
   a = b = c = pi * 32 * e; // is OK here, but the variables a, b, c,
                            // pi and e cannot be accessed after the
                            // curly bracket below.
}
```

1 Local variables can not be of type array or type record. Local pointer variables (records included) are allowed.

8.7 Functions

Many languages differentiate between *functions* and *procedures* (subroutines), but not pTALK. A function in pTALK is similar to a mathematical function, taking some parameters as input, performing arbitrary operations on those parameters and then returning a result. A very simple function is shown below. It takes a numerical input parameter called *x* in and then calculates and returns *x* to the power of 2.

EXAMPLE

```
// pTALK

float square( float x )
{
   return x * x;
}
```

The function is called ***square()***, and is of type **`float`**. The argument *x* is also of type **`float`**. The **`return`** statement calculates an expression and terminates the function, i.e. it lets the function return the calculated value as a result.

To create a more general function for calculating powers of *n*, rather than only squares, a function called ***power()*** could be defined that takes two arguments *x* and *n*. An example of this is provided below. The ***power()*** function uses a loop to build up the result. When the loop terminates, it returns the result as the value of the function itself.

EXAMPLE

```
// pTALK

float power( float x, int n )
{
   float product = 1;
   int   i       = 1;

   while ( i <= n )
   {
      product *= x;
      i++;
   }
   return product;
}
```

The ***square()*** function can now be defined in terms of the ***power()*** function. This approach is not recommended as it is relatively inefficient, but it illustrates how a function can call another function in order to perform the tasks that it is intended to carry out.

EXAMPLE

```
// pTALK

float square( float x )
{
   return power( x, 2 );
}
```

Void Functions

Not all functions have, or need, return values. Function that do not return anything are called *void functions*, indicated by a *void return type*. This resembles the concept of procedures or subroutines in some other languages.

Argument Passing

pTALK passes arguments *by value*, which means that the values of the *actual arguments* are copied into the *formal arguments* when the function is called. To illustrate this, we'll create a function that doubles the value of its argument. Here's a first attempt:

EXAMPLE

```
// pTALK

void doubleIt( float x )
{
   x = 2 * x;
}

   :
   :
float  var = 5.5;
doubleIt( var );
```

One might think that the above function does what was intended, i.e. that the variable **`var`** will have the value **`11`** after ***doubleIt()*** is called. However, that assumption would be incorrect because the value of the actual argument **`var`** is copied into the formal argument *x*, and so it is *x* that is doubled to **`11`**. In the example above, the value of **`var`** remains unaltered.

If we require a function that changes the values of the actual variables that are passed to it as arguments, then the arguments must be passed as pointers. Passing a pointer to a variable means to pass a *reference* to it. Below is a version of the ***doubleIt()*** function that actually doubles the value of the variable passed to it:

EXAMPLE

```
// pTALK

void doubleIt( float* x )
{
   *x = 2 * *x;
}

   :
   :
float  var = 5.5;
doubleIt( &var );
```

The formal argument *x* is declared as a *pointer to a float*, written as **`float*`**. When the function is called, we pass a *pointer* to the variable **`var`** (a reference to it). This is written as **`&var`**.

Recursive Functions

A *recursive function* is a function that calls itself. This is very useful for functions where the route to the solution is recursive by nature. A classical example is the mathematical function *factorial*, which calculates 1*2*3*...*n for some value of n. The factorialfunction can be formally defined as

$$0! = 1$$

$$1! = 1$$

$$n! = n \times (n-1)!$$

A recursive pTALK function that look similar to the mathematical definition above can be written to calculate *n*!:

EXAMPLE

```
// pTALK

int fact( int n )
{
   if ( n == 0 || n == 1 ) return 1;        // 0! = 1 and 1! = 1
   return n * fact(n-1);                    // n! = n * (n-1)!
}
```

8.8 Member Expressions

All objects in ProcSee are children of other objects. The great grand ancestor of all objects is *::* or *root*. To refer to these parents and their children from within pTALK, the **.** dot operator[1] is used. For example, in the expression **A.B**, the object **B**, whose parent is **A**, is being referred to. The parent can be any object (or pointer to an object) that has children. Typical examples are record variables, pictures, and instances. The child can be a record field, an attribute or even a function. Examples follow:

SYNOPSIS

```
// pTALK

alarmPicture.xSnap = 10;           // xSnap is an attribute in the picture

pumps[10].flow = 34.4;              // flow is a record field

if ( Valve1.isOpen() ) ...          // isOpen is a function

components.list.numberOfItems - 10 // the dot operator can be nested
```

8.9 For Loops

A *for-statement* is typically used to step through a data structure of fixed length, such as an array. In general, nothing can be done with a **for** statement that cannot be done with a **while** statement. Nevertheless, it can be used to write more compact code, and some people find it easier to use as it, to some extent, reduces the risk of errors. (especially where **continue** statements are used in loops that increment / decrement a variable as it is easy to forget to increment it before a **continue** in a **while** loop).

The synopsis is as follows:

SYNOPSIS

```
// pTALK

for ( S1 ; E1 ; E2 ) S2;

S1;                 // S1 is a statement executed once only
while ( E1 )        // E1 is the loop condition evaluated before each loop
{
    S2;             // S2 is the body of the for loop
    E2;             // E2 is an expression that is evaluated after each loop
}
```

1 The C/C++ **->** arrow operator is optional in pTALK. The **.** dot operator can be used everywhere the arrow operator can be used and vice versa. This is unlike C/C++, where the arrow operator must have a pointer left operand and the dot operator must have a non-pointer operand.

The synopsis above also compares the *for* statement with an ordinary *while statement*. The expression *E1* is the loop condition, and the loop runs as long as *E1* evaluates to true. The **`for`** statement is special in that it contains a statement *S1* to be executed once before the loop, and also an expression *E2* to be evaluated at the end of each iteration. The initial statement *S1* is typically used to declare and initialize one or more loop counters, while the expression *E2* is typically used to increment the loop counter. An example follows:

EXAMPLE

```
// pTALK

int n   = 100;
int sum =   0;

for( int i = 1; i < n; i++ )
{
   sum += i * n;
}
```

The statement *S1* and the expressions *E1* and *E2* are optional. If the loop condition E1 is omitted, the result is an infinite loop; so **`for(;;)`** is identical to **`while(1)`**.

Note that **`break`** and **`continue`** statements also apply to for loops. The **`break`** statement immediately breaks out of the for loop, while the **`continue`** statement transfers control to *E2* followed by *E1*[1].

8.10 The Switch Statement

The *switch statement* is the most advanced construct in pTALK. There are three different forms of the statement, of which only one is defined in C/C++.

Interval Testing[2]

Using the **`switch`** statement to do *interval testing* is quite intuitive and easy to learn. A typical example of its use would be to test the value of a variable and do different things depending on the value.

1 This is the only exception to the comparison with the while-statement. Note that the expression *E2* is evaluated after a continue statement.

2 The interval form of the switch statement is not defined in C/C++.

Here is an example:

INTERVAL TESTING

```
// pTALK

switch ( level )        // check the value of variable level...
{
 case [0,99]:           // if level is in the interval [0,99]...
   alarmColour = green;
   break;
 case [100,199]:        // if level is in the interval [100,199]...
   alarmColour = yellow;
   break;
 default:               // otherwise...
   alarmColour = red;
   popupAlarmPicture();
   break;
}
```

In the example above, if the value of **level** is in the interval between **0** and **99** inclusive then **alarmColour** is set to **green**. If the level is between **100** and **199**, then **alarmColour** is set to **yellow**, otherwise **alarmColour** is set to **red** and an alarm picture is displayed (by calling a function that performs that task). This kind of interval testing is common in user interfaces, hence the development of a special construct in pTALK to support it.

Note that the expression in the parenthesis after the **switch** statement can be arbitrarily complex. It is only evaluated once, and then the resulting value is tested against the intervals. The upper and lower limits of each interval can also be complex expressions.

Using breaks

Each action block should normally end with a **break** statement. Although it is not strictly necessary to specify a **break** at the end of the last action block of a **switch** statement, it is considered good practice to do so. An accidentally omitted **break** statement is a common programming error that can often be difficult to locate, as a missing **break** statement can give unexpected results in code that comes after the **switch** statement has completed.

Equality Testing

The second **switch** statement form is that found in C/C++. It tests an expression against a series of values.

Operator Testing[1]

Operator testing is a natural extension of *equality testing*. While equality testing does just that (tests whether a value is *equal* to the alternatives), operator testing tests whether a value is *less than*, *greater than*, (or *whatever than)* the alternatives.

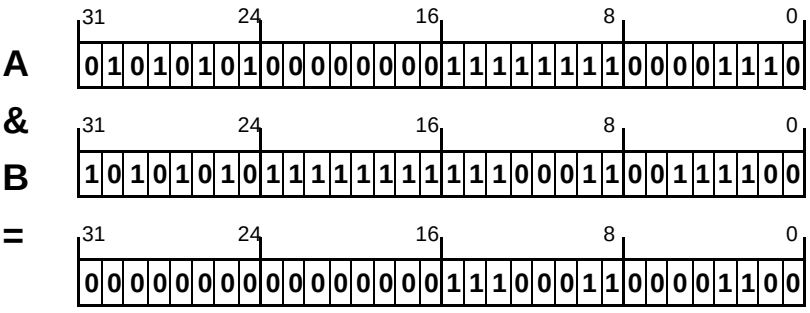
1 Operator testing is a variant of the switch statement that is not found in C/C++.

8.11 Bitwise Manipulation

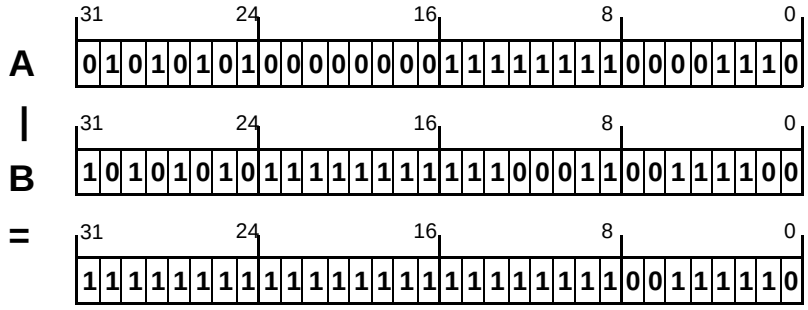
pTALK contains a family of operators to deal with bit manipulation. Bitwise operations have a close resemblance to logical operations; bitwise *AND*, bitwise *OR*, and bitwise *NOT*. There is also a bitwise *exclusive OR* operator. The operators are **&**, **|**, **~**, and **^**, respectively. They normally deal with 32 bit unsigned integers operands. The figure below should illustrate the difference between these logical operations:

Figure 24

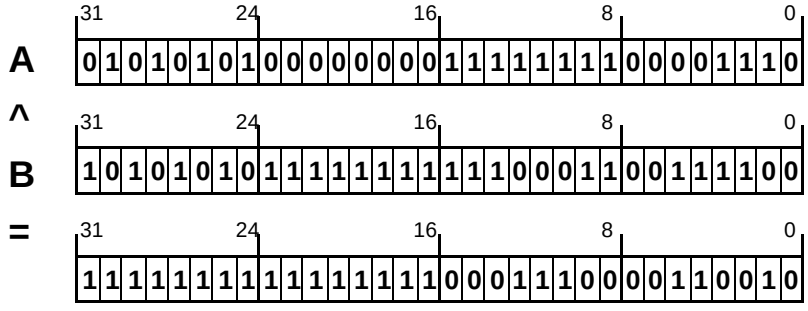
*The bitwise AND (**&**) operator returns a result where a given bit n is 1 if both corresponding bits in the operands are also 1. The bit is set to 0 otherwise.*



The bitwise OR (|) operator returns a result where a given bit n is 1 if at least one of the corresponding bits in the operands are also 1. The bit is set to 0 otherwise.



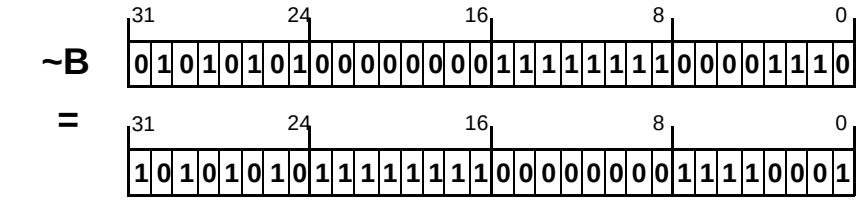
The bitwise exclusive OR (^) operator returns a result where a given bit n is 1 if one and only one of the corresponding bits in the operands are also 1. The bit is set to 0 otherwise.



Only one of the bitwise operators is unary, meaning that it takes only one operand. The bitwise NOT (**~**) operator return *s* a result where all bits in its operand are inverted. This is shown in Figure 25.

Figure 25

The bitwise NOT (~) operator return a result where all bits in its operand are inverted.

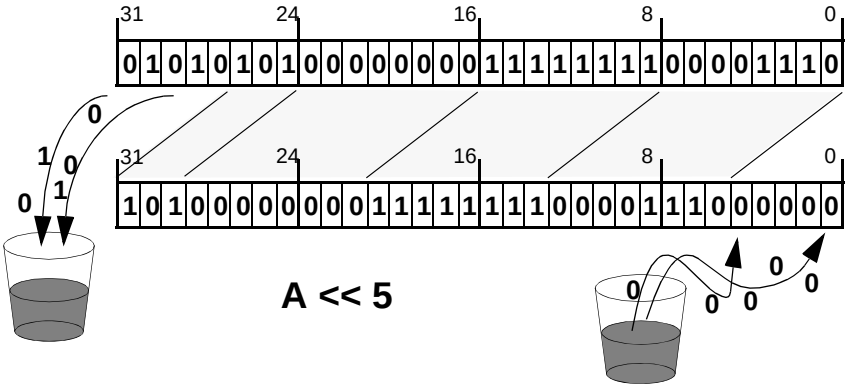


Shifting Bits

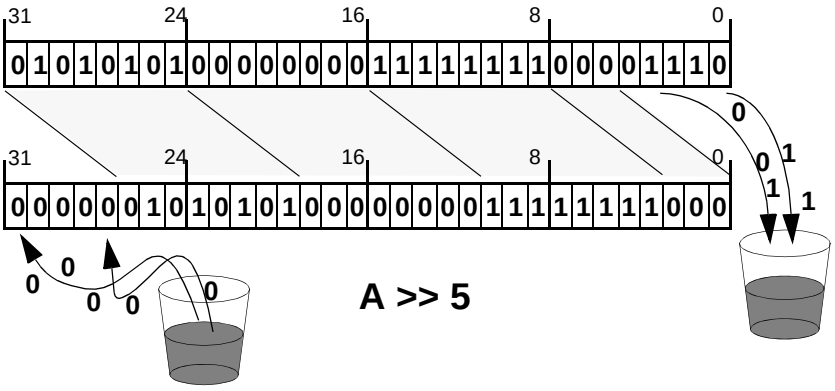
Operators exist to shift the bits in an integer to the left or right. This is illustrated in Figure 26.

Figure 26

The shift left (<<) operator shifts all bits in its first operand to the left a number of positions given by the second operand. The bits that is "left over" is discarded, and the new places are filled with zeros.



The shift right (>>) operator shifts all bits in its first operand to the right a number of positions given by the second operand. The bits that is "left over" is discarded, and the new places are filled with zeros.



Note that shifting bits *n* positions to the left is identical to multiplying the number by 2^n. Shifting bits *n* positions to the right is identical to dividing the number by 2^n.

Bitwise Assignment

Bitwise operators can be combined with assignment just like the arithmetic operators. These operators are **&=**, **|=**, **^=**, **~=**, **<<=**, and **>>=**.

Examples of Bitwise Operations

Bitwise operations can be combined and used to manipulate the bits in integer attributes and variables. Below are some examples of how to use bitwise operators to manipulate data bit by bit.

EXAMPLE

```
// pTALK

int A;

A |= 0b_10_0111;      // insert the given bits into A

A |= 1<<5;            // set bit number 5 in A

A ^= 1<<5;            // toggle bit number 5 in A

A &= 0xd6a;           // clear all bits in A that is not in 0xd6a

A = 1<<23 | 1<<0;     // A has a value where only bits 23 and 0 are set

A = ~(1<<23 | 1<<0);  // Bits 23 and 0 are the only that are not set
```

8.12 Dynamic Binding[1]

Special language constructs are used to handle *dynamic binding*. ProcSee uses dynamic binding extensively to connect application data to user interface components. The colour of a component can, for example, be bound to some application data, as shown in the example below:

EXAMPLE

```
// Document file

rectangle R
{
   foregroundFillColour = `level > limit ? red : green`;
      :
      :
}
```

In the above extract from a document file, the foreground fill colour of the rectangle **R** is dynamically bound to an expression. The effect is that the colour will be **red** *whenever* the value of **level** is greater than the value of **limit**, otherwise it will be **green**.

1 Dynamic binding through the use of the **:=** and **@** operators is not found in C/C++.

Dynamic Assignment

Setting up dynamic bindings can, of course, be done using the editor (GED), or by editing the document file directly. But, it can also be done at run-time using pTALK. A special assignment operator (**:=**) can be used to do this, as in the following fragment of pTALK code:

EXAMPLE

```
// pTALK

R.foregroundFillColour := "level > limit ? red : green";
```

The right side of the expression can be anything that returns a character string.

Getting a Dynamic Binding

It is possible to query a binding using the **@** operator. If the **@** operator is applied to an attribute that is dynamically bound, the pTALK source code is returned.

EXAMPLE

```
// pTALK

char* theSourceCode = @ R.foregroundFillColour;
```

Assuming that the dynamic binding from the previous example still applies, **@R.foregroundFillColour** will return the string **"level > limit ? red : green"**. If the attribute is not dynamically bound, i.e. contains a constant value, the **@** operator will return an empty string, **""**.

8.13 Generating Code at Run-Time[1]

Languages like C, C++, Pascal, and Fortran are compiled languages. This means that all source code must be prepared prior to compilation and linking, and no changes can be made to the source while the program is running. The write-compile-run cycle must be obeyed. pTALK is considerably more flexible, as new source code can be introduced and run whenever needed. Examples of this have been in the previous section, where dynamic binding were introduced.This section introduces two more related mechanisms.

1 None of the features introduced in this section are available in C or C++.

Building new Source

a push button class Consider the design of a push button class that is to be used in a user interface. Push buttons could be used as part of menus or forms to enable the end user to perform certain actions by clicking on them with the mouse button. However, in order to create a general-purpose button class, it is necessary to express an action, but the action that will eventually be bound to each instance of the push button is unknown.

The solution to this problem is to express the action as text and store it in a character string attribute in the button class. A different text can then be assigned to each instance of the button class. The text string can then be executed as pTALK source code using one of the following two functions:

execute() The standard pTALK function ***execute()*** executes a text string. The routine parses the string as pTALK source code and executes it.

executeAsync() ***executeAsync()*** is an asynchronous version of ***execute()***.

Run-Time Lookup

from string to object Sometimes the name of an object might only be available as a text string, but to perform an operation on an object it is first necessary to retrieve the actual object. This can be done using pTALK's **@** lookup operator. The example below shows how to call the ***open()*** function of an instance of a class **`Valve`**, given the name of the instance stored in a variable **`CurrentValve`**:

EXAMPLE

```
// pTALK

char* CurrentValve;
   :
   :
(Valve @ CurrentValve).open();
```

The left side of the lookup operator must be a type name, and the right side of the operator can be any expression that evaluates to a character string. The result of the expression is a pointer to the object with a name given by the string.

8.14 Several Ways to do The Same Thing

In a relatively general purpose language such as pTALK, there are several ways of doing the same thing. This flexibility should be used to choose the most effective and readable way to solve a given task. The example below demonstrates this:

EXAMPLE

```
// pTALK

// Alternative I: if statement
if ( level <= 100 )
   alarmColour = green;
else
   alarmColour = red;

// Alternative II: conditional expressions
alarmColour = level <= 100 ? green : red;

// Alternative III: logical and expression
alarmColour = red;
level <= 100 && alarmColour = green;

// Alternative IV: switch statement
switch ( level ) <=
{
   case 100: alarmColour = green; break;
   default : alarmColour = red;
}
```

9 Basic User Interface Components

This chapter covers the basic building blocks needed to create a graphical user interface. In ProcSee, the basic building blocks can be put into two groups: resources (windows, colours, patterns, fonts, linestyles, and cursors) and other components (processes, libraries, record types, attributes, functions, and dialogues).

9.1 The Building Blocks of a User Interface

In ProcSee, some basic user interface components are needed before the implementation of a user interface can begin. The basic interface components are resources such as *windows*, *colours*, *patterns*, *fonts*, *linestyles*, and *cursors*, and other components such as *libraries*, *record types*, *resource styles*, *attributes*, *functions*, and *dialogues*.

On X-Windows, the resources that cannot be defined / modified interactively from within GED are *cursors*, *patterns*, *linestyles*, *resource styles* and *records*.

The main resource file for an application is called <*application name*>.pctx, and the corresponding document file is called <*application name*>.Tdoc.

User Interface components that can be defined / modified from within GED are

- colours *(section 9.3 "Defining Colours in GED")*
- fonts *(section 9.4 "Defining Fonts in GED")*

- patterns
- line styles
- cursors
- resource styles *(section 9.9 "Resource Styles")*
- windows *(section 9.10 "Defining Windows in GED")*
- layers *(section 9.12 "Layers")*
- libraries *(section 9.13 "Libraries")*
- attributes, functions, and dialogues *(section 9.14 "Processes")*

The basic user interface components should preferably be defined as early as possible in the user interface development process.

9.2 Defining Resources

The resources (*colours, fonts, cursors, patterns, linestyles* and *windows*) are defined in a document file called <*application name*>.Tdoc, as illustrated below:

When this document file is compiled using the *pcc* program, a file named **`Tutorial.pctx`** will be generated. That file is a binary version of the ProcSee resources defined in the document file.

Some of the resources can be defined / modified interactively from within GED

- colours
- fonts
- windows

Defining Cursors

User defined cursors can be added in the main resource file, as illustrated in the example above. A cursor is defined by specifying the font index of the desired cursor, the foreground colour, and the background colour. You can also define your own bitmap cursors.

Two pTALK functions are available for changing between cursors and modifying colours of existing cursors; **`defaultCursor(...)`** and **`setCursorColours(...)`**. **`setCursorColours(...)`** only works on an unix platform. For a detailed description of these functions, refer to the *ProcSee Reference Manual*. The cursors can also be set on individual windows, see section 9.10 "Defining Windows in GED"

```
// Configuration language

application Tutorial
{
 colour black { red = 0;      green = 0;    blue = 0;      type = 0; }
 colour white { red = 65535;  green = 65535;blue = 65535;  type = 0; }
 colour red   { red = 65535;  green = 0;    blue = 0;      type = 0; }
 colour cyan_red_green
 {
  type = 2;
  rgb { red=0;      green=65535;  blue=65535;  time=500; }
  rgb { red=65535;  green=0;      blue=0;      time=300; }
  rgb { red=0;      green=65535;  blue=0;      time=300; }
 }

 font time_b_12
 {
   name = "-adobe-times-bold-r-normal--12-120-75-75-p-70-iso8859-1";
 }

 font time_14
 {
   name = "-adobe-times-medium-r-normal--14-140-75-75-p-77-iso8859-1";
 }

 display display1
 {
   connection = "";
   width = 1280;
   height = 1024;
   resizeMode = 0;
 }

 window win0
  {
   parent = "display1";
   x = 0;
   y = 0;
   width = 1142;
   height = 870;
   backgroundColour = -1;
  }

 pattern MyPattern
 {
   patternBitmap = "MyPatternDefinition";
 }

 cursor ProcessingCursor
 {
   fontCursor = 150;      // see: xfd -fn cursor
   foregroundColour = 1;
   backgroundColour = 0;
 }

 cursor myBitmapCursor
 {
    bitmapCursor = "Cursorfile.xbn";
 }
}
```

Defining Patterns

By editing the **.Tdoc** file, user defined patterns may be added to the applications resources, as illustrated in the example above,

In the procsee directory - examples / patterns, some predefined patterns can be found.

An example of including three of those patterns, **Raster0**, **Wave1** and **Brick** in the resource file, is shown below.

EXAMPLE

```
pattern Raster0 { patternBitmap = "./patterns/Raster0.bm"; }
pattern Wave1   { patternBitmap = "./patterns/Wave1.bm"; }
pattern Brick   { patternBitmap = "./patterns/Brick.bm"; }
```

9.3 Defining Colours in GED

To modify or create a new colour, select the **Colours** item in the View menu from GED's main window(Figure 27). The colours

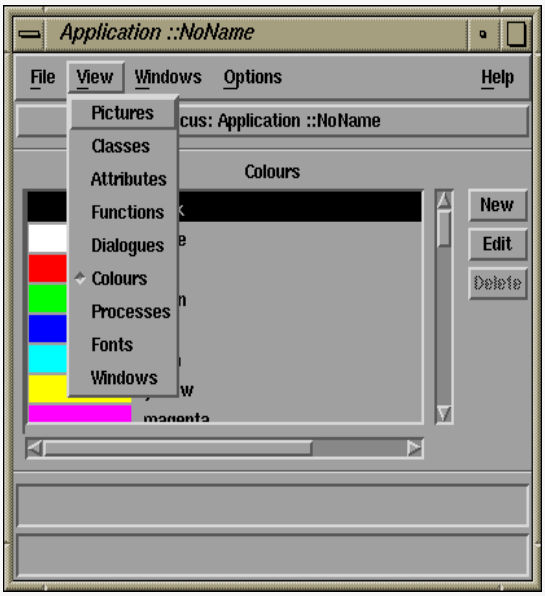


Figure 27

The figure shows GED's main window and how colours can be selected from the View menu. Colours can be created and modified by selecting the New or Edit button on the right side of the window, and the Colour editor (Figure 29) will be displayed.

that are already defined will be listed. Three buttons for colour manipulation are located on the right side of the colour list. To edit an existing colour select the colour by clicking on the colour rectangle or text, and the selection will be highlighted. After a colour is selected click on the **Edit** button to display the *Colour Editor* (see Figure 29).

The *Colour Editor* can also be displayed by selecting the **New** button.

To create a new colour, select the **Colours** node in the Main window as shown in Figure 28. If you expand this node you will see the already defined colours as subnodes of the **Colours** node. To edit an existing colour, right click on the colour name and select **Edit...**, or double click on the colour name to display the *ColourEditor* (see Figure 29)

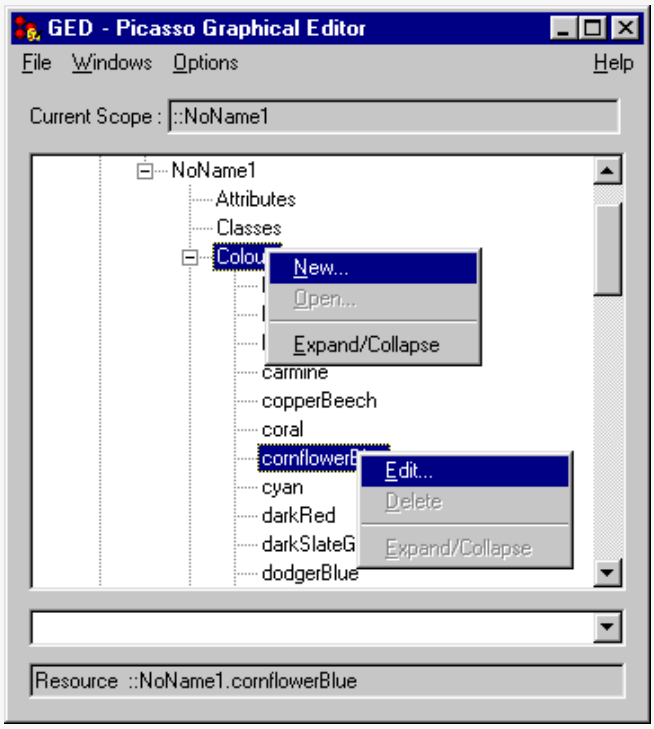


Figure 28
The figure shows GED's main window and how colours can be selected from the Tree view. Colours can be created by selecting New... and modified by selecting Edit... on the popup menu, and the Colour editor (Figure 29) will be displayed.

The *Colour Editor* can also be displayed by selecting **New...** from the popup menue that appears when right clicking on the **Colours** node.

The *Colour Editor* supports two colour systems: *RGB* (Red Green Blue) and *HLS* (Hue Lightness Saturation).

The name of the colour can be specified in the text input field labelled *Colour*.

Five colour information rectangles are at the top portion of the colour editor. The colour rectangle labelled **Restore** displays the last accepted colour. The rectangle labelled **Inverse** displays the inverted colour. The resulting Lightness value will be 100% - current Lightness value. The larger rectangle in the middle, displays the current colour. Note that the rectangle displays the currently selected segment of a blink colour only and not the resulting blinking colour. The rectangle labelled **Complement** displays the resulting complement colour of the current colour. The rectangle labelled **Accept** displays the current colour including blinking colours. By selecting the **Accept** button the current changes to a colour are installed, and the new definition of the colour is used in any of the pictures displayed. This is a temporary installation of the colour,

and the original colour can be reinstalled by selecting the **Cancel** button.By clicking on any of the buttons **Restore**, **Inverse**, **Complement**, and **Accept** the current colour is set accordingly. When clicking on the **Grab** button a cross cursor appear. Move the cross over the place where the colour is that you want to grab. Left click on the mouse to set the selected colour into the colour editor.

Colour components (**Red, Green, Blue, Hue, Saturation**, and **Lightness**) can be changed by clicking on the arrow sections at the end of the scrollbars, or by moving the slider inside the scrollbars. The coloured rectangles at either end of the scrollbars indicate the resulting colour if the maximum or minimum value of that colour component is selected.

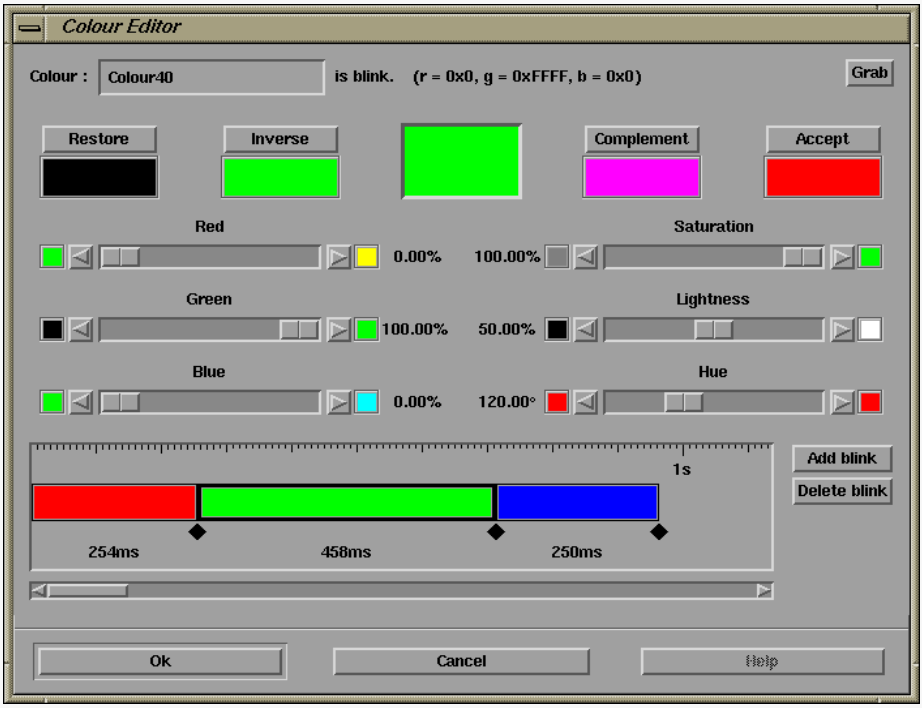


Figure 29 *The figure shows the Colour Editor. The editor supports two colour systems: RGB and HLS for editing and creating new colours. Blink segments can be added to a colour by selecting the Add blink button and removed by selecting the Delete blink button.*

Defining a blink colour

A blink colour can consist of an arbitrary number of blink segments with intervals from 1 millisecond and up to 10 seconds, and any number of blinks

To add a new blink segment, select the **Add blink** button and a new blink segment will be added. The blink segment interval can be changed by clicking on the diamond on the right side of the segment. There is no diamonds on the Windows version, only rulers. While depressing the mouse button, moving the mouse to the desired blink interval. In this case the interval of the current segment will change, and thus affect the total blink interval. If the <Shift> key is pressed while changing the interval, the total blink interval is kept constant, and the two adjoining blink intervals are changed.The blink interval for a segment is displayed below the blink segment.

A blink segment can be deleted by first selecting it and then selecting the **Delete blink** button.

To change the colour of a blink segment, select it by clicking on it using the left mouse button. The colour can then be changed using any of the colour components.

The created / modified colour can be installed by selecting the **OK** button. No changes are made to the colour definition if the **Cancel** button is selected.

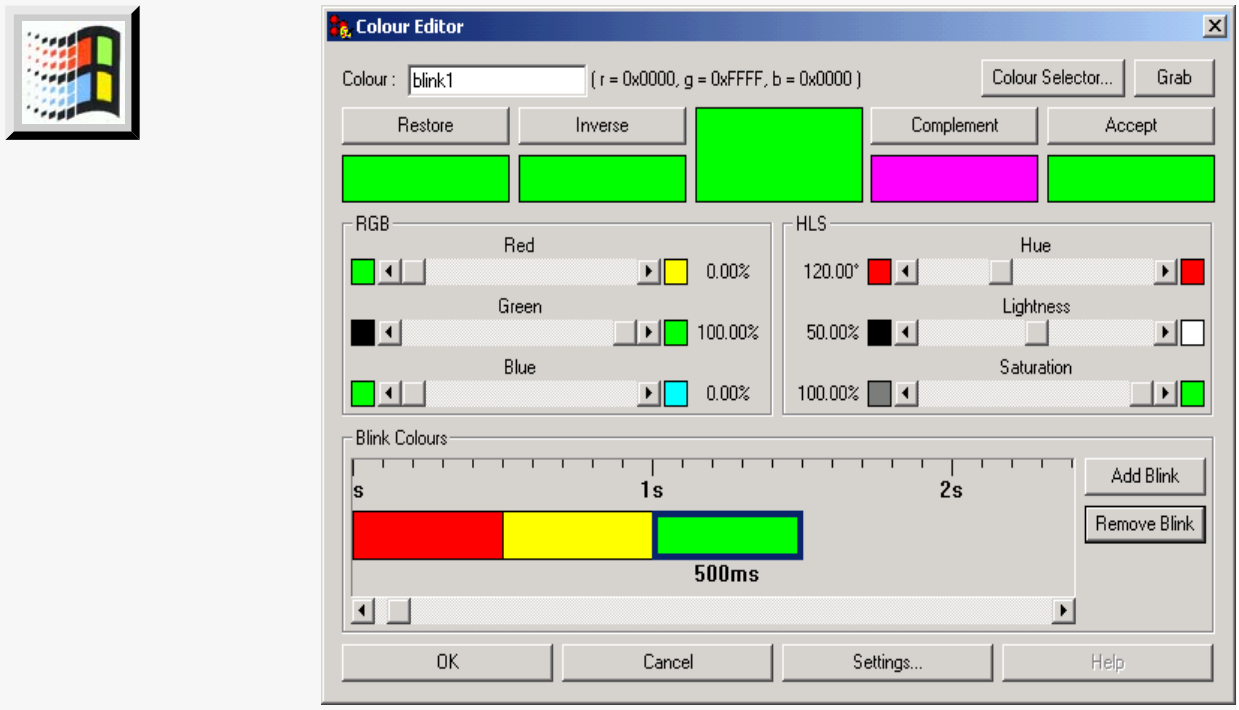


Figure 30 *The Colour Editor.*

Two additional buttons are implemented in the Windows versior the Colour Selector button and the Settings button.

By pressing the **Colour Selector** button in Figure 30 a standard Mi crosoft Windows Colour Selector dialogue will appear.See Figur 31.

By pressing the **Settings...** button in the colour editor a dialogu window will appear. See Figure 32.

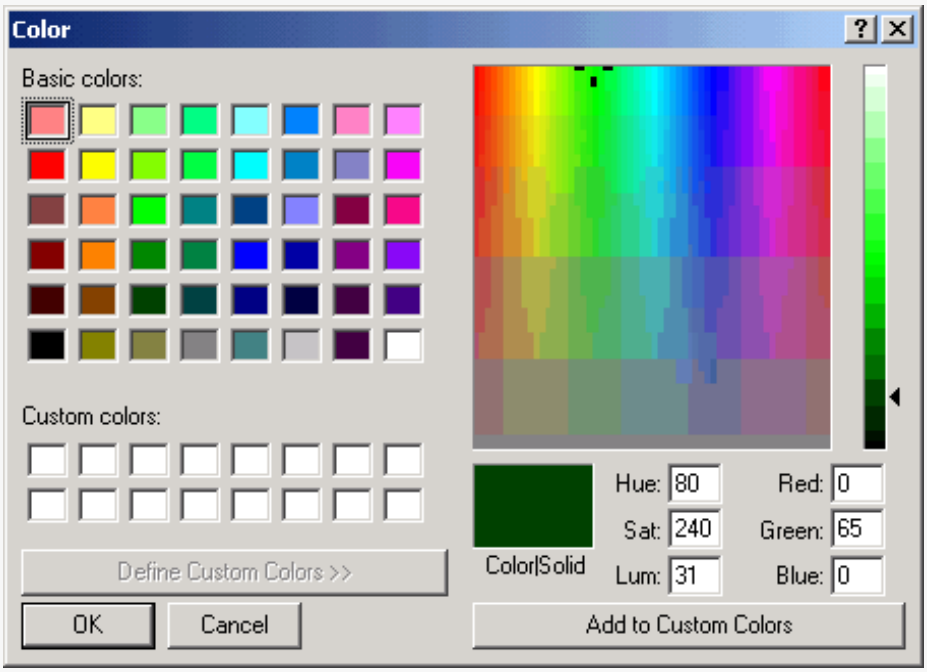


Figure 31 *The Microsoft window standard Colour Selector dialogue box.*

Clicking on the **Basic colours** will move a cursor round in the big colour palette rectangle to the corresponding colour. By clicking in the palette rectangle, different colours can be chosen. The colour is stored in **Custom colours** by clicking on **Add Custom Colours** button. The **Colour | Solid** rectangle is always presenting the target colour. **OK** button transfer the chosen colour into the Colour editor.

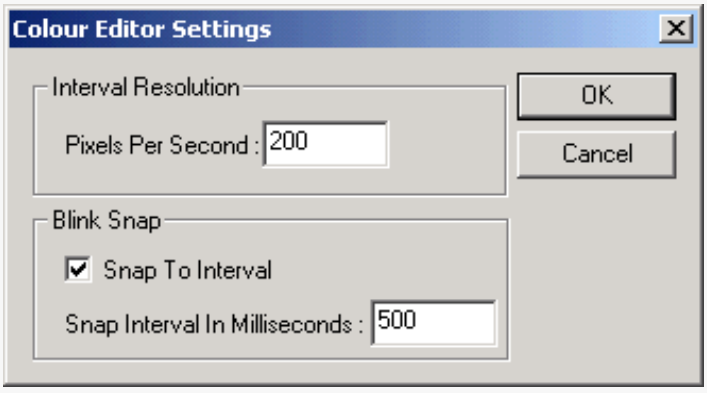


Figure 32 *The Colour Editor Settings dialogue.*

In this window you can adjust the length of the Blink Colours area and the snap interval on the ruler.

Interval Resoulution sets the resolution in pixels per seconds in the Blink Colour area in Figure 30.

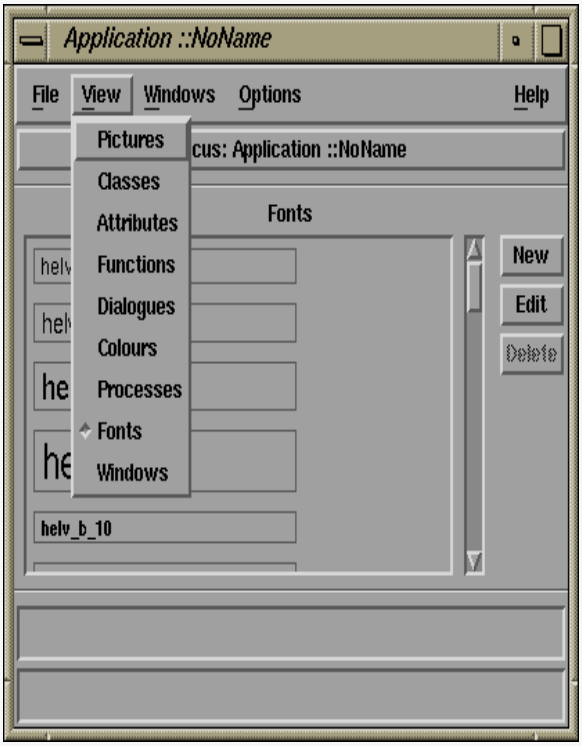
Blink Snap sets the ruler snapping in the Colour Editor Blink Colours window.

9.4 Defining Fonts in GED

To modify or create new fonts select the **Font** item in the view menu from GED's main window (Figure 33). The fonts available in the application will be displayed in a selectable list.

Figure 33

The figure shows GED's main window and how fonts can be selected from the View menu. Fonts can be created and modified by selecting the New or Edit button located to the right of the font list.



Three buttons for font manipulation is located on the right side of the font list. The delete button is not available. To edit an existing font select one of the fonts in the font list and press the **Edit** button or double click on the selected font. To create a new font just press the **New** button. If a font is selected in the font list the new font will be equal to the font just selected but with a new resource name. The created font will be displayed last in the font list.

The *Font Viewer* is a selection tool helping the user to find a proper font based on properties that the user enters. It is not an editor like the other resource tools in GED (colour- and window editor). No manipulation of character bitmaps or other properties are implemented.

The *Font Viewer* is divided into three parts (Figure 34). The upper part is where the user restricts the number of fonts listed in the system font list. The middle part is where the user selects a font from a list of fonts with the given properties that the user has specified. The last part is a preview of the current and last accepted font. First a detailed description of the upper part of the *Font Viewer*.

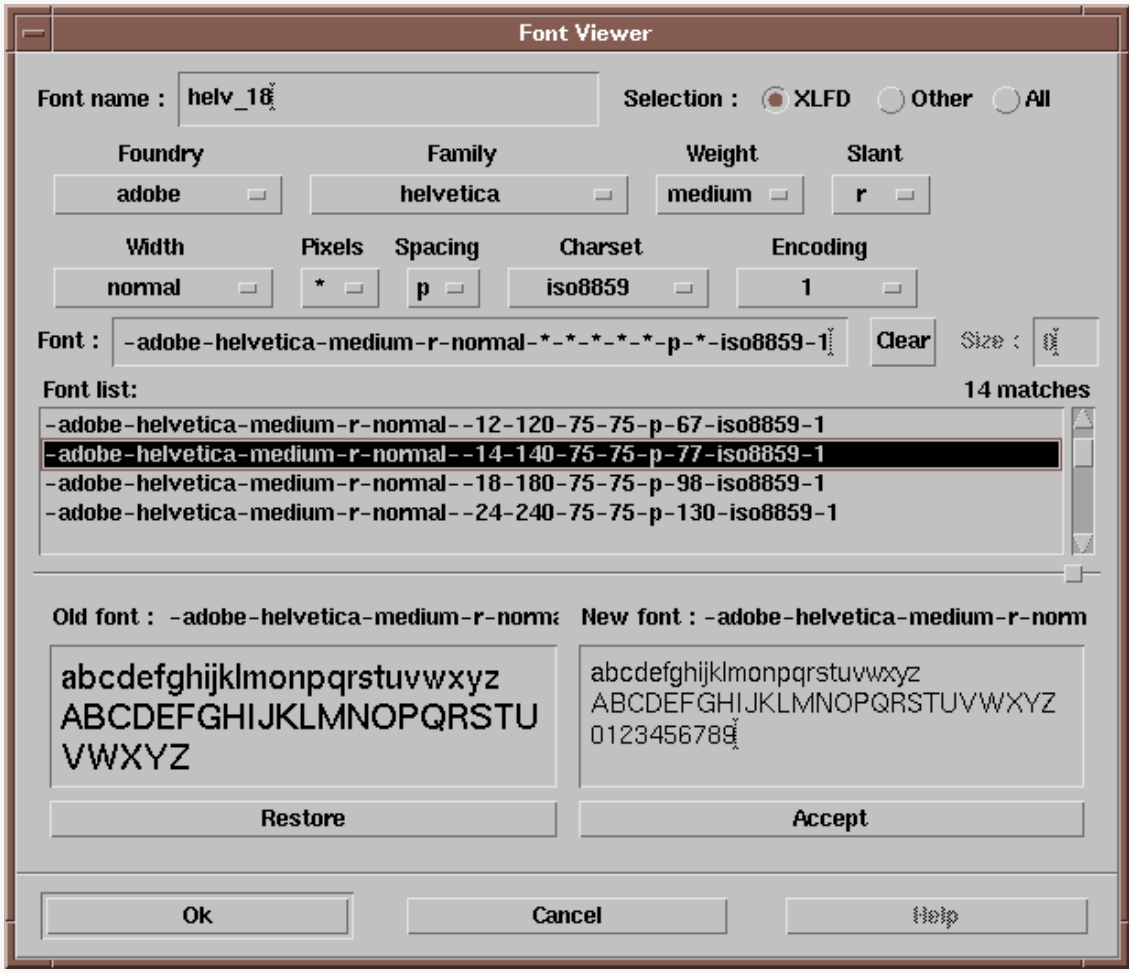


Figure 34 *The figure shows the Font Viewer. Both ordinary fonts and XLFD fonts are supported. In addition is scalable fonts supported where the font size is specified in the size input field. The area above the accept button (restore button) is a text input field displaying the font just selected. The text in this area can be changed by the user.*

The text input field in the upper left corner is where the name of the font resource is specified. This is the name that is used in the *rtm*.

The three radio buttons to the right of the text labelled *Selection* are used to specify the search method. If the radio button **XLFD** is selected the option menus specifying the font properties are enabled. XLFD fonts is an abbreviation for *X Logical Font Description* and is standard on all X-servers. Font names like *-adobe-helvetcia-medium-r-narrow--25-180-100-100-p-111-iso8859-1* is an example of an XLFD font, while the font *12x8* is not. The following properties can be specified using the option menus:

- Foundry
- Family
- Weight
- Slant
- Width
- Pixels
- Spacing
- Charset
- Encoding

By pressing one of the option menus below the text labels listed above, a set of available property fields will be enabled. Selecting one of the font property fields will narrow the number of fonts in the font selection list. If another option menu is selected only values for fonts which match the previous selected fields will be selectable. To make other fields available you must deselect some other properties by choosing the '*' entry for that field.

Pressing **Other** or **All** will disable the XLFD menu options and the text field labelled *Font* will be enabled. Any character entered in this input field will change the selectable font list. This text field accepts a limited regular expression, where an asterix (*) matches zero or more characters. The question mark (?) matches one character. The difference between the choices *Other* and *All* is that *Other* only displays non-XLFD fonts, while *All* means match all fonts available on the X-server.

Pressing the **Clear** button will clear all fields and all fonts for the specified search method will be available in the font list. The XLDF option menus will also be cleared, meaning that all property fields will be set to '*'.

The input field to the right of the label *Size* will only be enabled when a scalable font is selected. A scalable font is a font that can be rescaled to any size. To enable this field arm the **XLFD** radio button in the Selection group and select 0 in the option menu labelled *Pixels*. This field only accept digits. Digits entered in this text input field will immediately change the font in the text field above the **Accept** push button.

The middle part of the *Font Viewer* consists of a scrollable list of available fonts with the properties the user has entered. These fonts can be selected using either the pointer or the keyboard. Selecting a font in this list will change the font preview above the **Accept** button. The number of fonts matching the regular expression or the XLFD font specification will always be displayed to the right above the font list.

The bottom part of the *Font Viewer* is divided into an accept and a restore area. The accept area displays the font just selected in the font list. Pressing the **Accept** button will change the font in the *rtm* temporarily. Pressing the **Restore** button will restore the font in the *rtm* as it was before the **Accept** button was pressed.

The text fields above the **Accept** and **Restore** buttons are enabled and the text displayed in these areas are just an example text. This text can easily be modified.

The **Ok** and **Cancel** buttons located at the bottom of the *Font Viewer* make permanent changes to the font resource. Pressing the **Ok** button will accept the font resource and close the *Font Viewer*. Pressing the **Cancel** button will ignore all changes made in the viewer and restore the font to the value it had before the *Font Viewer* was started.

To create a new font, select the **Fonts** node in the Main window as shown in Figure 35, right click on it and select **New...**. If you expand this node you will see the already defined fonts as subnodes of the **Fonts** node. To edit an existing font, right click on the font name and select **Edit...**, or double click on the font name to display the *Font Viewer* (see Figure 36)

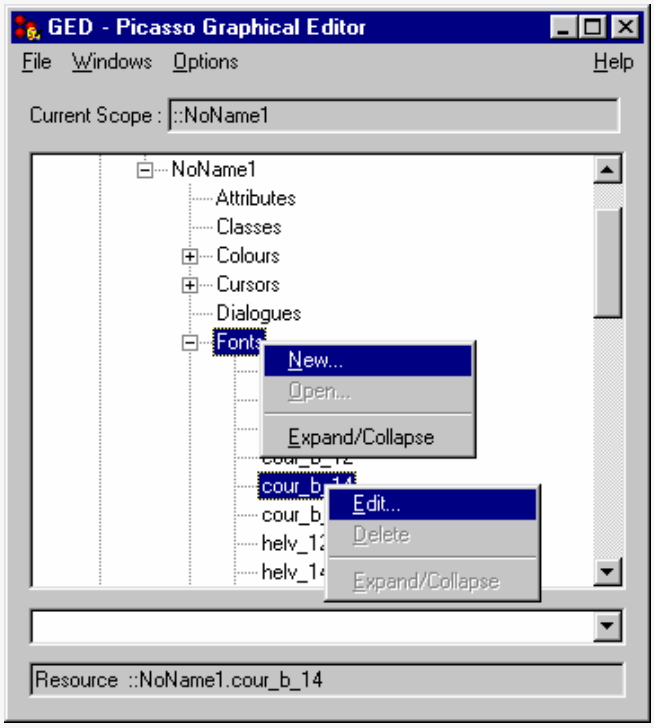


Figure 35
The figure shows GED's main window and how fonts can be selected from the Tree view. Fonts can be created by selecting New... and modified by selecting Edit... on the popup menu, and the Font editor (Figure 36 and Figure 37) will be displayed.

The *Font Viewer* is a selection tool that helps the user to find a proper font based on the properties selected from the available fonts

The *Font Viewer* consists of two different dialogues. The first one is a standard Windows *Font Selection Dialog*, with an extra field to enter the ProcSee name for the font (see Figure 36).

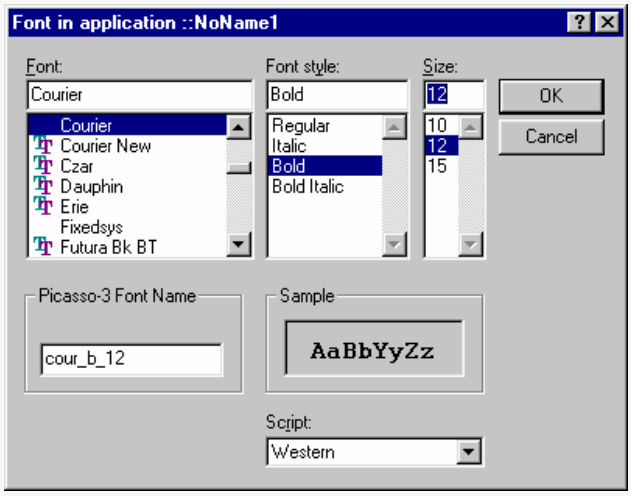


Figure 36
The figure shows the Font Viewer. All Windows fonts are supported. In addition XLFD and Non-XLFD Unix fonts are supported, but they use a different dialogue (see Figure 37)

To create a new font, just select the font wanted, and then select the proper style and size. Make sure to enter a ProcSee name for the font, to be able to use it in the system later.

The other part of the *Font Viewer* is the Unix->Windows conversion dialogue (see Figure 37). This will show the font as it was created on X-Windows if it can't find an exact match for the Unix font (for information on this, see the X-Windows specific part of this chapter). This font will be converted into the nearest matching Windows font when used, but it can be spesified to be an exact Windows font by pressing the **Choose font...** button and selecting the desired font from the *Font Selection Dialogue*. When this is done, the *Font Viewer* will open directly into the Windows *Font Selection Dialog* the next time you select to edit the font.

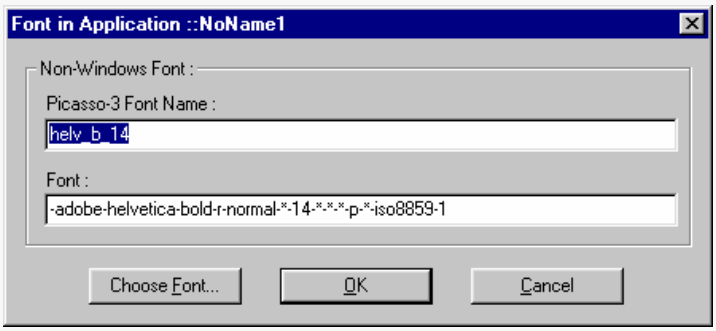


Figure 37
The figure shows the Font Viewer when you view fonts that are created on Unix. You can edit the Unix font or change it to be a proper Windows font.

You can also choose to edit the Unix-font by modifying the attributes in the Font text field and then pressing the **OK** button.

The **OK** button in the *Font Viewer*s make permanent changes to the font resource and closes the *Font Viewer*. and the **Cancel** button will ignore all changes made and restore the font to the value it had before the *Font Viewer* was started.

9.5 Defining patterns in GED

To create a new pattern, select the **Patterns** node in the Main window as shown in Figure 38, right click on it and select **New...**. If you expand this node you will see the already defined patterns as subnodes of the **Patterns** node. To edit an existing pattern, right click on the pattern name and select **Edit...**, or double click on the pattern name to display the *Pattern Editor* (see Figure 36)

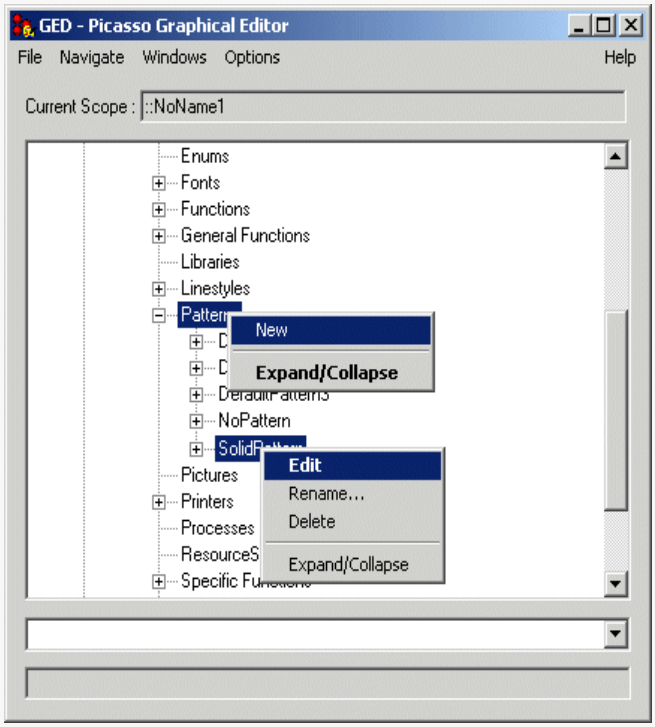
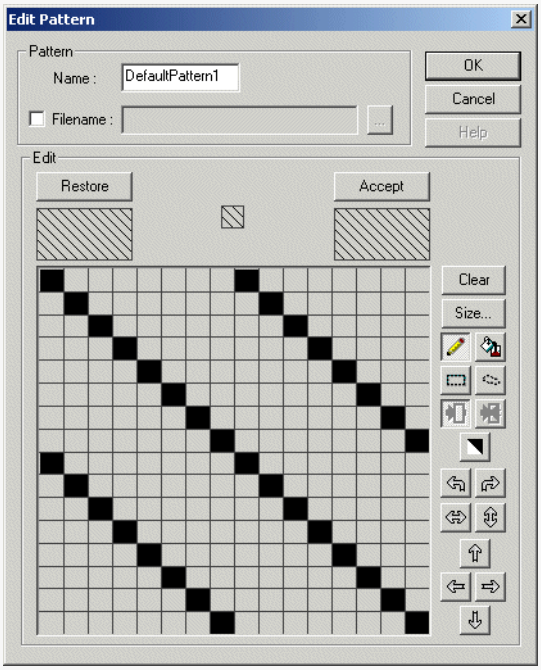


Figure 38
The figure shows GED's main window and how patterns can be selected from the Tree view. Patterns can be created by selecting New... and modified by selecting Edit... on the popup menu. The Pattern editor is shown in Figure 41.

Figure 39
The figure shows the pattern editor. The editor supports basic editing functionality, plus selection, rotation, etc.



The field **Name** is where the name of the pattern is specified. This is the name used in ProcSee and it must be unique within the application or library. This field must be filled in when a new pattern is created. When editing an existing pattern this field is filled in with the name of the pattern. The pattern will be renamed if the name is changed. Notice that the input field is disabled if the pattern is part of a resource style. For more information about resource styles, see section 9.9 "Resource Styles".

The pattern can be saved in a separate file or in the application .pctx or library .plib file. The pattern is by default not saved to a separate file. To enable this feature click on the check box located to the left of **Filename**. The text field to the right of **Filename** will be enabled together with the button labelled "…". Either fill in a file name or press the "…" button. This will open the standard file selection box in Windows.

The pattern editor has a preview feature which enables the users to view a small section of the pattern as the pattern is being edited. All changes made to the pattern will automatically be updated and displayed in the preview area. Another very useful feature is the ability to see how the pattern looks like when it is used in a ProcSee picture. Pressing the **Accept** button will create or modify the pattern in ProcSee. It is then possible to see the pattern used in its real environments. The **Restore** button will restore the pattern in ProcSee as it was before the **Accept** button was pressed.

To clear the pattern, press the **Clear** button. A question dialogue will be displayed where you must accept the action. To change the size of a pattern, open the **Pattern Size** dialogue by pressing the **Size...**button. The dialogue is illustrated in Figure 40. Enter the new size of the pattern and press the **Ok** button. The predefined sizes accessible from this dialogue are 8 by 8, 16 by 16 and 32 by 32. The pattern editor will try to retain as much of the old pattern as possible when the pattern is resized.

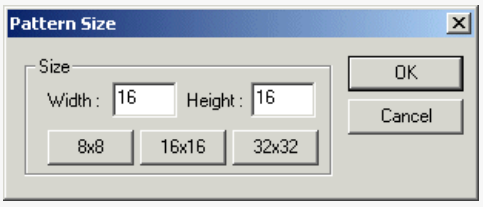
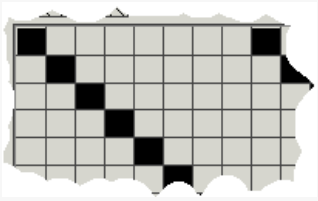


Figure 40 *This figure shows the pattern size dialogue which is used to change the size o a user defined pattern.*

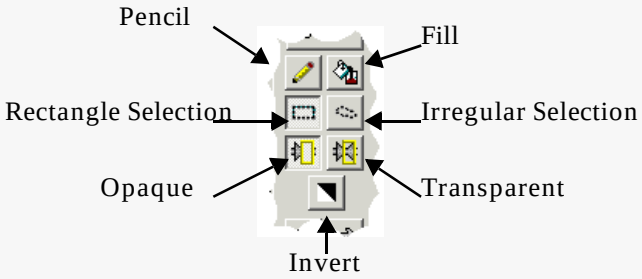
To edit the pattern press the left or right button in the editing area, see Figure 41.This is a rectangular area where black rectangles indicate that cells are on (foreground), otherwise off (background). To change the pattern just press the mouse using the left or right button in the cell you want to change. A left mouse button click will set the cell(s), while a right mouse button will clear the cell(s). This applies to both the pencil and the flood fill tool. See Figure 42. With the pencil tool only single cells can be set. The flood fill tool will replace all cells within a closed area. Notice that the entire pattern will be changed if the area is not closed.

Figure 41
This figure illustrates the pattern editing area. Black rectangles indicates that the cells are on.



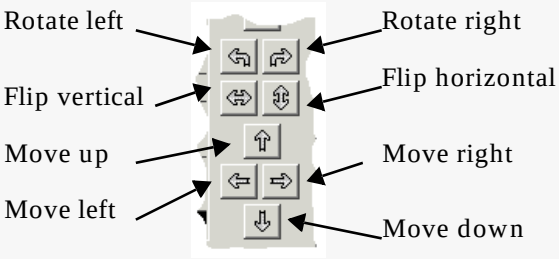
The regular and irregular selection tools are used to select parts of the pattern. The selected areas can only be used in a move operation. Select an area, press the left mouse button inside the area and move the selected area around inside the pattern. If the opaque button is on the background area will be moved as well. Select the transparent button if only the foreground cells should be moved. To invert the pattern press the invert button.

Figure 42 *This figure displays the editing tools available in the pattern editor.*



The arrow buttons located in the right bottom area of the pattern editor are used when rotating, flipping and moving the pattern.

Figure 43 *This figure displays the roation, flipping and move buttons available in the pattern editor.*



The **Ok** and **Cancel** buttons make permanent changes to the pattern resource. Pressing the **Ok** button will accept the pattern and close the **Pattern Editor**. The pattern will be saved to the file specified if the *Use file* check box is enabled. Pressing the **Cancel** button will ignore all changes made in the editor and restore any changes made to the pattern.

The pattern editor is not implemented on the X-Windows versions of GED. To add a new pattern on the X-Windows version use a text editor and add the pattern. For a description of the pattern syntax used in the Tdoc files, see the configuration language in the ProcSee Reference Manual.

9.6 Defining gradient patterns in GED

The general definition of a gradient is a smooth linear blending of two or more colours. A gradient in ProcSee can be used as fill- or line- patterns in shapes or as background in pictures. It can be used wherever regular patterns can be used. Gradient patterns are just like regular patterns. They have to be created in the symbol-table before they can be used. Actually, a gradient pattern is a special type of pattern, with the same meta-class, Pattern, and functions as regular patterns.

In ProcSee, a gradient pattern is a blending of colours, where each colour is associated with a given position, also called a blend stop, Figure 44. These blend stops produces a sorted blend list, where the sort key is the blend stop positions. These positions are in the range [0, 1], where the value 0 is first in the list and the value 1 is last in the list. Positions outside this range are illegal.

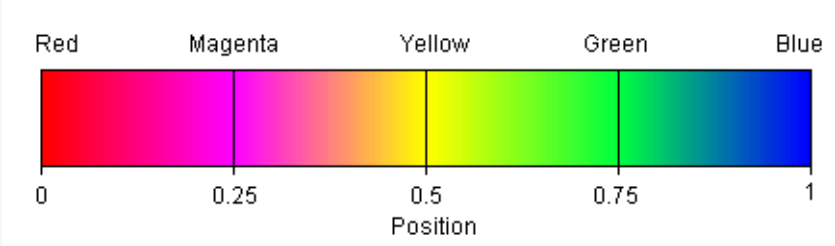


Figure 44 *This figure illustrates the relationship between the positions and the colours the gradient is composed of.*

Each blend stop is either a colour or a factor. A colour blend stop has the following properties:

- colour definition
- transparency
- foreground colour
- background colour

The colour property defines the RGB (red, green, blue) components of the colour, where each RGB component in the editor is displayed as a value in the range [0, 0xFFFF]. The transparency property determines whether the gradient is opaque, transparent or semi transparent. 0 means opaque, while 0xFFFF means transparent. Any value between these limits produces a semi transparent colour.

It is also possible to create a blend colour based on the colours of the shape or the picture the gradient is assigned to, and the selected colour. The ratio of mixing these colours is determined by the foreground- and background properties. In GED is these colour properties displayed in percentage. Notice that the sum of these colour properties cannot exceed 100%. If the sum is 100%, it means that the colour is got entirely from the shape or picture. Any value greater than 0% creates a blending of the selected colour and the shape or the picture colours. Users creating colour blends do not need to consider that values must be within certain limits, because GED does all the range checking. Therefore, it is recommended that gradient patterns are created using GED's built-in gradient editor.

A blend stop of type factor calculates a colour based on the previous and next colour in the blend list. The blending ratio is determined by the property, factor. GED displays values and accepts values in the range [0, 1]. A value of 0 and 1 means, use the previous and next blend stops in the blend stop list, respectively. Any value between these limits produces a blending of the next and the previous colour in the blend list. Blend stops of type factor in positions 0 and 1 are special cases. No blend factors exist at positions outside this range. Therefore, when using blend factors at these limits, the colour definitions are, determined by the property factor, from the shape's or picture's foreground or background colour. The colour definition is entirely or partly got from the shape's or picture's foreground attribute if the position is 0 and the property factor is unequal to 1, and from the background colour if the property factor is unequal to 0 in position 1.

When the gradient pattern is created, ProcSee automatically add blend stops of type factor if the position 0 or 1 has not been added to the blend list. Therefore, it is easy to create a two colour gradient pattern using a shape's or picture's foreground and background colour. Just create a new gradient pattern without adding any blend stops. ProcSee will automatically do that when it creates the gradient.

Note that a pTALK function is available to distinguish a regular pattern from a gradient pattern. The name of this function is Pattern::isGradient(). It returns the value 1 if the pattern is a gradient pattern, otherwise the value 0.

Creating a gradient pattern

To create a new gradient pattern, right click the Patterns node in the main window in GED and select New Gradient..., see Figure 45. This action opens a new Gradient dialogue editor as shown in Figure 46.

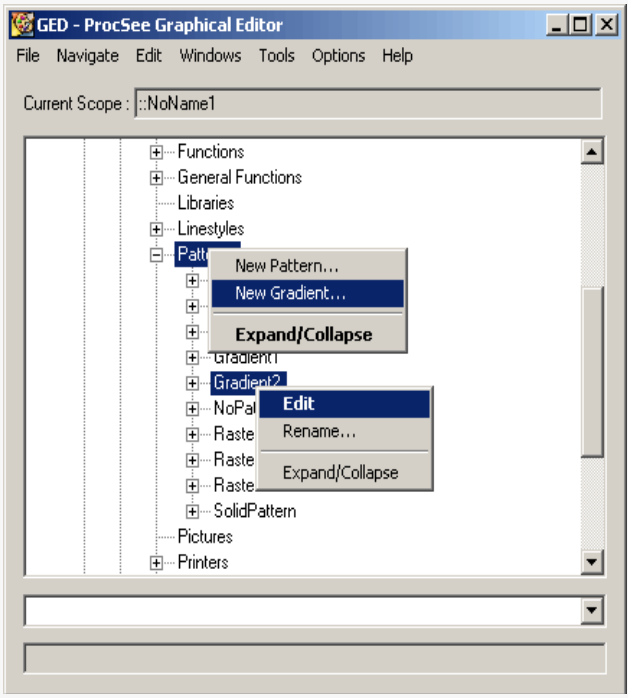
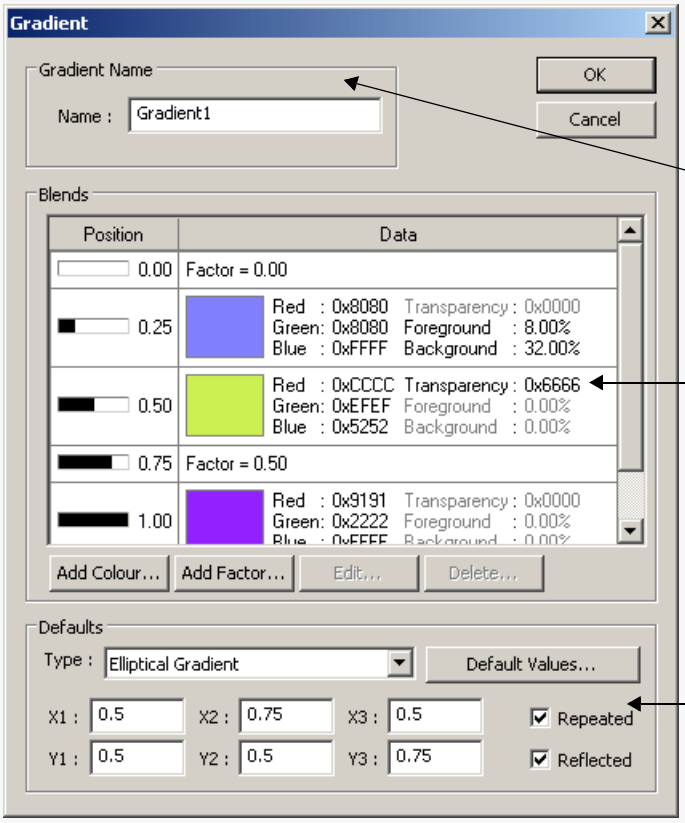


Figure 45
GED has built-in support for creating gradient patterns. To create a new gradient pattern right click on the Patterns node and select New Gradient... To edit an existing gradient pattern right click on the gradient pattern and select Edit... The gradient pattern editor is shown in Figure 41.

To edit an existing gradient pattern, expand the Patterns node and double click on a gradient node. Note that both regular patterns and gradient patterns are displayed under the Patterns node. When an existing gradient pattern is opened in the Gradient editor, the properties of the gradient pattern are displayed and editable in the editor. The exception is the name. When an existing gradient is edited it is not possible to change its name in the editor. Renaming can only be applied when selecting Rename… from the popup window when right clicking on a gradient pattern.



Upper part, where the gradient name is specified.

Middle part, where the blend stops are created and modified.

Lower part, where the gradient's default values are specified.

Figure 46
This figure shows the gradient editor in GED. The editor is divided in tree parts. The upper part is where the name is specified, the middle part is where the blend stops are created or edited, and the lower part is where the default values used by the gradient are specified.

The Gradient dialogue editor as shown in Figure 46 is divided in three main parts. Each part is described thoroughly in turn in the next sections. These are:

- Gradient name
- Gradient blends
- Gradient default values

Gradient name

In the upper part of the editor is the name of the gradient pattern specified. Type the name of the resource in the text input field next to the name label. The name is used to identify the gradient pattern in the pTALK gradient function, and is given as the first argument to the gradient function.

A shape's *fillPattern* attribute can be specified like this, if the name of the gradient pattern is *Gradient1*:

```
S.fillPattern = 'gradient( Gradient1 )';
```

For more information about the pTALK **gradient(...)** function, see the Gradient reference in the standard library section in the Reference Manual.

The OK button is disabled until a name is specified. Note that all gradient patterns must be given unique names. The input field is disabled when editing an existing gradient pattern.

Gradient blends

In the middle part of the Gradient editor is the gradient blend stops created and edited. A list is displayed, as shown in Figure 46, where each item in the list is either a blend colour or a blend factor. The items in this list are sorted in ascending order, where the key is the blend stop position. The positions are in the range [0, 1], where position 0 is first in the list and position 1 is last. The item positions are displayed in the left column in the list, and are common for blend colour and blend factor. The position is displayed as a bar and the value given in textual format. It helps visualize the position of the blend stop in the gradient. An empty bar indicates that the blend stop is in position 0, while a filled bar indicates that the item is in position 1.

The information displayed for each item depends on the blend stop type. The following information is provided when the item is of type blend colour, see Figure 47. In the column to the right of the position column is the colour information displayed. The selected blend colour is visualized in a rectangular area. The red, green and blue components of the selected colour, in hexadecimal digits, are displayed next to the rectangle. The column to the right provides textual information about the transparency, the foreground and the background attributes. The transparency is displayed in hexadecimal digits, and is in the range [0, 65536]. The foreground and background attributes are displayed in percentage.

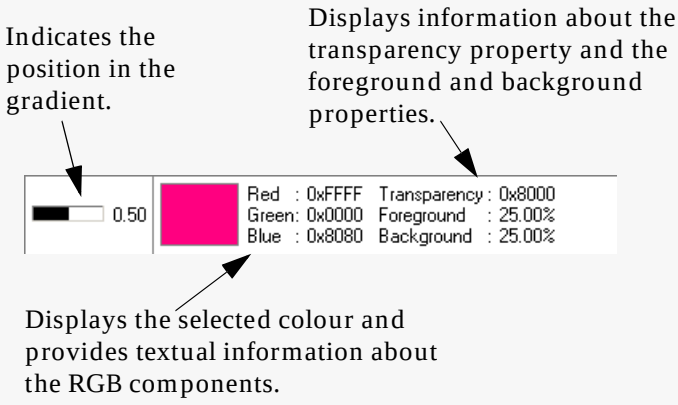
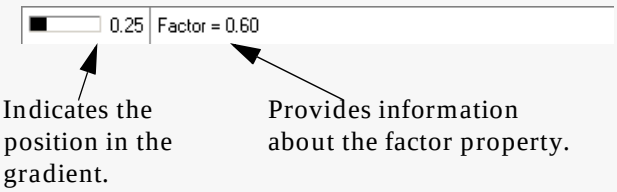


Figure 47
This figure illustrates the information provided when the blend stop is of type blend colour.

The following information is provided when the blend stop item is of type factor. The column to the right of the position column provides information about the blend factor. This factor is displayed in the range [0, 1], where 0 means use the colour of the previous blend stop, while 1 means use the colour of the next blend stop.

Figure 48
This figure illustrates the information provided when the blend stop is of type blend factor.



The row of buttons below the blend stop list provides functionality for modifying the items in the list, that is add, edit or delete. Pressing the buttons with the label Add Colour... or Add Factor... opens new dialogue windows where the user can add a new blend colour or a new blend factor, respectively. These editors are described next in this section. The next buttons named Edit... and Delete... are enabled only when one of the items in the list are selected. Pressing the Edit... buttons opens the blend colour dialogue editor or the blend factor dialogue editor. The type of editor opened depends on whether it is a blend colour or a blend factor item that is selected in the list. The editors used to edit a blend stop are the same used when creating a new one.

Creating a blend colour

The blend colour dialogue window is opened either when pressing the Add Colour... button or the Edit... button, while a colour blend stop item is selected in the list. This window is shown in Figure 49. The position slider and the input field defines the position in the blend stop list, and must be in the range [0, 1]. Either use the slider or use the input field to enter the value. The position is by default set to the value 0.5 when a new blend colour is created. The lower part of this window is divided in three main parts.

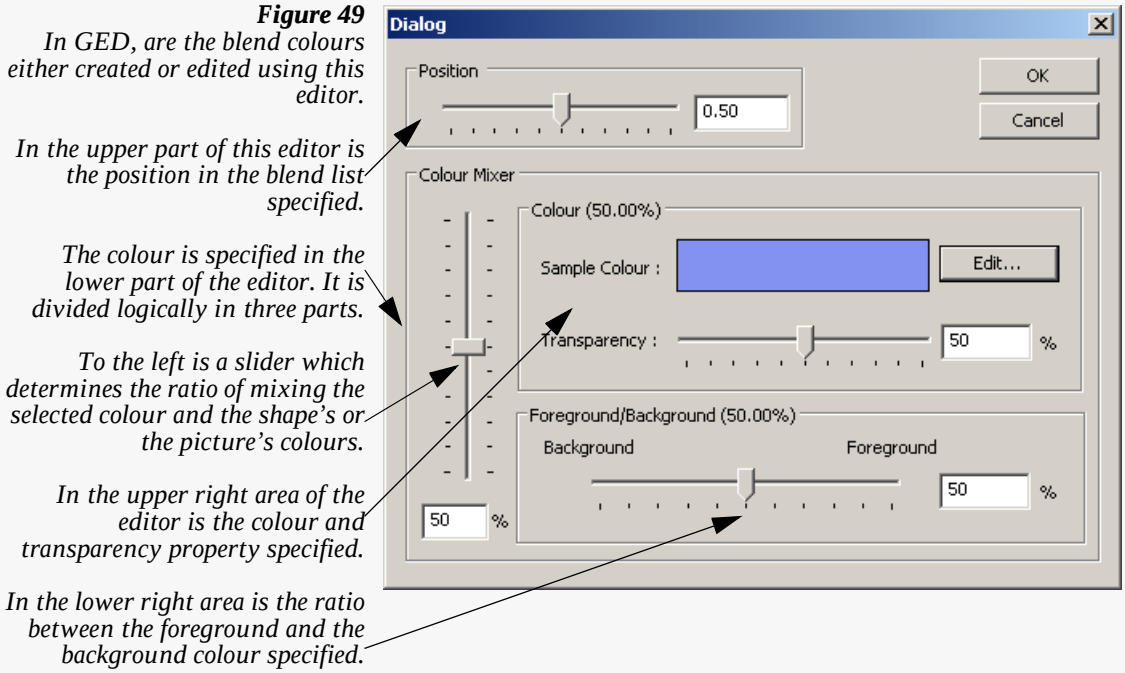


Figure 49
In GED, are the blend colours either created or edited using this editor.

In the upper part of this editor is the position in the blend list specified.

The colour is specified in the lower part of the editor. It is divided logically in three parts.

To the left is a slider which determines the ratio of mixing the selected colour and the shape's or the picture's colours.

In the upper right area of the editor is the colour and transparency property specified.

In the lower right area is the ratio between the foreground and the background colour specified.

To the left in this window are a colour mixer slider and an input field located. The colour mixer is used when calculating a colour based on the selected colour and a shape's or a picture's foreground and/or background colour. This value is in percentage and must be in the range [0, 100]. The colour of the blend stop is equal to the specified colour when this value is set to 100%, and to the shape's or picture's colours when this value is set to 0%. Any value between these limits produces a colour based on the selected colour and the shape's or the picture's colours.

The colour is specified in the upper right part of the colour mixer area. Pressing the Edit... button opens the Microsoft Windows Colour picker window. This window will not be described further in this User's Guide. For more information about this window, see Microsoft User's documentation. The selected colour is displayed in the selected colour area next to the Edit... button. The colour can be opaque, transparent or semi-transparent. The value of the transparency property can be changed by the transparency slider or the input field located below the selected colour area. This value is in percentage and must be in the range [0, 100]. A value of 0% produces an opaque colour, 100% produces a transparent colour. Any value between these values produces a semi-transparent colour.

In the lower right part of the colour mixer area is the ratio between the foreground and the background colour specified. Drag the slider or type the value in the input field next to the slider to change this ratio. A value of 0% means use the background colour, while a value of 100% means use the foreground colour. A colour is calculated based on both the shape's or the picture's foreground and background colour if the value entered is between these limits. The ratio depends on this value.

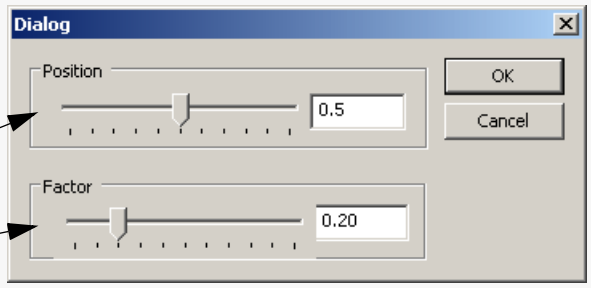
Creating a blend factor

To open the blend factor dialogue window, press either the Add Factor... button, or the Edit... button while a blend factor item is selected in the list. The window is shown in Figure 50. The upper part of this window is where the user changes the position of the blend factor. For more information about this, see the previous paragraph. The factor is set in the lower part of this window. To change the value of the factor use either the slider or enter a value in the text input field located to the right of the slider. The value entered must be in the range [0, 1]. For more information aboth this property, see section 9.6 "Defining gradient patterns in GED".

Figure 50
In GED, are the blend factor either created or edited using this editor.

In the upper part of this editor is the position in the blend list specified.

In the lower part is the blend factor specified.



Default values

In the lower part of the Gradient editor is the gradient pattern's default values specified, see Figure 46. These default values are used if they haven't been specified in the flags argument in the **gradient(...)** pTALK function.

The drop down field next to the type label is where the default gradient type is specified. Press the field or the arrow button next to the field to get the options for this property. The following gradient types are available:

- Linear gradient
- Elliptical gradient
- Circular gradient
- Rectangular gradient
- Square gradient

The default vectors used by the gradient are specified in the input fields located to the right of the labels named X1, Y1, X2, Y2, X3, Y3. These input fields specify two vectors. The vector coordinate (X1, Y1) are the vectors' origin. The first vector is (X1, Y1) to (X2, Y2), and the second vector is (X1, Y1) to (X3, Y3). These values are normally specified in the range [0, 1], where the value 0 is left or top, and the value 1 is right or bottom. This depends on whether it is the x or the y direction that is specified. The vector point (0, 0) is in the upper left corner of the shape, and the vector point (1, 1) is in the lower right corner of the shape. Note, when ProcSee calculates the size and direction of the gradient, it uses the shape's bounding box (not its shape, but a rectangular area) or the picture's dimension.

All input is initially set to default values when creating a new gradient pattern. These input values can be changed by the user. To change one or more of these coordinate vectors, click on an input field, and type a new value. Notice that the default value of each input field depends on the gradient type and the repeated and the reflected flags. These input fields are automatically changed when all input fields are set to their default values and the user changes the gradient type, the reflected flag or the repeated flag.

Whether it is four or six input fields available depends on the gradient type. Changing the gradient type to elliptical or rectangular enables the second vector's end point (X3, Y3). The second vector is applicable only for these gradient types.

To reset the input fields to ProcSee's default values, press the Default Values… button.

To enable or disable the gradient pattern's repeated or reflected flags, press the label or square box named repeated or reflected, respectively. When the repeated flag is enabled a gradient is created where the gradient is repeated until the shape or the picture is-filled using the gradient. The number of repetitions depends on the size of the vectors. When the reflected flag is enabled the gradient is reflected about the gradient vector(s). These flags do not change the visual appearance of the gradient if the vector(s) cover the bounding box of the shape or the picture.

9.7 Defining Linestyles in GED

To create a new linestyle, select the **Linestyle** node in the Main window as shown in Figure 51, right click on it and select **New...**. If you expand this node you will see the already defined linestyles as subnodes of the **Linestyle** node. To edit an existing linestyle, right click on the linestyle name and select **Edit...**, or double click on the linestyle name to display the *Linestyle Editor* (see Figure 52)

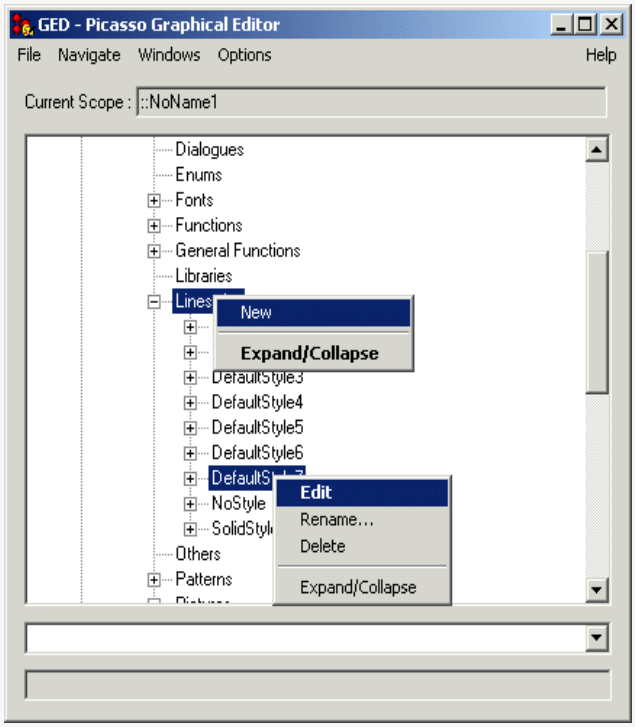
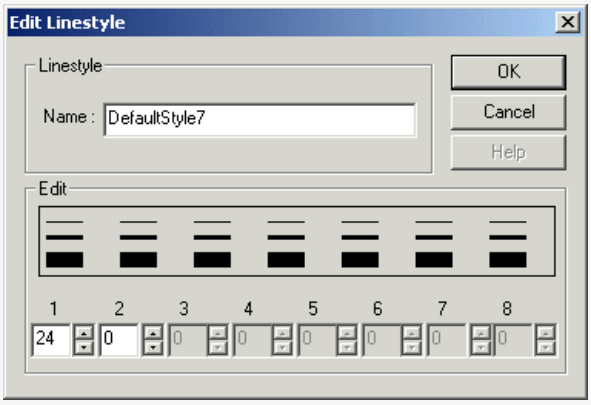


Figure 51
The figure shows GED's main window and how linestyles can be selected from the Tree view. Linestyles can be created by selecting New... and modified by selecting Edit... on the popup men. The Linestyle editor is shown in Figure 52.

The input field **Name** is where the name of the line-style is specified. This is the name used in ProcSee and it must be unique within the application or library scope. This field must be filled in when a new line-style is created. When editing an existing line-style this field is filled in with the name of the line-style. The line-style will be renamed if the name is changed. Notice that the input field is disabled if the line-style is part of a resource style. For more information about resource styles, see section 9.9 "Resource Styles"

Figure 52
This figure displays the line-style editor used in GED.



The edit area is composed of a preview area and eight editing fields. The preview area graphically displays current line-style using different line widths. This preview area will automatically be updated when the line-style is being changed.

The line-style is changed using one of the eight input fields or spin buttons. The first value specifies the length of the first dash used by the line, the second value specifies the length of the first space, and so on. A zero value indicates end of line-style. To specify a line-style with 5 dashes, 5 space and so on use 5 in the first input field and 0 or 5 in the next. The spin buttons have values ranging from 0 to 255.

The **OK** button in the *Linestyle Editor* will make permanent changes to the linestyle resource and close the *Linestyle Viewer*. The **Cancel** button will ignore all changes made and restore the linestyle to the value it had before the *Linestyle Editor* was started.

The linestyle editor is not implemented on the X-Windows versions of GED. To add a new linestyle on the X-Windows version use a text editor and add the linestyle. For a description of the linestyle syntax used in the Tdoc files, see the configuration language in the ProcSee Reference Manual.

9.8 Defining Cursors in Ged

To create a new cursor, select the **Cursors** node in the Main window as shown in Figure 53, right click on it and select **New...**. If you expand this node you will see the already defined cursors as subnodes of the **Cursors** node. To edit an existing cursor, right click on the cursor name and select **Edit...**, or double click on the cursor name to display the *Cursor Editor* (see Figure 36)

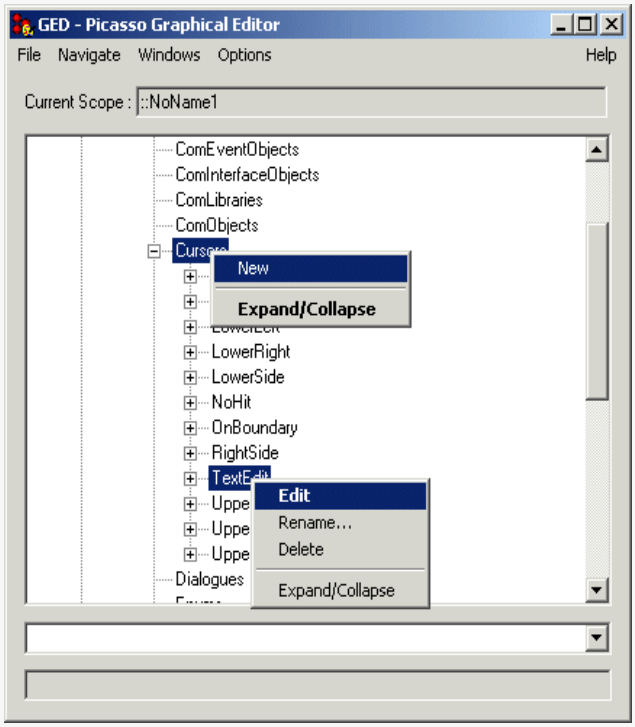
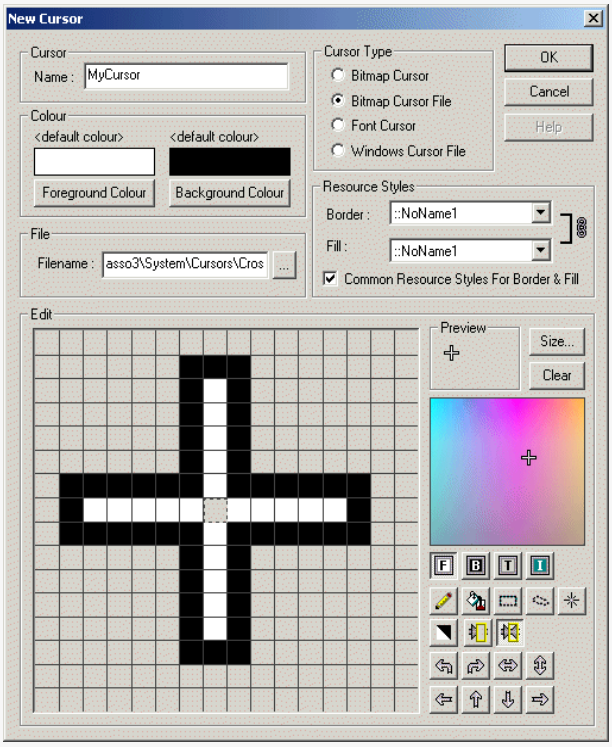


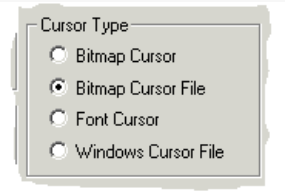
Figure 53
The figure shows GED's main window and how cursor can be selected from the Tree view. Cursors can be created by selecting New... and modified by selecting Edit... on the popup menu. The Cursor editor is shown in Figure 41.

Figure 54
The figure shows the cursor editor. Different cursor types can be selected. The editor also supports basic editing functionality, plus selection, rotation, etc., when editing cursor bitmaps.



The *Cursor Editor* displays various parts of the editor depending on the cursor type selected in the *Cursor Type* group. ProcSee supports all the cursor types listed in Figure 55. Left clicking on any of the radio buttons in this group will change the appearance of the *Cursor Editor*. Each of the different cursor types will be described in detail later in this section. Before continuing the common parts of the editor needs an explanation. These parts are fixed and will not change when the different cursor types are selected.

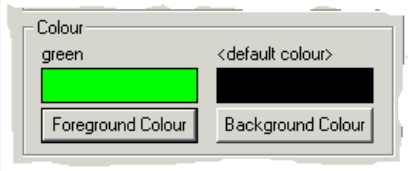
Figure 55
This figure displays the cursor types available in ProcSee. Change the cursor type by left clicking on the radio buttons.



The field **Name** is where the name of the cursor is specified. This is the name used in ProcSee and it must be unique within the application or library. This field must be filled in when a new cursor is created. When editing an existing cursor this field is filled in with the name of the cursor. It is not possible to rename the cursor from this dialogue. Renaming of cursor are done from the popup menu displayed when clicking on the cursor node in the tree-view in GED, see Figure 53. Notice that the input field is also disabled if the cursor is part of a resource style. For more information about resource styles, see section 9.9 "Resource Styles".

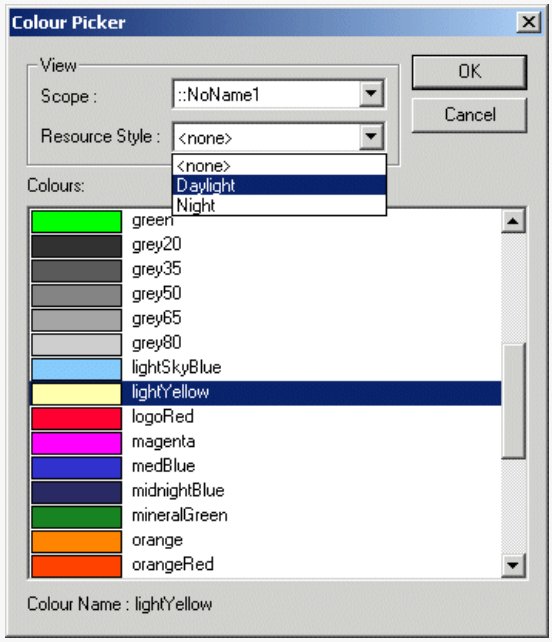
Cursors can be displayed both on Windows and X-Windows using different colours. This feature is also supported in the *Cursor Editor*. Notice however that colours can only be set when using bitmap cursors on Windows. The colour group where colours can be changed is shown in Figure 56. Default colours used by the cursors in ProcSee are black (background) and white (foreground). Current foreground and background colours are displayed in the two rectangular areas above the two colour buttons. Above the colour preview areas are the name of the colours. When default colours are used the name <*default colour*> will be displayed, otherwise the name of the selected colours. To change the cursor's foreground colour or background colour, click on the *Foreground Colour* or *Background Colour* button using the left mouse button. When one of these two buttons are pressed the *Colour Picker* dialogue shown in Figure 57 will be displayed.

Figure 56
This figure displays the colour group in the Cursor Editor.



The cursor's background and foreground colour can be changed from the *Colour Picker* dialogue, see Figure 57. This dialogue is divided into the following parts, *View* and *Colours*. The ProcSee scope and resource style can be selected from the *View* group. The scope can be changed to application or a library from the *Scope* combo-box. Use the *Resource Style* combo-box to change the resource style to use. In Figure 57 the following resource styles are available, *Daylight* and *Night*.

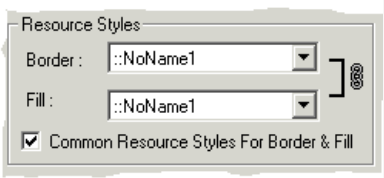
Figure 57
This figure shows the Colour Picker used in the Cursor Editor for selecting the cursors' foreground and background colour.



All colours defined in current scope are displayed in the list in the lower part of the *Colour Picker* dialogue. Each line displays the colour together with the colour name. Select one of the colours in the list using the left mouse button. The currently selected colour is displayed below the colour list. Press **Ok** or **Cancel** to close the dialogue. **Ok** will accept the colour selected.

The *Resource Style* group is used to visually see how the cursor looks like when using different resource styles, see Figure 58. The resource style will not be part of the colour name when the cursor is saved in ProcSee. If for instance the resource style is changed and the *Cursor Editor* is closed the resource style will be ignored. The next time the editor is opened the cursor will have the application's or library's colours and not the resource style which where set last time the cursor was modified. The cursor may have different resource style for border (foreground) and fill (background). Unselect the check-box "*Common Resource Style for border & fill"* to use different resource style for border and fill. For more information about resource styles, see section 9.9 "Resource Styles".

Figure 58
From the Resource Style group different resource styles can be selected to view how the cursor will appear when different styles are used in the application or the pictures.



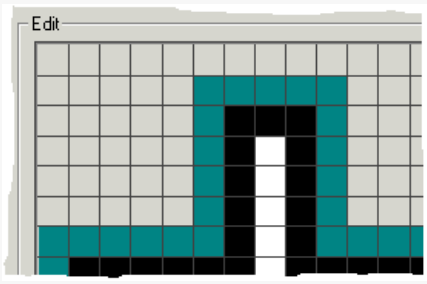
When *Bitmap Cursor* or *Bitmap Cursor File* is selected in the *Cursor Type* group the *Cursor Editor* dialogue will expand including an editor. The lower part of Figure 54 shows the edit area. The edit area can be divided into three parts, which are editing, preview and buttons. Each of these parts will be described below.

The preview area makes it possible to view every little change made to the cursor. All changes made to the cursor will automatically be updated and displayed in the preview area. The cursor is displayed in its' actual size. In addition is a preview test area displayed where the cursor can be tested on various colours. Just press down the left mouse button and move the cursor around inside the test area and the cursor will be moved to the mouse position. This enables the user to get an impression how the cursor will look like when different colours are used as background, see Figure 59.

Figure 59
This figure shows the preview area in the cursor editor. Use the mouse cursor to move the cursor to different positions in the preview test area to get an impression how the cursor will look like when used on various background colours.

To edit the cursor press the left or right button in the editing area, see Figure 60.This is a rectangular area where the cells can have two colours, foreground and background, plus transparent and inverse transparent. The colours are selected in the topmost button row, see Figure 61. To change the cursor just press the mouse using the left or right button in the cell you want to change. A left mouse button click will set the cell(s) to the selected colour, while a right mouse button will clear the cell(s) using the transparent colour. This applies to both the pencil and the flood fill tool. With the pencil tool only single cells can be set. The flood fill tool will replace all cells within a closed area. Notice that the entire cursor will be changed if the area is not closed.

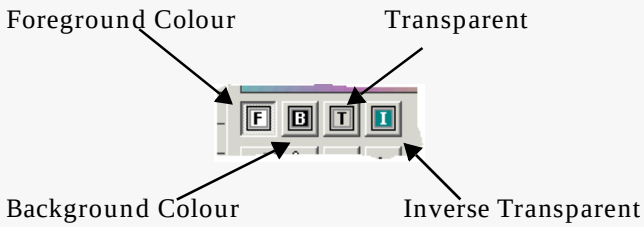
Figure 60
The editing area in the cursor editor. Pressing left mouse button will set the cell to either foreground, background, transparency or inverse transparency colour. Right mouse button will set transparency colour.



To clear the cursor, press the **Clear** button. A question dialogue will be displayed where you must accept the action. To change the size of a cursor, open the **Cursor Size** dialogue by pressing the **Size...**button. The dialogue is illustrated in Figure 40. Enter the new size of the cursor and press the **Ok** button. The predefined sizes accessible from this dialogue are 8 by 8, 16 by 16 and 32 by 32. The cursor editor will try to retain as much of the old cursor as possible when the size is changed.

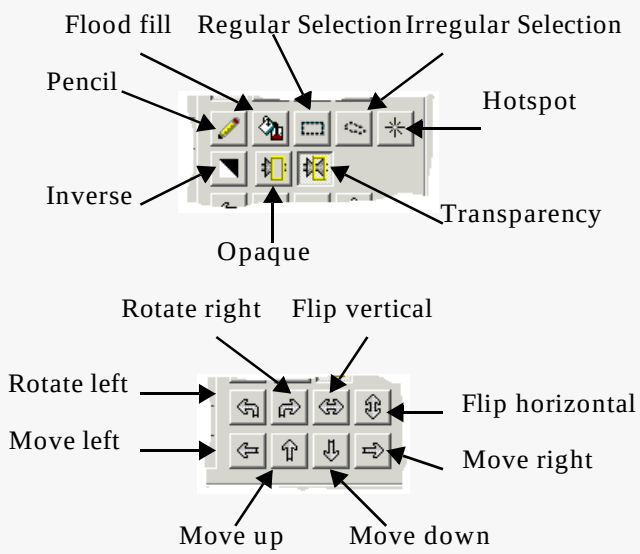
The colours used when editing the cursor are displayed in the first button row, see Figure 61. The button marked *F* is the foreground colour. In the editing area and the preview area the colour selected for the foreground colour will be used when displaying the cursor. The same applies for the next button marked with a *B* for background. The next button marked *T* is the transparency colour and the last button marked *I* is the inverse transparency colour.

Figure 61
The colour button row in the cursor editor.



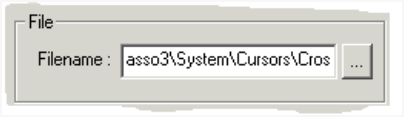
The regular and irregular selection tools are used to select parts of the cursor. The selected areas can only be used in a move operation. Select an area, press the left mouse button inside the area and move the selected area around inside the cursor. If the opaque button is on the background area will be moved as well. Select the transparent button if only the foreground cells should be moved. To set the cursor's hotspot x, y position, select the hotspot button and use the left mouse button to select a hotspot position in the editing area. To invert the cursor press the invert button. The last two button rows contain buttons for rotation, flipping and move operations.

Figure 62
This figure gives a graphical and textual description of the buttons available in the cursor editor



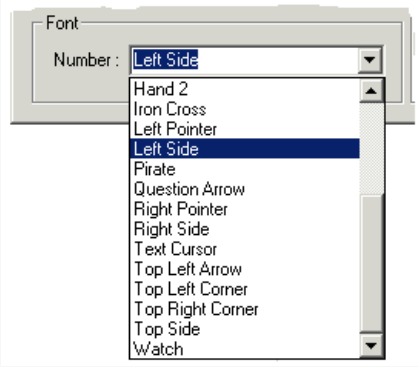
When *Bitmap Cursor File* is selected in the *Cursor Type* group an input field will be displayed where a file name must be entered, see Figure 63. The cursor will be saved to this file when the **Ok** button is pressed.

Figure 63
Enter the name of the bitmap file or the Windows cursor file in the input field. Pressing the button marked ... will open the Windows standard file selection box.



The predefined font cursors will be displayed in a combo box when selecting *Font Cursor* in the *Cursor Type* group box. Select one of the cursors from the list or enter an integer value in the input field, see Figure 64.

Figure 64
Select a cursor from the list or enter an integer value in the input field.



The radio button marked *Windows Cursor File* in the *Cursor Type* group box is Microsoft Windows specific. An input field where the Widows file can be entered is displayed when this radio button is selected, see Figure 63. In the input field enter the name of a Windows .cur or .ani file.

The **Ok** and **Cancel** buttons make permanent changes to the cursor resource. Pressing the **Ok** button will accept the cursor and close the **Cursor Editor**. The cursor will be saved to the file specified if *Bitmap Cursor File* is selected in the *Cursor Type* group. Pressing the **Cancel** button will ignore all changes made in the editor and restore any changes made to the cursor.

The cursor editor is not implemented on the X-Windows versions of GED. To add a new cursor on the X-Windows version use a text editor and add the cursor. For a description of the cursor syntax used in the Tdoc files, see the configuration language in the ProcSee Reference Manual.

9.9 Resource Styles

Sometimes the resources defined in the application or library should have other values than the default. One example is when a picture is to be printed on a black and white printer. Instead of changing the global resources before printing, and then changing them back, ProcSee has a consept of resource styles. This makes it possible to change a set of resources for a specified operation, like exporting to PostScript for printing or export to image. One can also set a resource style on a specific window for display on the screen, or one could change the resource style globaly.

A resource style contains resources with the same names as the global resources that should be overridden. An example is if a global colour named pictureBackgroundCol is gray, but should be displayed as white, a resource style containing a colour named pictureBackgroundCol with white colour could be created. Then this resource style could be specified in a parameter to e.g. the exportPostScript function, or in the useResourceStyle function on the application, or Window.

A resource style can use other resource styles, so that not every resource have to be specified if there already exists a resource style with the same resource modifications as you want. An example of this is when you have a resource style with grays for all the colours, and you want some colours, like alarm colours to be coloured, you can create a new resource style with only the alarm colours defined, and use the resource style with grays from the new resource style.

If a resource style uses more than one resource style, the overriding resource will be taken from the topmost resource style specified where this resource is found.

On X-Windows, the resource styles must be defined in the .Tdoc files. Refer to *"The Configuration Language"* in the ProcSee Reference Manual.

On Microsoft Windows, resource styles can also be defined in GED.

To create a new resource style, select the ResourceStyles node in the Main window of GED as shown in Figure 65, right click on it and select New... In the window that pops up (see Figure 66), you should give a name to the new resource style. You can also use other existing resource styles in this resource style, by moving existing resource styles from the Available list to the Use list at the bottom of the Window with the << button, or don't use it with the >> button. The order of these can be changed by using the Up and Down buttons. When you click on the OK button in this window, you have created an empty resource style.

Figure 65
The ResourceStyles node in the main window of GED.

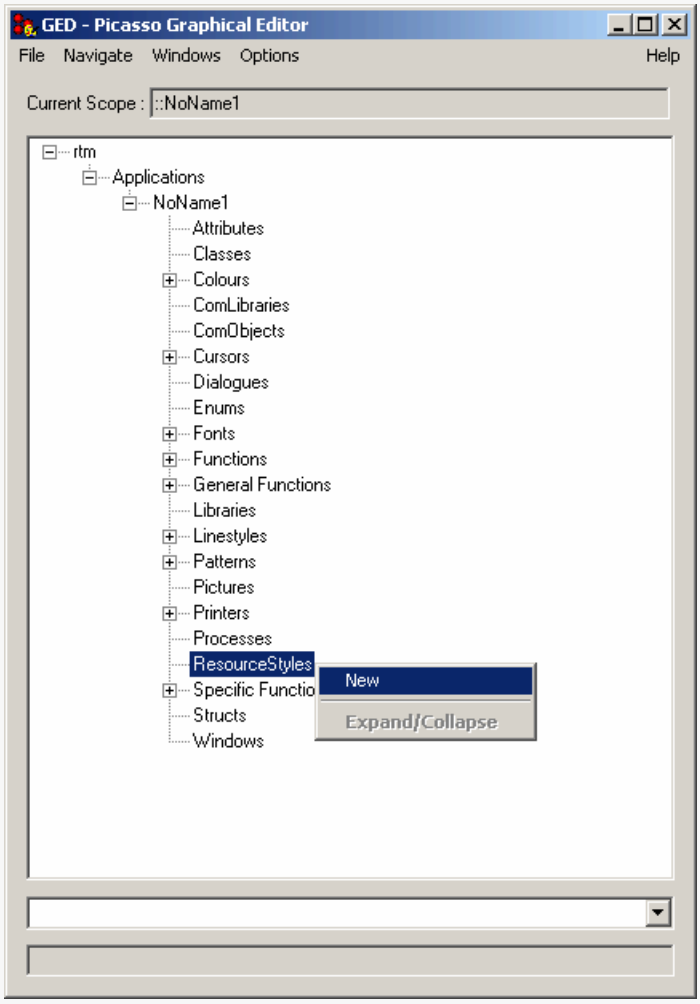
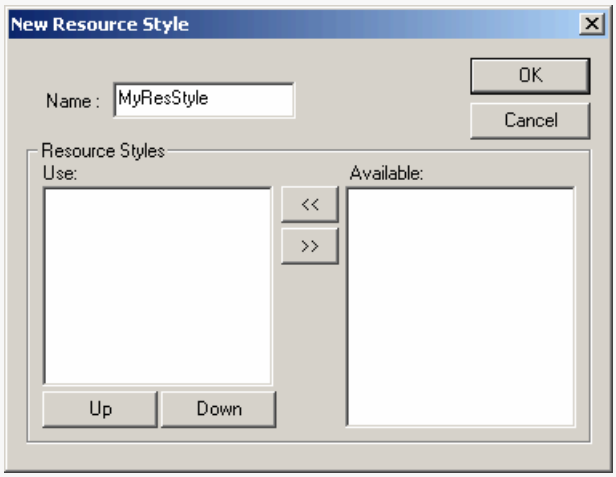
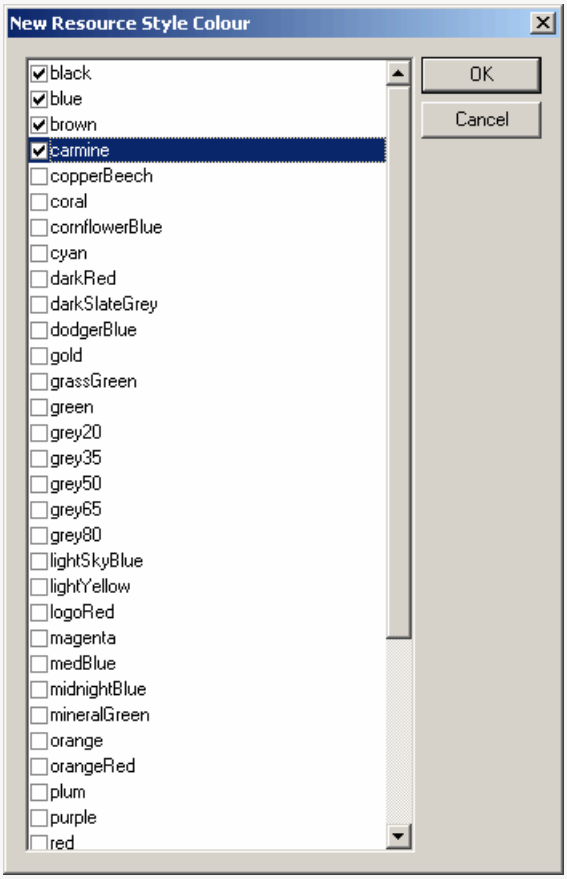


Figure 66
The New Resource Style Window.



To fill the resource style with the resouces that should be overridden, right click on the Colour heading under the ResourceStyle you added, and select new. The window that pops up (See Figure 67), allows you to select the colours you want to override in this resource style. This can be done multiple times, to add more colours to the resource style.

Figure 67
The New Resource Style Colour Window.



When you click on OK, the resources will be added to the selected resource style. You can then change each resource, like the global resources, by right clicking the resource under the resource style in the main window tree in GED, and selecting edit from the pop-up menu. You can also remove resources from the resource style, by right clicking the resource, and selecting Delete from the pop-up menu.

9.10 Defining Windows in GED

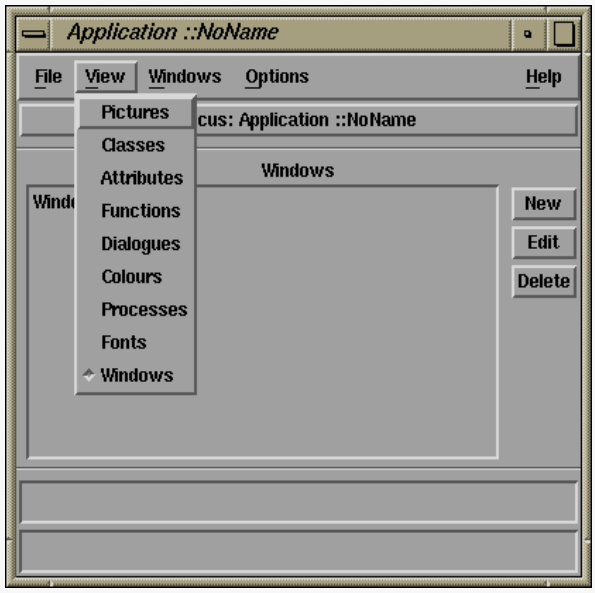
To be able to present a picture created in the drawing editor you need a window. A window is a dedicated area on the screen for an application to be displayed. All windows and icons on the screen

are managed by the systems window manager. The system's window manager program handles the window positioning on the screen, which functions to allow on windows, and the window decoration.

In GED there is a specialised *Window Editor* for creating new windows and modifying existing ones. To modify or create a new window, select the **Windows** item in the **View** menu from GED's main window (see Figure 68).

Figure 68

The figure shows GED's main window and how the Windows item can be selected from the View menu. Windows can be created and modified by selecting the New or Edit button on the right side of the window, and the Window Editor (Figure 70) will be displayed



In the select-area in GED's main window a list of available windows is displayed for the current application. If a new window is desired press the **New** button. The *Window Editor* will now be displayed containing default values, including the window name that is the string "*Window*" concatenated with a serial digit. For editing a window one of the displayed windows listed in GED's main window have to be selected before the **Edit** button is pressed. This will bring up the *Window Editor* containing the settings assigned for the actual window (see Figure 70). For deleting an existing window, select the actual window from the listed once and press the **Delete** button.

In GED there is a specialised *Window Editor* for creating new windows and modifying existing once. To create a new window, select the **Windows** node in the Main window as shown in Figure 69. Right click on the node and select **New...** from the popup menu.

If you expand the **Windows** node you will see the already defined windows as subnodes of the **Windows** node. To edit an existing window, right click on the window name and select ***Edit...***, or double click on the window name. This will bring up the *Window Editor* containing the settings assigned for the selected window (see Figure 70)

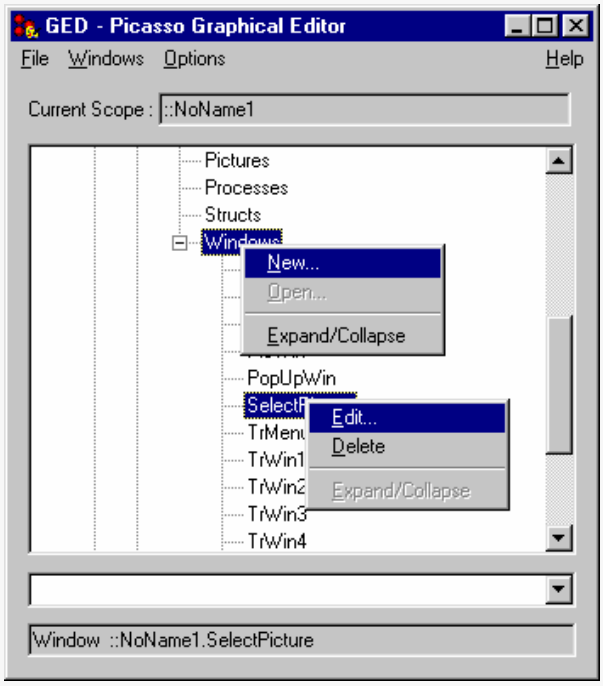


Figure 69
The figure shows GED's main window and how windows can be selected from the Tree view. Windows can be created by selecting New... and modified by selecting Edit... on the popup menu, and the Window editor (Figure 70) will be displayed.

If a new window is created, the *Window Editor* will now be displayed containing default values, including the window name that is the string "*Window*" concatenated with a serial digit. For deleting an existing window, select the actual window from GED's main window, right click on it and select **Remove** from the popup menu.

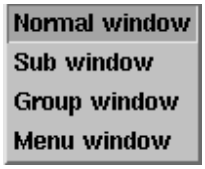
Figure 70

The Window Editor to the right presents the current setting for a normal top-level window owned by the running application loaded in GED.

In this editor all the window's attributes can be defined dependent on which type of window is selected and which parent it is connected to.

The editor is divided into four separate sections in addition to the three buttons at the bottom. The topmost section presents the window name aligned with the two buttons **Restore** and **Accept**. The **Restore** button brings the settings back to the most resent accepted setting carried out by pressing the **Accept** button. Hence the **Accept** button sends the current settings to the window manager to be reflected in the actual window.

The second topmost section contains; the type setting for the window, the parent to connect to, and the window title. In addition the window's geometry (x,y,width,height) can be redefined in the four input fields aligned to the right in this section.



The *Type* attribute controls the behaviour of the window. Four types are available

- *Normal window*: is the default window type. A top-level window is entirely controlled by the window manager where all the advanced settings can be manipulated.
- *Sub window*: is controlled by a parent window and have no decoration or functions attached to it.
- *Group window*: is controlled by a group owner that have to be specified in the *Group* attribute. Have no minimize button, which means that only the group-owner can minimize the group. These windows will also appear on top of the owner.

- *Menu window*: is not controlled by the window manager, hence no window decoration is available. Used in pull-down or pop-up menus that will appear only for a relative short period of time.

The **Parent** attribute should be the name of a display, or the name of another window. The **title** attribute contains the title to show in the frame of the window. If the **title** attribute is empty, the window will display the name of the picture displayed in the window, or the name of the window if no picture is displayed in the window.

For more information on these attributes and on types of window managers in use, see the *ProcSee Reference Manual* on the *Window* topic in section "*The Configuration Language*".

The third section displays the current settings for the four attributes **Decoration**, **Functions**, **Input mode**, and **Cursor**. In addition there is an "**Advanced...**" button for manipulation on the actual properties for each of the attributes. When pressing this button the window "*Advanced Window Settings*" will appear (see Figure 71). The **Decorations** and **Functions** attributes in the main *Window Editor* window states the current settings done in the advanced window. In Figure 70 the properties setting for both the **Decorations** and **Functions** attributes are **ALL**. Which means that all decorations and functions are *on* for this specific top-level window. Again this is further explained in the *ProcSee Reference Manual* on the *Window* topic in section "*The Configuration Language*". The two next attributes, **Input mode** and **Cursor**, directly maps to the setting in the advanced window.

The second last section presents a row of buttons to carry out operations on the created or modified window. The **Map** button displays a newly created or iconified window on the screen or inside a parent window. The reverse operation is carried out with the **Unmap** button. The **Iconify** button makes an iconic presentation of the window some obvious place on the desktop. And the **Raise** and **Lower** buttons places the window on top or behind all windows on the screen. These window operations are functionally the same as the functions that can be selected from the menu appearing when activating a dedicated area on the window frame (if it's a *Normal* or *Group* type of window).

To exit the *Window Editor* press the **Ok** button and all the changes made to a window will take place. On the other hand if the **Cancel** button is pressed all the changes will be discharged even if an **Accept** operation has taken place.

Figure 71

The window "Advanced Window Settings" to the right presents the current setting for a normal top-level window.

This window have all the decorations on and all the functions are available.

The cursor type can be selected from a list of system dependant cursors.

The "*Advanced Window Settings*" form in Figure 71 have properties that can be toggled on and off for both the **Functions** and the **Decorations** attributes. These properties are not available if the window type is either sub window or menu window. The **Cursor type** selection menu consists of the cursors defined in the current *RTM* application *(see section 9.2 "Defining Resources")*. When the mouse pointer enters the window the cursor will change to the selected type.

Note that the changes made in the "*Advanced Window Settings*" form will not be applied to the window before the **Accept** button in the *Window Editor* is pressed.

The **Input mode** attribute in Figure 71 is by default *Modeless*. This is the standard mode where all windows can receive keyboard and mouse inputs from the system. The option *System mode* defines that the window manager restricts all inputs to this window (note that this option depends on the window manager). The two middle options are not available in the RTM.

The attribute **Background colour** changes the colour in the window. A value of -1 means that the background colour of the picture displayed in the window is used. This is the default setting. The window's background colour will only be used if no picture is displayed in the window or the size of the picture is smaller than the window size.

To see the effects of the changes made in the *Window Editor* just click the **Accept** button. Hence there is no need to close the window to verify the new settings.

To see the effects of the changes made to the window, click the **Close** button in the *Advanced Window Editor*, followed by a click on the **Accept** button in the Window Editor. The last action will apply the changes made to the window.

9.11 Display

To create a new Display object, right click on the Displays node in the main window in GED and select New… from the popup menu (see Figure 72). To edit an existing Display object, expand the Displays node and right click on the name of the Display object to edit. From the popup menu select Edit…. Both of these actions open the Display editor in GED (see Figure 73).

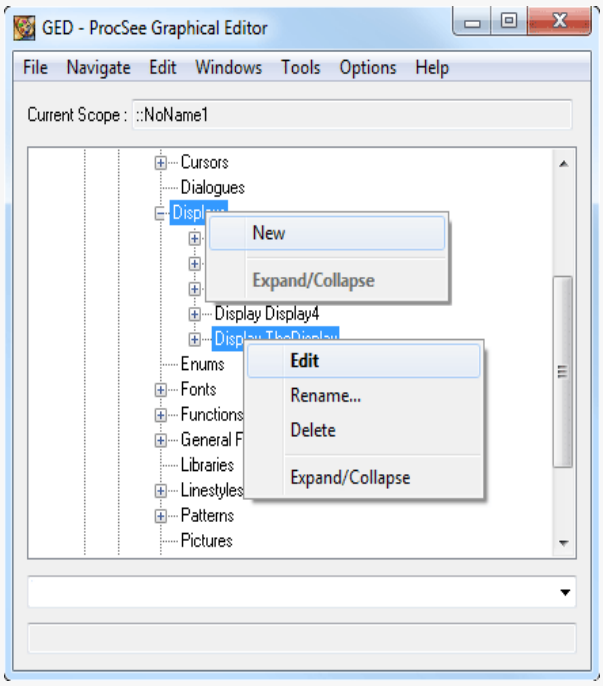


Figure 72 *This figure illustrates the New... and the Edit... Display object's pull-down menus which is displayed when right clicking on the Displays node and the display name node, respectively.*

The Display editor is divided in five parts. Each of these will be described thoroughly in this section.

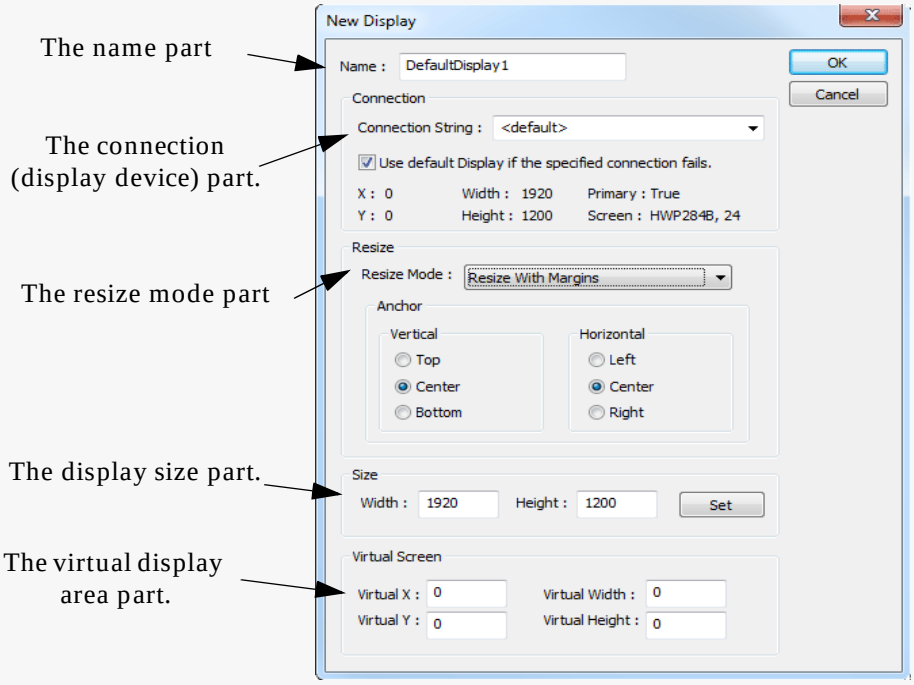


Figure 73 *This figure shows the Display editor in the graphics editor.*

The name section is located in the upper part of the editor. This is where the name of the Display object is entered. Notice that the name input field is read only when editing an existing Display object, while this name is mandatory and have to be specified when a new Display object is created. To rename an existing Display object, select Rename... from the popup menu when right clicking on the display name in the main window in GED. The name of the Display object must be unique within the application scope. Notice that the ProcSee RTM automatically creates default displays if an application not supporting the Display object is loaded into the RTM. The names of these displays are DefaultDisplay followed by a sequence number, like DefaultDisplay1, etc.

The display device is specified in the connection section. The drop-down list located to the right of the connection string label provides all the displays available on the current machine. Select one display from this list or enter a display device string in the input field. Information about the selected display is shown in the area below the connection string field. The information provided is the position, size and whether the display is the primary display or not. In addition the monitor name and the size in inches is displayed. The change made to the connection string will modify the Display object's *connection* attribute. Select the check box "*Use default Display if the specified connection fails*" to fallback to a default display if the specified connection string isn't a legal display device on the running machine.

Different resize modes can be specified in the resize section. These changes will modify the Display object's resizeMode attribute. A resize mode can be selected in the drop-down list located to the right of the Resize Mode label. All the resize modes supported by ProcSee is provided in this list. The resizing functionality affects all the windows and the pictures associated with the Display object. Note that it doesn't have any effect unless the display size is specified.

Selecting No Resize (0) means that the display (windows, pictures) maintains its' original size (design time size), i.e. no rescaling.

Select Resize With Margins (2) to rescale the display and maintain the aspect ratio. This mode resizes the Display object to fit the display size (or virtual display area). The Anchor section is enabled when this resize mode is selected. The default is that the display is anchored to the center of the display, both in the vertical and horizontal direction. This default can be changed by selecting Top, Center or Bottom in the vertical anchor button group, or Left, Center or Right in the horizontal button group. The enum AnchorStyle defines these constants which can be used programmatically from the pTALK language. To change the resizeMode attribute from the pTALK language, take the value 2(Resize With Margins) and perform a bitwise or operation between this value and the constants found in the enum AnchorStyle.

Select Resize No Margins (1) to resize the Display object to the display size (or virtual display area). This mode doesn't preserve the aspect ratio if the display size ratio at design time was different from current display size ratio.

In the Size section the original display size is specified (design time). Width and height is initially set to the display size when a new Display object is created. Click on the Set button to set the width and height attributes to the properties of the selected display. Notice that these values modify the Display object's width and height attributes, respectively.

The Virtual Area section defines an area (display surface) on a display where the ProcSee RTM is allowed to display its' pictures. All windows associated with the Display object are resized and repositioned to fit this area. Note that the virtual display area functionality is disabled if one of the input fields Virtual Width or Virtual Height is set to the value 0, or the Resize Mode (resizeMode attribute) is set to No Resize. These input fields update the Display object's attributes virtualX, virtualY, virtualWidth and virtualHeight.

The following figure (Figure 74) illustrates a multi-display application and shows how this application can be displayed on a laptop (single monitor) using the virtual display area functionality. The display is divided in four parts, where each Display object occupies one fourth of the display. In this example the size of the monitors are 1280x1024 and for convenience the monitor size of the laptop is set to be the same. The labels show the settings for the virtual display area for each of the four Display objects on the laptop.

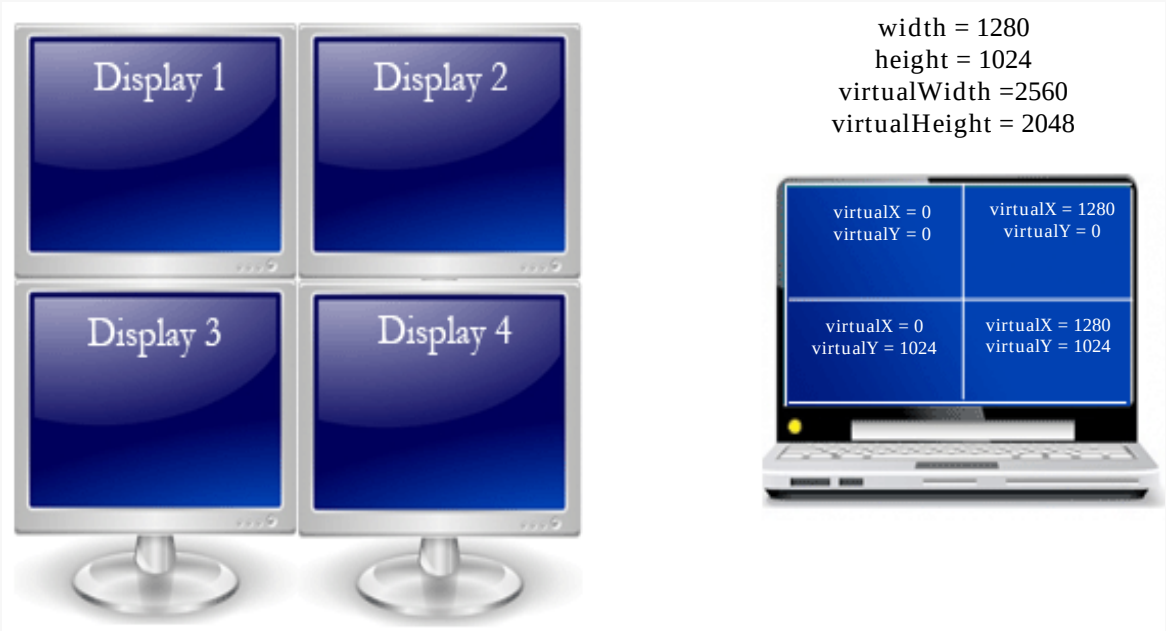


Figure 74 *This figure illustrates four Display objects displaying ProcSee windows (left part). To the right is a laptop where the same windows are displayed on a single display. The necessary Display object settings are shown inside the virtual display areas on the laptop.*

9.12 Layers

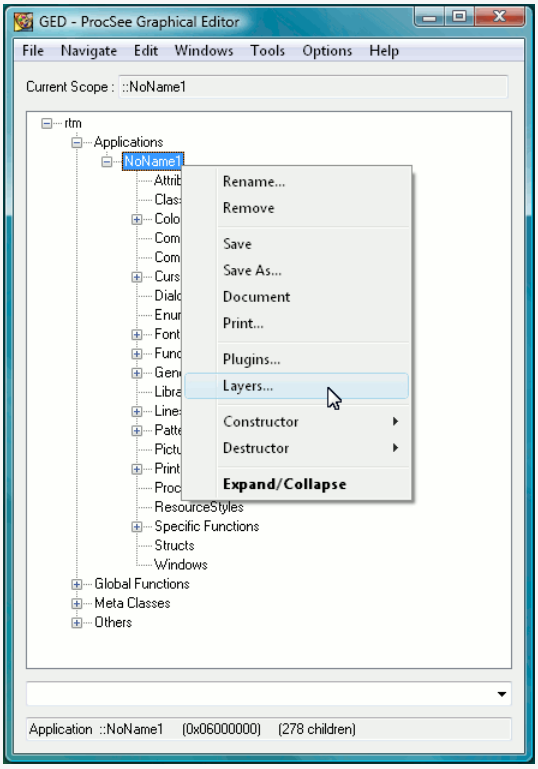
Shapes can be connected to layers. This makes it possible to create pictures where shapes are drawn in different layers. The layers are defined in the application and can be defined in the libraries. Layers in libraries are used by the shapes in the classes in the library. The order of the layers in the application defines the order of the layers used when drawing shapes. The order of the layers in the library is

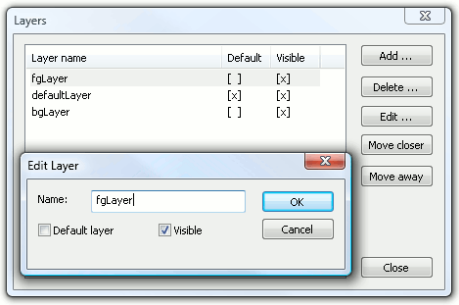
only for use if the library is loaded into a new application where the layers in the library is not defined. The application layers contains connections to the library layers, using the application layer's *aliases* attribute in the .Tdoc files.

In Windows, the layers can be defined using GED.

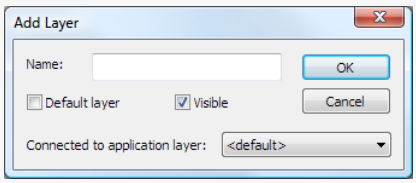
By selecting Layers in the menu for an application or library, the layers can be edited.

The window that opens, contains a list of the layers. Here layers can be added, removed, edited, and the order of the layers can be changed.





When adding and editing layers in a **library**, you can specify what application layer that should connect to the layer. The default is to create a layer in the application with the same name as the library layer, but you can also choose to let an existing application layer connect to the library layer you create. This is done with the combobox at the bottom of the Add / Edit layer dialogue.



In X-Windows, the layers can be defined in the .Tdoc file, see the Reference Manual.

9.13 Libraries

reuse

Libraries are containers that can hold a number of graphic class definitions. A graphic class is a reusable user interface component that can serve as a template for a number of instances of this class.

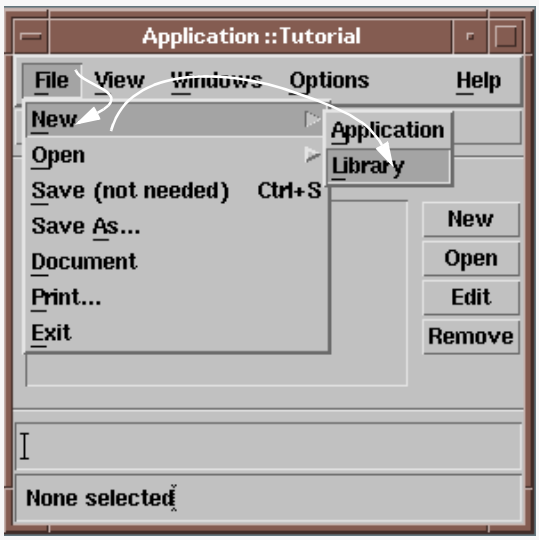
When creating libraries it is important to use an explanatory symbolic name on the library, that conveys what kind of classes are contained in the library.

Creating a New Library

To create a new library, select the **File** menu from the GED main window.Then select **New** from the File menu followed by **Library**, as illustrated in Figure 75 The newly created library is given the name **`NoNameLib.`**.

Figure 75

The figure illustrates how a new library can be created, by selecting New and then Library from the File menu in GED main window.



To create a new library, select the application you want the library in, the right click on the **Libraries** node and select **New...** as shown in Figure 76. The newly created library will be given the name **`NoNameLib`** concatenated with a serial digit.

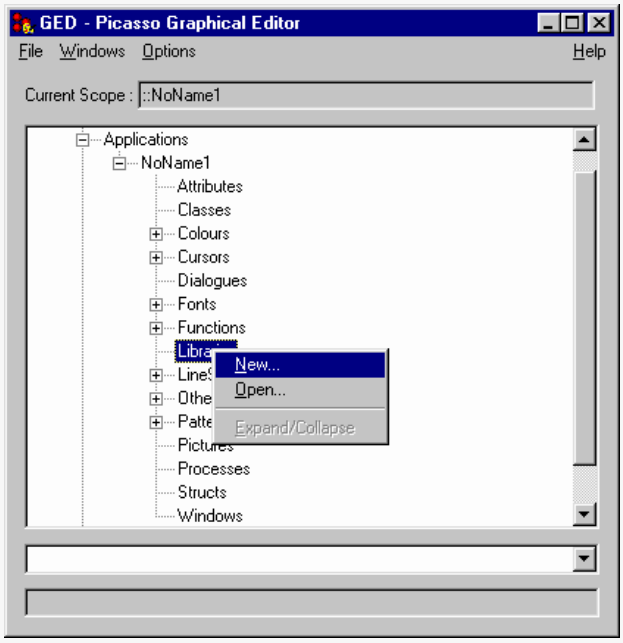


Figure 76

The figure illustrates how a new library can be created, by selecting New... on the popup menu from the GED main window.

Add Resources to a Library

In the current release of ProcSee only colours and fonts can be interactively added to a library using GED. Other resources such as patterns and linestyles must be added to the Tdoc version of the library using a text editor.

To add a new colour or font to a library, make sure the library is selected for editing by addressing the **Focus** button, and selecting the desired library from the *GED Focus* window (see Figure 77). Make sure the **Colours** or **Fonts** item on GED's View menu is selected to view the colours or fonts contained in the library. .

To add a new colour to a library, select the **Colours** node under the library you want to add the colour to. Then right click the **Colours** node and select **New...**. .

To add a new font, follow the same procedure as for colours, but select the **Fonts** node instead of the **Colours** node

The definition of colours in a library is similar to that of an application. For a description of how this is done, please refrer to section 9.3 "Defining Colours in GED"

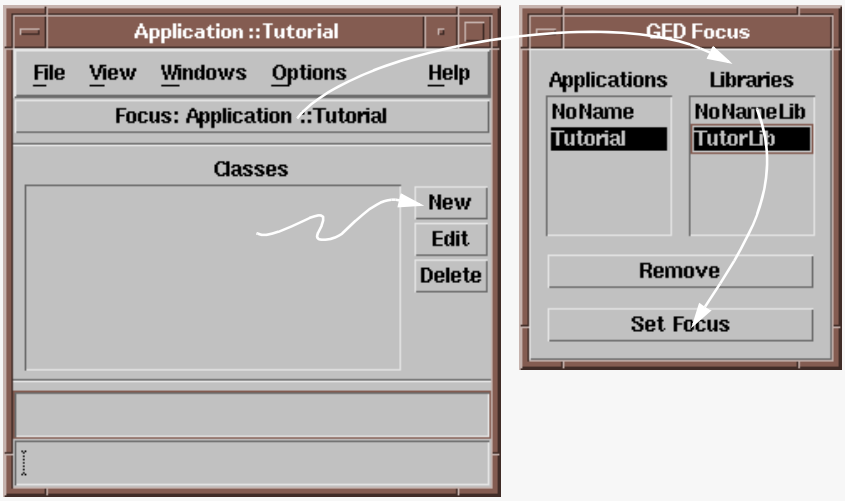
Note that if resources are contained in a library, the resources must be referred to by their symbolic names, otherwise it is not guaranteed that the correct resource is used.

Add Classes to a Library

To add a new class to a library, make sure the library is selected for editing. To tell *GED* that you want to work on a library, you must focus on the library. To do this, select the **Focus** button. When the *GED Focus* window pops up, select the desired library from the library list, and select the **Set Focus** button as illustrated in Figure 77.

Figure 77

To select a library for editing, select Focus from the main GED window. From the GED Focus popup window select the desired library from the list of available libraries. Finalize the selection by clicking on the Set Focus button To add a new class click on the New button in GED's main window, and the drawing editor will be started.

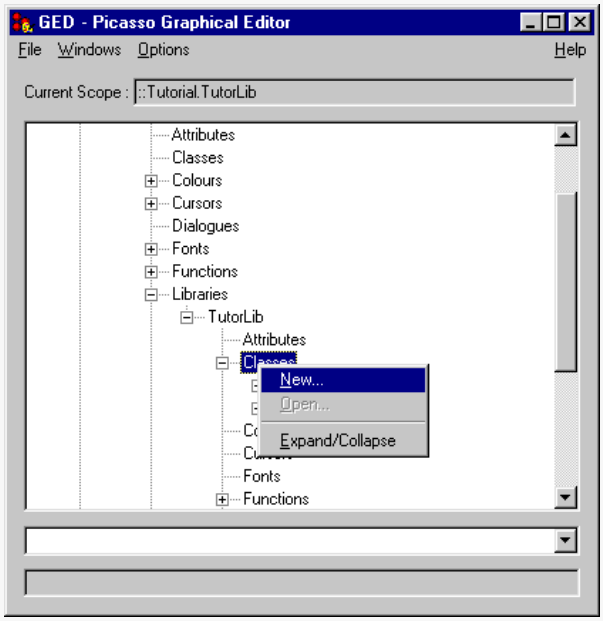


A new class can now be created in the *Drawing Editor*. For a detailed description refer to chapter 12 *"Pictures and Classes"* on page 165.

To add a new class to a library, open the library in the application you want to add the class to. Select the **Classes** node in the library and right click it. Select **New...** to create the new class as shown in Figure 78. This will start the *Drawing editor* where you can create your class.

Figure 78

The figure illustrates how a new class in a library can be created, by selecting New... on the popup menu from the classes node in your library in the GED main window.



Saving a Library

To save a library, make sure the focus is set to the correct library. Select the **Save** item from the file menu, and the library will be saved to a file with the library name as file name.

If you want to save the library to a file using a different name, select **Save As...** from the file menu, and a *Save As* window will pop up. Specify the new name of the library and the directory path to where you want the new library file to reside.

To save a Library, Locate the library you want to save in GED's main window. Right click on the library node of the library you want to save. Select **Save** from the popup menu.

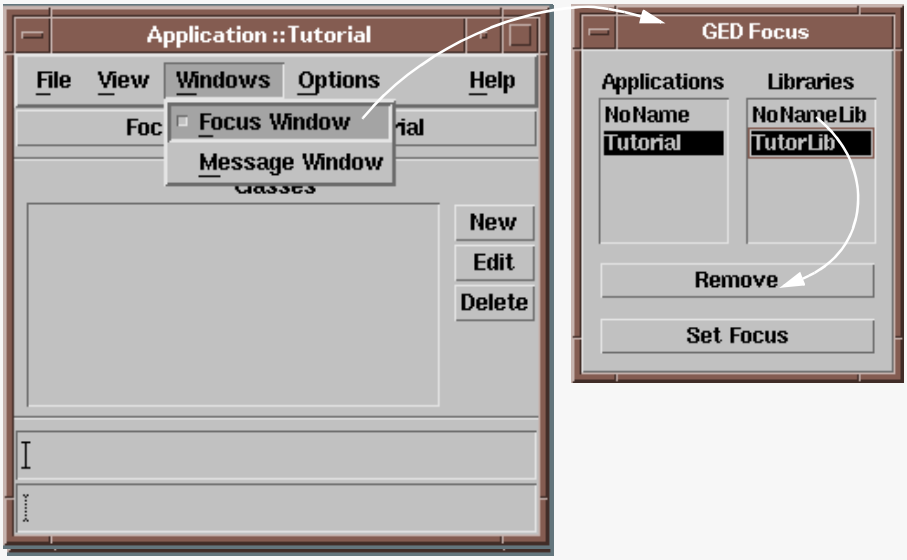
If you want to save the library to a different file name, you select **Save as...** from the popup menu, and specify a filename and a directory path in the *Save as* dialogue.

Removing a Library

To remove a library, go to the **GED Focus** window. If the window is not already displayed, select **Focus Window** from the Windows menu in GED's main window.Select the desired library from the library list, and select the **Remove** button.

Figure 79

The figure illustrates how a library can be removed. From the GED Focus Window, select the desired library, and click on the Remove button.



To remove a library, locate the library you want to remove in the tree. Right click on the desired library and select **Remove** from the popup menu. Then confirm removal by ansering **Yes** and cancel removal by answering **No** to the removal question,

9.14 Processes

A *process* in ProcSee is a representation of a single user program (a program written in Fortran, C, C++, etc.). The process can contain record (struct) definitions, variable definitions, and remote function declarations, using pdat files read by the RTM. The pdat file format is described in "The ProcSee database file format (.pdat)" chapter in the reference manual. The records, variables, and functions should all be defined in the corresponding user program.

A remote function is made available to ProcSee through the use of the API-function **`PfRegisterFunction`** from the user program, and hence will not do anything when called from pTALK unless the user program is connected to ProcSee.

```
// a process record and variable definition
// this file is named TutorProcDb.pdat

struct PUMP
{
 char name[20];
 float flowRate = 0.8;
 int32 state = 0;
};

float t1_level = 100.0;
float t2_level = 0.0;
int32 v1_state = 0;
```

In an application (see chapter 7 *"User Programs in ProcSee"* on page 45), the process is used as described below. A process with the same name as the application is defined in the application document file as described in Ex 1 below. Additional processes with names different to that of the application are defined in the application document file as described in Ex 2 below. A record definition

```
// configuration language
// the following is an excerpt of a file called Tutorial.Tdoc

// Ex 1

application Tutorial
{
 ...
 database "TutorProcDb.pdat";
 ...

}

// Ex 2

application Tutorial
{
 ...
 process TutorialProcDb2
 {
  database "TutorProcDb2.pdat";
 }
 ...
}
```

in any process document file will become a global resource for the whole application.

GED provides a *Process Viewer* and a *Process Editor* for working on processes. The *Process Viewer* presents the process variables and functions as currently defined in the RTM with their current values.

Ged provides a *Process Editor* and a *Variable/Function Viewer* for working on processes. The *Variable/Function Viewer* present the variables and functions that are currently defined in the *rtm* with their current values.

The *Process Editor* is a text editor that loads the process definition from a pdat file. Any comments included in the pdat file will be preserved.

To work on processes, make sure the focus is set to the application and select the **Processes** item from the View menu, and the currently defined processes will be listed (see Figure 80).

To work on processes, expand the **Processes** node to view the defined processes. Select the desired process and right click on it. Then select **Edit...** from the popup menu.

In the current version of the *Process Editor* records or structures can not be worked on interactively, but would have to be created, modified, or deleted through the use of a regular text editor.

Adding a New Process

To add a new process to an application, select the **New** button on

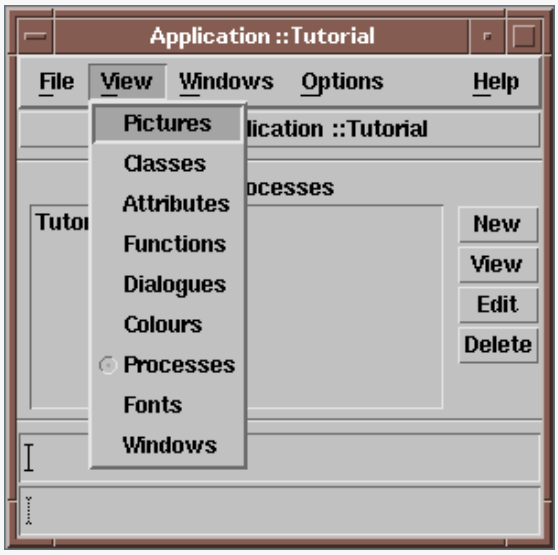


Figure 80

The figure shows how processes can be viewed from GED's main window, by accessing the Processes item on the View menu. Processes can be created and modified by selecting the New, View, Edit, and Delete buttons on the right side of the window, and the Process Viewer or the Process Editor will be displayed.

the right side of the window, and specify the process name in the *New Process window*. The process is created by selecting the **OK** button. To abort the creation of the process select the **Cancel** button.

To create a new process, locate the **Processes** node beneath the ap-

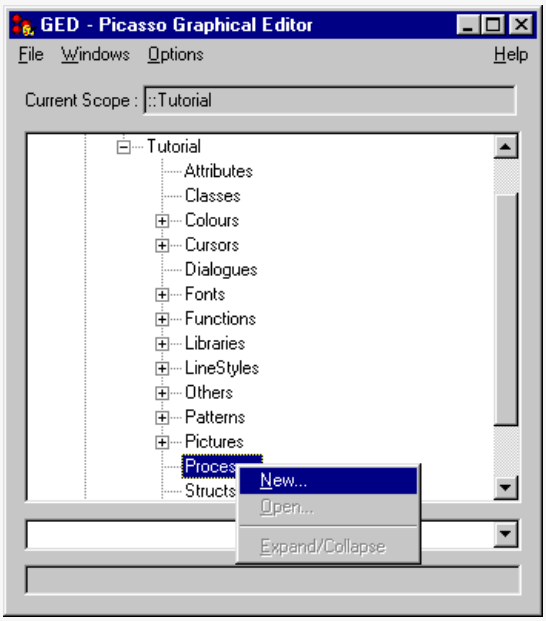


Figure 81

The figure shows how processes can be created from GED's main window, by right clicking the Processes node in the application and selecting New... in the popup menu.

plication node you want the process in. Right click the **Processes** node and select **New...** from the popup menu as shown in Figure 81. Enter the process name in the *New Process Window*. The process is created by selecting the **OK** button. To abort the creation of the process select the **Cancel** button

Note that the process files (.pdat files) have not yet been created. This must be done in the *Process Editor*.

Creating and editing data files for a process

Once a process has been created, the data files or pdat files can be specified. This is done using the *Process Editor*.

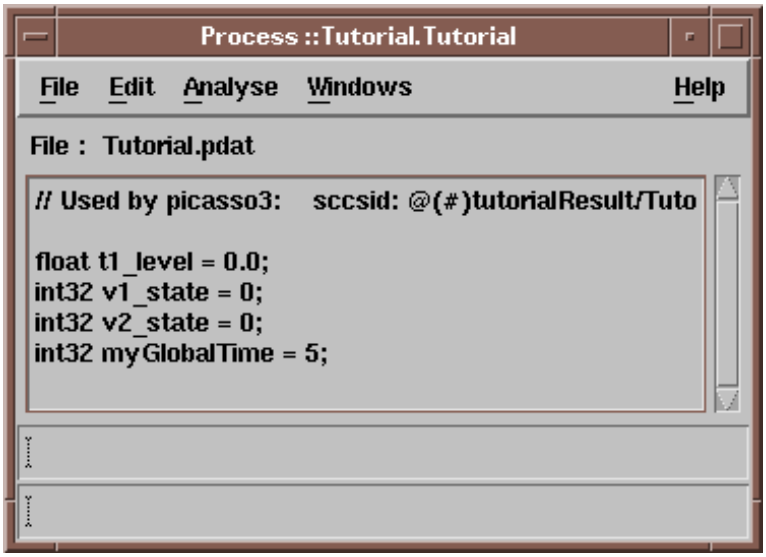
The *Process Editor* is displayed by first selecting the desired process and then addressing the **Edit** button on the right side of the GED main window (see Figure 82).

The *Process Editor* is displayed by selecting the desired process, right clicking it and selecting **Edit...** from the popup menu, or by double clicking the desired process.

To create a new pdat file for the process select the **New** item on the

Figure 82

The figure illustrates the Process Editor. The editor is used to add and modify variables in a process. The set of variables for a process can be defined in several data file (pdat files). The pdat files constituting a process can be managed from the File menu.



File menu. This must be done whenever a new process is created, or if a process not yet contains any pdat files. A new pdat file named NoName1.pdat will be created. At this point new process variables can be added using the pdat file syntax (see the *ProcSee Reference Manual* for a description of the syntax). To save the variables to file select the **Save** or **Save As...** items on the File menu, and specify the name of the pdat file.

If the process consists of several pdat files, individual files can be selected for editing by addressing the **Edit** item on the **File** menu.

The **Open** item on the **File** menu will load an already existing pdat file. If the pdat file with the same name already is contained in the process, a question will pop up asking whether a reload is desired.

To check the syntax of the pdat files, select the **Check Current** item on the **Analyse** menu. Any error messages can be view in the *Syntax Analyser Window* (located on the Windows menu). By addressing the error in the list, the cursor will jump to the line containing the error in the pdat file. To check the syntax of all pdat files, select the **Check All** item on the Analyse menu. When the syntax of all pdat files are checked, the pdat files should be installed in the process in the RTM. pdat files can be installed by selecting the **Install** item on the Analyse menu.This will install all pdat files constituting the process. All pdat files can be saved and installed by selecting the **Save All & Install** item on the File menu. The install command will not install any variables in the RTM if any errors are discovered.

Process variables defined in pdat files are only available after installation of the files.

pdat files can be removed from a process by selecting the **Remove** item on the **File** menu. Note that this will only remove the pdat file (and its content) from the process contained in the RTM. If the pdat file has been saved, it will not be removed from the harddisk.

On the **Edit** and **Remove** items on the **File** menu, an asterix(*) next to the pdat file indicates that the file has been changed.

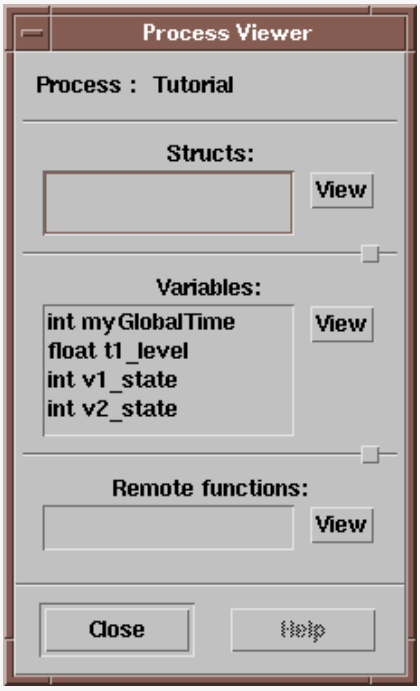
An asterix in the process editor window title field indicates that the process has been changed and that an install is needed.

Viewing process variables

To view variables and functions defined for a process, select the desired process, and then address the **View** button on the right side of the GED main window. This will display the *Process Viewer* (see Figure 83). The viewer can be used to look at current values of process variables. Details about Structs, Variables, or Remote Functions can be looked at by addressing the corresponding **View** button on the right side of the viewer window. The **Structs:** window of the process viewer will show all structs defined in the application, since structures are global to an application.

Figure 83

The figure illustrates the Process Viewer. The viewer can be used to look at current values of process variables. By selecting a Struct, Variable, or Remote Function, details can be looked at by addressing the corresponding View button on the right side of the Process Viewer.



To view variables and functions defined for a process, select the desired process and expand it. This will show two new sublevels : **Variables** and **Remote Functions**. To view them siply right click on the item you vish to view and select **View...** from the popup menu (see Figure 84). To view structs right click on an item under the **Structs** node in the application and select **View...** from the pop-up menu.

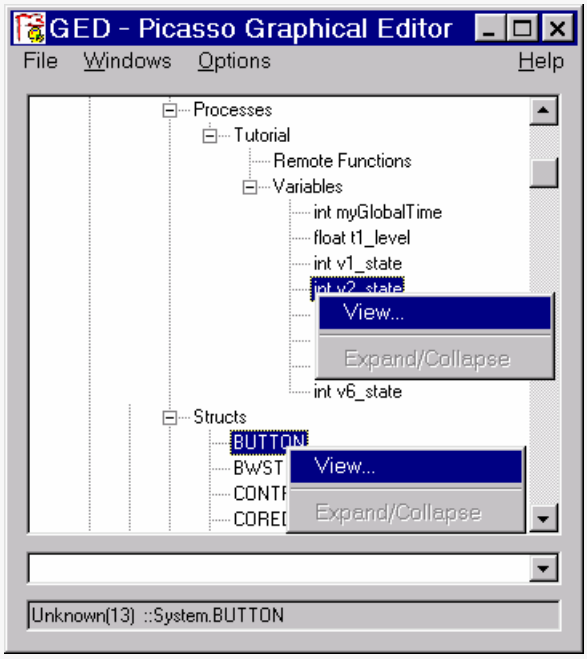


Figure 84

The figure shows how process variables can be viewed from GED's main window, by right clicking the desired variable's node or the struct node in the application and selecting View... in the popup menu.

9.15 Attributes, Functions and Dialogues

The attributes, functions, and dialogues described here are global to an application in ProcSee, and are therefore defined under an application.

Attributes

An attribute should not be confused with a process variable, since a process variable is merely a reflection of a corresponding variable in a user program.

UI variable

An attribute is a variable *local to the user interface*, and does not have a corresponding variable in a user program. An attribute can be of any of the basic data types in ProcSee as listed below: An attribute is used whenever local variables are needed in an application, picture, or class to store information needed within the user interface only.

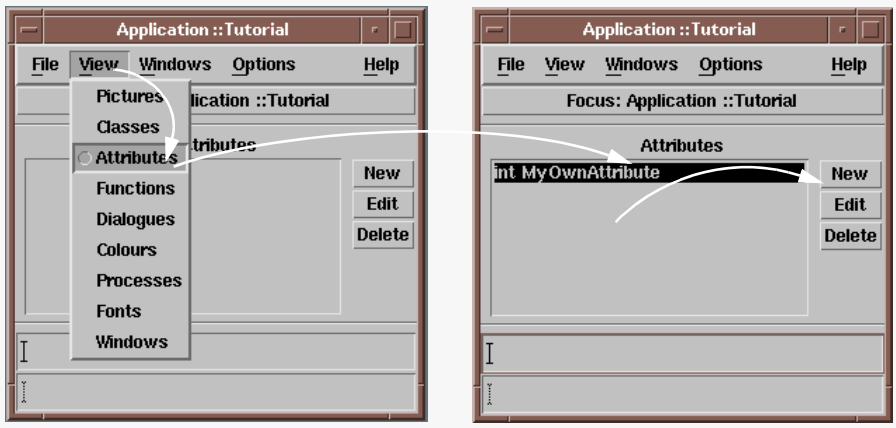
```
// legal attribute data types.

float
double
char
short int
int
unsigned short int
unsigned int

// legal attribute data types also include pointers to (*) and arrays
// of ([]) the above listed data types.
// In addition attributes can be of List, HashTable, TimerAt, and
// TimerInterval data types, and pointers to any ProcSee data type.
```

create an attribute

An attribute can be defined using GED. To create a new attribute, select the **View** menu from GED's main window, then select **Attributes** from the View menu as illustrated in Figure 85, then select

Figure 85

To create a new attribute for an application, set GED's main window to display attributes, by selecting Attributes from the View menu.Existing attributes for a selected application will be listed. A new attribute can be created by clicking on the New button.



the **New** button next to the Attributes list. An *attribute Editor* window will then pop up. The attribute definition can be typed in the window labelled *Attribute:*. Figure 87 illustrates the definition of an attribute called **`MyOwnAttribute`**.

An attribute can be defined using GED. To create a new attribute, select the **Attriutes** node in GED's main window. Right click it and

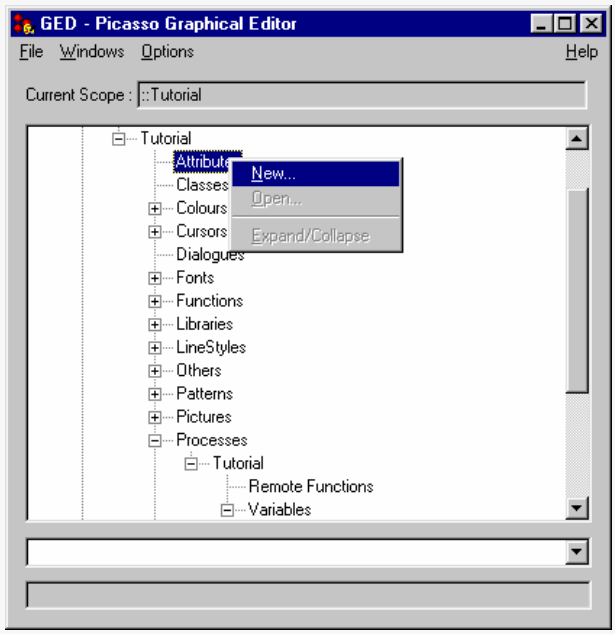


Figure 86

To create a new attribute for an application, select the Attributes node in GED's main window. Right click the node and select New.... Existing attributes for a selected application will be listed if the Attributes node is expanded.

select **New...** from the popup menu (see Figure 86). An *attribute Editor* window will then pop up. The attribute definition can be typed in the window labelled *Attribute:*. Figure 87 illustrates the definition of an attribute called **`MyOwnAttribute`**.

To check if the defined attribute is syntactically correct, click on the **Compile** button. If the message *Compiled OK* is displayed in the *Compile messages:* text field, the attribute definition is syntactically correct and can be installed by clicking on the **OK** button. This will close the attribute editor. If an error message is displayed in the *Compile messages:* window, take a closer look at your attribute definition and look for the error(s) (Standard editing keys can be used to change the attribute definition).

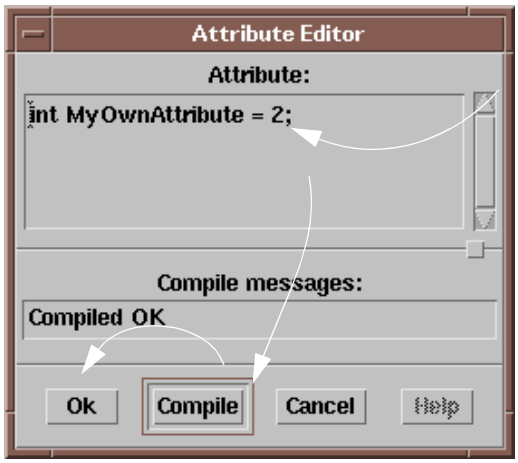
An attribute can be connected to a dynamic value. Assume that a user program contains a variable **`t1_level`**. To connect the attribute **`MyOwnAttribute`**, defined in Figure 87, to the user program variable, the following must be written:

```
int MyOwnAttribute = `t1_level`;
```

Remember to put the dynamic variable in backquotes `.

Figure 87

The figure shows the Attribute Editor window. An attribute is defined in the input field labelled "Attribute:". To check if the attribute definition is syntactically correct, click on the Compile button, and the message "Compiled OK" should be displayed in the Compile messages field. To close the Attribute editor window and install the new attribute, click on the OK button.



edit an attribute

To edit on an existing attribute, select the **View** menu from GED's main window, and then select **Attributes**, as illustrated in Figure 85. Select the desired attribute from the attribute list and click on the **Edit** button next to the attribute list window, and the *Attribute Editor* window will pop up.

To edit an existing attribute, select the desired item, right click on it and select **Edit...** from the popup menu, or double click the desired attribute. The *Attribute Editor* will pop up with the selected attribute.

Make the desired changes, and click on the **Compile** button to check the syntax of your changes. Install the changes by clicking on the **OK** button (see Figure 87).

delete an attribute

To delete an attribute, select the **View** menu from GED's main window, and then select **Attributes**, as illustrated in Figure 85. Select the desired attribute from the attribute list and click on the **Delete** button next to the attribute list window,A *Delete question* window will pop up. Confirm the delete operation by clicking on the **Delete** button. Cancel the delete operation by clicking on the **Cancel** button.

To delete an attribute, select the node you want to delete from the **Attributes** node, right click on it and select **Delete**. A *Delete Question* window will pop up. Confirm the delete operation by clicking the **Yes** button, and cancel the delete operation by clicking the **No** button.

Functions

A function is a named collection of pTALK statements. Functions defined in a ProcSee application are accessible from all parts of the interface within a particular application.

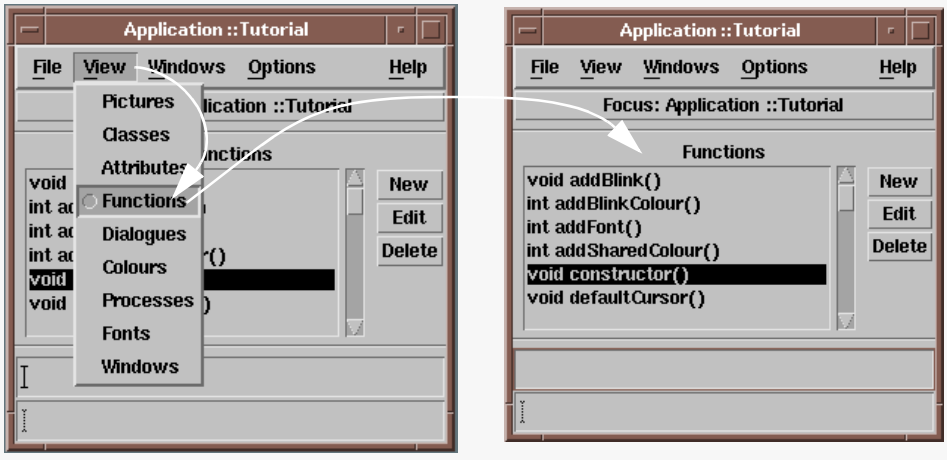
When should a function be defined? Generally, a function should be defined whenever a certain sequence of pTALK statements is repeated several times within the interface, or whenever the readability of a sequence of pTALK statements can be improved by incorporating the pTALK statements in a function. For a detailed description of the syntax of the pTALK language refer to chapter 8 *"pTALK, the ProcSee Language"* on page 55.

create a function

To create a new function, select the **View** menu from GED's main window. From the View menu, select **Functions**, as illustrated in Figure 88. Already defined functions will be listed. Among these are some predefined functions.

Figure 88

The figure illustrates how GED is set to edit on application functions.



To create a new function, click on the **New** button and a *Function Editor* window will pop up.

To create a new function, select the **Functions** node in GED's main window. Right click on it and select **New...** in the popup menu (see) and a *Function Editor* window will pop up.

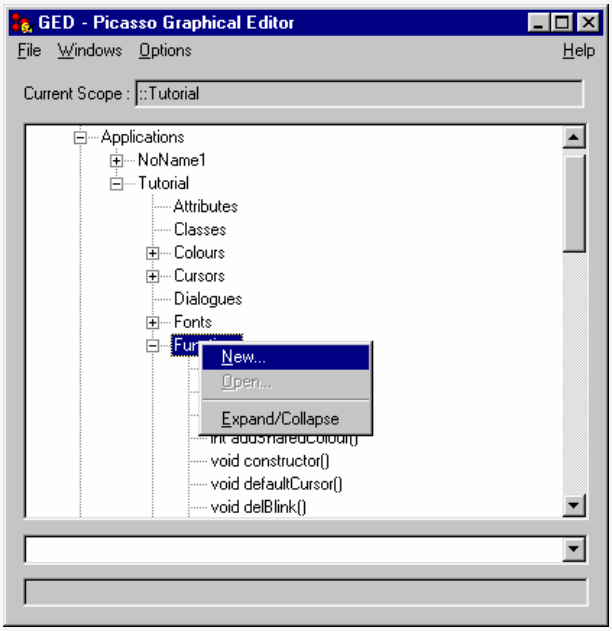


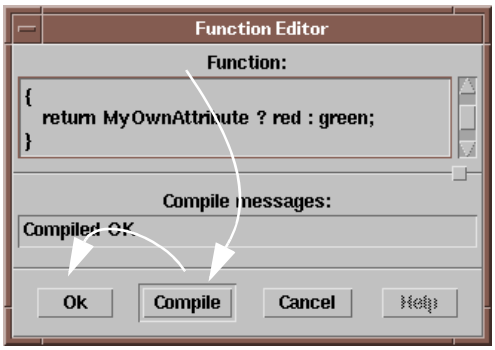
Figure 89

To create a new function for an application, select the Functions node in GED's main window. Right click the node and select New.... Existing functions for a selected application will be listed if the Functions node is expanded.

Type the function definition in the window labelled *Function:* (see Figure 90). After the function has been defined, check the syntax by clicking on the Compile button. If the message *Compiled OK* is displayed in the *Compile messages:* window, the function is syntactically correct, and can be installed by clicking on the **OK** button (this will close the Function editor window). If, after selecting the **Compile** button, an error message is displayed in the *Compile messages:* window, go back to the function definition, and make the appropriate corrections, before selecting **Compile** and **OK**.

Figure 90

A new function is defined by typing the function definition in the Function: window, selecting the Compile button, and installing the function by selecting OK.



edit a function

To edit an existing function, select the View menu from GED's main window. From the **View** menu, select **Functions**, as illustrated in Figure 88. The existing functions will be listed in the *Function:* window. Select the desired function for editing, and click on the **Edit** button.

To edit an existing function, select the function you want to edit from GED's main window. Right click on it, and select **Edit...** from the popup menu, or you can double click on the node you want to edit.

The Function editor will pop up. Make the desired changes to the function definition. Check the syntax of the changes by clicking on the **Compile** button. If the message *Compiled OK* is displayed in the *Compile messages:* window, the function is syntactically correct, and can be installed by clicking on the **OK** button. If any error messages are displayed in the *Compile messages:* window, make the appropriate corrections, **Compile** and click on **OK** to install the function.

delete a function

To delete an existing function, select the View menu from GED's main window. From the **View** menu, select **Functions**, as illustrated in Figure 88. The existing functions will be listed in the *Function:* window. Select the desired function to delete, and click on the **Delete** button. A *Delete question* window will pop up. To confirm the deletion of the function, click on the **OK** button. To cancel the delete operation, click on the **Cancel** button.

To delete a function, select the function you want to delete from the **Functions** node, right click on it and select **Remove**. A *Delete question* window will pop up. Confirm the delete operation by clicking the **Yes** button, and cancel the delete operation by clicking the **No** button.

Dialogues

A dialogue is a description of an interaction between a user and the ProcSee system. A dialogue consists of an event/action pair. The event can be one or several of the following standard event functions in combination with any legal logic expression in the pTALK language.

In the above standard button functions, legal values for the button arguments are 1, 2, or 3. Referring to the Left, Middle, and Right mouse button respectively, and 0 for any button. For the key func-

```
// standard event functions

int     buttonPressed( int button );
int     buttonReleased( int button );
int     buttonDoubleClicked( int button );
int     buttonTripleClicked( int button );
int     buttonRepeated( int button );
int     keyPressed( int key );
int     keyReleased( int key );
int     cursorMoved();
nt      windowEntered();
int     windowLeft();
int     windowClose();
int     windowMoved();
int     windowResized();
int     shapeEntered();
int     shapeLeft();
int     focusLost();
int     trendUpdated();
```

tions, the key argument corresponds to an X-Window keysym value for a keyboard key. Three methods for finding the keysym value are:

- Use the X-Windows demo program **`xev(1)`**. For information on how to use the program, refer to program documentation.
- Use the X-Windows command **`xmodmap -pk`**. This will give you a list of keys defined in X and their corresponding keysym values. For a detailed description refer to manual pages for **`xmodmap`**.
- Use the *Triggers:* pop-up menu in the dialogue editor.

For a detailed description of the standard event functions, refer to **`EVENT(3)`** in the *"Standard pTALK Functions Manual pages"* in the Reference Manual.

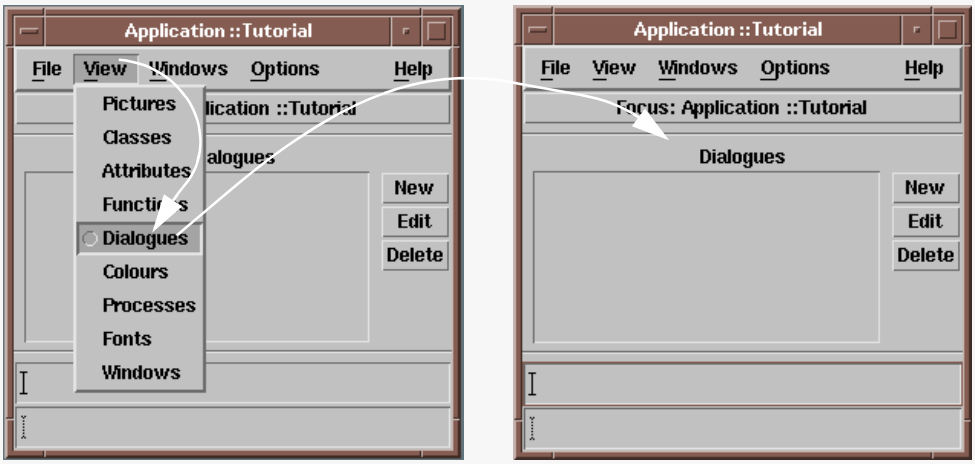
The action part in a dialogue is a pTALK statement. For a detailed description of the syntax for the pTALK language, refer to chapter 8 *"pTALK, the ProcSee Language"* on page 55.

create a dialogue

To create a new dialogue, select the View menu from GED's main window. From the **View** menu, select **Dialogues**, as illustrated in Figure 91. Previously defined dialogues will be listed.

Figure 91

The figure shows how GED is set to operate on dialogues.

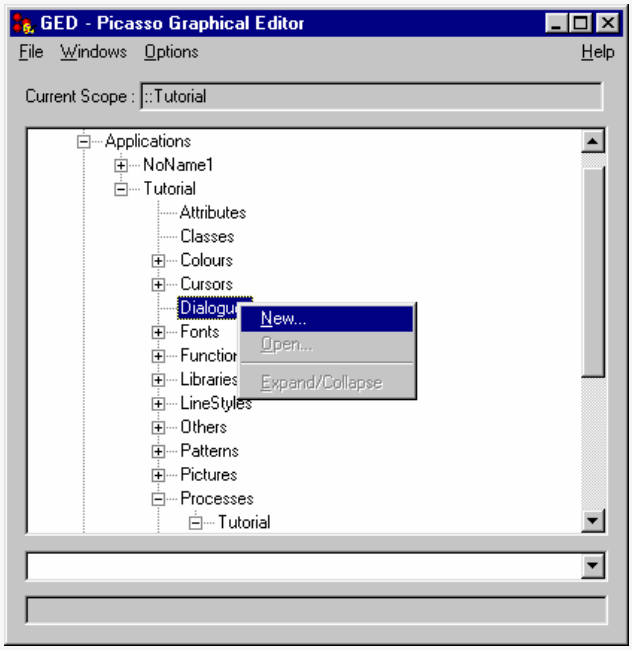


To create a dialogue, click on the **New** button next to the dialogues list, and a *Dialogue Editor* window will pop up (see Figure 93).

To create a new dailogue, select the **Dialogues** node in GED's main window. Right click on it and select **New...** in the popup menu (see Figure 92) and a *Dialogue Editor* window will pop up (see Figure 93).

Figure 92

To create a new dialogue for an application, select the Dialogues node in GED's main window. Right click the node and select New.... Existing dialogues for a selected application will be listed if the Dialogues node is expanded.

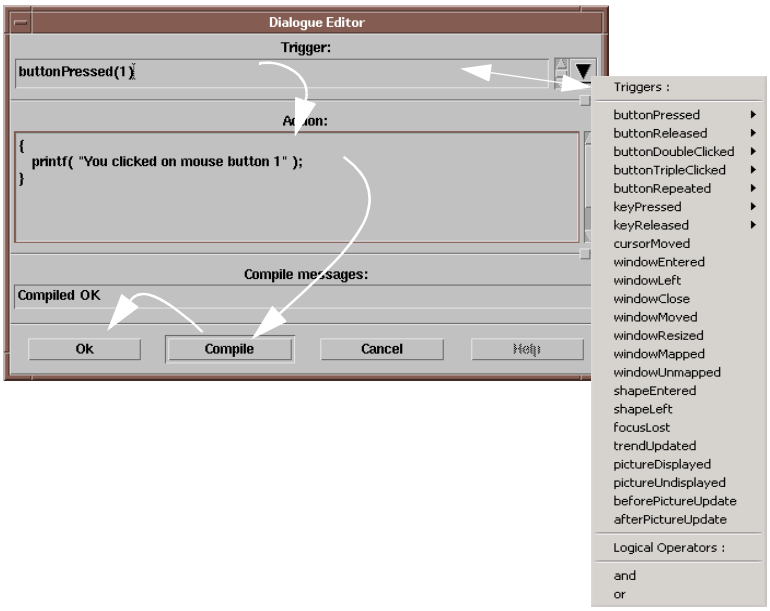


Specify the event in the *Trigger:* input field, and the action in the *Action:* input field.

A *Trigger* can be specified by typing the desired event function or by using the *Triggers:* pop-up menu on the right side of the dialogue editor.

Figure 93

The figure illustrates how a dialogue is defined in the Dialogue editor. After defining the event and the action, check the syntax by selecting the Compile button. If correct, install the dialogue by selecting the OK button. If any error messages appear, make appropriate corrections, Compile and install by selecting OK.



By selecting any of **buttonPressed**, **buttonReleased**, **buttonDoubleClicked**, **buttonTripleClicked**, or **buttonRepeated** from the *Triggers:* pop-up menu (see Figure 94 and Figure 95), a button designator menu will pop up. By selecting one of the button designators the button event function with correct parameter will be copied into the *Trigger:* input field in the dialogue editor. (If **buttonPressed**, then **Any** is selected, **buttonPressed(0)** will be copied into the *Trigger:* field.)

Figure 94

The figure illustrates the Trigger: selection pop-up menu, and the corresponding sub-menu for all button event functions.

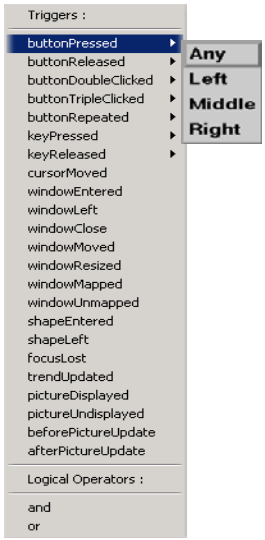
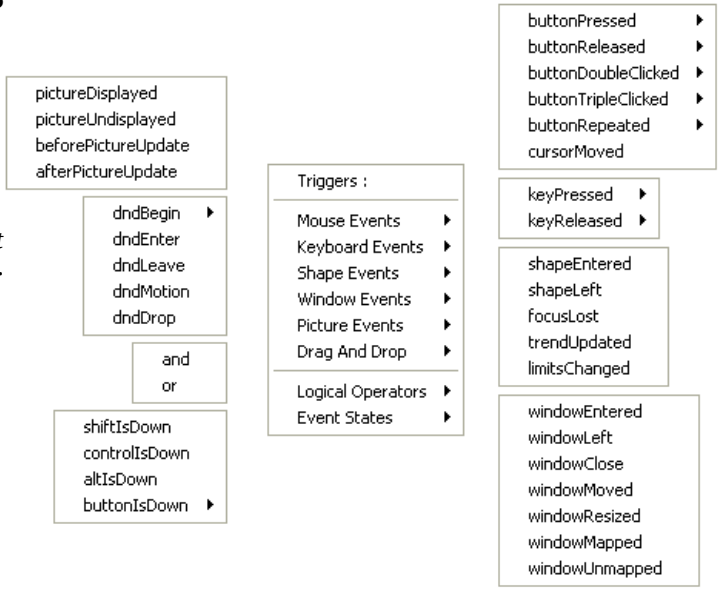


Figure 95

The figure illustrates the Trigger: selection pop-up menu, and the corresponding sub-menu for all button event functions.



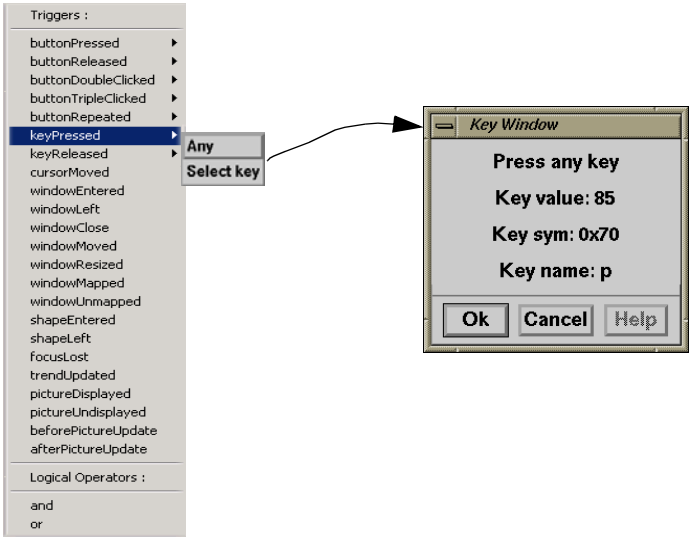
By selecting any of **keyPressed** or **KeyReleased** from the *Triggers:* pop-up menu (see Figure 96) a key selection menu will pop up. If the Select key option is selected a *Key Window* will pop up. When a key is pressed in the window, information about the key is displayed.

Clicking on the **OK** button in the *Key Window* will close the window and copy the selected key event function with the selected *keysym* value into the *Trigger:* field in the *Dialogue Editor.*

Selecting the **Cancel** button will close the *Key Window* and leave the *Trigger:* input field in the *Dialogue Editor* unchanged.

Figure 96

The figure illustrates the Trigger: selection pop-up menu, and the corresponding sub-menu for the two key event function. The Select key option will display the Key Window, where detailed information about keys can be found by pressing keys in the dialogue.



To check the syntax of the dialogue, click on the **Compile** button. If the dialogue is syntactically correct, the message *Compiled OK* should be displayed in the *Compile messages:* text field. To install a syntactically correct dialogue, click on the **OK** button and the Dialogue editor window will be closed. If an error message is displayed in the *Compile messages:* text field, make the appropriate corrections in the dialogue definition, select **Compile** and install the dialogue by clicking on the **OK** button.

edit a dialogue

To edit an existing dialogue, select the View menu from GED's main window. From the **View** menu, select **Dialogues**, as illustrated in Figure 91. Previously defined dialogues will be listed. Select the desired dialogue for editing, and click on the **Edit** button next to the dialogue list, and the *Dialogue Editor* will pop up. Make the desired changes, **Compile** and install the edited dialogue by clicking on **OK**.

To edit an existing dialogue, select the node for the dialogue you want to edit, right click on it, and select **Edit...** from the popup menu, or you can double click on the node you want to edit. The *Dialogue Editor* will pop up. Make the desired changes, **Compile** and install the edited dialogue by clicking on **OK**

delete a dialogue

To delete a dialogue, select the View menu from GED's main window. From the **View** menu, select **Dialogues**, as illustrated in Figure 91. The defined dialogues will be listed. Select the dialogue to be deleted, and click on the **Delete** button next to the dialogue list. A Delete question window will pop up. To confirm the deletion, click on the **Delete** button. To cancel the deletion, click on the **Cancel** button.

To delete an dialogue, select the node you want to delete from the **Dialogues** node, right click on it and select **Remove**. A *Delete Question* window will pop up. Confirm the delete operation by clicking the **Yes button**, and cancel the delete operation by clicking the **No** button.

9.16 Constructors and Destructors

A *constructor* is a user supplied function that is used to perform an initialisation job.

A *destructor* is a user defined function that is used to perform some clean-up operations when a user interface component cease to exist.

Constructors and destructors can be defined in:

- applications
- libraries
- pictures
- classes
- trend logs

Whenever an application, a library, etc. is loaded from file, a user defined function named constructor is searched for and executed. The syntax is plain pTALK with all its possibilities for calling other functions.

The signatures of the constructor and destructor functions can be as follows:

```
CONSTRUCTOR DEFINITION

void constructor()
{

   ... constructor body

}


DESTRUCTOR DEFINITION

void destructor()
{

   ... destructor body

}
```

A typical constructor job for an application would be to set default values for process variables and to load and display one or several initial pictures.

Figure 97

Constructors are defined in the same way as other pTALK *functions using GED. This figure shows a typical constructor for an application.*

10 Undo

Undo is a feature in ProcSee which helps users erase the last changes made to applications, libraries and pictures. With this functionality enabled, the developer can easily revert back to an earlier stage in the development cycle. In ProcSee the undo functionality is available at every scope in the object hierarchy.

Undo is a feature in ProcSee which helps developers to negate the last changes made to applications, libraries and pictures. When this feature is enabled, users can correct mistakes and try out different aspects of the user interface without the risk of loosing the original work. That is, undo can be used to revert back to an earlier stage in the development cycle.

ProcSee supports redo in addition to undo. Redo is available to negate the last undo actions made, which is the opposite of undo. Note that the redo history will be cleared when the user performs an action which is added to the undo list.

To undo the last action made, select *Undo* in the *Edit* menu in GED. To redo the undo action, select *Redo* from the same menu. See Figure 98. For easier access to the undo and redo feature, shortcut keys are available in the drawing editor in GED. The shortcut keys are *ctrl-Z* for undo and *ctrl-Y* for redo. Note that the undo and redo functionality is not available when the picture is in test mode.

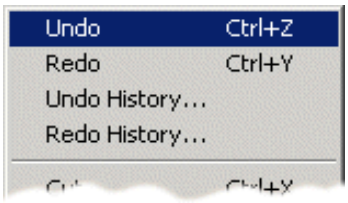


Figure 98*The undo menu in the graphics editor GED.*

ProcSee has separate undo lists for applications, libraries and pictures. Changes made to a picture will not affect the undo history of another picture, library or application.

ProcSee maintains a history of undo actions. The user can undo not only the last action, but a series of actions. To view this undo history, select *Undo History...* from the *Edit* menu in GED, see Figure 99. This dialogue displays the description of the undo actions. From the same dialogue, redo actions can be viewed under the *Redo History* tab. To undo or redo a series of actions in one operation, select the actions in the history list and press the *Undo* or *Redo* button.

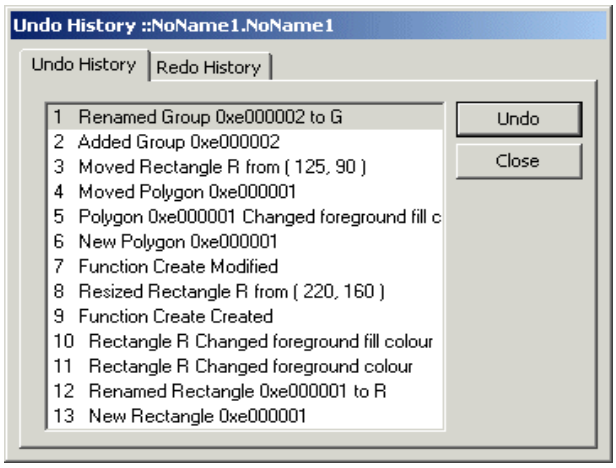


Figure 99*The Undo History dialogue. The descriptive names of the actions are displayed in the list. In this dialogue one or several undo actions can be undone in one operation by selecting the undo action to revert to and pressing the Undo or Redo button. Selecting The Redo History tab will display the redo actions.*

ProcSee creates readable descriptions of the undo actions when they are added to the undo history list. Examples include *"Executed TestFunc()"*, *"Attribute MyAttr Created"*, *"Renamed Rectangle 0xe0000001 to R"*, *"Test Mode"*, etc. These descriptions are displayed in the *Undo History* dialogue.

The *Undo History* dialogue behaves differently depending on whether the dialogue is opened from the drawing editor in GED, or from the main window in GED. If the *Undo History* is opened from the drawing editor, the dialogue will always show the undo action for that particular picture. If it is opened from the main window, it will show the undo history for the node selected in the tree view. That is, the undo history list will change when the developer selects a new node in the tree view. Note that there is one *Undo History* dialogue in the main window, while there is one *Undo History* dialogue for each picture opened in the drawing editor.

When a picture is in test mode and the undo functionality is enabled, a special undo action is created, called *"Test Mode"*. All changes made to the picture while it is in test mode will be added to the *"Test Mode"* action. This action will be added to the undo history list when the picture is changed to edit mode and modifications have been made to the picture during the period it was in test mode.

Changes made to a picture due to dynamic bindings will not create undo actions. Lets say that a rectangle's *foregroundFillColour* is dynamically dependent on the process variable *SB232*, i.e.:

```
R.foregroundFillColour = `SB232 > 100 ? red : blue`.
```

An undo action will not be created when the process variable *SB232* changes and the foreground fill colour of the shape *R* changes from red to blue or vice versa. The situation is different if a function is called which modifies the foreground fill colour of shape *R* directly, like:

```
void set()
{
  R.foregroundFillColour = SB232 > 100 ? red : blue;
}
```

An undo action will in this case be created when the foreground fill colour is changed.

Editing a ProcSee application connected to an external source updating process data, like a simulator or a data acquisition system, should be problem-free when the undo feature is enabled. However, it can in rare circumstances, be necessary to disable for instance the undo feature in the applications, libraries or pictures. Situations where this can be necessary are when the performance is dramatically reduced or the increased memory consumption is a problem. Enabling and disabling the undo functionality is done from the *Undo Setting* dialogue. For more information how to customize the undo functionality in ProcSee, see Undo Settings on page 161.

Note that the undo feature is only enabled when using the graphics editor GED on the Microsoft Windows platforms. The undo functionality will automatically be disabled in all applications, libraries and pictures when GED terminates.

10.1 Undo Settings

To open the *Undo Settings* dialogue, select *Undo Settings...* from the *Options* menu, see Figure 100. In this dialogue undo settings can be changed from their default values.

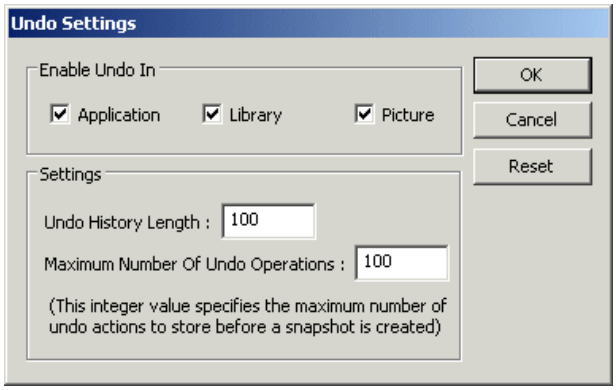


Figure 100*The Undo Settings dialogue which can be used to customize the undo functionality in ProcSee.*

The undo feature will by default be enabled in applications, libraries and pictures. To disable or enable the undo feature, check or uncheck the check boxes belonging to the *Application*, *Library* or *Picture*, respectively.

The undo history length can be user configured from this dialogue. When the number of undo actions exceeds the undo history length, the oldest undo actions will be deleted when new undo actions are added to the list. The default value is 100.

An undo action contains one or several undo operations. Undo operations are the basis operations in an undo action. If a picture designer calls a function named *"Create"*, which creates and initialises several shapes, a single undo action will be created with the descriptive name *"Executed Create()"*. This single undo action stores several undo operations. Examples of undo operations are, shape created and shape properties changed.

The maximum number of undo operations specifies the maximum number of operations an undo action can store before a snapshot is created. When a snapshot is taken, a copy of the entire picture, library or application (minus library entries, picture entries and process entries) will be created. A snapshot will consume a lot of memory and may reduce performance when the snapshot is taken. Sometimes, it is therefore necessary to increase this limit to reduce the possibility of creating snapshots. Normally, this limit should be left as is, but it can be increased in circumstances where many snapshots are created. The default value is 100.

The *Reset* button resets the undo settings to the defaults.

11 Active Shapes

Active Shapes is a feature in **ProcSee** *which enhances the selection functionality in the drawing editor.*

Active Shapes is a feature in ProcSee utilized by GED when a picture is displayed in the drawing editor. It is an editing functionality, and can only be applied to pictures in edit mode. The main purpose with active shapes sets is to enhance the user-friendliness of the editing features in the editor. This functionality does not influence, interfere or reduce the performance of a running system, because it is only utilized by the drawing editor in GED.

The definition of an active shape set is a group of shapes in a picture which can be active or inactive independent of other sets. Only shapes in active sets can be selected. Several active sets can be active simultaneously. It represents an important concept where shapes can be grouped together to improve the selection functionality in the editor. Only shapes in active shapes sets can be selected and modified from the selected properties or the graphic attributes window in GED. Mouse and keyboard events are transient in inactive sets, i.e. events are not detected by inactive shapes but propagated until an active shape is hit, if any.

Shapes do not have to be added to an active shapes set to be selectable. GED provides a way to activate all shapes independent of whether they are part of an active shapes set or not. This set is called *"All"* in GED. ProcSee also offers a special active shapes set which is the inverse of the other active shapes sets, i.e. all shapes not included in any other sets. This feature has proven to be very useful in many situations, for instance when selecting shapes or adding shapes to active shapes set(s). Notice that this special active shapes

set is always added. The name of this set in GED is, *"Shapes Not In Any Active Shapes Sets"*. On the other hand, a shape can be added to several sets if it simplifies the editing.

Shapes in inactive active shapes sets are by default visualized using semi-transparent colours, i.e. see-through shapes. The default for inactive shapes are neutral colours with 75% transparency (compared to the original colour). This default can be modified from the advanced editing settings dialogue, see *"Advanced Editing Settings..."* on page 227.

The active shapes sets are persisted to file, both binary (.ppic) and document files (.Tdoc). When the file is loaded the active shapes sets are loaded as well, i.e. the sets remain valid through different editing sessions.

11.1 Why using active shapes?

From a GUI designer's perspective, active shapes set offers many advantages over traditional editing providing better control over the shapes to edit. Next follows a few examples to better illustrate how powerful the active shapes sets feature is.

For instance, selecting a shape can be difficult if several shapes are located at the same or nearly at the same position. In these situations active shapes is a nice feature. It is much easier to select the actual shapes if only the active shapes sets where the shapes of interest are located, are enabled. Using the active shapes sets feature prevents the inactive shapes to impact on the selection.

Another example is to use active shapes sets to prevent shapes from being moved by accident, i.e. it can be used to lock shapes to fixed positions. To use this feature add the shapes to be locked to an active shapes set and disable this set.

12 Pictures and Classes

A picture is a collection of graphic elements such as rectangles, lines, circles, etc. In addition to these, a picture can also contain attributes, functions, and dialogues. A class is, in many respects, similar to a picture. A class can however be instantiated many times. This chapter describes how to create pictures and classes using the ProcSee Graphics Editor (GED).

12.1 Composite User Interface Components

When creating a graphic user interface, graphic elements such as rectangles, circles, lines, and polygons are put together to convey a certain message to the user of the interface. In ProcSee this collection of graphics is called a picture.

A class is, in many respects, similar to a picture. The major difference is that a class can serve as a template for several instances of that class.

Pictures and classes consist of primitive shapes[1] and instances[2], and may also contain user defined attributes, functions and dialogues. Pictures and classes have their own coordinate system, snap settings, and background colours. All of these things may be edited in the picture and class editor part of the ProcSee editor, hereafter called the *drawing editor*.

1 Primitive shapes include graphic elements such as rectangle, line, polygon, circle, circular arc/slice, ellipse, elliptic arc/slice, text, and dialogue area.

2 An instance is a specific occurrence of a class.

Pictures and classes contain collections of graphical objects in addition to attributes, functions, and dialogues. The user interface designer can add attributes, functions, and dialogues to the pictures and classes as needed, to supplement those that are predefined.

In the following sections, pictures, classes, and their attributes will be described. A explanation is also given of how to create pictures and classes using GED.

12.2 The Coordinate System

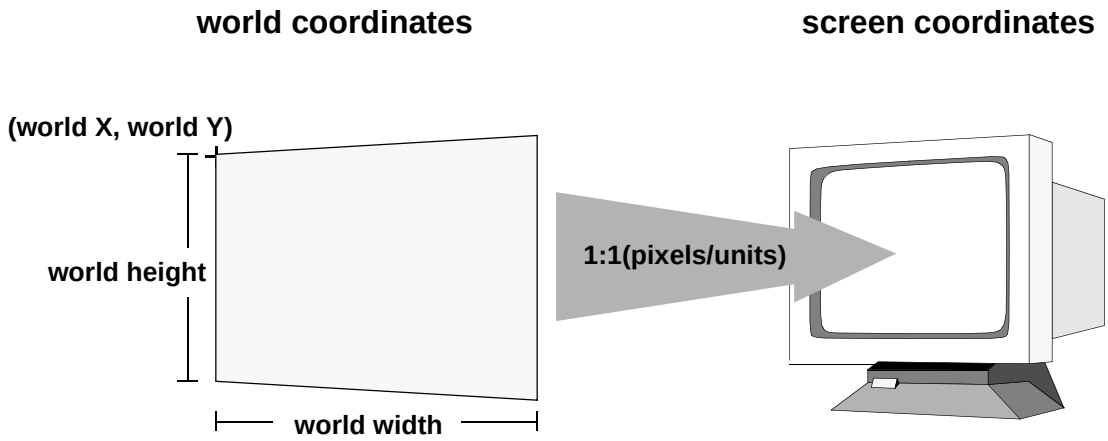
A picture has its own, user defined coordinate system, referred to as the *world coordinate system*. The world coordinate system is mapped onto the underlying screen coordinate system. The relation between the screen and world coordinate system is determined by the *1:1(pixels/units)* factor, which is the number of screen coordinates to each world coordinate.

The default values of the world coordinate system are

world X = 0, world Y = 0

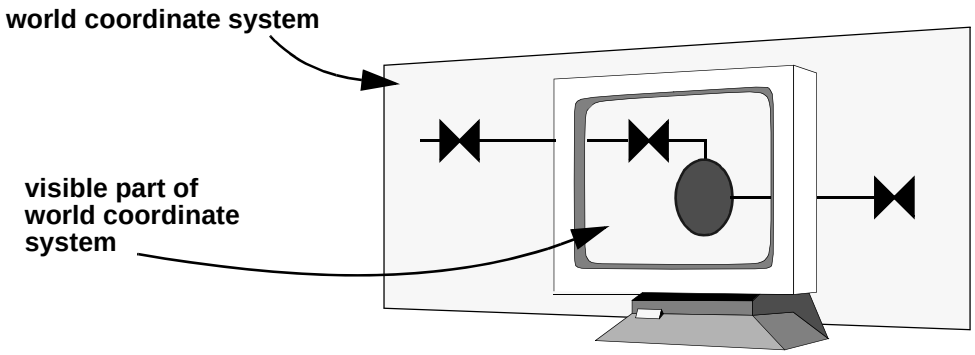
world width = width of screen in pixels

world height = height of screen in pixels



The world coordinate system is user defined, and can therefore be larger than that of the screen coordinate system, as illustrated below. The factor *1:1(pixels/units)* determines the size of world units

relative to screen units. A predefined picture function *pan*, can be used to move the visible part of the picture around in the world coordinate system.



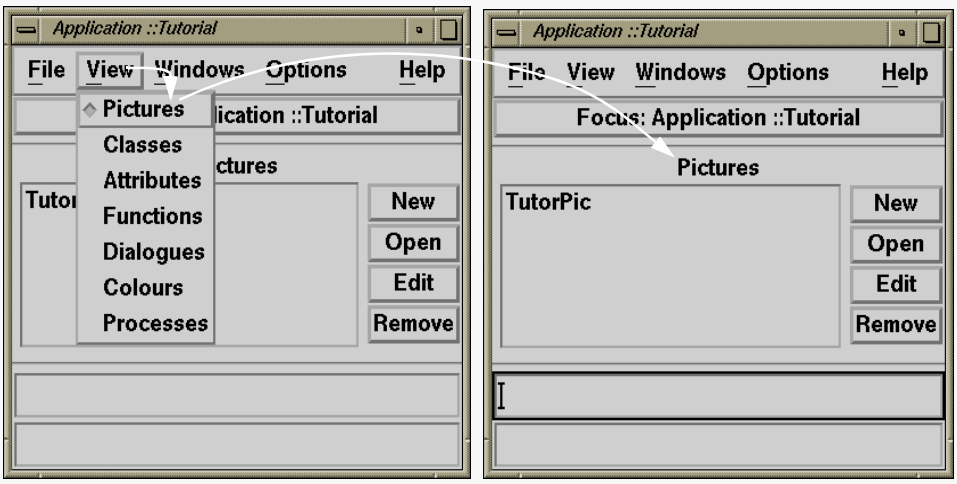
12.3 Working with Pictures

Creating a New Picture

Before you create a new picture, ensure that the active application in GED is the one that you want to add a new picture to. To prepare to create a new picture select the **View** menu from GED's main window, then select **Pictures** from the View menu, and a list of existing pictures will be displayed (see Figure 101).

Figure 101

The figure illustrates how GED is set to operate on pictures. Select Pictures from the View menu, and the existing pictures will be listed.



To create a new picture, click on the **New** button next to the picture list, and the *Picture Drawing Editor* window will pop up.

The newly created picture will be given the name **`NoName`**. You should give the picture a proper name. This can be done by going to the **File** menu in the drawing editor, and from the **File** menu selecting **Save As...**. A Save As window will pop up. Type the path to the directory where you want the picture to be saved in the input field labelled *Path:*. Type the name of the picture in the input field labelled *Save as Picture:*. Click on the **OK** button to save the new picture. To cancel the save operation, click on the **Cancel** button (see Figure 102)

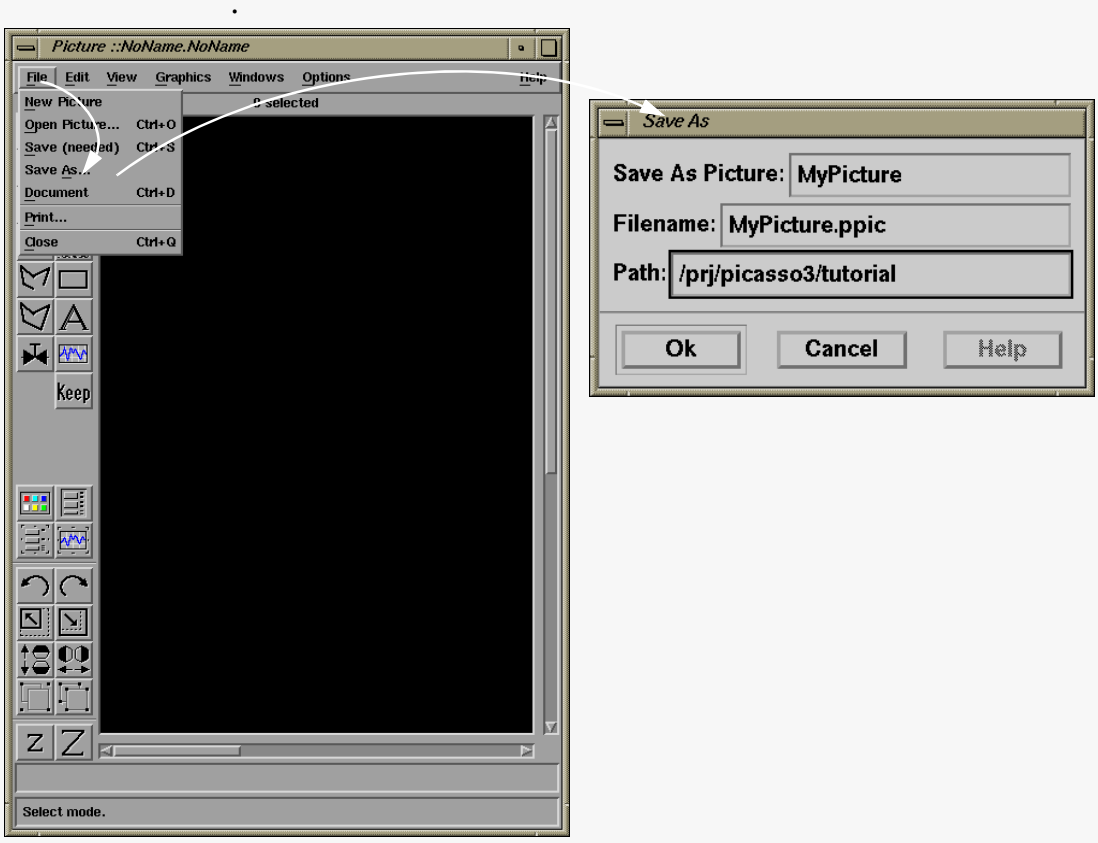


Figure 102 *The figure illustrates how a picture is saved to a picture file with a new name. Select the File menu from the drawing editor, and select Save As... . In the Save as pop up window, specify the path to where the new picture will reside, and a new picture name. To save the picture, click on the OK button. To cancel the save operation, click on the Cancel button.*

To Create a new picture, select the **Pictures** node in the application you want the picture in, right click it, and select **New...** from the popup menu (see Figure 103). This will start the *Picture Drawing Editor*.

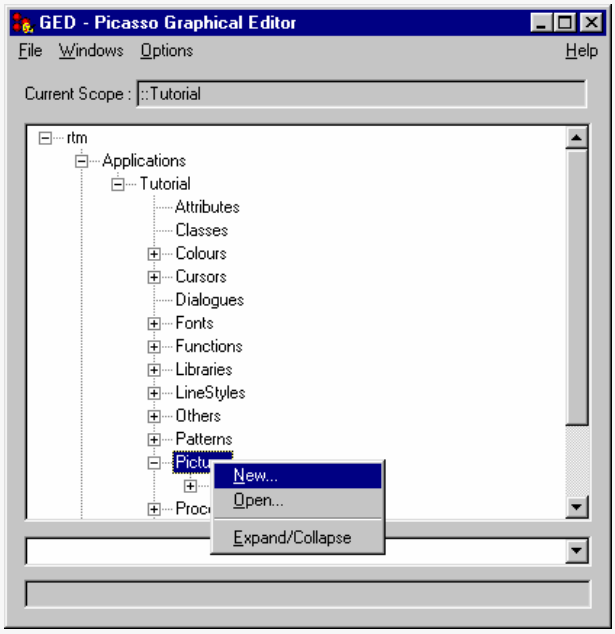


Figure 103

The figure illustrates how GED is set to operate on pictures. Select the Pictures node and right click it. Select New... from the popup menu to create a new picture. The existing pictures will be listed under the Pictures node.

The newly created picture will be given the name **NoName**. You should give the picture a proper name. This can be done by going to the **File** menu in the drawing editor, and from the **File** menu selecting **Save As...**, or you can right click on the picture

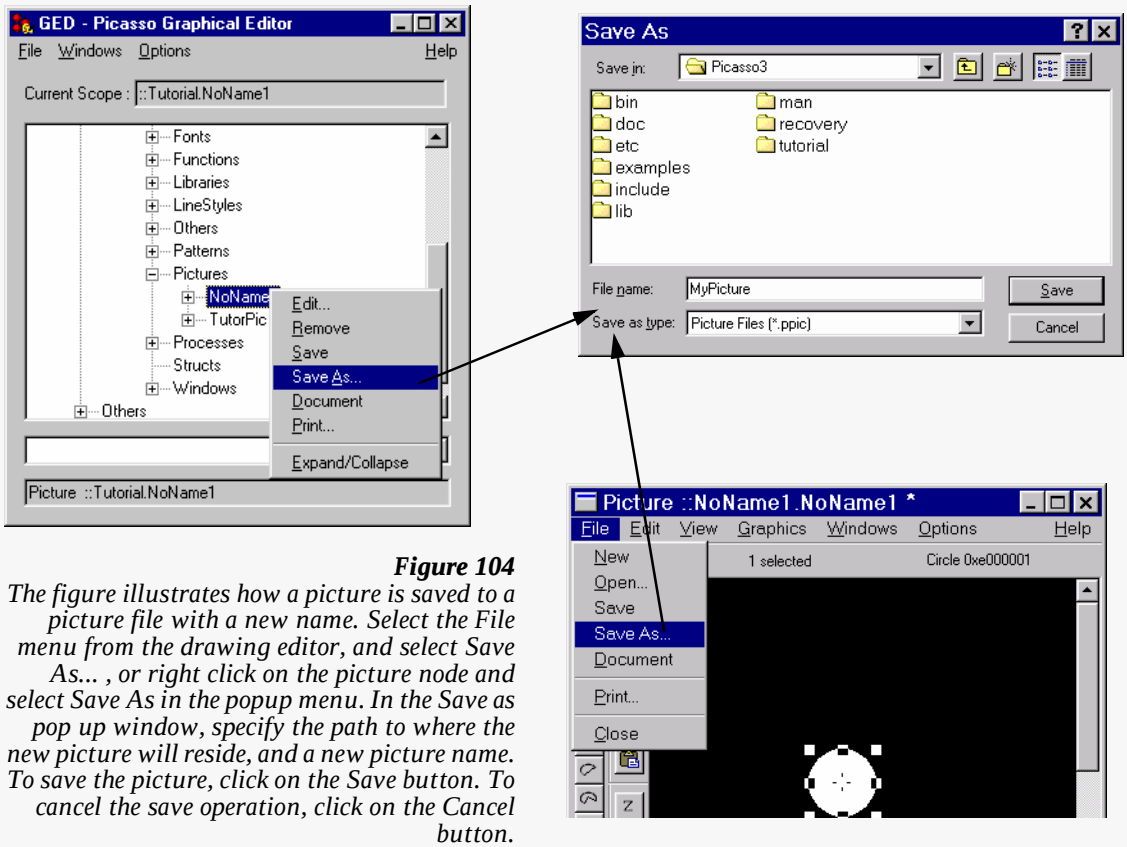


Figure 104

The figure illustrates how a picture is saved to a picture file with a new name. Select the File menu from the drawing editor, and select Save As... , or right click on the picture node and select Save As in the popup menu. In the Save as pop up window, specify the path to where the new picture will reside, and a new picture name. To save the picture, click on the Save button. To cancel the save operation, click on the Cancel button.

node in the GED main Window and select **Save As...** from the pop-up menu. A *Save As* window will pop up. Select the directory where you want the picture to be saved and enter the name of the picture. Click on the **Save** button to save the new picture. To cancel the save operation, click on the **Cancel** button (see Figure 104)

In chapter 13 *"The Drawing Editor"* on page 179, you will find a description of how to use the drawing editor to draw the actual graphics in a picture.

Editing an Existing Picture

To edit a picture select the **View** menu from GED's main window. Select **Pictures** from the View menu, and a list of the loaded pictures will be displayed (see Figure 101). Select the picture that you wish to work on from the picture list, and click on the **Edit** button next to the picture list. The *Drawing Editor* window will pop up, with the selected picture loaded in the editor.

To edit a picture, expand the **Pictures** node. The pictures listed here are all the pictures loaded at the time. Select the picture you want to edit, right click on it and select **Edit...** from the popup menu, or double click on the node. The *Drawing Editor* will pop up with the selected picture loaded in the editor.

Loading a Picture from File

To load a picture from file into the *Drawing Editor*, select the **Open** button from GED's main window (see Figure 101). A file selection window will pop up. Select the desired picture file by clicking on the file name. To load the file click on the **OK** button, and the drawing editor will be started with the specified picture loaded. To cancel the LoadFromFile operation, click on the **Cancel** button.

To load a picture from file into the *Drawing Editor*, select the Pictures node and right click on it. Then select **Open...** from the popup menu. Select the desired file in the file selection dialogue that pops up. To load the file, click the **OK** button, and to cancel the operation, click the **Cancel** button. This will start the *Drawing Editor* with the specified file.

Pictures can also be loaded from the *Drawing Editor* by selecting **Load...** in the **File** menu.

Removing a Picture

To remove a picture select the **View** menu from GED's main window. Select **Pictures** from the View menu, and a list of the already loaded pictures will be displayed (see Figure 101). Select the desired picture to be deleted from the picture list, and click on the **Remove** button. A Delete question window will pop up. To confirm the deletion, click on the **OK** button. To cancel the delete operation, click on the **Cancel** button.

To remove a picture, expand the **Pictures** node to get a list of all loaded pictures. Select the desired picture to be removed, and right click on it. Select **Remove** from the popup menu. A Delete question window will pop up. To confirm the deletion, click on the **Yes** button. To cancel the delete operation, click on the **No** button.

Note: The remove operation will not delete any picture files, it will merely unload the picture from ProcSee.

Useful Standard Picture Functions

The predefined picture functions and attributes are described in detail in the pTALK *Standard Library Functions* in the *Reference Manual.* This section gives some examples on how to use the most commonly used predefined picture functions.

The following examples assume that a picture named **`DemoPicture`** exists in the currently loaded application in GED. The examples below show how the DemoPictures functions can be used in the pTALK language. A detailed description of the pTALK language is provided in chapter 8 *"pTALK, the ProcSee Language"* on page 55.

name

To get the name of the picture, use **`DemoPicture.name()`**. This function will return the name of the picture in a character string.

window operations

The picture contains some standard functions that operate on the window that the picture is displayed in. If the picture is not currently displayed, these function will have no effect. To bring the picture to the front on the screen, use **`DemoPicture.raise()`**, this will place the window that the picture is displayed in on top of all the other windows on the same level in the window hierarchy. The opposite effect can be achieved using **`DemoPicture.lower()`**.

Many modern window managers allow a title on windows. In ProcSee, the default title on a window is the name of the picture being displayed in the window. It is often desirable to use a more explanatory title on a window. This can be done using the *title()* function. For example, **`DemoPicture.title("Main picture for project XYZ")`**.

The picture and its window can be moved to a new location on the screen using **`DemoPicture.move(200, 300)`**. This would move the upper left corner of the picture to the position X = 200, Y = 300.

To resize the window that a picture is being displayed in, use the *resize()* function. For example, **`DemoPicture.resize(400, 400)`**, would set the new size of the window to width = 400 and height = 400.

The Picture constructor() and destructor()

The picture constructor() and destructor() functions are user defined functions with a predefined signature: **`void constructor()`** and **`void destructor()`**.If these functions are defined for a picture, they will be executed automatically. The **`constructor()`** will be executed whenever the picture is loaded from file. The **`destructor()`** will be executed whenever the picture is deleted, using either the **Remove** command in Ged or the pTALK function **`remove()`**.

Note that there is a difference between loading and displaying a picture. If a picture has been loaded but is not displayed, the constructor is NOT executed when the picture is displayed. However, the constructor is called when a picture is loaded from file.

12.4 Using Classes

In many respects, a class is similar to a picture. However there is one major difference:

template A class can serve as a template for several instances or individualized copies of the class. A class can be defined as an independent user interface component, encapsulating implementation details. The ability to use a class as a general description for several specific instances, makes it easier and more efficient to develop and maintain a graphical user interface.

A class is created as a picture. That is, attributes, functions, dialogues, and the graphic shapes are all defined in a similar manner to that of a picture. However, they are saved collectively as a class.

class libraries Related classes are stored in class libraries, for reuse and easy retrieval. A number of class libraries can be used in an application.

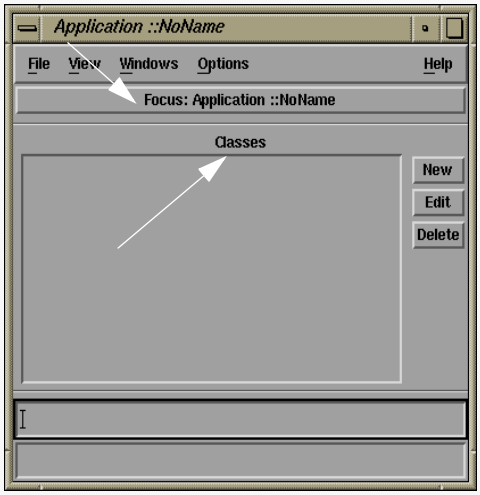
reuse If classes in libraries are designed independently from their surroundings, libraries containing generic user interface classes can be reused across several projects and/or be used to support different user interface standards.

Creating a New Class

To create a new class, make sure that GED is in class view mode by setting the focus onto the desired library, or by selecting **Classes** from the **View** menu. When the **New** button on the right side of the GED window is clicked, the *drawing Editor* is started.

Figure 105

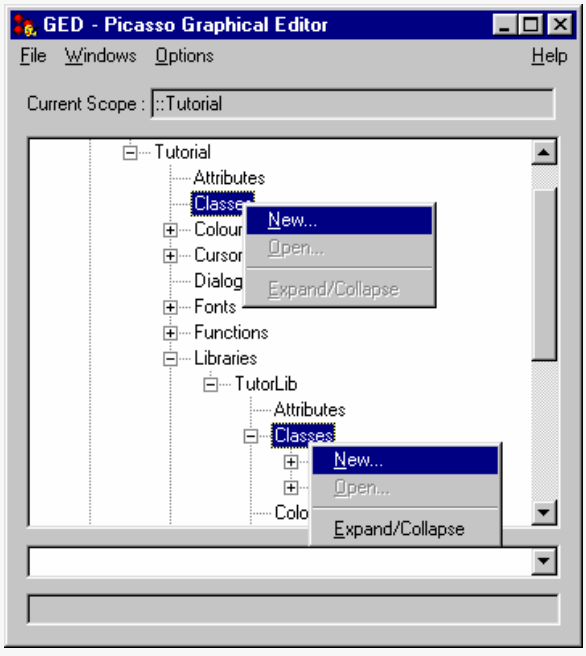
To edit or delete already existing classes, or to create new classes, set GED in class view mode by setting focus on a library, or by selecting classes from the View menu.



To create a new class, right click on the **Classes** node in your application or on the **Classes** node in the library where you want the class. Select **New...** from the popup menu (see Figure 106), and the *Drawing Editor* is started.

Figure 106

The figure illustrates how to use GED to create classes. Select the Classes node in the application or the desired library, and right click it.. Select New... from the popup menu to create a new class The existing classes will be listed under the Classes node.



A new class is defined in the drawing editor in a manner similar to that for a picture. For a description on the drawing editor refer to chapter 13 *"The Drawing Editor"* on page 179.

Editing a Class

To edit an existing class, make sure that GED is in class view mode by setting the focus onto a desired library, or by selecting **Classes** from the **View** menu. The classes that are available will be listed in a text field labelled *Classes*. Select the class that you wish to work on by clicking on it's name using the left mouse button, and then the **Edit** button on the right side of the GED window. The *Drawing Editor* will be then started and the selected class is loaded in the editor for editing.

Alternatively, a class can be loaded in the drawing editor by double clicking on it's name.

To edit a class, expand the **Classes** node on the application level or on the library level, depending on where the desided class is located. The classes listed here are all the classes that are available in the libraray / application. Select the class you want to edit, right click on it and select **Edit...** from the popup menu, or double click on the node. The *Drawing Editor* will pop up with the selected class loaded in the editor.

To insert a class into a library , select **Insert** or **Insert As...** from the **File** menu in the drawing editor. If the class is loaded into the drawing editor from a library, the class will be inserted into that library. If the class was not loaded as part of a library, it will be inserted under the application level, similar to that of global attributes, functions, and dialogues as described in section 9.15 *"Attributes, Functions and Dialogues"* on page 144.

Note that whenever a class is inserted into a library as described above, the class is only put into its container; a library or application. In order to save a class to file, the application or library, in which the class is a member, must be saved. Refer to section 7.6 *"Saving an Application"* on page 51 and *"Saving a Library"* on page 136.

Deleting a Class

To delete a class, make sure GED is in class view mode by setting the focus onto a desired library, or by selecting **Classes** from the **View** menu. The available classes are listed in the text field labelled *Classes*. Select the desired class by clicking on the class name using the left mouse button, and then select the **Delete** button on the right side of the GED window.A *Delete Question* window will pop up. To confirm the deletion of the class select **OK**, to cancel the deletion select **Cancel**.

To delete a class, locate the desired class in the application **Classes** node, or the library **Classes** node. Right click on the class that is to be removed, and select **Delete** from the popup menu.A *Delete Question* window will pop up. To confirm the deletion of the class select **Yes**, to cancel the deletion select **No**.

The Class constructor() and destructor()

The class constructor() and destructor() functions are user defined functions with a predefined signature: **`void constructor()`** and **`void destructor()`**. If these functions are defined for a class, they will be executed automatically for instances of this class. The **`constructor()`** will be executed whenever an instance is created. The **`destructor()`** will be executed whenever an instance is deleted.

Class Coordinate System

A class has a coordinate system of its own, similar to that of a picture. The coordinate system can be changed in the drawing editor. The *reference point* of a class is (0,0) in the class coordinate system. To position the graphics of a class properly according to the reference point, do the following:

- Change the coordinate system so that (0,0) is somewhere near the middle of the *Drawing Editors* drawing area. This is done by selecting the **Class settings...** item from the **Options** menu in the drawing editor. Make the appropriate changes in the *Class Settings Window*.
- Set a suitable snap interval, and select all of the graphic elements in the class. The snap interval is used to define a grid that makes editing easier.
- Place the cursor on the graphics where you want the class reference point to be. Now, while keeping the mouse button down, move the mouse cursor to the (0,0) position. Note that the current mouse cursor position is displayed in the upper left corner of the drawing editor.

12.5 Attributes, Functions, and Dialogues

User defined attributes, functions, and dialogues can be defined for both pictures and classes. This is done in a similar manner to that described in section 9.15 *"Attributes, Functions and Dialogues"* on page 144, but with the Picture Properties / Class Properties window, described in *"Picture Properties..."* on page 195, instead of the main window as the starting point.

13 The Drawing Editor

The drawing editor is part of the graphics editor (GED), and is used to create graphical pictures and classes, as well as attributes, functions, and dialogues for both pictures and classes.

13.1 Introduction

The *drawing editor* is the part of the *graphics editor* that is used to create the actual pictures and classes. Graphics are drawn in a similar manner for both pictures and classes. The only difference is in the save operation. A picture is saved to a picture file, and a class is inserted into a library that can be saved to a library file.

Figure 107 shows the drawing editor. It has a menu bar at the top, containing a File, an Edit, a View, a Graphics, an Options, and a Help menu.

The drawing editor is also used to define attributes, functions, and dialogues for both pictures and classes.

testing

An important feature of the drawing editor is the test facility. This allows the user interface designer to test and verify the functionality of an interface while designing it. To switch between the design and test modes, the user interface designer simply clicks on a button.

The following sections explain the editor menus and commands.

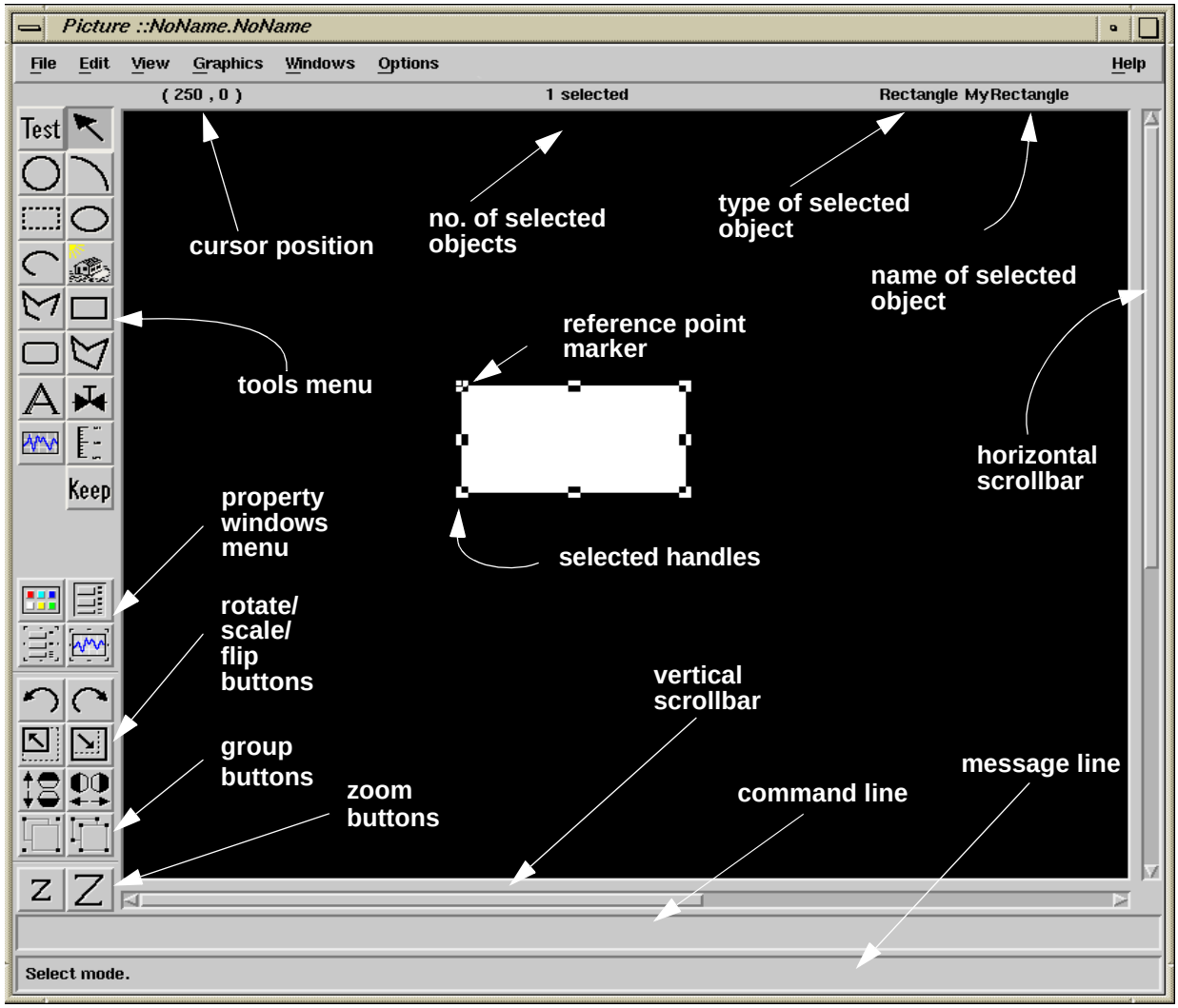


Figure 107 *The figure shows the ProcSee drawing editor, highlighting the major editor information and command areas.*

The NT version has a broader set of toolbars. Including a filebar and an editbar.(Figure 108)

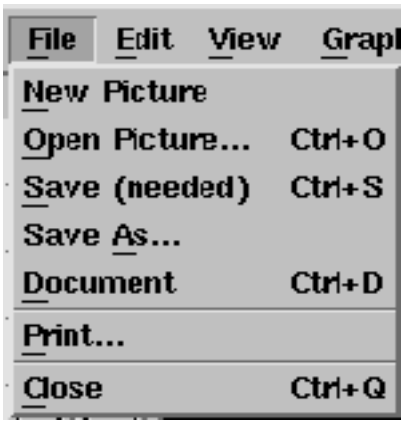
Figure 108
The toolbars give shortcuts to new, open, save, cut, copy & paste as an option to clicking in the menu

13.2 The File Menu

The File menu offers several choices for pictures as illustrated in Figure 109. Each choice will be described in turn.

Figure 109

The File menu contains choices for creating a new picture, opening an existing picture, saving a picture, saving a picture with a new name, creating a document file of a picture, and closing the drawing editor.



New Picture

To create a new picture, select the **New Picture** item. This will create a new picture called **`NoName`** in the editor. If any unsaved changes have been made to a previously loaded picture, a *save question* window will pop up, and the old picture can be saved before the new one is created.

Open Picture...

To open an existing picture, select the **Open Picture** item. A *file selection window* will pop up, and the desired picture file can be selected from the list of available pictures. To load the picture, click on the **OK** button. To cancel the operation, click on the **Cancel** button.

Save

To save a picture, select the **Save** item. If a picture has not been assigned a name different from **NoName**, a *save as* window will pop up, and you will be prompted to specify a path to where you want to save the picture and to give the picture a name.

Save As...

To save a picture with a new name select the **Save As...** item. A *save as* window will pop up, and you will be prompted to specify a path to where you want to save the picture and to provide a name for the picture. To save the picture, click on the **OK** button. To cancel the save operation, click on the **Cancel** button.

Document

To make a document file of a picture, select the **Document** item. A document file of a picture is a text file containing the complete picture definition, written in the ProcSee Configuration Language. The document file will be placed in the same directory as the original picture file, and will be called <*picture name*>.Tdoc.

Print...

To make a PostScript[1] dump of a picture, select the **Print** item, and the *print picture window* will pop up. To print the PostScript dump of the picture on a suitable printer select one of the available printers from the printer list. To print the picture to a PostScript file, select the *Print to file:* toggle, and specify a name for a file to be printed to. To execute the print command, select the **Print** button.

1 PostScript is a registered trademark of Adobe Systems Incorporated.

To cancel the print command, select the **Cancel** button.

Figure 110

The figure shows the Print Picture window and the Printer Settings window. A picture loaded in the drawing editor can be printed to a printer by selecting the desired printer, or printed to a file by specifying the filename. From the Printer Settings window the layout and the paper colour tuning can be specified.

To adjust the printer settings for a selected printer, select the **Printer Settings** button, and the Printer Settings window will pop up. In the Picture Settings window paper size (or the size of the PostScript dump) and margins can be specified. The *Colour tuning(%):* area is used to tune colours for printers. The colours shown on a computer screen often vary from those provided by a colour printer. The colours can be tuned to the desired extent to make colour printouts look as similar to the picture on the screen as possible.

The *Print Command:* item specifies the command to be used to print the picture to a printer. Available option characters are *%P* and *%F*. In the execution of the command *%P* is replaced with the printer name, and *%F* is replaced with the PostScript dump of the picture currently displayed in the drawing editor.

Comments specific to a printer can be added in the *Printer Info:* area.

To add a new printer to the printer list, a file named *<printerName>.pprn* should be placed in the directory where the graphics editor is started. To add a printer named *printer1*, a file named *printer1.pprn* should be created. If the file is empty, default values will be used for the printer settings. The layout of the .pprn file is described in the *"ProcSee Reference Manual, exportPostScript()"*.

Close To leave the drawing editor, select the **Close** item. If the currently displayed picture contains any unsaved changes, a Save question window will pop up, and you can save the picture by clicking on the **OK** button. To exit without saving the picture, click on the **Don't Save** button

Configure printing on Windows NT

ProcSee only supports postscript printers, and prints by copying to the printer port **LPT**.

In a network, you have to do the following to get printing to work from GED: Make sure you know the name of the printer(s) you want to use, and what server they are connected to.

Start the command line shell ("Command Prompt" in the **Start Menu**)

Use the NET USE shell command to map a printer to a parallell-port. This is the way to do it:

net use LPT<a number of your choice (1-9)>: \\<server-name>\printername /persistent:<**yes**/**no**>

It is recommended that you use **LPT2** and upwards, since your computer probably already has a parallellport attached (this is **LPT1** by default).

/persistent:**no**

- Means that the printer mapping is for this session only. The next time you restart / log on to your computer, the printer mappings will be lost.

/persistent:**yes**

- Means that your computer remembers the connection the next time you restart / log on to your computer.

Example:

If your Printer is called "ColourPrinter2", and the server it is attached to is called PrintServer and you want it attached to **LPT2**, the command line would look as follows:

```
net use LPT2: \\PrintServer\ColourPrinter2
```

Then you have to configure GED to use LPT2. This is done in the print menu.
Click the "Printer settings" button, and on the "Print Command" line, replace **LPT1**: with **LPT2**: and click **OK**.

Printing should now work correctly.

13.3 The Edit Menu

The edit menu contains editing support functions, such as Cut, Copy, Paste, etc. (Figure 111). Each of the Edit menu choices will be described below. To use any of the editing functions, one or several graphic objects must first be selected. To select an object, set the drawing editor in edit mode by clicking on the arrow icon on the tools menu. When the drawing editor is in edit mode, select an ob-

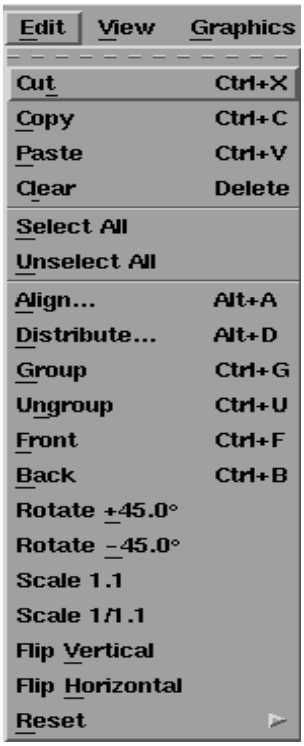


Figure 111

The figure shows the Edit menu in the drawing editor. The Edit menu contains editing support functions.

ject by clicking on it using the mouse pointer (refer to Select Tool in section 13.5 *"The Graphics Menu"* on page 211). A selected object will be displayed with its editing handles (small squares displayed on the object).

Cut

To remove one or more selected objects from a picture, select the **Cut** item. The selected objects will be removed from the picture, and placed on the clipboard. Objects that are cut to the clipboard can be pasted back into a picture later using the **Paste** command.

Copy

To copy one or more selected objects in a picture, select the **Copy** item. The selected objects will be copied to the clipboard. To put a copy of the objects into a picture, use the **Paste** command.

Paste

The **Paste** command will copy whatever is on the *clipboard* to a picture. The objects that are copied from the clipboard to the picture will be selected in the picture (the editing handles are displayed if the picture is in edit mode).

Clear

Same as **Cut**, but the removed objects will not be placed on the clipboard.

Select All

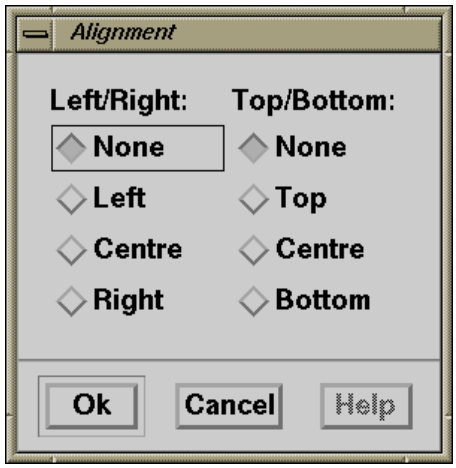
To select all of the elements in a picture, select the **Select All** command. All elements in the picture will then be displayed with their editing handles.

Unselect All

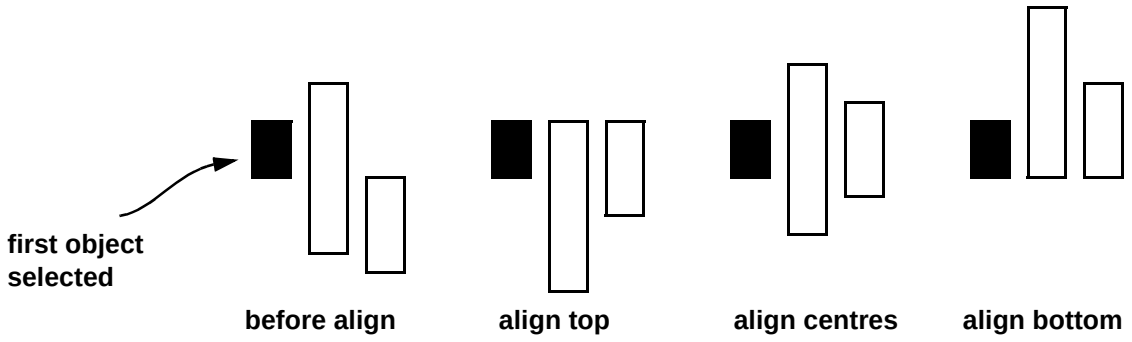
To unselect all elements in a picture select the **Unselect All** command.

Align...

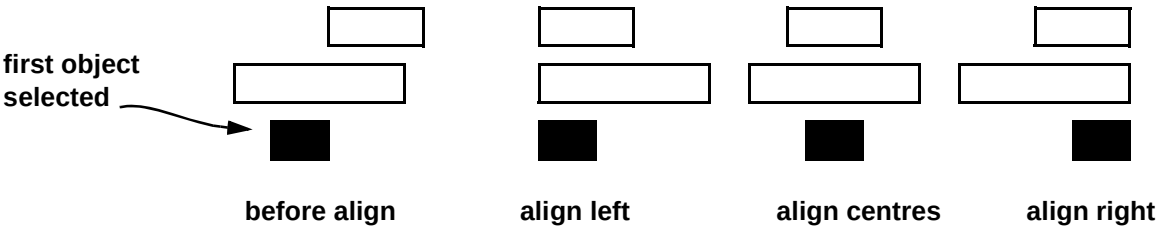
To align objects, use the **Align** command, and the *Alignment window* will appear. ProcSee aligns objects according to the first object selected. Objects can be aligned horizontally and vertically, with centre and sides as alignment base. If you want to align in one direction only, specify None for the direction that is not required in the Alignment window.



To align objects along a horizontal line, select Top, Centre, or Bottom from the Top / Bottom area of the Alignment window (remember to select None on the Left / Right area)



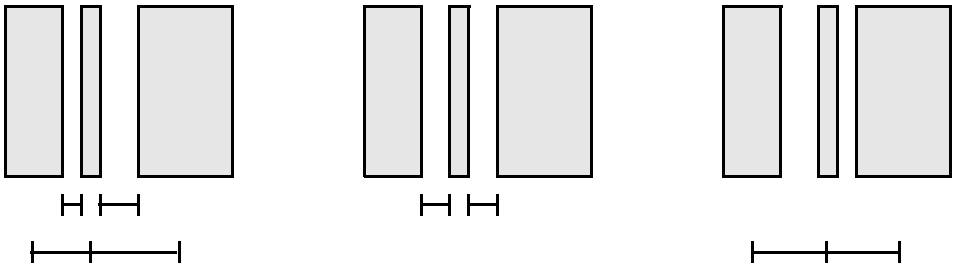
To align objects along a vertical line, select Left, Centre, or Right from the Left / Right area of the Alignment window (remember to select None on the Top / Bottom area).



Distribute...

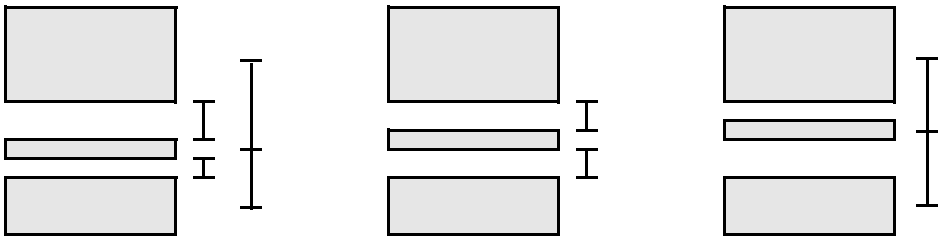
To distribute objects, select the **Distribute** command. Objects can be distributed horizontally and vertically, with centre and edges as distribution base. If you wish to distribute in one direction only, select None for the direction that is not required.

To distribute objects horizontally, select Edge or Centre from the Left / Right area and None from the Top / Bottom area of the Distribution window.



before distribute **distribute edge** **distribute centre**

To distribute objects vertically, select Edge or Centre from the Top / Bottom area and None from the Left / Right area of the Distribution window.



before distribute **distribute edge** **distribute centre**

Group

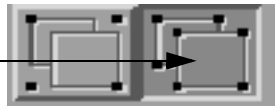
To group two or more selected objects, select the **Group** command. Objects that are grouped are treated as one object by all editing commands. Alternatively, the group button in the lower left corner of the drawing editor can be used, as illustrated below.

Button to group the selected objects.

Ungroup

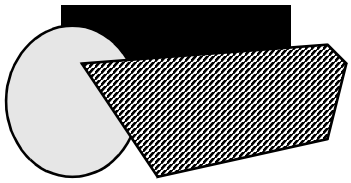
To ungroup grouped objects, select the **Ungroup** command. The involved objects will then be treated as individual objects. Alternatively, the ungroup button in the lower left corner of the drawing editor can be used, as illustrated below.

Button to ungroup the selected grouped objects.

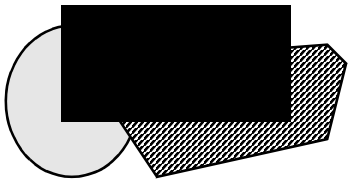


Front

In a picture, objects have an ordering according to their sequence of creation. To bring an object to the front of a picture, select the **Front** command.



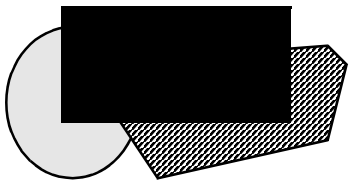
The rectangle is behind the ellipse and the polygon



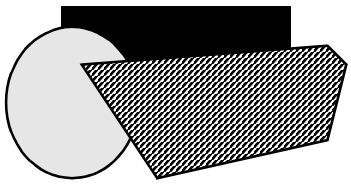
After using the Front command on the rectangle, the rectangle is on top of the ellipse and the polygon

Back

To move an object to the back of a picture select the **Back** command.



The rectangle is on top of the ellipse and the polygon

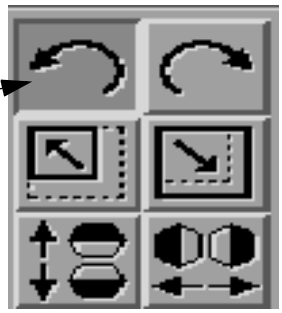


After using the Back command on the rectangle, the rectangle is behind the ellipse and the polygon

Rotate +45.0°

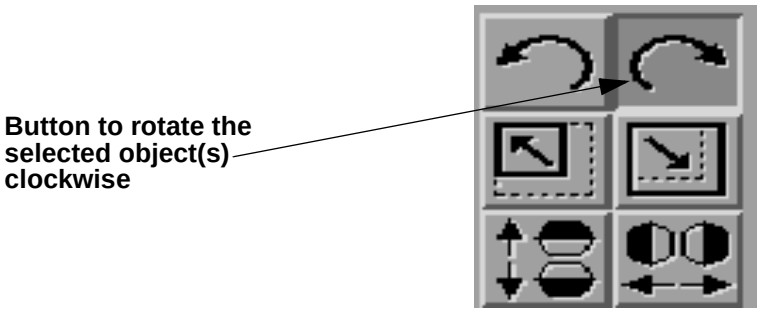
This command will rotate the selected object(s) with an angle of +45 ° (counter clockwise). The angle can be specified in the **Editing Options** window, see section 13.7 *"The Options Menu"* on page 220. Alternatively, the counter clockwise rotate button in the lower left corner of the drawing editor can be used, as illustrated below.

Button to rotate the selected object(s) counter clockwise



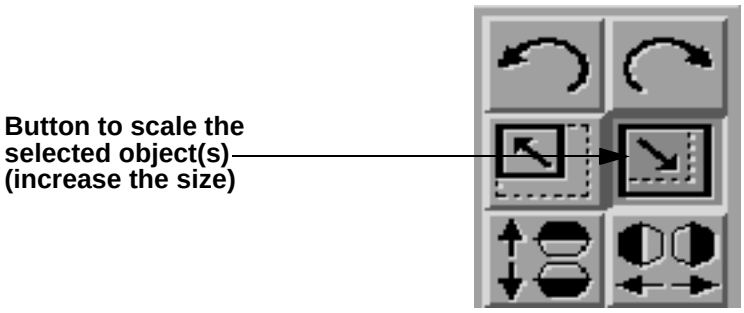
Rotate -45.0°

Same as above, except the selected object(s) is rotate with a negative angle (clockwise). Alternatively, the clockwise rotate button in the lower left corner of the drawing editor can be used, as illustrated below



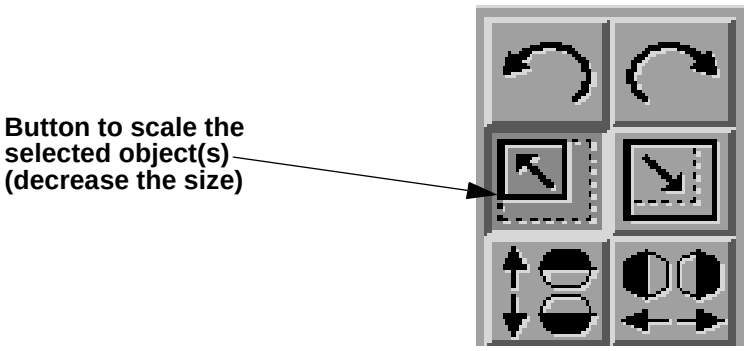
Scale 1.1

This command will scale the selected object(s) with a factor of 1.1. The scale factor can be specified in the **Editing Options** window, see section 13.7 *"The Options Menu"* on page 220. Alternatively, the scale 1.1 button in the lower left corner of the drawing editor can be used, as illustrated below



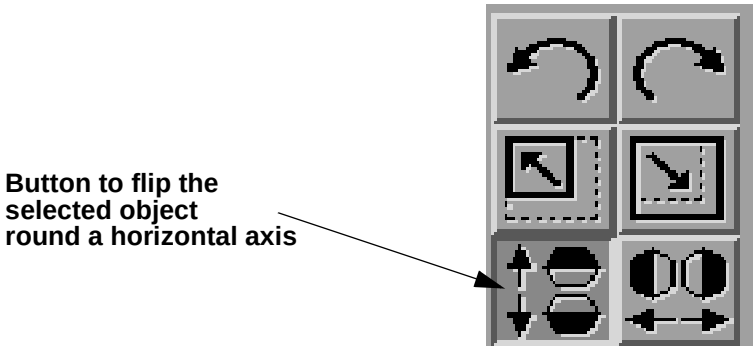
Scale 1/1.1

This command will scale the selected object(s) with a factor of 1 / 1.1. The scale factor can be specified in the **Editing Options** window, see section 13.7 *"The Options Menu"* on page 220. Alternatively, the scale 1 / 1.1 button in the lower left corner of the drawing editor can be used, as illustrated below



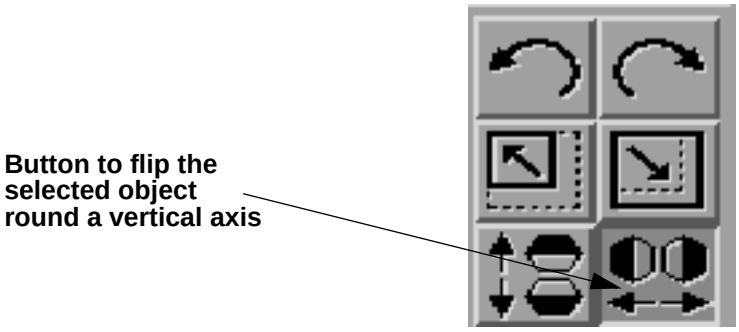
Flip Vertical

This command will flip the selected object around a horizontal axis.Alternatively, the vertical flip button in the lower left corner of the drawing editor can be used, as illustrated below.



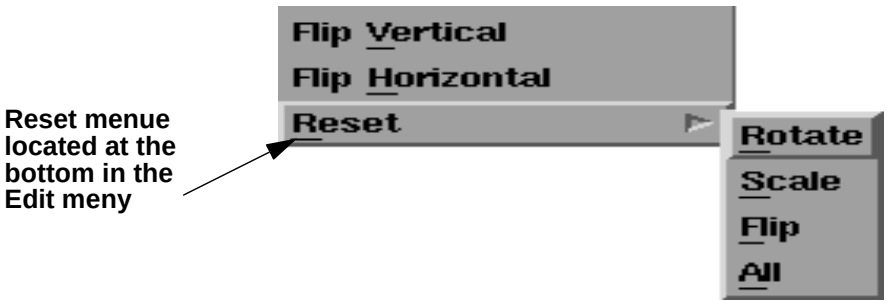
Flip Horizontal

This command will flip the selected object around a vertical axis. Alternatively, the horizontal flip button in the lower left corner of the drawing editor can be used, as illustrated below.



Reset

With this command, either the rotate, scale, flip or all three actions



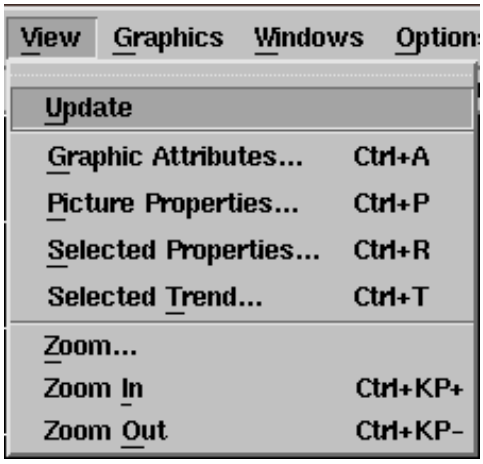
will be reset to its original positions.

13.4 The View Menu

The View menu contains choices for displaying and changing at-

Figure 112

The View menu contains choices for displaying graphics attribute for selected objects, and for displaying picture data, as well as setting the zoom factor for the picture.



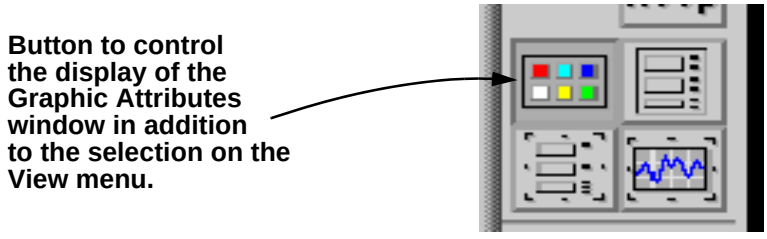
tributes and dialogues for selected objects, as well as displaying and changing attributes, functions, and dialogues for a picture. The zoom factor for the picture can be set by choosing the **Zoom** item.

Update

To refresh a displayed picture, as well as update associated property windows in the drawing editor, choose the **Update** command.

Graphic Attributes...

To pop up the graphic attributes window, select the **Graphic Attributes** item. The graphic attributes window can be displayed through an alternative method, by selecting one of the *property window buttons* located in the lower left corner of the drawing editor, as illustrated below.



The graphic attributes menu can be used to change the properties of one or more selected objects. The small window on the right side of Figure 113 shows the available attribute values for the selected attribute.

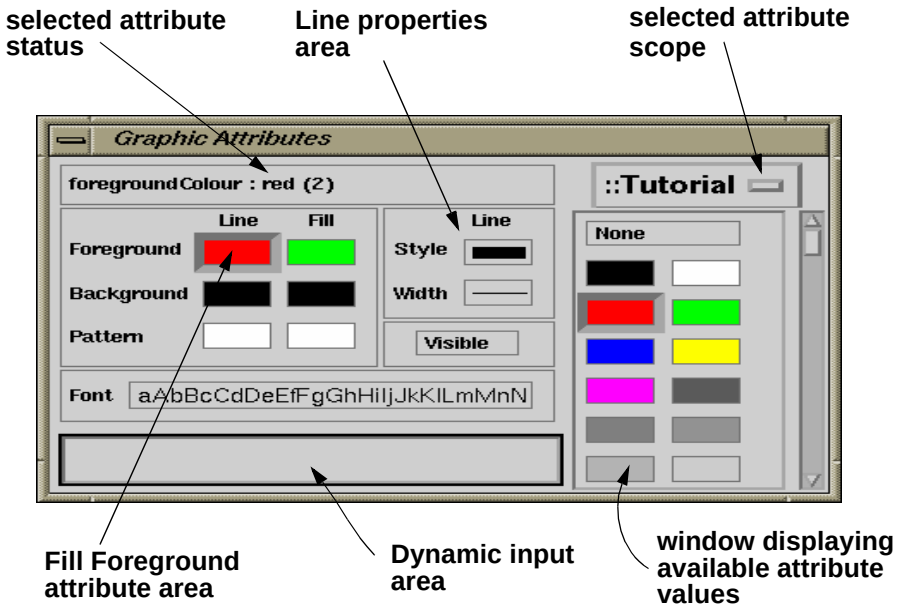


Figure 113

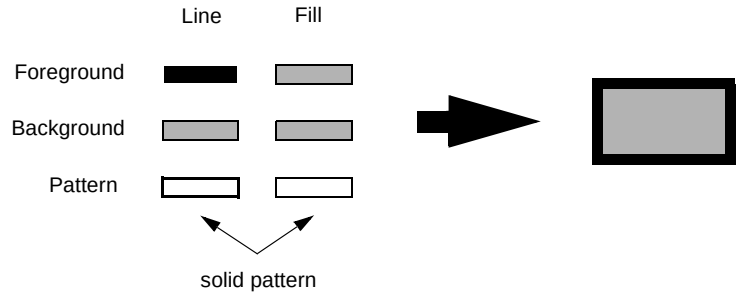
The figure shows the Graphic Attributes property window. This property window is used to set/change the graphics properties for one or several selected objects.

The graphic attributes window can be used to set the default attribute values for an editing session. Set the attributes to the desired default when no objects in a picture are selected.

The attribute values window can display a colour palette, a font palette, a pattern palette, a line width and style palette, and a visibility palette. The font palette contains the fonts specified in the application resource file. The colour palette contains the colours defined in the application resource file. The visibility palette contains only two choices: visible and invisible. To set an attribute, click in the appropriate attribute area (Line Foreground, Fill Background, Line Width, etc.), and select an attribute value from the attribute values window.

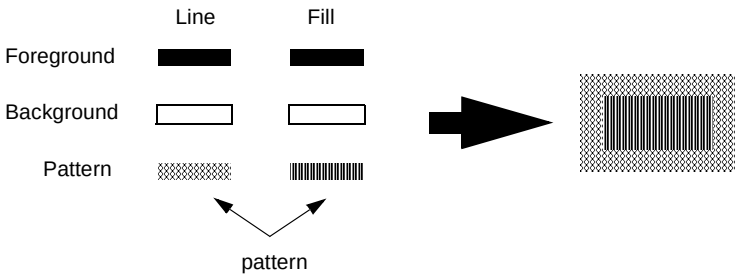
The graphic attributes window is also used to select the scope of the resources such as colours, fonts, patterns, etc. The default scope is the application. Resources can also be defined in libraries, and the resources in a library is selected by addressing the attribute scope button in the upper right corner of the graphic attribute window.

To draw a rectangle with a solid colour border and a solid colour foreground, select solid for Line Pattern and Fill Pattern. In this case

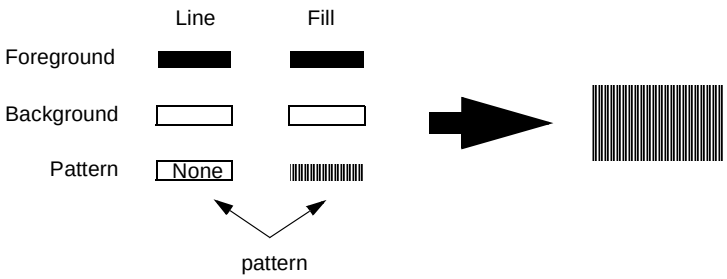


the Background attribute will not have any effect since the pattern attribute is set to Solid.

To draw a rectangle with a border pattern and a foreground pattern, select the desired patterns for both Line and Fill. Then select the desired colours for Line Foreground and Background, and Fill Foreground and Background. In the example below, the Foreground colours have been set to black and the Background colours have been set to white. The rectangle is drawn using patterns with the selected colours.



To draw a rectangle without a border, set the Line Pattern attribute to None, as illustrated in the following example.

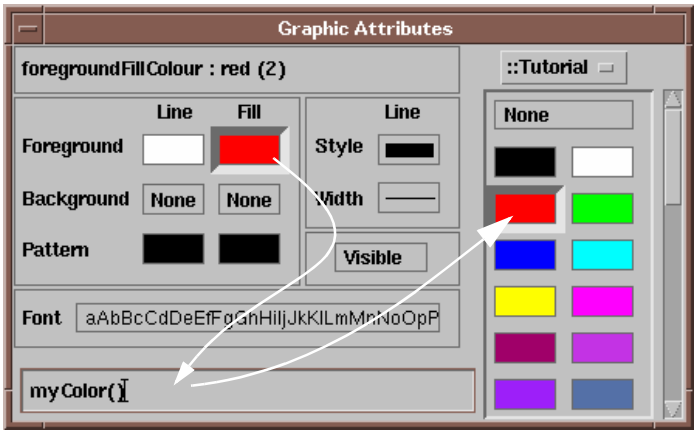


To set an attribute to a dynamic value, select the attribute by clicking on the appropriate attribute area. To indicate a dynamic attribute value, type the desired statement in the pTALK language in the

dynamic input area. End the input of the pTALK statement by pressing the Return key on the keyboard. Now, the attribute will reflect the evaluated value of the user defined function. In the example in Figure 114, the Fill Foreground attribute is dynamically connected to the user defined function **`myColor()`** which produces a red colour for the foreground fill attribute.

Figure 114

The figure illustrates how an attribute can be dynamically connected to a user defined function myColor().

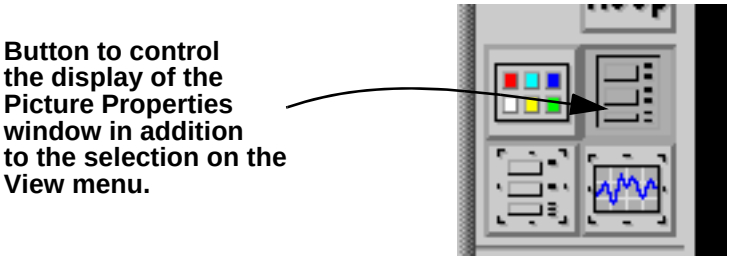


To assign a gradient to a pattern attribute, select line- or fill pattern by clicking on the appropriate attribute area. Next, click on a gradient pattern in the pattern palette. This action sets the line- or fill pattern attribute dynamically equal to a gradient. The shape is displayed with the gradient initialized using default values. To change the gradient pattern's default values, set the keyboard focus to the dynamic input area, and type the name **gradient** followed by the pattern name. After the pattern name, enter appropriate arguments and press return. This may change the visualization of the gradient. For more detailed description of the pTALK function gradient, see the Gradient description in the standard library reference in the Reference Manual.

Picture Properties...

To create, edit, or delete picture properties, such as attributes, functions, and dialogues for the loaded picture, select the **Picture Properties** item and the Picture properties window will pop up (see

Figure 115 on page 197). The picture properties window can also be displayed through an alternative selection method, using one of the property window buttons as illustrated below:



A picture contains predefined attributes, 5 of which concern rotation and scaling of the picture. The *xReference* and *YReference* attributes specify the reference point for rotation. The *rotationAngle* attribute specifies the rotation angle in degrees. The *xScaleFactor* and *yScaleFactor* specify the scaling in the x and y directions respectively.

For a description of how to create, edit, and delete attributes, functions, and dialogues, refer to Attributes, functions, and dialogues in chapter 5.

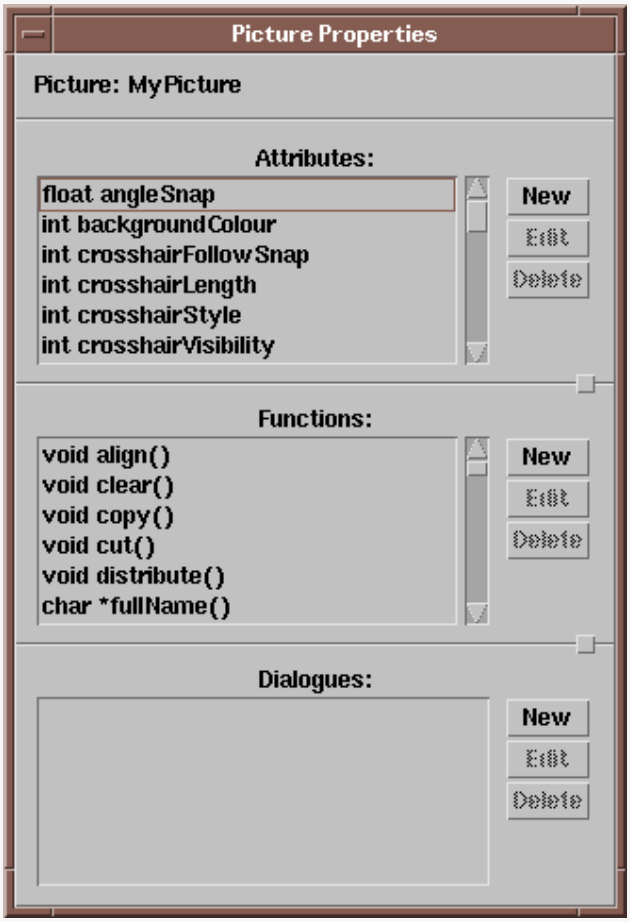


Figure 115

The figure shows the Picture Properties window, where picture attributes, functions, and dialogues can be created, edited, and deleted.To operate, select the desired data item in one of the lists, and select the desired operation on the buttons next to the list.To create a new data item, click on the appropriate new button.

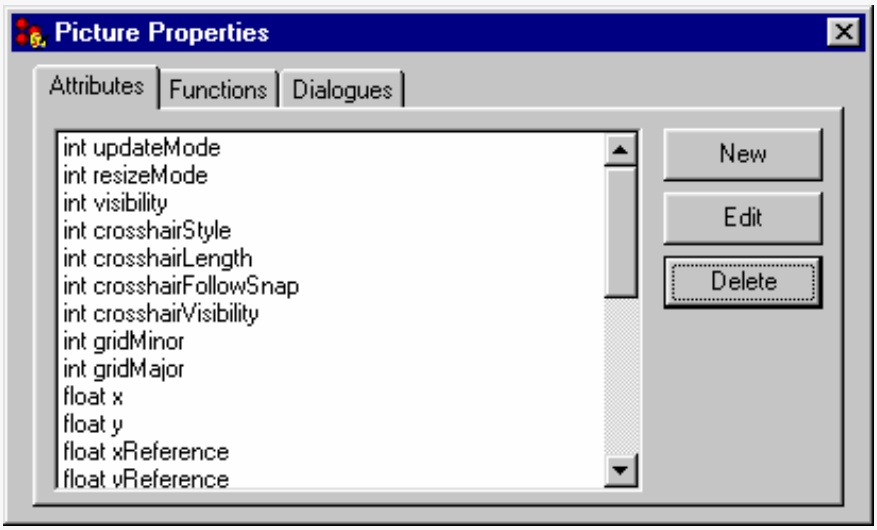


Figure 116

The figure shows the Picture Properties window, where picture attributes, functions, and dialogues can be created, edited, and deleted.To operate, select the desired data item in one of the lists, and select the desired operation on the buttons next to the list.To create a new data item, click on the new button.

Selected Properties...

To edit the properties of a single selected object, select the **Selected Properties** item, and the Selected properties window will then appear (Figure 117). New values can be assigned to the attributes of

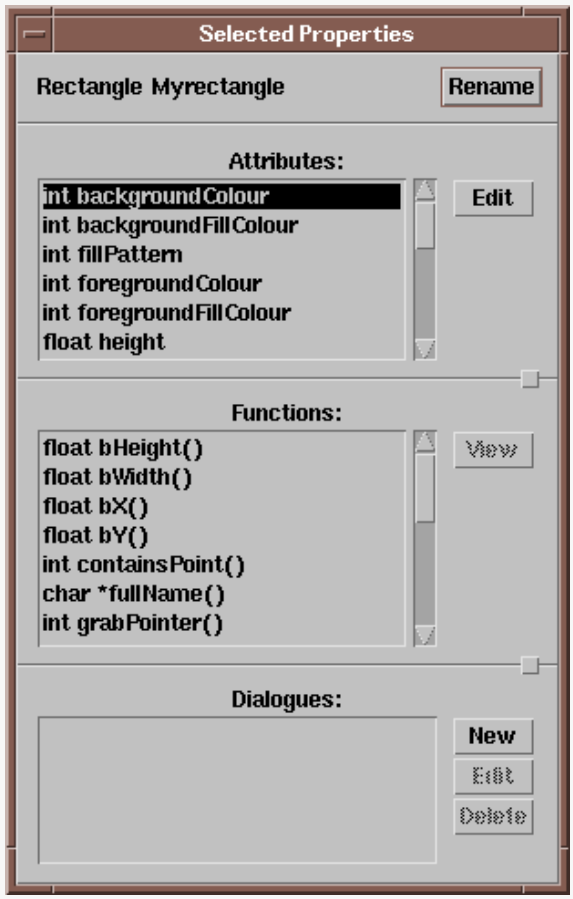


Figure 117

The figure shows the Selected Properties window. The window will show the attributes, functions, and dialogues for the selected object. The type of object and a possible name will be displayed. If the selected object does not have a name, an identification number will be displayed. In the example the selected object is of type Rectangle and the name of the rectangle is MyRectangle.

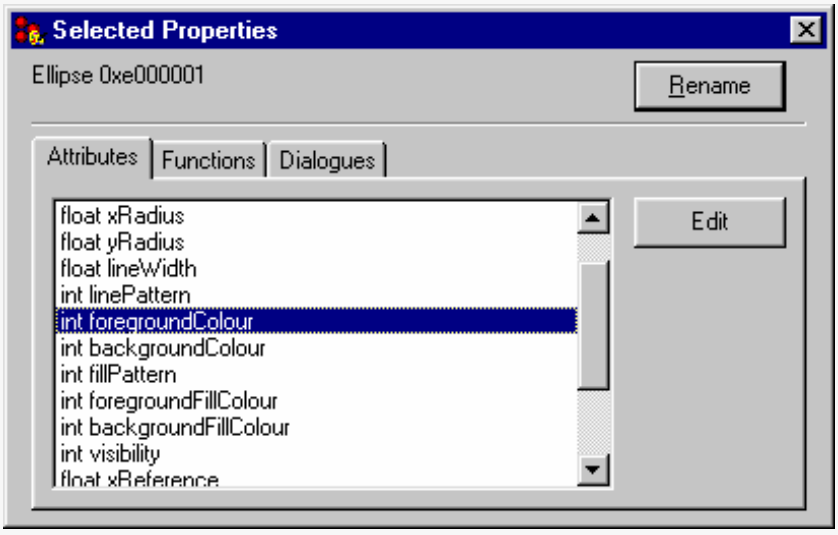
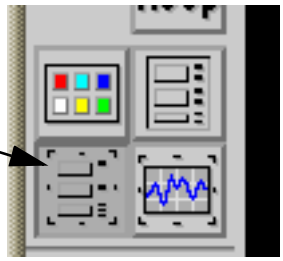


Figure 118

The figure shows the Selected Properties window. The window will show the attributes, functions, and dialogues for the selected object. The type of object and a possible name will be displayed. If the selected object does not have a name, an identification number will be displayed. In the example the selected object is an Ellipse.

the selected object. The attribute definitions themselves cannot be changed or deleted. Functions defined for a selected object can be viewed, but not modification. Dialogues can be freely created, edited, and deleted. Alternatively, the selected data window can be displayed by selecting one of the property window buttons as illustrated below.

Button to control the display of the Selected Properties window in addition to the selection on the View menu.



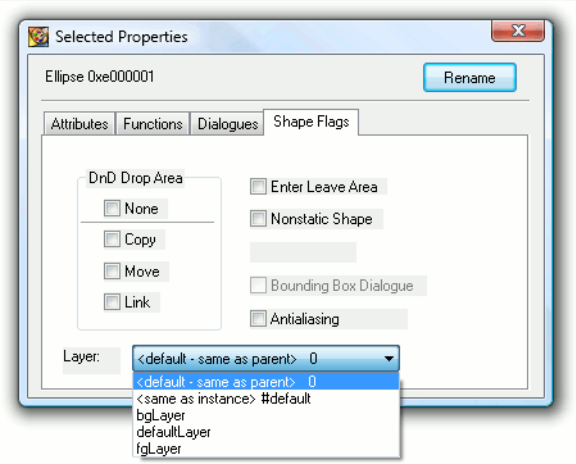
All graphic objects contain predefined attributes, 5 of which concerns rotation and scaling of the object. The *xReference* and *YReference* attributes specifies the reference point for rotation. The *rotationAngle* attribute specifies the rotation angle in degrees. The *xScaleFactor* and *yScaleFactor* specifies the scaling in x and y direction respectively.

For a description of how to create, edit, and delete attributes, functions, and dialogues, refer to section 9.15 *"Attributes, Functions and Dialogues"* on page 144.

To rename a shape press the rename button in the selected properties dialogue. From the window that appear type in the unique name that you want to give the shape and press the **Ok** button.

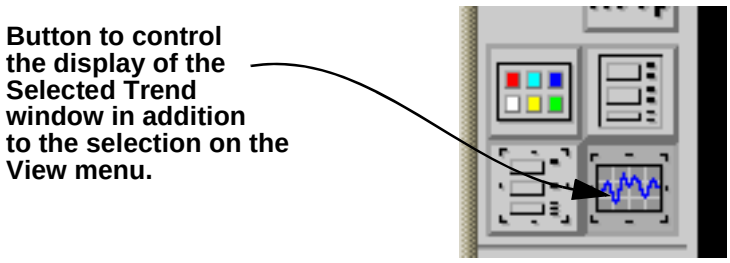
Figure 119

The figure shows the Shape Flags pane of the Selected Properties window. Here it is possible to mark the selected shape as a Drag and Drop target, and to set other shape flags, like antialiasing. The layer to use for this shape can also be set here. The default is to use the same layer as the parent shape (if the shape is inside e.g. a group shape. For shapes at the picture level, the default is to use the default layer.)



Selected Trend...

To edit the properties of a single selected trend object, select the **Selected Trend** item, and the Trend properties window will pop up (Figure 120). The selected trend window can alternatively be displayed by selecting a property window button as illustrated below.

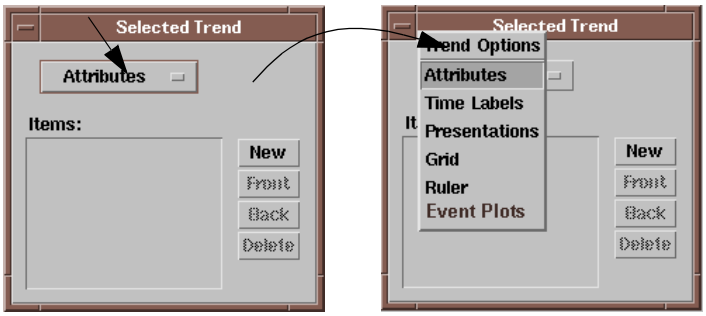


A trend diagram currently consists of five parts: global trend attributes, a number of trend presentations or variables to be trended, a number of time labels, a number of trend rulers, as well as trend grids.

The trend log definition, the trend configuration file, and the trend diagrams are described in detail in *"The ProcSee Reference Manual, The Configuration Language"*

Figure 120

The figure shows the Selected Trend menu. Trend attributes, trend presentations, and trend labels can be created, deleted, and modified from this menu



Trend Attributes

To edit any of the trend properties, select the desired part from the Trend Properties menu. The background of a trend consists of a rectangle or an ellipse. You can modify any of the background attributes, such as foregroundColour, backgroundColour, foregroundFillColour, backgroundFillColour, pattern, fillPattern, linePattern, and lineWidth.

The attributes that are specific to the trending are *timespan*, *trLog*, *orientation*, *scroll*, and *updateMode*.The *timespan* attribute specifies the timespan in seconds for which the trend diagram will show his-

toric data. The *trLog* attribute specifies the name of the trend log from where the trend diagram should retrieve data. The trend log should be defined in the main resource file.

The *orientation* attribute specifies whether the trend should be trended left-to-right, right-to-left, or circular. The scroll behaviour (scrolling of trend curve when trend diagram is filled) is specified in the *scroll* attribute. The *updateMode* specifies whether the trend should be updated as a pixmap and if overlapping objects should be redrawn or not.

These attributes can be edited as described for *Selected Properties.*

For a detailed description of the trend attributes and their usage, refer to the *ProcSee Reference Manual*.

Trend Presentation If the trend presentation item is selected, a list of available **Trend Presentation** items will be displayed in the *Trend Properties* menu. By selecting one of these, the available attributes will be displayed in the *Selected Properties* menu. Trend presentations can be added to a trend diagram by selecting the **New** button on the right side of the trend properties window. A trend presentation can be renamed by selecting the desired trend presentation in the trend properties menu, and then selecting the **Rename** button from the selected properties menu.

Currently, historic trends can be presented as lines, filled curves, or filled curves with changes in steps. To modify the appearance of a trend presentation, attributes such as foregroundColour, backgroundColour, linePattern, and lineWidth can be changed.

The attributes that are specific to a trend presentation are *variable, logFrequency, upperLimit, lowerLimit, autoScale, presentation, upperBand,* and *lowerBand.*

The *variable* attribute specifies the variable to be trended either as a character string or as a pTALK expression. The *logFrequency* specifies (in seconds) how often the variable is logged and plotted. There should be an entry in the trend variable configuration file corresponding to the *variable* and *logFrequency* specified in the trend presentation.

The *upperLimit* and *lowerLimit* attributes specifies the value range for which the variable will be plotted. The *autoScale* attribute specifies whether the trend presentations should be automatically scaled to the curve´s highest and lowest value and thus ignoring the trend presentation attributes *upperLimit* and *lowerLimit*. In the case of an automatic scale operation the *upperBand* and *lowerBand* attributes are used to determine the new value range.

The *presentation* attribute specifies how the trend values should be presented graphically. The available options are line, filled curve, or filled curve with changes in steps.

For a detailed description of the trend presentation attributes and their usage, refer to *The ProcSee Reference Manual*.

Trend Time Label If the trend time label item is selected, a list of available **time label** items will be displayed in the *Trend Properties* menu. By selecting one, the available attributes will be displayed in the *Selected Properties* menu. Time labels can be added to a trend diagram by selecting the **New** button on the right side of the trend properties window. A time label can be renamed by selecting the desired tag in the trend properties menu, followed by selecting the **Rename** button from the selected properties menu.

To modify the appearance of a time label, attributes such as foregroundColour, backgroundColour, theFont, and format can be changed. The formats supported are similar to those supported by the function *strftime(3C)*.

The attributes specific to the trending are *alignment*, *interval* and *offset*. The *alignment* attribute specifies how the individual time labels are aligned. The available options are left, right, or centred. The *interval* attributes specify (in seconds) the interval between time texts in the time label. The *offset* attribute specifies the distance from the trend area to the time label. If this is a negative value, the time label is placed above the trend area, otherwise it is placed below.

The attributes can be edited, as described for *Selected Properties.*

For a detailed description of the time label attributes and their usage, refer to *The ProcSee Reference Manual*.

Trend Ruler If the trend ruler item is selected, a list of available **Trend Ruler** items will be displayed in the *Trend Properties* menu. By selecting one, the available attributes will be displayed in the *Selected Properties* menu. Trend rulers can be added to a trend diagram by selecting the **New** button on the right side of the trend properties window. A trend ruler can be renamed by selecting the desired ruler in the trend properties menu, followed by selecting the **Rename** button from the selected properties menu.

To modify the appearance of a trend ruler, attributes such as foregroundColour, backgroundColour, linePattern, and lineWidth can be changed.

An attribute specific to a trend ruler is *rulerIsSelected*. If the rulerIsSelected attribute is set to 0, the ruler is disabled.

The attributes can then be edited as described for *Selected Properties.*

For a detailed description of the trend ruler attributes and their usage, refer to *The ProcSee Reference Manual*.

Trend Grid If the trend grid item is selected, a list of available **Trend Grid** items will be displayed in the *Trend Properties* menu. By selecting one, the available attributes will be displayed in the *Selected Properties* menu. Trend grids can be added to a trend diagram by selecting the **New**

button on the right side of the trend properties window. A trend grid can be renamed by selecting the desired grid in the trend properties menu, followed by selecting the **Rename** button from the selected properties menu.

To modify the appearance of a trend grid, attributes such as foregroundColour, backgroundColour, linePattern, and lineWidth can be changed.

The attributes specific to a trend grid are *depthPosition, noOfLines, timeInterval*, and *timeOffset*. The *depthPosition* attribute determines where the grid is placed; behind or in front of the curves. The number of horizontal lines is determined by the *noOfLines* attribute, whereas the number of vertical lines is determined by the *timeInterval* attribute. The *timeOffset* attribute specifies position of the first vertical grid line relative to the earliest time currently shown in the trend diagram.

For a detailed description of the trend grid attributes and their usage, refer to *The ProcSee Reference Manual*.

Event Plots

If the event plot item is selected, a list of available **Event Plots** items will be displayed in the *Trend Properties* menu. By selecting one, the available attributes will be displayed in the *Selected Properties* menu. Event plots can be added to a trend diagram by selecting the **New** button on the right side of the trend properties window. An event plot can be renamed by selecting the desired event plot in the trend properties menu, followed by selecting the **Rename** button from the selected properties menu.

The attributes specific to the event plot shape are *event, shape, theTrendPres* and *offset*. The *event* attribute specifies the name of the event type. The *shape* attribute specify the shapes to create for the events. Either a pTALK class or a pTALK function. The attribute *theTrendPres* specifies whether the event shapes should follow the amplitude values of the trend presentation or not. The *offset* attribute specifies the distance from the position the shapes had been given unless this *offset* had not been specified. If this is a negative value the event shapes are moved up, otherwise down.

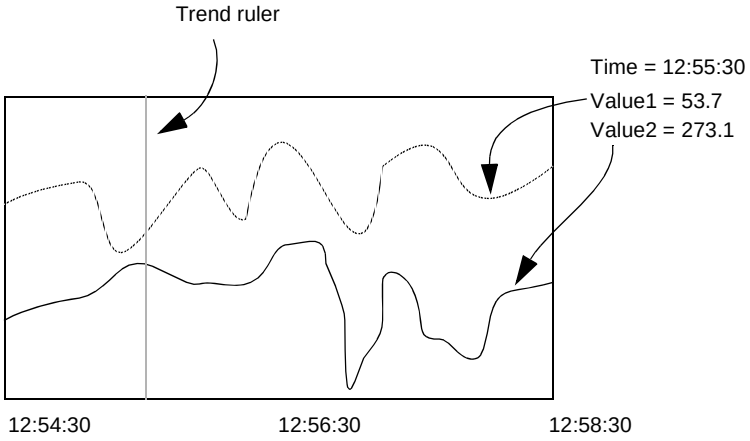
The attributes can then be edited as described for *Selected Properties.*

For a detailed description of the event plot attributes and their usage, refer to *The ProcSee Reference Manual*.

Using the Trend Ruler Typically, one trend ruler is used per trend diagram. The trend, trend presentation, and trend ruler contain a set of functions that can be used to define the operation of the trend ruler.

Figure 121

The figure illustrates a typical use of the trend ruler. As the mouse cursor is moved across the trend diagram, the Value1, Value2, and Time texts are updated accordingly. In this example Value1 contains the current value of the dashed trend line, while Value2 contains the current value of the solid trend line



Below is a description of a typical trend ruler set up.

The trend ruler is visible only when the mouse cursor is within the trend diagram area, and is moved according to the mouse movements. As the mouse cursor is moved, the Time, Value1, and Value2 texts are updated with the current values of the trend curves.

Three user defined dialogues are specified in the trend diagram to control the trend ruler; one that turns the trend ruler on as the mouse cursor is moved inside the trend diagram, one that turns the trend ruler off as the mouse cursor is moved outside the trend diagram, and one that moves the trend ruler as the mouse cursor is moved within the diagram.

In this example, three text elements (Time, Value1Txt, and Value2Txt) are used to display the resulting trend ruler values as the ruler is moved across the trend diagram.

"" on page 363 contains a detailed example of how the trend ruler can be used in a trend diagram. This example relates to the illustration given in Figure 121.

Using Historic Trends The creation of historic trend is performed in three steps:

- Define a trend log in the main resource file.
- Define the variables to be trended in a trend configuration file.
- Create graphic trend diagrams in pictures.

In the current version of ProcSee, only trend diagrams can be specified and edited using the graphics editor. The trend log definition and the trend variable configuration file both have to be defined in the document (Tdoc file) format using a text editor. The document format is compiled using *pcc(1)*.

An introduction to the ProcSee trend features is given in chapter 16 *"Historic Trends"* on page 237.

Active Shapes & Layers...

In the drawing editor click on the *Active Shapes & Layers* button in the edit toolbar or the *Active Shapes & Layers*... item in the *View* menu. This action opens the *Active Shapes & Layers* dialogue window, see Figure 122.

This button controls the Active Shapes & Layers dialogue window.

All active shapes features supported by ProcSee are provided in this dialogue. In addition the layers created in the application and the libraries are displayed. Notice that new layers cannot be created, removed or modified in this dialogue. This functionality is available from the *Layers* dialogue window in the GED's main window.

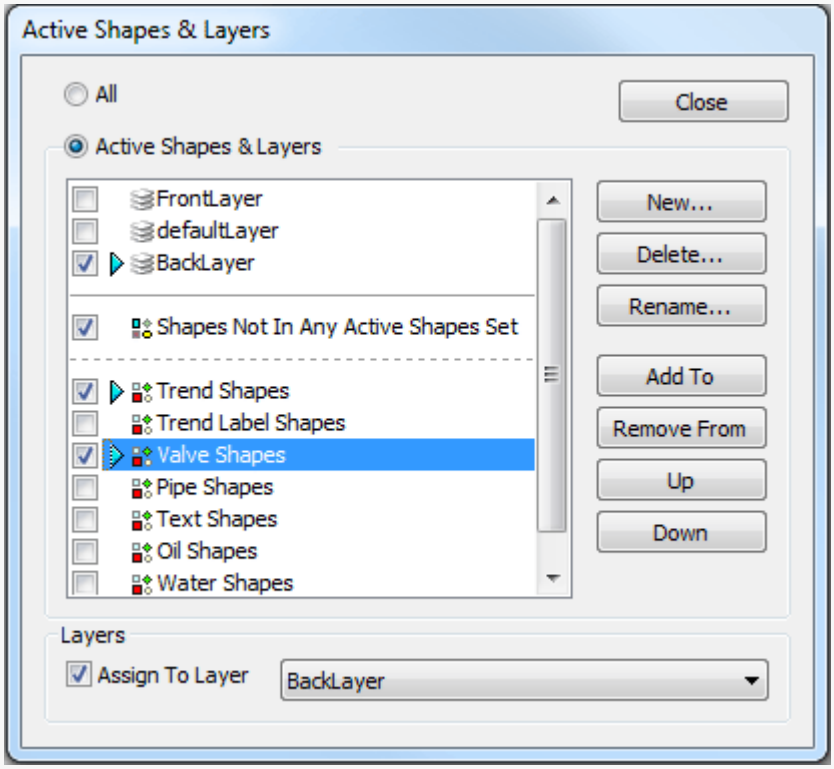


Figure 122
This figure illustrates the Active Shapes & Layers dialogue in the drawing editor.

The *Active Shapes & Layer* list displays the layers in the application and the libraries, and the active shapes sets created in the picture. This list is divided in three parts separated by horizontal lines (separators). The upper part displays the layers, the middle part displays the shapes not in any active shapes sets, and the lower part displays the active shapes sets. Different icons are used for the different parts in order to visually distinguish the different type of entries from each other, see Figure 107.

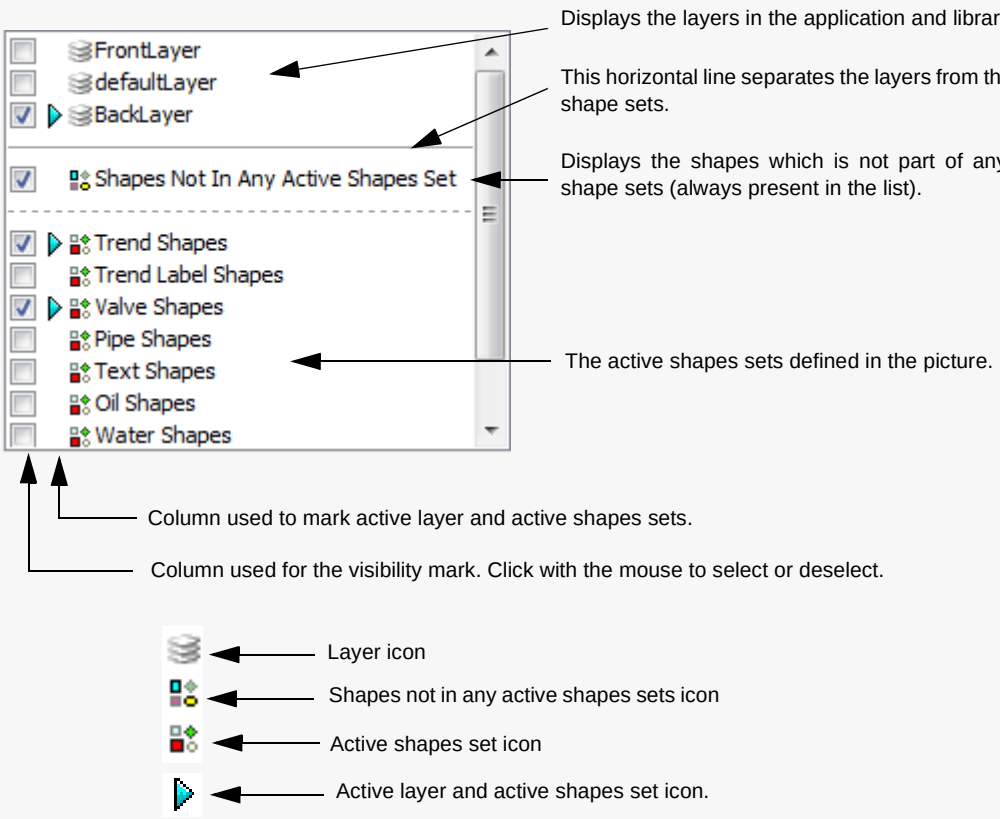


Figure 123 *This figure illustrates and describes the Active Shapes & Layer list.*

The *Active Shapes & Layers* dialogue window provides the following functionality (described in detail later in this section):

- Activate or deactivate layer and active shapes set.
- Create new, delete and rename active shapes set.
- Add shapes to, and remove shapes from layer and active shapes set.
- Move active shapes entries up or down in the active shapes set list.
- Assign layer automatically to new shapes created in the editor.

Activate or deactivate layer and active shapes set

The dialogue provides functionality to activate or deactivate layer and active shapes set. The radio buttons *All* and *Active Shapes & Layers* are mutually exclusive, meaning that only one of these can be selected simultaneously.

When *All* is selected, all shapes in the picture are active and selectable.

When selecting the *Active Shapes & Layers* radio button, only shapes in layers and active shapes set entries marked with the visibility mark are active and selectable. An entry is marked if the visibility check box associated with an entry is checked, see Figure 123. Click on the check box to activate or deactivate a layer or an active shapes set entry in the list.

Create new, delete and rename active shapes set

Click on the *New*... button to create a new active shapes set. This action opens the *New Active Shapes* dialogue box, see Figure 124.

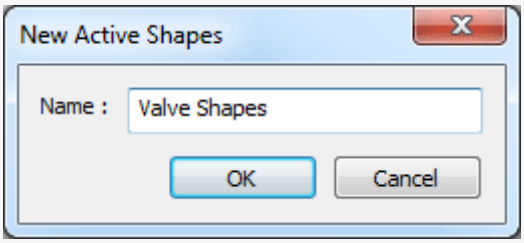


Figure 124 *The New Active Shapes set dialogue window.*

In the text field enter the name of the active shapes set and press the *OK* button. A new active shapes set will be created in the picture with the specified name. Notice that spaces are legal characters in an active shapes set name. Each active shapes set must have unique names. The shapes selected in the picture will automatically be assigned to the new active shapes set.

Press the *Delete*... button to delete an active shapes set. A confirmation dialogue box will appear on the screen. Press the *Yes* button to accept, or the *No* button to reject the delete operation.

To rename an existing active shapes set, press the *Rename*... button. This action displays the rename dialogue box where the new name of the active shapes set can be entered, see Figure 124.

Notice that none of these features can be applied to layers.

Add shapes to and remove shapes from layer and active shapes set

Pressing the *Add To* button adds the selected shapes in the picture to the layers or the active shapes sets in the list marked with the active mark symbol, i.e. a right arrow symbol. The active mark symbol for layers is mutually exclusive, meaning that a shape can be assigned to one and only one layer.

The active layer can be changed from the combo box located to the right of the *Assign To Layer* check box in the lower part of the dialogue. For active shapes sets, the active mark is always marked if the visibility check box is marked.

Notice that the special active shapes set, *Shapes Not In Any Active Shapes Sets*, cannot be marked as active. This set only contains shapes not included in any of the other sets. Shapes cannot be added to, or removed from this set. The contents of this set depends entirely on the other active shapes sets.

Adding shapes to active entries are a four-step process. First, mark the active shapes sets and/or the layer in the list with the active icon. Next, make the *"Shapes Not In Any Active Shapes Sets"* entry visible, or select the *"All"* radio button in the upper part of the dialogue. Next, select the shapes in the picture to add to the layer or the active shapes sets. Finally, press the *Add To* button. These steps will add the selected shapes to the active entries.

To remove shapes from a layer or active shapes sets just select the shapes to remove in the picture and press the *Remove From* button. This will automatically remove the shapes from the active shapes sets. Notice that the layer settings for a shape will be cleared if and only if the active layer in the dialogue matches the shape's layer setting.

Move active shapes entries up or down in the active shapes set list

Active shapes sets can be moved up or down in the active shapes sets list. Pressing the *Up* button will move the selected entry one step up, while the *Down* button will move the selected entry down one step. This functionality does not influence on how the shapes in the picture are displayed. These buttons are used to group active shapes sets in the list.

Assign layer automatically to new shapes created in the editor

The *Assign To Layer* check box is used to automatically assign the active layer to new shapes created in the picture. To disable this functionality deselect the check box. The layer to assign to the new shapes is changed by selecting the layer in the combo box next to the check box.

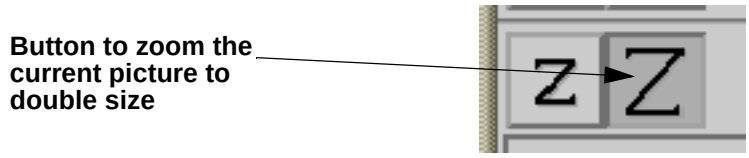
Zoom...

To zoom in or out a picture, select the **Zoom** item, and the zoom window will pop up. The picture currently loaded in the drawing editor can be zoomed by entering a percentage in the *Zoom size(%):* area, or by selecting from the predefined zoom sizes in the pop-up menu. The **Reset** button will set the zoom size to 100%. The **Zoom In** button will double the current zoom size, and the **Zoom Out** button will cut the current zoom size in half

To do the actual zooming, select the **OK** button

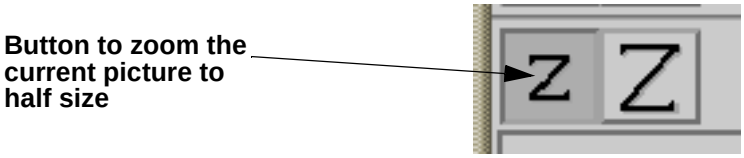
Zoom In

This selection will zoom the currently loaded picture to double of the current zoom factor. This command is similar to the larger of the zoom buttons in the zoom buttons menu on the lower left side of the drawing editor, as illustrated below.



Zoom Out

This selection will zoom the currently loaded picture to half of the current zoom factor. This command is similar to the smaller of the zoom buttons in the zoom buttons menu on the lower left side of the drawing editor, as illustrated below.

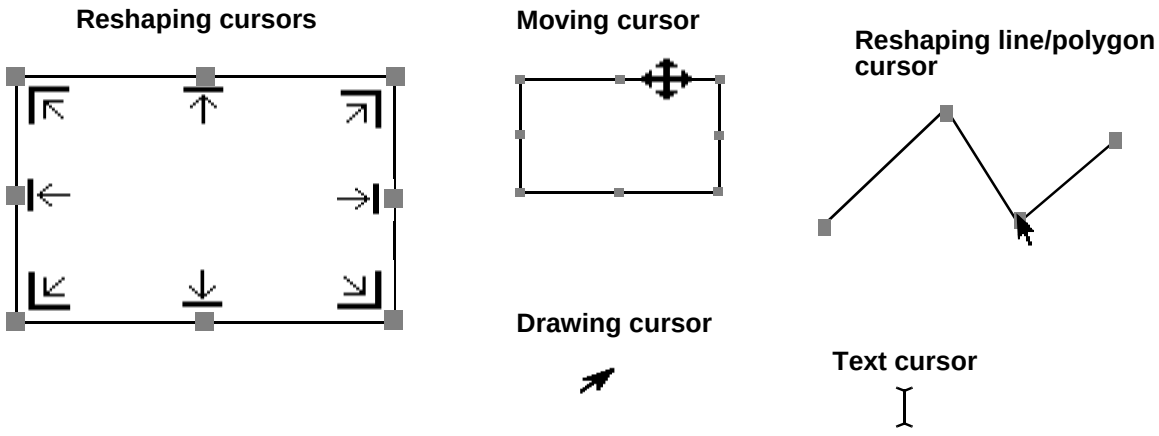


13.5 The Graphics Menu

The Graphics menu contains choices for drawing graphic objects. As illustrated in Figure 125, the Graphics menu has an equivalent menu in the tools menu, which is always accessible on the left side of the drawing editor.

The drawing editor will display helpful instructions in its message line as graphics commands are selected. For example, if you draw a circle, the message line will display: *Circle: Press left button for centre point, then drag.*

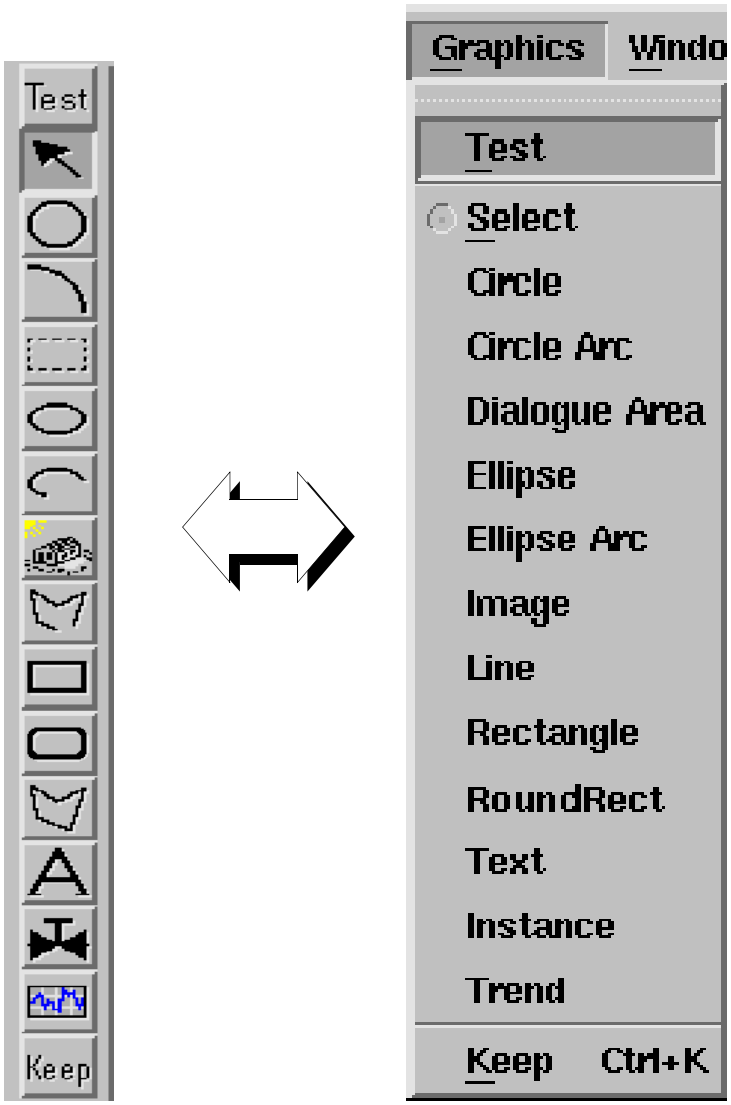
Below is a figure illustrating the different cursors used in the drawing editor. The cursor will change when moved into the corresponding editing handles areas.



Constant lineWidth To create shapes with constant line widths, the lineWidth attribute can be set to a negative value. This will ensure a non-scalable line width. A lineWidth of -6 will give a constant width of 6 that is independent of zoom factor.

Figure 125

The figure illustrates the Graphics menu, and the equivalent Tools menu.The Keep command is used to keep the current tool. If the rectangle is the current tool, and keep is activated, several rectangles can be created without having to select the rectangle button or menu item for each draw operation.



Editing Composite Shapes

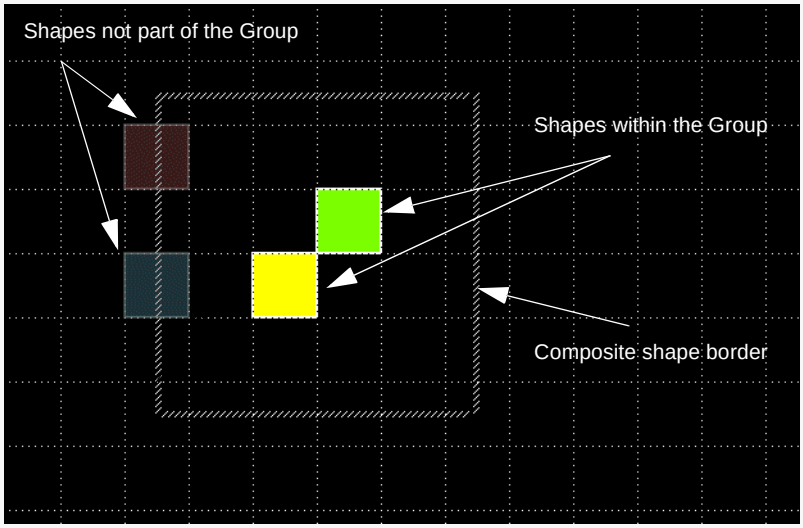
A composite shape in ProcSee is a shape containing other graphical objects, like Rectangle, Circle, etc. Examples of composite shapes in ProcSee are; Group, Trend and Instance. Sub-shapes are not directly accessible from the picture, which makes it more difficult from the editor to modify these shapes. Clicking with the left mouse button on a shape within for instance a Group, will not select the sub-shape, but instead the Group instance itself.

However, ProcSee provides functionality to edit composite shapes in the drawing editor. This functionality is by default enabled. Double clicking with the left mouse button on a Group or Trend shape will make the sub-shapes available for editing, just like any other shape located in the picture. When editing shapes in ProcSee, only shapes on the same level (in a hierarchy) are accessible from the drawing editor. To change the level, moving upwards in the hierarchy, double click with the left mouse button on a composite shape. Pressing the escape key will go back one level in the shape hierarchy (moving downwards). Notice that ProcSee only support composite editing of the Group and the Trend shapes.

Transparent colours are by default used to improve the users' understanding of the shapes which are available for editing in a picture. Shapes not directly available for editing are by default marked using transparent colours (this default can be changed). A shaded border is drawn around the area of the composite shape (with an offset), see Figure 126. This border will automatically be updated when sub-shapes are added, moved or resized. This functionality cannot be disabled.

Figure 126

This figure illustrates an active Group shape containing two rectangles. The shapes outside the Group are drawn using transparent colours.



To modify the default composite editing setting, see *"Advanced Editing Settings..."* on page 227.

Test

The **Test tool** is used to set the drawing editor into test mode. In this mode the graphical interface can be tested. Clicking on an object will NOT select an object as it would for the Select tool (described below) in edit mode. If the object clicked on has a dialogue where

the event is buttonPressed, the corresponding action will be executed. In the test mode, the command line can be used to set attribute / variable values interactively to verify dynamic relationships in an interface. The command line accepts any legal pTALK statement or expression. If a pTALK expression is entered, the result of the expression will be displayed in the message line.

Select

The **Select tool** is used to select one or more objects in a picture. It can be used in two ways:

- Click on the object you want to select. If you wish to modify the selection by adding or removing an object, hold down the <shift> key while clicking on it.
- To create a selection rectangle, hold the left mouse button down while moving the mouse. Move it until the desired objects are within the selection rectangle borders and then release the mouse button. If you wish to modify the selection afterwards, by adding or removing an object, hold down the <shift> key while clicking on the desired object. You can, of course, select or deselect any number of objects in this manner.

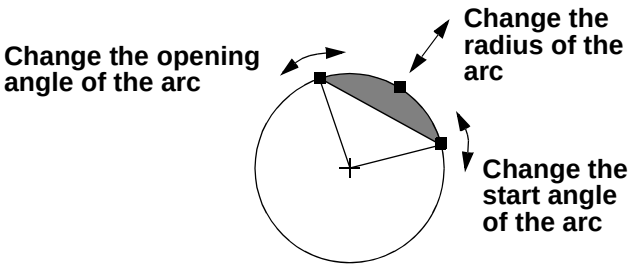
An object's editing handles will be displayed if it is selected. The mouse cursor will change as it is moved around the object and over its selection handles.

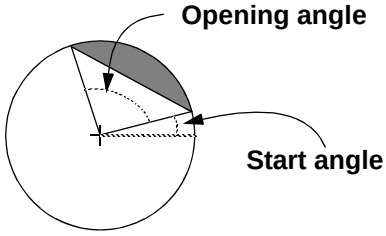
Circle

To draw a circle select the **Circle tool**. Place the mouse cursor in the picture where you want the centre of the circle. Then click on the left mouse button, and drag the circle to the desired size. The new circle will be displayed with its editing handles. To modify the circle, place the cursor over one of the editing handles and, while keeping the left mouse button down, drag to the new position.

Circle Arc

To draw a circle arc, select the **Circle Arc tool**. Place the mouse pointer in the picture where you want the centre of the circle arc. While keeping the left mouse button down drag to the desired size.





To modify the arc, move the mouse cursor into the appropriate editing handle and while keeping the left mouse button down, drag to the new shape / position. To ease the drawing of a circle arc, snap interval can be specified for both the angle and the radius. This option is located in the Options menu (refer to section 13.7 *"The Options Menu"* on page 220).

A circle arc can be drawn as a chord (illustrated above) or a slice.

Prior to the drawing operation, you must choose whether you want to draw a chord or a slice. This choice can be set in the Options menu (described in the following section)

Circle arc chord and circle arc slice are two different options in the "*Graphics Menu*"

Dialogue Area

The dialogue area tool is used to create a rectangular, sensitive area in a picture. The area is invisible when in test / run mode. To create the sensitive area, select the **Dialogue Area tool**. Place the mouse

cursor in the picture where you want the dialogue area. Then, while keeping the left mouse button down, drag the area to the desired size. To connect a dialogue to the area, bring up the **Selected Properties** window, and define the desired dialogue(s) as described in chapter 9 *"Basic User Interface Components"* on page 150.

Ellipse

To draw an ellipse, select the **Ellipse tool**. Place the mouse cursor in the picture where you want the centre of the ellipse to be. While keeping the left mouse button down, drag the ellipse to the desired size. To edit the ellipse, move the mouse cursor over one of the editing handles, and drag to the new shape / size, with the left mouse button down.

Ellipse Arc

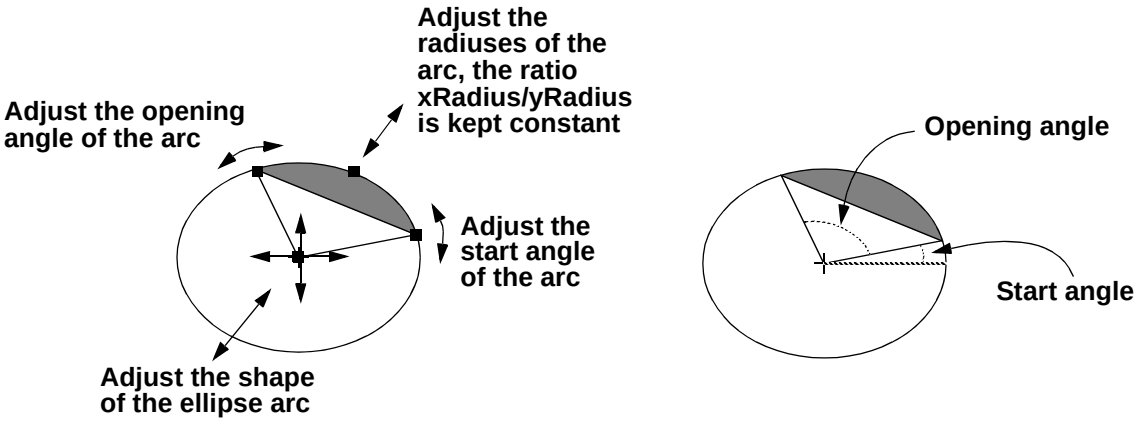
To draw an ellipse arc, select the **Ellipse Arc tool**. Place the mouse pointer in the picture where you want the reference point of the ellipse arc to be. With, the left mouse button down, drag to the desired size.

To modify an arc, move the mouse cursor over the appropriate editing handle and, with the left mouse button down, drag to the new shape / position. To ease the drawing of an ellipse arc, a snap interval can be specified for both the angle and the radius. This option is located in the Options menu.

An ellipse arc can be drawn as a chord (see circle arc) or a slice

Prior to the drawing operation, you must choose whether you want to draw a chord or a slice. This choice can be set in the Options menu (described in the following section)

Ellipse arc chord and ellipse arc slice are two different options in the "*Grahpics Menu*"



Image

The image tool lets you load a graphical image in to a picture.

To load an image, select the **Image tool** and a file selection box will pop up. Select the desired image file, and then click on the **OK** button to load the image. Use the left mouse button to position the image in the picture.

Line/Polyline

To draw a line with one or more line segments, select the **Line tool**. Use the left mouse button to click in the picture where you want the first point of the line. Move the mouse to the next position, click on the left mouse button to place the second line point, etc. To end the drawing of a line, click on the right mouse button. To edit a line (make sure the line is selected), move the mouse cursor over one of the editing handles, and it will change into a line editing cursor. With the left mouse button down, drag the line point to the desired position.

You can also add and delete points from the line. To delete a point, click the left mouse button over the point that is to be removed while keeping the control key down. To add a new point to a line, clicking the left mouse button over the desired line segment while keeping the control key down.

Rectangle

To draw a rectangle, select the **Rectangle tool**. Place the mouse cursor in the picture where you want to place the upper left corner of the rectangle, and with the left mouse button down, drag to the de-

sired size. To modify the rectangle, move the mouse cursor over one of the editing handles. With the left mouse button still down, drag to new size / position.

RoundRect

To draw a roundRect, select the **RoundRect tool**. Place the mouse cursor in the picture where you want to place the upper left corner of the roundRect, and with the left mouse button down, drag to the desired size. To modify the roundRect, move the mouse cursor over one of the editing handles. With the left mouse button still down, drag to new size / position.

Polygon

To draw a polygon, select the **Polygon tool**. Using the left mouse button, click in the picture where you want the first vertex of the polygon. Move the mouse to the next position, click on the left mouse button to place the second polygon vertex, etc. To end the drawing of a polygon, click on the right mouse button. To modify a polygon (make sure the polygon is selected), move the mouse cursor over one of the editing handles, and the mouse cursor will change to the polygon editing cursor. With the left mouse button still down, drag the polygon vertex to the desired position.

Points can be added and deleted in the same manner as for lines. See above for details.

Text

To place a line of text in a picture, select the **Text tool**. Using the left mouse button, click in the picture where you want the lower left corner of the text. The text "*enter a text and press <return>*" is displayed, with the text cursor at the start position of the text. Type your text and end the text input by pressing the <return> key on the keyboard. When you press the <return> key, the text cursor will be removed and the text will be displayed with its editing handles.

To edit or modify the actual text, double-click on it. The text cursor will be displayed. Use the arrow key to move the text cursor, or move the mouse pointer to the desired position and click on the left mouse button. To remove the text cursor, thus completing the modification process, press the <return> key.

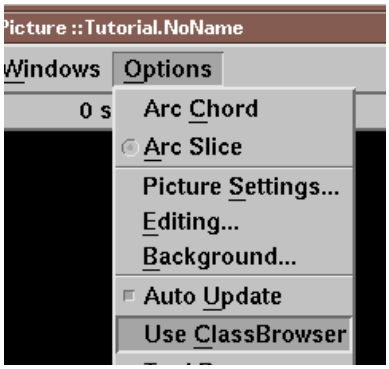
If the text is connected to a dynamic attribute, it will need to be modified via the *"Selected Properties"* window (refer to section 13.4 *"The View Menu"* on page 192). When adding dynamics to a text object, remember to specify an appropriate string format.

Instance

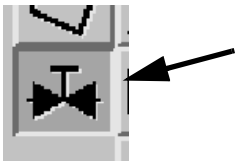
The classes may be presented in either an **Instantiate Class** window or in a **Class Browser** window. The selection is done in the **Options** pull-down menu. The **Instantiate Class** window will be used if **Use Class Browser** is *not* selected, see Figure 127. This selection must be done before the **Instance tool** is selected.

Figure 127

The Use ClassBrowser option in the Options pull/ down menu (default), and the Instantiate Class window.

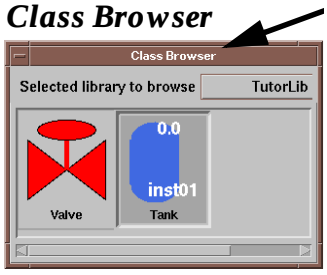


instances



Select the **instance tool** in the drawing editors Tools menu.

Class Browser



If **Use Class Browser** is selected, a browser with available libraries and classes appears.

Select the library where the class resides by clicking on the desired library name with the left mouse button. When a library has been selected, the classes contained in the library will be listed on the right side of the Instantiate Class window.

Select a class by clicking on the desired class' name with the left mouse button. Instances are placed in a picture by clicking on the left mouse button.

Trend

To place a trend diagram in a picture, select the **Trend tool**. Place the mouse cursor in the picture where you want to place the upper left corner of the trend. When the left mouse button is pressed down, a default trend diagram will be placed in the picture.To edit the trend object, move the mouse cursor over one of the editing handles. With the left mouse button pressed down, drag to adjust the size or position.

Keep

To keep the current tool active for subsequent drawing operations, select the **Keep tool**. For example, when drawing a rectangle without the Keep tool activated, the rectangle tool will be deactivated after each rectangle has been drawn and the select tool will become active, but, if the Keep tool is active, the rectangle button will remain active, usually until the keep tool is deactivated.

13.6 The Windows Menu

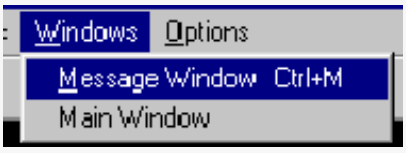
The Windows menu (see Figure 128 and Figure 129) contains one choice for displaying a message window and for Windows NT, also one choice for displaying the main GED window. The message window contains error messages as well as other information messages from the ProcSee RTM.

Figure 128

The figure illustrates the Windows menu in the X Windows (UNIX) version.

Figure 129

The figure illustrates the Windows menu in the Microsoft Windows NT version..



13.7 The Options Menu

The Options menu contain choices for picture configuration and the configuration of some graphic object defaults.

Figure 130

The figure shows the Options menu for the X Windows (UNIX) version.. The menu contains picture configurations and configuration items for some graphics commands.

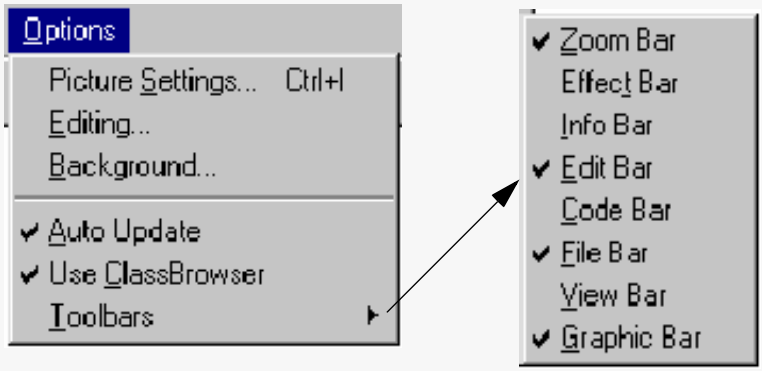


Figure 131

The figure shows the Options menu for the Windows NT version. The menu contains picture configurations and configuration items for some graphics commands.

Arc Chord

To set the default draw method for circle and ellipse arcs to chord mode, select Arc chord item from the Options menu.

Arc Slice

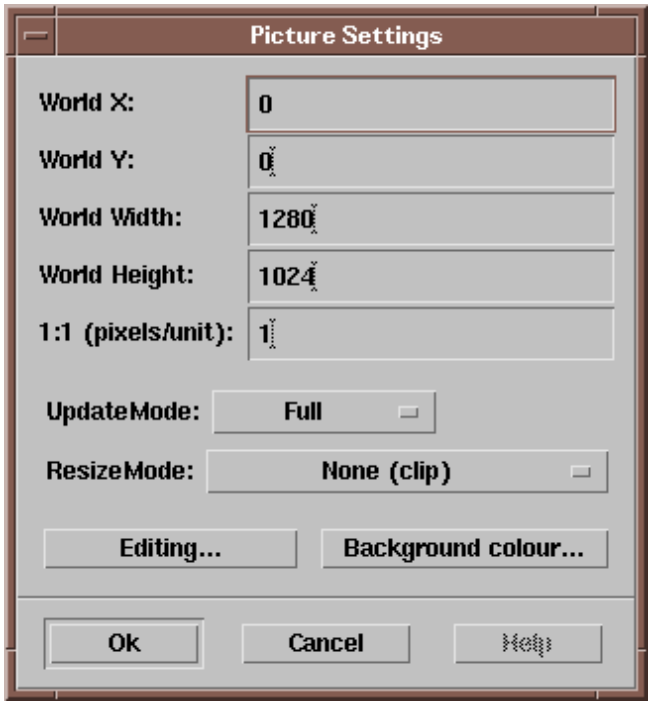
To set the default draw method of circle and ellipse arcs to slice mode, select Arc slice item from the Options menu.

Picture Settings...

To configure the picture settings, select the Picture settings item from the Options menu. The Picture Settings window will then pop up (Figure 132).

Figure 132

The figure shows the Picture Settings window. The picture coordinates are set in the World X, World Y, World Width, and World Height attributes, with World X and Y the upper left corner of the picture.To set the ratio between the screen coordinate system and the picture coordinate system, specify the new ratio in the (pixels/unit) attribute field .



Editing and Background colour buttons are greyed out in the *"Picture Settings"* dialogue.. Use the Options menu to activate these dialogues.

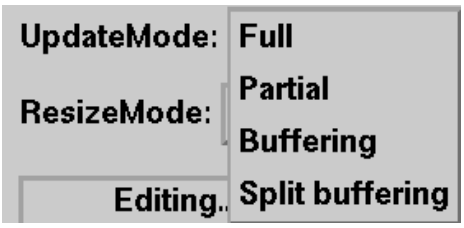
The coordinate system of a picture can be changed by modifying the World X, World Y, World Width, and World Height attributes. The upper left corner of a picture is given by World X and World Y. The size of a picture is given by World Width and World Height.

A picture coordinate system is defined independently of the underlying screen coordinate system (see section 12.2 *"The Coordinate System"* on page 166), often referred to as a pixel coordinate system (the resolution of the computer monitor hardware). To get a proper mapping of the picture coordinate system to the screen coordinate system, a ratio between the two is used. The ratio is defined as the number of pixels to a picture unit.

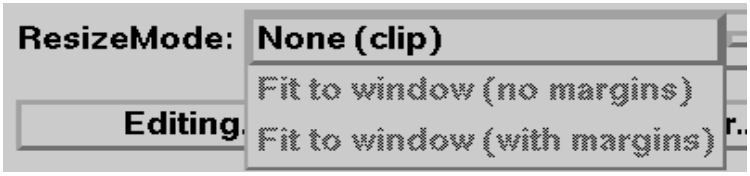
The picture can be configured to use one of four alternative update modes. Full update mode will guarantee that graphics are correctly updated regardless of dynamic changes. In Partial update mode, overlapping graphics are NOT guaranteed to be updated correctly.

Buffering mode and Split buffering mode also guarantee that graphics are correctly updated regardless of dynamic changes, but in these modes the graphics are drawn in a pixmap, and then copied into the window. The Buffering mode operates on one large pixmap, whereas the Split buffering mode operates on several small pixmaps.

The advantage of partial update mode, for some applications, is that it is faster than full update mode. The advantage of the buffering modes is that they will provide a smoother update of the picture. The buffering modes require more memory than the other modes, and may result in a heavier load on some systems.



A picture's relation to the window that it is displayed in can be configurable by changing the ResizeMode attribute.



The resize mode can be set to None, meaning that the picture will be clipped to fit within the window in which it is displayed in. In the current version, the Fit to window options are not available.

The Editing... and Background... buttons in the Picture Setting window have the same effect as the Editing... and Background... options in the Options menu.

Editing...

The **Editing Options** window is used to specify snap, grid, and crosshair settings (Figure 133).

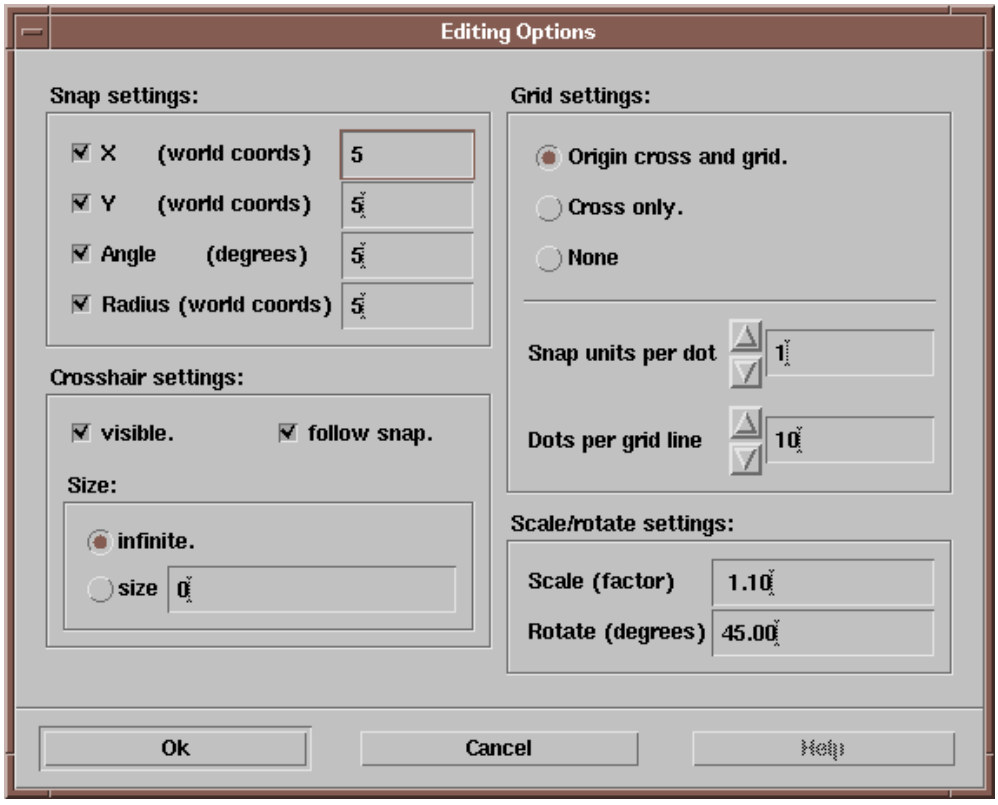


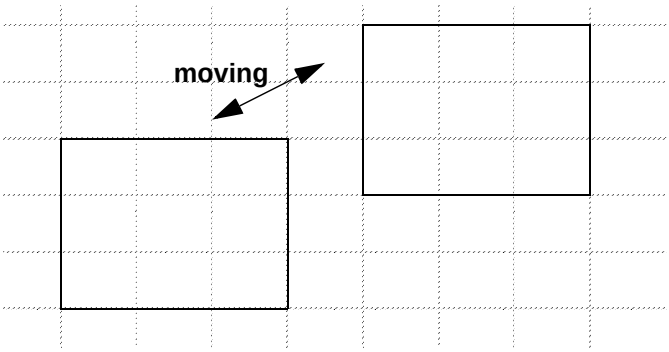
Figure 133 *The figure shows the Editing Options window. Snap, crosshair, and grid settings can be specified.*

Specify the snap settings to place an invisible grid[1] over a picture for easier editing. Snap intervals can be set for coordinates, angles, and radii. To set a snap interval, type the snap interval in the appropriate input field. The example in Figure 133 shows a snap of 5 for the x and y coordinates.

To deactivate a snap setting, click on the appropriate label (X, Y, Angle, or Radius), and a small square on the left side of the label will change to its deactivated mode. This is shown for Angle and Radius in the example in Figure 133.

1 The grid is, of course, visible when a picture is in edit mode

If a snap setting of 5 is used for the X and Y coordinates, a graphic shape such as a rectangle will be moved, or modified in any direction, in steps of 5.



To enable a crosshair cursor, select the visible toggle button in the Crosshair settings section. If the crosshair cursor should adhere to the snap settings, turn the follow snap toggle on. If the follow snap toggle is off, the crosshair cursor will have continuous movement.

The size of a crosshair cursor can be specified in world coordinate units. If an infinite size is selected the crosshair will span the visible part of a picture.

To set a visible grid, select the Origin cross and grid option. This will display grid lines with the coordinate (0, 0) clearly marked. Note that the origin cross is only visible if the origin is in fact in the visible part of a picture. If a picture's coordinate system is defined as follows (see Figure 132):

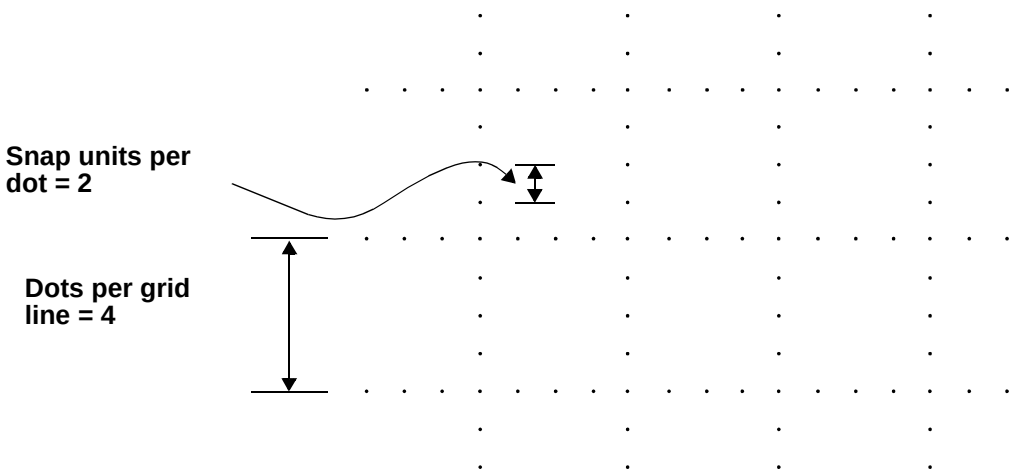
World X: -300
World Y: -300
World Width: 600
World Height: 600

the origin cross will be displayed in the centre of the picture.

If an only an origin cross is desired, select the Cross only option.

The None option indicates that no grid lines are required.

The appearance of the grid is determined by the Snap units per dot and the Dots per grid line settings. The Snap units per dot indicates how many snap intervals there are between the dots of the grid lines. The Dots per grid line setting indicates how many segments there are between each major grid line.



To apply the new settings, click on the **OK** button. To cancel the settings operation, click on the **Cancel** button (the original settings will be unchanged).

Note that you can also move selected objects around a picture that is in edit mode using the cursor keys on your keyboard. This allows for more accurate positioning of objects than is possible just using the mouse. If the shift key is down while using the cursor keys the objects are moved using current snap settings, otherwise it is moved pixel by pixel.

Advanced Editing Settings...

The *Advanced Editing Settings* dialogue window in the drawing editor is used to change the settings used by *ProcSee* when editing pictures, see Figure 134. The following properties can be changed from this window:

- Whether composite editing should be enabled, or not.
- Whether active shapes should be edited on top of other shapes, or not.
- How to visualize inactive shapes in the editor.
- What invisible shapes to show when in edit mode.

Normally, these settings are left as is, but they can be modified from this dialogue.

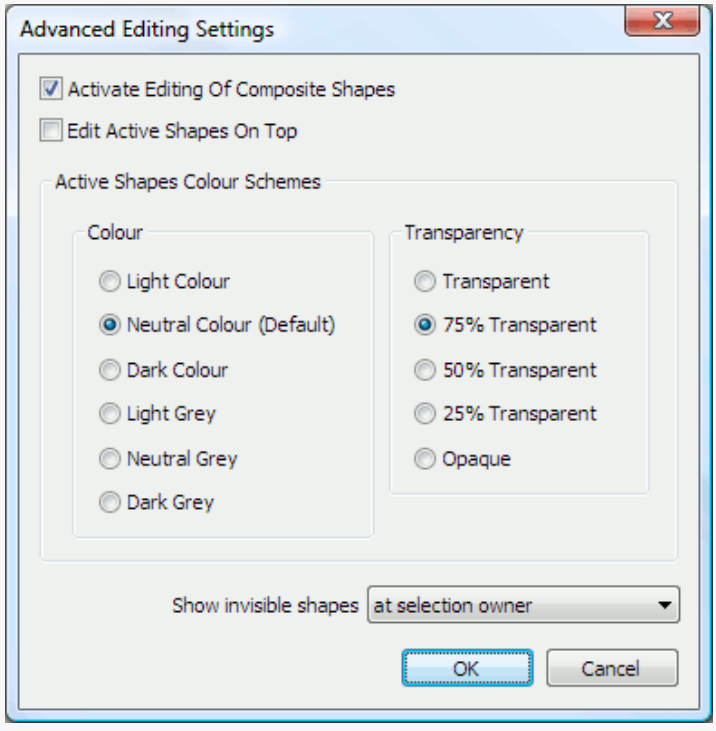


Figure 134
This figure illustrates the Advanced Editing Settings in GED. These settings are the default properties for the Advanced Editing Settings in ProcSee.

The check box *Activate Editing Of Composite Shapes* enables or disables the composite editing feature in ProcSee. The Group and Trend shape are the composite shapes supporting this feature. It is recommended that this check box is enabled, since it enhances the editing functionality in ProcSee.

The check box *Edit Active Shapes On Top* is a feature which makes it possible to edit active shapes on top of other shapes even if they are located below other shapes in the shape list order. It is by default off. This feature is used by composite editing, layer and active shapes set.

The *Active Shapes Colour Schemes* group defines how to transform the original shape colours to the colours to use when displaying the inactive shapes in ProcSee.

The left column of radio buttons (Colours) defines the colour to use. Two decisions about the original colours must be made:

- Whether to retain the colour or use a grey-scale colour.
- Whether the lightness of the original colour should be left as is, or made lighter or darker.

The default is to use a neutral colour (the colour is left as is).

The right column of radio buttons (Transparency) defines the transparency of the colour to use for inactive shapes. Select transparent if inactive shapes should be totally invisible, or opaque if the shapes should be drawn as if they where active. Select 25% to 75% to use semi-transparent colours. The default is 75% transparent.

Six different colour schemes and five transparency settings are available in this dialogue box. Notice that these colour settings are used by the active shapes functionality and the composite editing functionality in ProcSee.

The combo-box at the bottom of the dialogue window, is used to select how invisible shapes should be displayed when in edit mode.

There are 3 different possibilities: Show invisible shapes

- everywhere
- at selection owner
- at selection owner and parents

The setting "**everywhere**" makes all invisible shapes visible in edit mode (this is how it was in ProcSee up to and including version 3.8.2).

The selection owner is the picture level until the user goes into a composite shape (by double clicking on a composite shape). Then the selection owner is the composite shape.

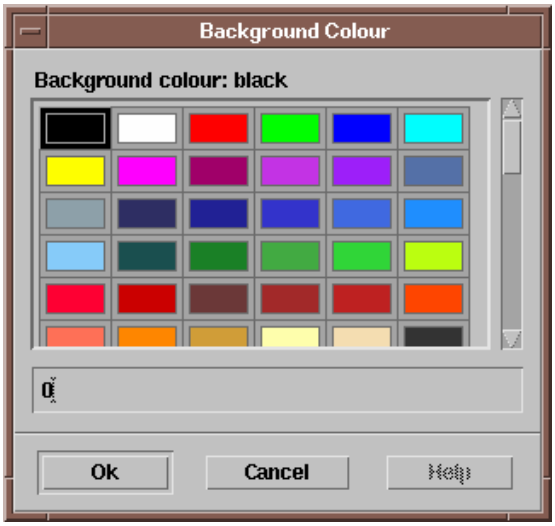
The setting "**at selection owner**" makes invisible shapes only visible at the selection owner. This is the default from ProcSee version 3.8.3. The setting "**at selection owner and parents**" also makes invisible shapes visible in the parents of the selection owner, like in the picture.

Background...

To select a different background colour for a picture, select the **Background...** option from the **Options** menu, and the Background window will pop up.

Figure 135

The figure shows the picture Background Colour window. To change the background colour of a loaded picture that is being edited, click on the desired colour. To fix the change in the picture, click on the OK button. To cancel the picture background setting operation click on the Cancel button.



To change the background colour of a loaded picture that is being edited, click on the desired colour. To fix change in the picture, click on the **OK** button. To cancel the picture background setting operation click on the **Cancel** button.

14 Testing the User Interface

In ProcSee, the testing of a user interface is made easier because a test facility is an integral part of the drawing editor. The interface designer can easily switch between designing and testing an interface, and all parts of the interface can be thoroughly tested at will.

14.1 Testing in the Drawing Editor

When designing user interfaces it is useful to be able to easily switch between *designing and testing*. In the ProcSee drawing editor, this is accomplished by setting the editor into it's test mode (see section 13.5 *"The Graphics Menu"* on page 211).

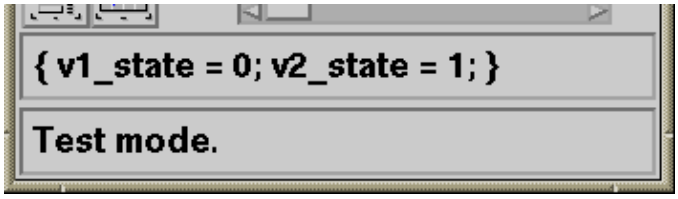
In the test mode, pTALK code can be written in the command line input field of the drawing editor, to assign new values to attributes and process variables used in the interface, as well as to issue arbitrary commands.

On the following pages, some examples of pTALK code will be given to illustrate the test mode features of the drawing editor.

Several pTALK statements (a compound statement) can be entered in the command line field. In this case, the pTALK statements should be enclosed in *curly braces* ***{ }***, as illustrated in Figure 136. In the case of a compound statement, no message is given in the message window.

Figure 136

When in test mode, pTALK *statements can be entered at the command line, to assign values to dynamic attributes in the interface. In this example two statements are entered, and hence enclosed in { }, according to the* pTALK *syntax.*



In the example above, two statements were included in the compound statement. However, if you use only one statement, the curly braces are not required. However, the statement must still be terminated by a semicolon (See the example in Figure 137). As in the above example, no message is given on the message line.

Figure 137

A single statement can be entered at the command line without curly brackets, but the statement still needs to be terminated by a semicolon.

In order to display the current value of an attribute / variable in the message window, enter an expression similar to the one shown in Figure 138. By typing arithmetic expressions at the command line, the test facility can be used as a calculator.

Figure 138

The current value of an attribute/variable can be displayed in the message window by including the attribute/variable in an expression, as illustrated in the figure.

.

The commands issued at the command line are stored in a command buffer. To recall the previous command, press <*ctrl-p*> (that is, hold down the ctrl-key and then press the p-key with the ctrl-key still down). To recall the next command after <*ctrl-p*> has been used, type <*ctrl-n*>.

The commands issued at the command line are stored in a command buffer. To recall the previous command, press the *arrow up* key and to recall the next command press the *arrow down* key

14.2 Testing with User Programs

If the actual user programs are available, they can be connected to the ProcSee Run-Time Manager, and the user interface can be tested in its real environment.

Dummy programs

However, it is often the case that the user programs are not available or are being developed in parallel with the interface. If this is the case it can be useful to develop dummy user programs that generate values for the variables used in the interface.

p3sim utility program

The ProcSee distribution includes a program called *p3sim* that can be used to easily generate a test environment for the user interface. The program provides generation of data for user specified variables. For a detailed description of the *p3sim* program, refer to *"The ProcSee Reference Manual"*.

By creating dummy programs, the user interface can be tested in an environment similar to the real run-time environment in which it will be used. Problem areas can be located and improved on before the actual user programs are connected to ProcSee for final testing.

Navigation in the main tree 15

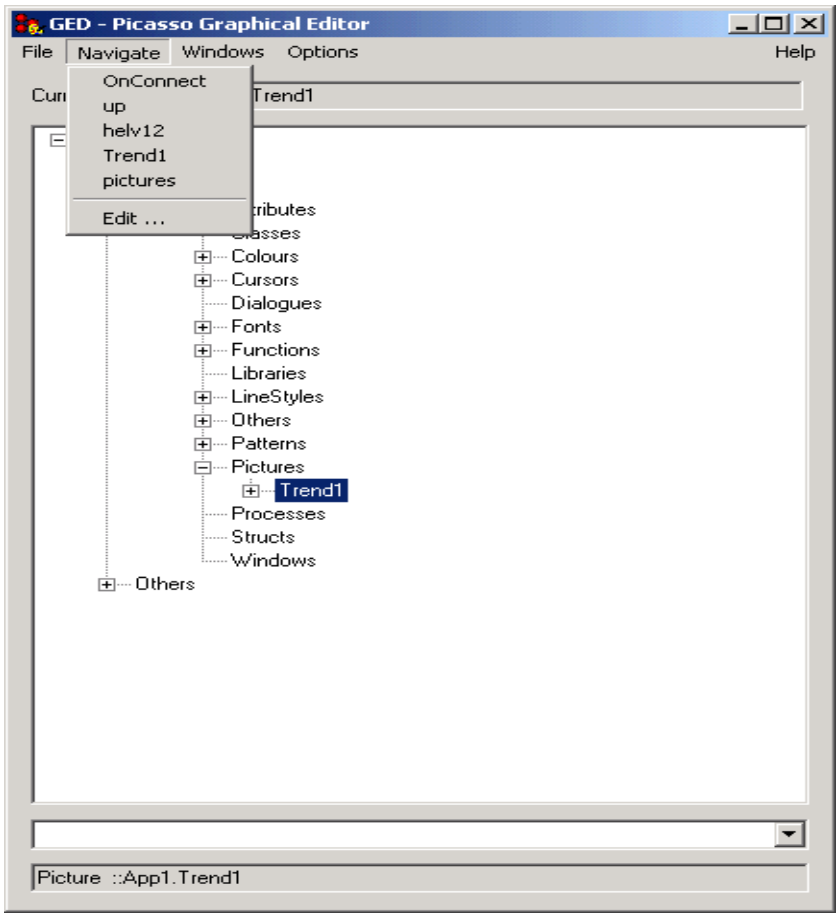
In the Windows version a navigation setup can be specified, this will make the navigation in the GED main window symbol-table tree easier

15.1 Using the navigation commands

When selecting the **Navigate** menu in GED's main window (See Figure 139) you get a list of previously added entries, and a choice for editing this menu.

Figure 139

Edit on the Navigation Commands.

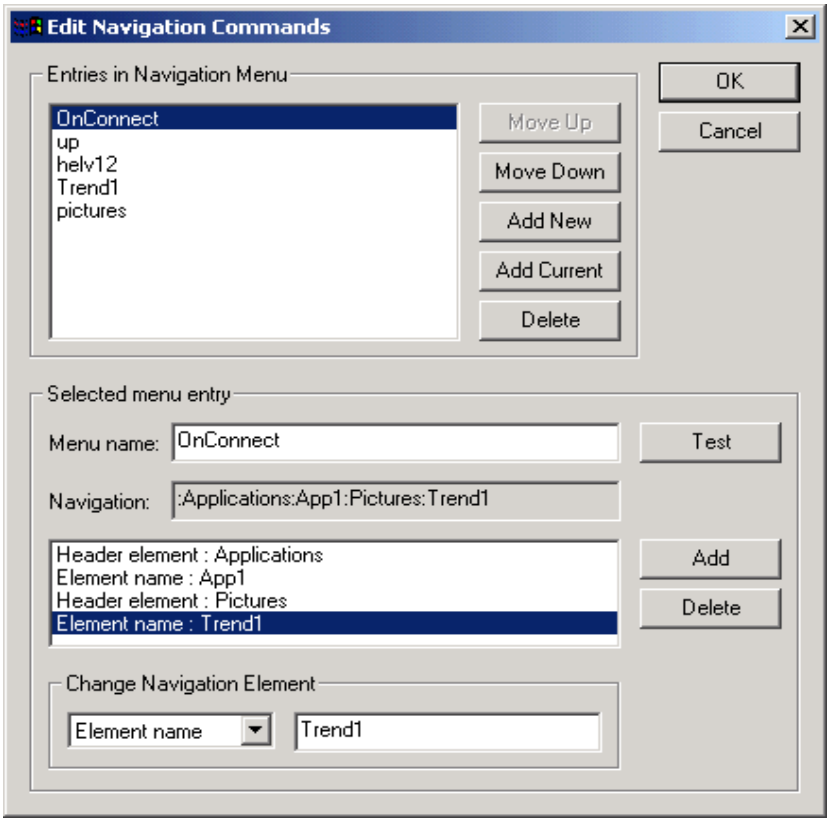


15.2 Edit the navigation commands

If you click on **Edit...** a **Edit Navigation Commands** window appear (Figure 140).The fastest and most convenient way to define a navigation entry to the **Trend1** picture in the above example, is to mark the **Trend1** picture in GED's main window before pressing **Edit...** in the **Navigate** meny and then press the **Add Current** button. A **Menu name** called Trend1 will be made and will be shown

Figure 140

The Edit Navigation Commands window.



from the **Navigate** menu in GED's main window after pressing the **OK** button. If you want this Trend1 picture to always be the one the editor is marking after the rtm is connected to the ged, change the **Menu name** to **OnConnect**. The menu can contain maximum 16 navigation entries, the oldest one will disappear if a new one is made.

The Navigate algorithm starts on the top of the tree, going down one by one level, as long as the element specified can be found.

The navigation elements of an entry can also be added manualy by pressing the **Add** button combined with the two **Change Navigation Element** input fields.

Historic Trends 16

This chapter gives an introduction to the historic trend concept in ProcSee. The historic trends consist of a trend logger as well as graphic trend diagrams that can be included in the user interface as any other graphic shape.

16.1 Introduction

The historic trend in ProcSee consists of a trend logger and a number of trend diagrams. The trend logger can log data for variables at regular intervals or events at random time stamps. Variables logged at regular intervals are presented as curves in historic trend diagrams. Events on the other hand can be visualized using any kind of predefined ProcSee shape or user defined class. The events are visualized in the same historic trend diagrams as the curves.

A trend diagram is a graphic presentation of the historic trend data and the events in the trend logger. Several variables and event types can be visualized simultaneously in a trend diagram. Plotting of variables and events are taken care of automatically once a diagram has been connected to a trend logger.

The examples in this chapter are described in the *ProcSee Configuration Language*. The attributes referred to in the examples can also be found in the attribute list available in the *Graphics Editor* (ged). For a description on how to create and modify trend diagrams interactively, refer to chapter 13 *"The Drawing Editor"* on page 179.

16.2 The Trend Logger

The trend logger in ProcSee is responsible for trending the historic trend variables to memory or disk at regular intervals and events at random time stamps. Depending on the usage, the trend logger can be configured in the following manner

- to be contained internally in the Run-Time Manager (RTM) taking care of displaying graphics etc. This is reffered to as an internal trend log.
- RTM to be set up as an external, dedicated trend log RTM taking care of logging historic data only. This is reffered to as an external trend log.

The graphics editor GED does not provide editing of trend loggers in the RTM. This has to be done manually in a text editor. Follow these steps when adding a trend logger. First, save the application from GED and create a Tdoc file (Using document from the file menu). Next, load the application resource file (.Tdoc) into a text editor. Then, add either an internal or external trend logger entry in the application. The entry *trendLogInternal* is used to set up an internal trend logger, while the entry *trendLogExternal* is used to identify a dedicated trend logger RTM in a display RTM. For more information about these entries, see the reference manual. The final steps, after saving the Tdoc file, are compiling the document file into a binary file (pctx) using the ProcSee compiler *pcc* and reloading the application resource file (pctx) into the RTM.

The internal trend logger is responsible for logging the variables to disk or memory. The historic trend variables are specified in a separate configuration file that the internal trend logger reads when the RTM program loads the application resource file. Several attributes must be specified for each trend variable in the configuration file. When visualizing trend variables in trend diagrams the name and log frequency are used to uniqely identify a historic trend variable in the trend logger. The internal trend logger is also responsible for logging events to file. The event types are specified in a separate configuration file. They can be specified in the same configuration file as the historic trend variables or in a separate event type configuration file.

To get connected to an external trend logger RTM, an external trend logger entry in the application context of the display RTM must be specified. The display RTM is a client to the external trend logger RTM. The name of the dedicated trend logger RTM is one of the attributes that must be specified in the trend log entry in the application context. This is used by the display RTM when connecting to the external trend logger.

The trend logging system in ProcSee is flexible. An RTM can be connected to one internal and several external trend loggers. There is no limit to the number of external trend loggers an RTM can be con-

nected to. In large scale systems it is recommended that the internal trend logger is used to trend either temporary trend variables or predicive trend variables. How to create predictive trend variables are described later in this chapter.

An external trend logger can run on a remote workstation. In projects with a lot of trend variables it is recommended to use one or several external trend loggers as this will remove load from the display RTMs. This will result in improved overall performance.

The trend logger can be configured to log variables for a number of frequencies, and hence provides a flexible way of viewing historical data with different resolution. Historic trend variables and events can be created during run-time. This is a feature that can be utilized when creating temorary trend variables. Temporary trend variables are not logged to disk. Trend variables of this kind are intended to be used for a limited time period and will be removed when the RTM terminates. The internal trend logger is normally used for logging temporary trend variables.

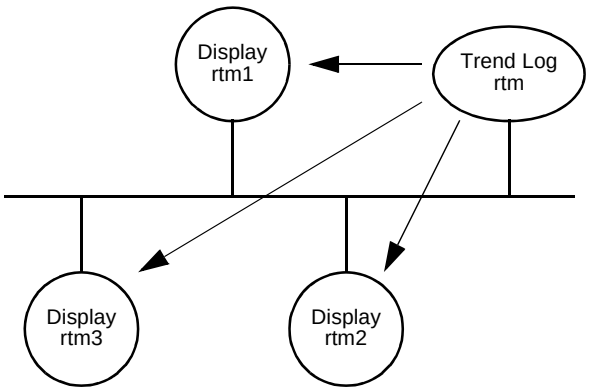
The data logged in the trend logger is normally raw application data that has not been processed by ProcSee before it is stored in the trend logger. In addition to storage of raw application data, the ProcSee trend system provides the following functionality

- storage of pre-processed data in the trend logger, i.e. If the application variable is measured in cubic inches, it can be converted locally in ProcSee to cubic cm before it is stored in the trend logger.
- process already stored trend data before visualizing the data in a trend diagram, i.e. the sum of two already stored variables can be visualized as one curve in a trend diagram.

The trend logger is designed to work in a distributed environment, and can support several display RTMs. Trend data can be logged

Figure 141

The ProcSee trend logger is designed to operate in a distributed environment, and can run as a dedicated trend log rtm, serving several display RTMs. The RTMs can run on different workstations in the network.



with a frequency of up to once per second. The trend variables can be specified to be logged either in memory or to disk. Since the log-

ging to disk is notably slower than logging in memory, it is recommended that the interval for logging to disk is not shorter than every 60 seconds.

Depending on the number of variables to trend and the complexity of the graphics to display, the trend logger should be configured to be internal or external.

Internal Trend Logger

In smaller applications and during prototyping phases, it is useful to utilize an internal trend logger. In large scale projects internal trend loggers are normally used to log predicitve and temporary trend variables.

An internal trend logger is part of a regular RTM. In this case the RTM takes care of both displaying graphics and logging historic data. To define an internal trend logger in your system, make sure the following is defined in your application resource file

EXAMPLE

```
application myApp
{
...
   trendLog internalLogger
   {
      diskLogLimIntvl = 60;
      timeTickIntvl = 5;
      timeMaster = 2;
      timeVariable = "trendTime";
      trendVariableFile = "./trendVars.ptrv";
      eventPlotFile = "./trendVars.pevl";
      directory = "../LOGDIR";
   }
```

The example above specifies that a trend logger named *internalLogger* should be created in the RTM. This trend logger has been configured to log variables in intervals of 5 seconds, and is triggered by an external task (timeMaster = 2). The variables and events to be trended in *internalLogger* are described in the files *trendVars.ptrv* and *trendVars.pevl*.

The binary log files will be put in the directory *LOGDIR*. If this directory does not exist the RTM will automatically create it. The name of the binary files are *<rtmName>.log* and *<rtmName>.dir* for the historic trend log variables. For each data type used in the event configuration file the RTM will generate new binary log files. If an event type is of type *int32* the binary log file will get the name *<rtmName>_int32.elog*.

For a detailed description of the trendLog definition refer to the *ProcSee Reference Manual*.

External Trend Logger

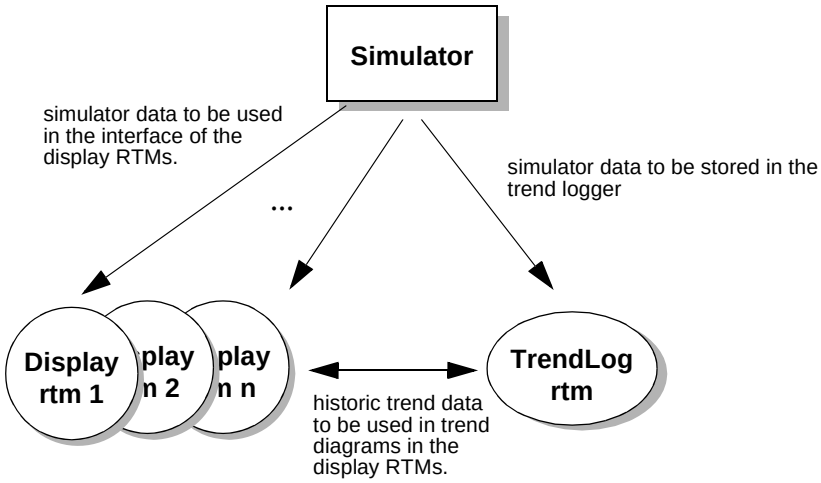
In larger scale system with a large number of variables and events to be trended, it is advantageous to use an external trend logger. An external trend logger is an RTM that is dedicated to trend logging only. This ensures that:

- a large number of variables can be trended with good performance.
- a large number of event types can be trended with good performance
- if several display RTMs are connected to an external trend logger, the trended data is synchronized for all RTMs
- as long as the trend log RTM is running, display RTMs can be started and stopped without affecting the logged data.

A typical configuration with an external trend logger would be one application program, several display RTMs, and one dedicated trend log RTM, as illustrated in Figure 142.

Figure 142

The figure illustrates a typical configuration with a dedicated trend log rtm, receiving data from an application program (in this case a Simulator). The trend log rtm communicates directly with the display rtms, and transfer trend log data as needed.



In order to realize the configuration in Figure 142, the following has to be taken care of

- make sure that the involved parts all refer to the same ProcSee application name.
- the dedicated trend log RTM has to be given a name, that should be used in the *trendLogExternal* entry in the resource file of the display RTMs.
- the name of the *trendLogInternal* and *trendLogExternal* entries in the resource files for the display RTMs and the trend log RTM should be the same, respectively.

The example below shows a typical example of a connection sequence to ProcSee, in this case to a trend logger RTM called *extLog*. It will also be initialized to be a server to the display RTMs.

EXAMPLE - Simulator-RTM connection

```
// The following should be included in the Simulator program code

// connects to the trend log rtm, extLog is the name of the
// trend log rtm, see trend log rtm example
PfInitialize("myApp", "Simulator", "extLog",...);

// The display RTMs will connect to the Simlator program.
PfInit("myApp", "Simulator",...);
```

The first ProcSee API call *PfInitialize()* establishes the connection to the trend log RTM *extLog*. Note that the trend log RTM is started with the command line option *-n extLog* (example below).

It is usual to let the application be a server for the RTMs if the number of display RTMs to be connected to the application program is not known in advance. The use of the API function call *PfInit()* will provide this. The example above uses this functionality. It is strongly recommended to use *PfInit()* in application programs because it simplifies and improves the flexibility in system configuration for large scale systems.

One question remains. How do we connect the display RTM to the application program? The pTALK function *connectToProcess()* is just what we are looking for. It establishes a connection to an application program. This function is normally put in the *constructor()* function in the RTM application resource file. The parameter to this function is the name of the process to connect to.

In both *PfInitialize()* and *PfInit()* the application name *myApp* is used. Note that this corresponds to the application name used in the resource files for the trend log and display RTMs in the examples

below. The resource file of the display RTM should contain a *trendLogExternal* entry and a *constructor()* that connects to the application program, as illustrated below.

EXAMPLE - excerpt of display RTM resource file Drtm.Tdoc

```
application myApp
{
...
   trendLogExternal externalLogger
   {
      extLogRTM = "extLog";
      extLogApp = "myApp";
   }
...
   function `constructor()
   {
      // Connect to process Simulator
      connectToProcess( "Simulator" );
   ...
   }`
}
```

In this example it is assumed that the display RTMs are started as:

```
> rtm -r Drtm.pctx -n <anyName>
```

, where *anyName* is a unique name.

The resource file of the trend logger RTM should contain a *trendLogInternal* entry as illustrated below.

EXAMPLE - excerpt of trend log RTM resource file TrLogrtm.Tdoc

```
application myApp
{
...
   trendLogInternal externalLogger
   {
      diskLogLimIntvl = 60;
      timeTickIntvl = 5;
      timeMaster = 2;
      timeVariable = "trendTime";
      trendVariableFile = "trendVars.ptrv";
      eventPlotFile = "eventVars.pevl";
      directory = "../LOGDIR";
   }
...
```

The trend log RTM must be started with the command as:

```
> rtm -r TrLogrtm.pctx -n extLog
```

Note that the application name *myApp* is used in all 3 examples. This is essential in order to be able to transfer data from the simulator to the RTMs.

In the above example the *trendLogInternal* attribute *timeMaster* is set to 2. This means that time in the trend log is governed by the application program (here the Simulator), and that the time has to be supplied in an application variable. This is specified in the *timeVariable* attribute, which is set to *trendTime*. The variable *trendTime* has to be included in the communication between the Simulator and the RTMs.

The trend log is ticked (the time is advanced, and historic data is logged) whenever the API function *PfFlush()* is called. For a detailed description, see the *ProcSee Reference Manual*.

The *timeTickIntvl* attribute has been set to 5 seconds in the examples. This means that the trend logger expects to be ticked every fifth second. The trend variables in the trend variable file should be specified with a log frequency equal to a multiple of 5.

16.3 Trend Configuration Files

In order to be able to log historical data for a process variable, the variable has to be specified in a trend logger's trend variable file. The trend variable entry in the configuration file contains a set of properties that must be set. The event types must also be defined in a configuration file. They can be defined in the same configuration file as the trend variables.

The configuration file must be created using a text editor. There is no support in the graphics editor GED to interactively create trend configuration files. It must be a pure text file with extension Tdoc. The ProcSee compiler *pcc* is used to compile the source file into binary configuration files. All the configuration data for the trend variables will be put in the binary file with extension ptrv and the event types in a binary file with extension pevl.

These file names is used by the internal trend logger when specifying the trend and the event configuration files. The *trendLogInternal* attributes *trendVariableFile* and *eventPlotFile* are used.

Trend Variable Configuration File

A trend variable entry contains the name of the variable, the log frequency, how much history should be stored, how the variable should be logged, and what the *max* and *min* values should be. Below follows an example trend variable file.

EXAMPLE - trend variable file

```
trendVarConfig trendVars
{
   trendVar trVar1{ lCycle=5; hist=600; type=5; lo=-999999; hi=999999; }
   trendVar trVar1{ lCycle=30; hist=3600; type=5; lo=-999999; hi=999999; }
   trendVar trVar2{ lCycle=5; hist=600; type=5; lo=-999999; hi=999999; }
   trendVar trVar2{ lCycle=30; hist=3600; type=5; lo=-999999; hi=999999; }
   trendVar trVar3{ lCycle=5; hist=600; type=5; lo=-999999; hi=999999; }
   trendVar trVar3{ lCycle=30; hist=3600; type=5; lo=-999999; hi=999999; }
}
```

In the example three different process variables are defined. Each of them are defined to be logged with two different frequencies, 5 and 30 seconds. The 5 second variables are specified to have a history of 10 minutes, whereas the 30 second variables are specified to have a history of one hour.

The trend name and frequency uniqely identifies a trend variable. This information is used by the trend presentation shape when requesting trend data. The trend presentation shape has two attributes that must be equal to an entry in the trend variable configuration file. These are *variable* and *logFrequence*. The value -1 for the attribute *logFrequence* has a special meaning in ProcSee. Setting this attribute to -1 will get the trend data with the best resolution based on the time span and requested time.

For a detailed description of the trend variable file, refer to the *ProcSee Reference Manaual*.

Event Configuration File

Event types must be specified in a configuration file just like trend variables. An event type entry contains the name of the event type, the data type and the history. The history is specified in seconds. Below follows an example of an event configuration file.

EXAMPLE - event configuration file

```
trendVarConfig eventVars
{
   eventVar event1{ dataType="int32"; hist=3600; }
   eventVar event2{ dataType="int32[10]"; hist=3600; }
   eventVar event3{ dataType="Alarm"; hist=3600; }
   eventVar event4{ dataType="float"; hist=3600; }
}
```

In the example above four event types are defined. Each of them with a history of one hour. The data types used in this example are int32, array of type int32 of size 10, struct of type Alarm and single float.

In contrast to historic trend variables, which can only log float values, an event type can log any data type. This includes data types of structures as well as primitive data types like int, float etc. It is also legal to log data of array types. The only restriction is pointer. Pointers are not allowed, neither in primitive data types nor in fields in structures.

16.4 Trend Diagram

A trend diagram is used in a display RTM to visualize the trend log data. The trend logger can either be an internal or an external logger. When connecting a trend diagram to a logger no knowledge is needed about the logger type because there are no differerences in the usage between internal or external logger. If running on computers with high load some minor delays can be apparent when requesting trend data when using an external trend logger.

A trend diagram can only get trend data from a single trend logger, even if the RTM is connected to several external trend loggers. A trend diagram is composed of several shapes. The Trend shape is the main shape and it provides the dimension of the visual trend area, except for the time labels. It is also the parent shape for other trend sub shapes. The following list is the trend shapes that the trend diagram can consist of:

time label text labels denoting the time of the trend diagram.

trend grid horizontal and / or vertical lines to easier identify time and value of a curve.

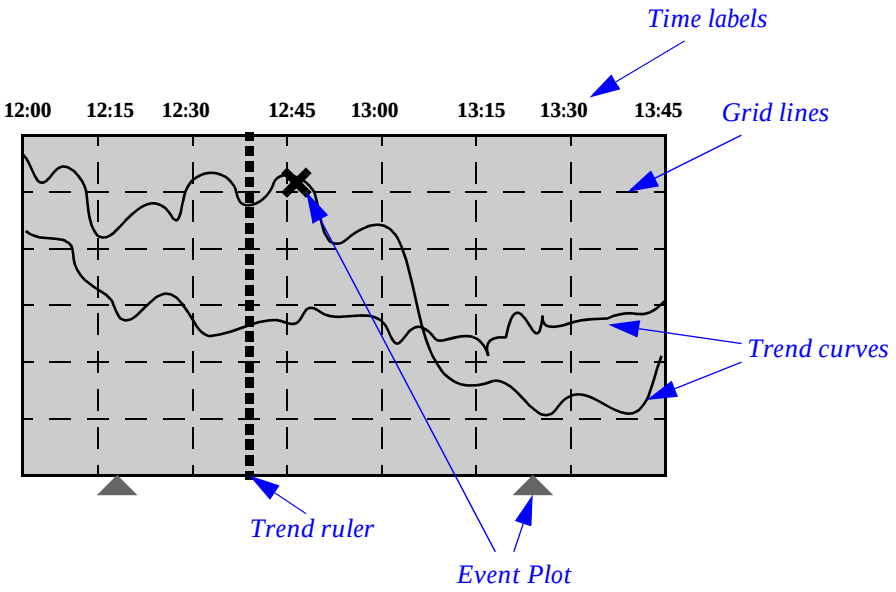
trend presentation a graphic representation of the historical data in question.

Event plots a graphic representation of the event data in question.

trend ruler a ruler that can be displayed in the trend diagram, and give exact read out of time and value for all curves currently displayed in the diagram.

A typical trend diagram is composed of one trend shape, one trend grid, one trend label, one trend ruler, several trend curves and several event plots. See Figure 143.

Figure 143
This figure illustrates a typical trend diagram with two trend curves, two types of event plots and a trend ruler.



All these shapes will be described later in this chapter, but first let us start looking at the trend shape.

Trend Shape

As described in previous section the trend shape is the main trend shape that provides the dimension for the trend area. It also serves as a container for other trend shapes. The trend shape alone does not contain functionality for displaying for instance trend curves. The trend presentation shape is responsible for this kind of functionality. Different kind of trend features are delegated to specific trend shapes. The trend sub shapes, listed in the previous section, are tightly connected to the trend shape. These shapes use information stored in the trend shape. Information like the trend logger, the size of the trend area, the time span etc. are used by the trend sub shapes.

The trend shape is responsible for connecting to the trend logger, specified with the attribute *trLog*, and for collecting trend variables and events types to be visualized in the diagram. It receives trend updates from the connected trend logger and delegates the data to the different trend shapes. The main purpose with the trend shape is to serve as the connection point between the trend logger and the trend sub shapes. Once a trend shape has been connected to a logger, the trend data is automatically transferred to the diagram on the requested intervals or when events are added to the trend logger.

It is usual to add dialogues on the trend shape. To move the ruler the dialogue trigger *cursorMoved()* is normally used. Also the dialgoue triggers *shapeEnter()* and *shapeLeft()* are used to change the trend ruler's visibility attribute when the mouse pointer is moved inside and outside of the trend diagram, respectively.

The trend logger to connect to is controlled with the attribute *trLog*. It must be a name of one of the trend loggers defined in the application. It does not matter if it is an external or an internal trend logger. All trend data received must come from a single trend logger. It is not possible to get trend data from different trend logger sources. Note that all trend variables and event types referenced in the trend presentation and event plot shapes must be defined in the same trend logger.

The attribute *timeSpan* controls the time window. It is specified in seconds. The trend diagram will display trend data for this time span.

The trend shape also controls the update mode. To prevent flickering on the screen when receiving updates the attribute *updateMode* can be used. It can be set to a mode that gives the optimal performance and/or flicker free updates. It is recommended to use one of the buffering update modes because these modes provide flicker free updates of the trend diagram.

In the following examples it is assumed that the trend diagram is connected to the internal trend logger *internalLogger*. It is not a complete example but illustrates the attributes used by the trend shape. It also demonstrates the use of the dialogue trigger *cursorMoved()* to move the trend ruler. In the example note that the *timeSpan = 240*. All variables that are later referenced in the trend presentations and

event plots should be included in the trend configuration files for this logger. A typical example of a trend entry in a picture Tdoc file could look like:

```
trend T1
{
   trLog = "internalLogger";
   x = 30;
   y = 100;
   width = 700;
   height = 350;
   visibility = 1;
   timespan = 240;
   orientation = 0;
   scroll = 0;
   foregroundFillColour = 11;
   backgroundFillColour = -1;
   foregroundColour = 21;
   backgroundColour = -1;
   lineWidth = 4;
   linePattern = 1;
   fillPattern = 1;
   updateMode = 6;

   trendPlot P1
   {
      // details in example below
   }

   trendLabels T1
   {
      // details in example below
   }

   trendRuler R1
   {
      // details in example below
   }

   trendGrid G1
   {
      // details in example below
   }

   eventPlot E1
   {
      // details in example below
   }

   dialogue
   {
      event = `cursorMoved() `;
      action = statement `{
                              R1.moveRuler();
                              theWindow().update();
                           }`;
   }
}
```

Trend Presentation

The historical data are presented in trend presentations. The names trend plots and trend curves are also used in the ProcSee documentation. Typically, a trend plot is connected to one trend variable that will provide the data for drawing of the curve between an upper and lower limit, providing the amplitude of the trend curve. However, it is possible to connect a trend curve to several trend variables, each of which can control different attributes of the trend curve. This feature is described in section section 16.5 *"Advanced Features"* on page 258, which treats the *tr()* function. For a detailed description, refer to the *ProcSee Reference Manual*.

A trend curve contains a number of attributes, of which the most important are:

variable contains the name of or a pointer to the trend variable that will be controlling the amplitude of the trend plot.

presentation determines how the trend plot should be presented graphically. There are nine different presentation forms available in ProcSee.

presentationFunc the type of presentation. Three different presentation forms are available in the RTM. Normal, natural logarithm and logarithm with base 10.

fillOffset it is used together with the attribute *presentation*. It controls the area to fill between the amplitude value and the value of this attribute. It can also be set dynamically equal to another trend plot in the same trend diagram. The area between the two curves will then be filled.

upperLimit the highest value the curve will be displayed for.

lowerLimit the lowest value the curve will be displayed for.

logFrequency the log frequency that is wanted for the trend variable(s) in this trend plot. If a frequency other than -1 is specified, there should exist an entry with a similar frequency for the variable(s) in the trend variable file.
If a log frequency of -1 is specified, the data set with the best resolution for this / these particular variable(s) is returned. The frequency should not be faster than the **timeTickIntvl** of the trend logger.

autoScale if the value is set to 0, autoscaling is off. Setting this variable to 1 will scale the curve automatically ignoring the attributes **upperLimit** and **lowerLimit**. If several curves are displayed in the same trend diagram setting the **autoScale** attribute to 2 will scale all the curves to ensure consistency between the trend curves.

upperBand setting this value will display a band between the edge of the trend diagram and the highest value of the curve. Only used if autoscale is on.

lowerBand setting this value will display a band between the edge of the trend diagram and the lowest value of the curve. Only used if autoscale is on.

Below follows an example of a trendPlot entry. This entry presents

```
trendPlot P1
{
   variable = "trVar1";
   presentation = 0;
   presentationFunc = 0;
   upperLimit = 100;
   lowerLimit = 0;
   visibility = 1;
   fillOffset = 0;
   logFrequency = -1;
   foregroundFillColour = 55;
   backgroundFillColour = -1;
   foregroundColour = 55;
   backgroundColour = 53;
   lineWidth = 0;
   linePattern = 1;
   fillPattern = 1;
   autoScale = 1;
   upperBand = 10;
   lowerBand = 10;
   xReference = 0;
   yReference = 0;
   xScaleFactor = 1;
   yScaleFactor = 1;
   rotationAngle = 0;
}
```

historical data for the variable *trVar1*. Since *trendPlot P1* is part of the *Trend T1*, the *trVar1* must be contained in the trend variable file specified for the *internalLogger* (see the trend variable file described above).

In the example *P1* is specified to use presentation = 0, that is a line. Furthermore, the data received for *trVar1* is assumed to be in the interval 0-100, given by *lowerLimit* and *upperLimit*. If a value outside the limit is received, it is presented as the limit value.

The log frequency is set to -1, specifying that the data set with the best resolution should be returned. If the log frequency had been set to 10, it would mean that an entry for *trVar1* with log frequency equal to 10 should exist in the trend variable configuration file.

Trend Ruler

In order to be able to display the exact time and value for a point on the trend curve, a trend ruler can be added to a trend diagram. Normally, it is desirable that the ruler moves as the mouse pointer is moved across the trend diagram. This is achieved by adding a dialogue to the trend diagram, as shown in the trend diagram example above.

The trend ruler contains one important attribute

rulerIsSelected this attribute is used to enable / disable the trend ruler. The default value is 1, enabled. A value of 0 indicates that the ruler is disabled.

The time and value for a given position of the ruler is available in each trend presentation. In the example the ruler time is found by executing the function *P1.getRulerTime("R1")*. The ruler value is found by executing *P1.getRulerValue("R1")*.

```
trendRuler R1
{
   visibility = 0;
   foregroundColour = 32;
   backgroundColour = -1;
   lineWidth = 0;
   linePattern = 1;
   rulerIsSelected = 1;
   xReference = 0;
   yReference = 0;
   xScaleFactor = 1;
   yScaleFactor = 1;
   rotationAngle = 0;
}
```

In order to display the time and value as the ruler is moved across the diagram, two text shapes can be used. The text shapes are not part of the trend diagram, but could be included in a picture or class. The example below assumes that the trend *T1* is part of the

```
picture MyTrends
{
   char tBuf[30];
   ...

   text Time
   {
      ...
      format = "%s";
      theText = `cftime(tBuf, "%H:%M:%S", T1.P1.getRulerTime("R1") ), tBuf`;
      ...
   }
   text Value
   {
      ...
      format = "%6.2f";
      theText = `T1.P1.getRulerValue("R1")`;
      ...
   }
   ...
}
```

picture *MyTrends.*

Trend Grid

To make it easier to identify time and value for trend plots in a diagram, trend grids can be added to the diagram. A trend grid is composed of vertical and horizontal lines.

The trend grid can be connected to a trend presentation, meaning that it will follow the maximum / minimum amplitude values of the curve. The attributes *autoStep* and *theTrendPres* must be set for this functionallity to be enabled. The major horizontal grid lines will be created in step of the value of the attribute *autoStep.* The attribute *noOfLines* specifies the number of horizontal grid lines between the major grid lines. The type of grid created depends on the *presentationFunc* set in the connected trend presentation. If the presentation function is either natural logarithm or logarithm with base 10 a logarithmic grid will be drawn.

Important attributes for the trend grid are

depthPosition this attribute specifies where the grid should be placed. The following values are available
0 - behind the trend plots
1 - on top of the trend plots, but behind the trend ruler
2 - on top of the diagram

noOfLines this attribute specifies how many horizontal lines are to be displayed. If auto step is enabled it determines the number of horizontal grid lines between the auto step. Otherwise, it determines the number of horizontal grid lines in the diagram. The lines are evenly distributed.

autoStep determines the value between major horizontal grid lines. Only available when the attribute *theTrendPres* is set to point to a trend presentation.

theTrendPres must be set to a trend presentation in the same trend diagram if auto step should be enabled.

timeInterval this attribute specifies the number of seconds between the vertical grid lines. A value of 0 indicates no vertical grid lines.

timeOffset the attribute can be used to start drawing vertical grid lines an offset into the diagram. If a timespan of 300 seconds is used for the trend, and the offset is set to 240, grid lines will be drawn in the area 240 - 300 seconds.

Below follows an example of a typical trend grid.

```
trendGrid G1
{
   depthPosition = 0;
   noOfLines = 5;
   autoStep = 0;
   theTrendPres = 0;
   timeInterval = 60;
   timeOffset = 0;
   visibility = 1;
   lineWidth = 0;
   linePattern = 1;
   foregroundColour = 14;
   backgroundColour = -1;
   xReference = 0;
   yReference = 0;
   xScaleFactor = 1;
   yScaleFactor = 1;
   rotationAngle = 0;
}
```

Time Labels

To indicate the time for a trend diagram, trend labels can be added. The time labels are automatically updated as new data is received from the trend logger, and is always guaranteed to be synchronized with the trend data.

The time labels can be placed above or below the diagram, and the format and interval between time labels can be configured through attributes. Important attributes of time labels are:

offset the vertical distance from the edge of the trend diagram. A negative value indicates that the time labels should be placed above the diagram.

alignment the attribute specifies how the text labels should be aligned horizontally. The following values are available: 0 - left justified, 1 - centered, and 2 - right justified.

interval number of seconds between each label.

theFont a font id or name. This font must be defined in the main resource file.

format the time format of the labels. The supported formats are similar to those of the strftime(3C) or cftime(pTALK) functions.

Below follows a typical example of a time label. In this example the

```
trendLabels T1
{
   offset = -20;
   alignment = 0;
   interval = 60;
   theFont = 0;
   format = "%H:%M:%S";
   visibility = 1;
   foregroundColour = 1;
   backgroundColour = -1;
   xReference = 0;
   yReference = 0;
   xScaleFactor = 1;
   yScaleFactor = 1;
   rotationAngle = 0;
}
```

time labels are placed above the trend diagram, indicated by the attribute *offset = -20*. The individual text labels are left justified, and displayed every 60 seconds. Since the trend *T1* (example above) is specified with a timespan of 240 seconds, there will be 5 time labels. Using the specified format, the time will be displayed as *"12:41:15"*, etc.

Event Plot

The event plot shape is used for visualization of event data logged in the trend logger. The event plot shape is connected to an event type in the trend logger. The name specified with the attribute *event*

must be equal to a name in the event configuration file. The event configuration file is described in chapter 16 *"Historic Trends"* on page 244.

The event plot shape itselft does not have a visible appearance. The event plot shape is just a container for other shapes. These shapes will automatically be moved when update messages are received from the trend logger. All events contain user defined data and are marked with a time stamp. This time stamp is used by the event plot to move the shape to the correct position within the trend diagram. The time stamp only gives the position in the x-direction (if we assumes that the trend is not rotated or is circular). The shapes will be removed by the RTM when the shape position, in the x-direction, is moved outside the trend diagram.

The question is now, what kind of shape can be displayed in an event plot? The answer to this question is any shape that can be created as a standalone shape in a picture. The attribute *shape* must be set to either a pTALK class or a pTALK function. The class must be defined in the application or in one of the loaded libraries. The pTALK function must return a pointer to a shape created with one of the predefined pTALK functions *new...()*, that is *newInstance()*, *newText()*, etc.

If the *shape* attribute is set equal to a pTALK class all the shapes in the event plot will be instances of this class. If this attribute is changed during run-time all the instances will be replaced with instances of the new class. The data and the time stamp stored with each logged event will not be available when instances are created using this method.

Setting the *shape* attribute equal to a pTALK function extends the flexibility of the event plot shape. The pTALK function will be called each time the event plot receives a new event from the trend logger. The pTALK function must have the following prototype:

```
Shape* function( char* ownerFullName,
                 int   timeStamp,
                 void* data );
```

, where *ownerFullName* is the fullname of the event plot shape. This argument must be the first argument in all the *new...()* functions, because the owner to the event plot shapes are the event plot itself. The *timeStamp* is the time when the event occured and *data* is the value stored with the event. A void pointer is used in the general function prototype. For instance, if int32 values are stored with an event type, a pointer to an int32 could have been used instead of a void pointer.

How does this extends the flexibility of the event plot shape? The answer is that the user defined pTALK function can create and return a pointer to shapes of any type. The data passed as argument to the function can be used to initialize an instance of a class or it can

be used to determine what kind of shape to create. It is not neccessary to set the position or visibility to shapes created in this function because they are automatically moved and made visible by the event plot shape.

An example of a user defined event function is given below.

```
Shape* evCreateAlarm( char* ownerFullName, int timeStamp, Alarm* alarm )
{
   AlarmClass* ai = newInstance( ownerFullName, "AlarmLib.AlarmClass",
                                 0, 0, 0 );
   ai->Data = alarm;
   ai->Time = timeStamp;

   return ai;
}

...
trend T1
{
   ...
   eventPlot E1
   {
      event = "ev1";
      shape = "evCreateAlarm";
      ...
   };
};
```

In this example instances of the class *AlarmClass* is created. The data used to initialize the instance is of type *Alarm* struct. This is a struct defined in one of the database files. The time stamp is also saved in the instance. The created instance *ai* is returned from the function.

The attributes *offset* and *theTrendPres* are used for positioning the shapes in the vertical or y-direction (if we assumes that the trend is not rotated or is circular). If *theTrendPres* is set to 0 the shapes will be position in the middle of the trend diagram in the vertical direction. Changing the *offset* attribute will move the shapes up or down. Negative values moves the shapes up and positive down. It is also possible to put shapes outside of the trend diagram. If the *theTrendPres* attribute is set to point to a trend presentation in the same trend diagram, then the position of the shapes will follow the amplitude value of the curve.

Important attributes of event plots are:

event the name of the event type. This event name must exist in the event configuration file.

shape the shape to create when events are generated. Either a pTALK class or a pTALK function.

theTrendPres points to a trend presentation if the shapes should follow the amplitude values of a curve.

offset offset added to the shapes in the vertical direction.

In the example the event shapes follow the amplitude curve to the trend presentation *PI* (attribute *theTrendPres*). The event type is *eALARM*. This name must exist in the event configuration file. The shapes are created in the pTALK function *evCreateAlarm*.

```
 eventPlot E1
{
    visibility = 1;
    event = "eALARM";
    theTrendPres = "P1";
    offset = 0;
    shape = "evCreateAlarm";
    xReference = 0;
    yReference = 0;
    xScaleFactor = 1;
    yScaleFactor = 1;
    rotationAngle = 0;
} // end of eventPlot
```

16.5 Advanced Features

In the following some advanced features of the ProcSee Trend System is described.

The *tr* Function

The standard pTALK function *tr()* must be given special attention. Unlike other pTALK functions the *tr()* function can only be used together with trend presentation shapes. If called outside the trend presentation scope, 0 is returned. Thus, it has a meaning only for trend presentation shapes.

Typically, a trend presentation is connected to one trend variable, that determines the shape/amplitude of the trend curve. In some situations it is also desirable to be able to connect a trend presentation to several trend variables, in order to better be able to reflect the state of the historic data.

The *variable* attribute determines the amplitude of the trend curve. The graphic attributes such as *foregroundColour*, *linePattern*, etc., can also be connected to trend variables using a special construct; the *tr()* function.

```
foregroundColour = `tr("trVar1")`;
```

The *tr()* function guarantees that this is not handled as regular dynamics, but that each historic data point is used to determine the foreground colour for each segment of the trend curve.

Note that the trend variables used in one trend presentation should have the same resolution (use the same log frequency).

In the example above the attribute *foregroundColour* was connected directly to the trend variable, this of course, requires that the variable *trVar1* contains a valid colour number. The *tr()* function can be used in any legal pTALK expression, such as

```
foregroundColour = 'alarm( tr("trVar1") )';
```

where *alarm()* is a user defined function that could look like

```
int alarm( float val)
{
   if ( val > 10 )
      return 1;
   else
      return 2;
}
```

In this example the *foregroundColour* of the trend presentation will toggle between 1 and 2 depending on the value of the trend variable *trVar1*.

If for instance the background colour of a curve depends on, lets say, two trend variables the *tr()* function can be used as in the example below. The user defined pTALK function *alarmcol()* accepts two float variables as arguments.

```
backgroundColour = 'alarmcol( tr( "trVar1" ),
                              tr( "trVar2" ))';
```

where *alarmcol()* must look like

```
int alarmcol( float val1, float val2 )
{
   ...
}
```

As described in section 16.2 trend variable data can be pre-processed before they are stored in the trend logger or when they are visualized in a trend diagram. The latter will be described in this section. Lets say that we wish to visualize the difference between two trend variables in the trend diagram. In a real life situation the variables would probably be determined by the end user. How can that be accomplished without trending every possible combination of differences between variables? The answer to the question is of course the use of the *tr()* function.

The *tr()* function is normally used on colour, line or pattern attributes. However, the *tr()* function can be used on the attribute *variable* as well. The resulting data type must be of type float. The following example illustrates the use of the *tr()* function to visualize the difference between the trend variables trVar1 and trVar2 in a trend diagram.

```
variable = `tr( "trVar1" ) - tr( "trVar2" )`;
```

If a user defined pTALK function is used to get the difference between two trend variables, the code may look like this:

```
variable = `diffunc( tr( "trVar1" ),
                     tr( "trVar2" ))`;
```

where *diffunc()* is a user defined function that could look like

```
float diffunc( float val1, float val2 )
{
  return val1 - val2
}
```

Trending Of Attributes

In addition to process variables, it is also possible to trend attributes. The trending of attributes is similar to that of process variables. The attributes must be included in the trend variable configuration file, and they must be specified with their full name.

The attributes can be connected dynamically to any legal pTALK expression, and hence this gives an opportunity to trend any expression of process variables.

In the example, the picture attribute *picTrVar1* is dynamically connected to the pTALK expression **`trVar1 + trVar2/2`**. The process variables *trVar1* and *trVar2* are in this case trended as well (see example in section 16.3 *"Trend Configuration Files"* on page 245).

Note that the *variable* attribute in the trendPlot is assigned to the fullName of the picture attribute, namely **`::myApp.trendPic.picTrVar1`**, similar to the specification of the attribute in the trend variable configuration file.

In the example below excerpts of the document (Tdoc) format is given. The example shows how a picture attribute can be trended and presented in a trend diagram.

EXAMPLE - trending a picture attribute

```
// trend variable configuration file

trendVarConfig trendVars
{
   ...
   trendVar ::myApp.trendPic.picTrVar1 { lCycle=5; hist=600; type=5; lo=-
999999; hi=999999; }
   ...
}

// trend picture

picture trendPic
{
   ...
   float picTrVar1 = 'trVar1 + trVar2/2';
   ...
   trend myTrend
   {
      ...
      trendPlot P2
      {
         ...
         variable = "::myApp.trendPic.picTrVar1";
         ...
      }
      ...
   }
   ...
}
```

Accessing Set of Logged Data

While the trend logger by default is logging data at a regular interval point by point, there is often a need to work on set of historical data. The term predictive data is often used as a notion for accessing set of data in the trend logger. A trend variable is said to be predictive if it is referred to in the trend variable specification file as being of predictive type (*type* = 7) or it is used in a call to the **newLogBuffer()** function. The functions **appendLogBuffer()** and **cancelLogBuffer()** will only work on predictive trend variables.

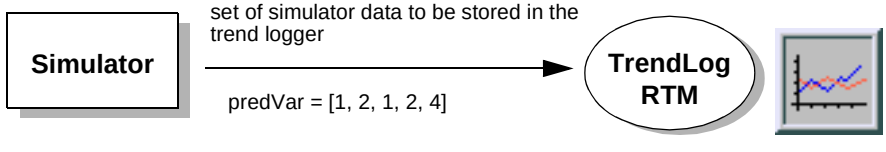


Figure 144
A simulator is adding set of predictive data to the trend logger. These values are compared to actual measured values.

An example can be a simulator that predicts values for process variables ahead of the current time. These values might be compared visually to the actual measured values for the same variable to verify the prediction. When the simulator has calculated more data for

the variable, these data can be appended to the current set. After a while, the whole data set might be deleted. In the following examples it is assumed an application named *myApp* and an internal trend logger named *myLogger*.

EXAMPLE - how to add new, append and cancel set of data to trend logger

```
myLogger.newLogBuffer("::myApp.myApp.predVar", -1, 10, tArr, vArr );
.
.
.
myLogger.appendLogBuffer("::myApp.myApp.predVar", -1, 10, tArr, vArr );
myLogger.appendLogBuffer("::myApp.myApp.predVar", -1, 10, tArr, vArr );
myLogger.appendLogBuffer("::myApp.myApp.predVar", -1, 10, tArr, vArr );
.
.
.
myLogger.cancelLogBuffer("::myApp.myApp.predVar", -1 );
```

In the example above, the trend log member functions **newLogBuffer()** and **appendLogBuffer()** requires 5 parameters:

- full name of trend variable
- the log frequency of this variable
- number of points to be added
- an array for the corresponding time stamps
- an array containing the calculated values

The same trend variable can be logged at different frequencies, and this is the reason why it has to be specified to uniquely define the variable. A ***lCycle*** value set to -1 can be used if the trend variable is logged at one frequency. Only the first two parameters are required for the function **cancelLogBuffer().** See the reference manual (Standard pTALK Functions - TrendLog) for further details

Another useful feature is to extract data from the trend logger and connect them to pTALK attributes or process variables. This can be done using the **getLogBuffer()** function.

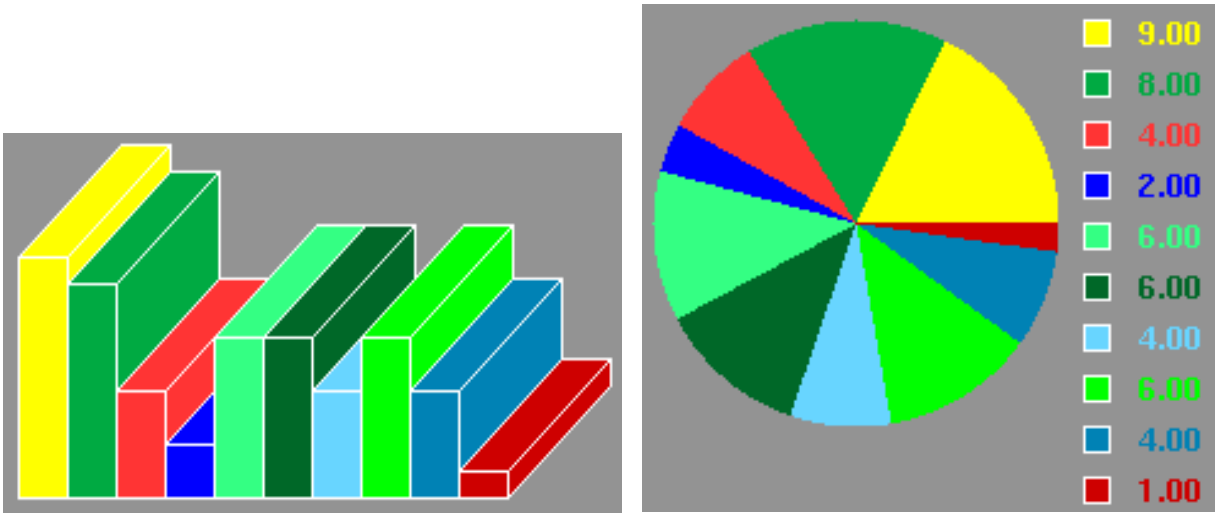


Figure 145 *Picture attributes and process variables can be connected to data that is present in the trend logger. This opens up for user defined presentation of logged data.*

The following parameters are required:

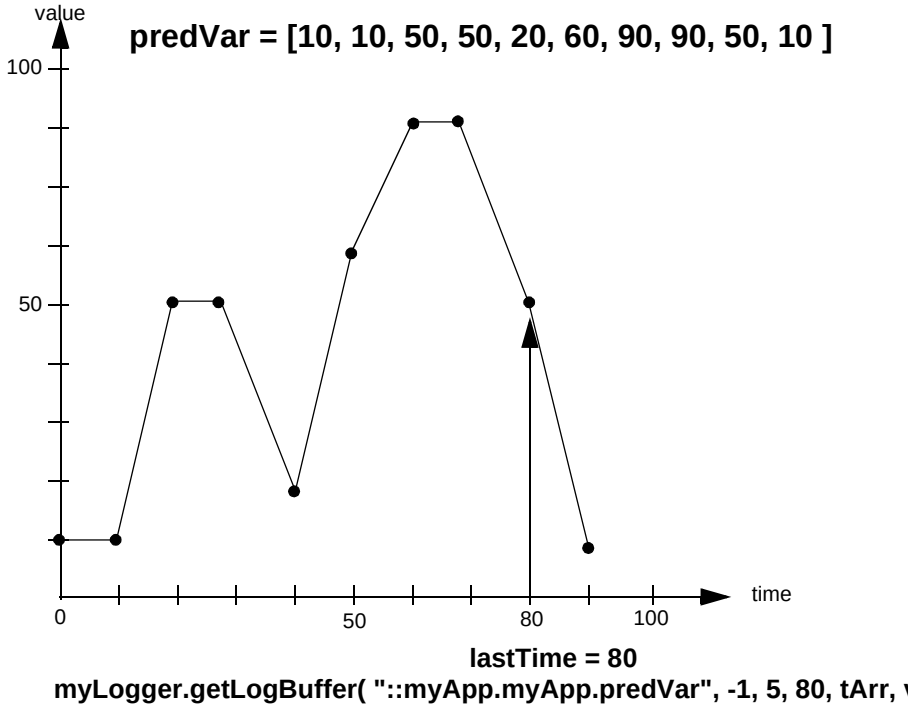
- full name of trend variable
- the log frequency of this variable
- number of points to be retrieved
- an array to store corresponding time stamps
- an array to store logged values

In the example below, 10 values are retrieved from (including) last time = *80* and backwards in time.

The reference manual (Standard pTALK Functions - TrendLog) ex-

Figure 146

The figure illustrates a trend variable with the associated log buffer. There are 10 values in the buffer at present. Log frequency is 10seconds, starting at time 0. The **getLogBuffer()** *function is asking for 5 values with lastTime at 80 to be returned in the two arrays tArr and vArr.*



plains how the **getLogBuffer()** function can be used together with trigger functions to extract data from the trend logger at each time tick in order to implement user defined trend classes. Be aware that this is opens up a wide range of possibilities on how to present historical data, but it can be quite time consuming.

COM 17

This chapter gives an introduction to the COM component support in ProcSee. Third party or user defined COM components can be used to extend the functionality in ProcSee. Both graphical user interface and non-graphical user interface COM components are supported. This technology is only available on the Windows platforms.

17.1 What is COM?

COM is a technology that was introduced by Microsoft in the middle of the 1990s. COM is an acronym for Component Object Model and it is as the name says a component based technology. COM components are generally small objects dedicated for a specific task which can be used in application development.

COM soon became a de-facto standard and this technology has become a powerful tool in application development world wide. It is probably one of the most important tools in commercial application development on the market. Using COM components can greatly reduce the development time and costs in a software project. It also simplifies debugging since many of these COM components are well tested components used by a huge number of developers world wide.

Thousands of COM components from different software vendors are available today which can be used in application development. Both commercial and non-commercial COM components exist on the market.

17.2 COM features

All COM components which are used in an application, are created with function calls supported in the COM library. They are not created using the ordinary memory allocation functions. A component can be running locally or remote. This is specified when the component is created. However, running a component on a remote machine is a feature that is not supported in ProcSee. It can also be created as a standalone process or as an in-process component (dll).

A COM component cannot be modified directly because it does not provide any support for this type of access. The component itself does not contain any functions. The question is then. How can a COM component be modified, because we want to be able to modify it? The answer to this question is through something that is called interfaces. All COM components provide a set of interfaces that can be accessed through function calls. IUnknown is an example of an interface. This is a special interface that must be supported by all components.

The IUnknown interface does not contain any functionality. This is just a base class for all the other interfaces. It is only used to query for other interfaces supported by the COM component. This query function is supported in ProcSee and in pTALK this function is called queryInterface(…). It returns a pointer to the interface that is requested by the application. For instance a COM component for accessing a database may have an interface called IQuery. All query functionality which the database component supports will be through this interface.

How can an application access functionality which a COM component provides through an interface? The answer to this is through functions and properties. An interface may contain both.

A COM function can be called like any other function in ProcSee. It can have both input and output parameters and can return values. A property is called attribute in ProcSee and it is used just like one. Some properties can be used both to set and get values (be an rvalue and lvalue). It means that properties can be put on both sides of the assignment operator. Notice that some properties may be read or write only.

The question is now. Can a COM component send information back to the container, ProcSee? As you might have guessed, of course it can. The next question is then, how? An interface, as described above, is a communication path from the container to the component. COM supports something called event interfaces. It is an interface but used in the opposite direction, that is from the component to the container, ProcSee. An event interface exposes functions just like an ordinary interface. When a component calls a function in a container it is said to be firing an event. What happens when a push-button is being clicked on? An event function will be

called in the container, which is a pTALK function in ProcSee. For instance a push-button will fire an *OnClick(int buttonNo)* event to ProcSee when it is being clicked on.

A COM component can have a graphical user interface. These components can be inserted into pictures in ProcSee just like any other shape. They are called ComShapes in ProcSee (Microsoft call these components for controls or ActiveX controls). An example of a COM component with a graphical user interface is for instance a push-button. Also non-graphical user interface components can be used in ProcSee, these components are called ComObjects. A component accessing a database is an example of a component of this type.

17.3 COM components supported by ProcSee

Most of the COM components on the market today support something called automation. Automation compatible components are components which can be used in scripting languages. Microsoft Visual Basic and Borland Delphi are examples of programming languages that support these types of components.

Only COM components which comply with the automation technology can be used in ProcSee. Automation compatible components expose their unique features to ProcSee through type information. All the type information that the component exposes must be available at run-time when inserting a COM component in ProcSee.

Only interfaces that support automation will be available in ProcSee. That is, interfaces which are derived from IDispatch (Pure Automation or Dual interfaces). Custom interfaces (Derived from IUnknown) provided by a component will not be available through the pTALK language in ProcSee.

17.4 COM types

Some COM concepts must be described before continuing with how to use COM components from the graphics editor GED. This section will briefly describe the COM types used in ProcSee. For a detailed descriptions, see section 17.6 *"COM definitions in the symbol table"* on page 289, section 17.7 *"Using COM in ProcSee"* on page 293 and section 17.8 *"Predefined COM Data Types in ProcSee"* on page 298.

Before a COM component can be used in ProcSee it has to be known to the pTALK language. When a COM component is inserted into ProcSee the entire definition will be created in the symbol-table. The type information provided by the components which are used to create the types in ProcSee, are called COM definitions.

There is a distinction between COM definitions and objects of the COM definitions in ProcSee. These differences will be described next in this section.

The COM definitions used in ProcSee and a short description of each of these types follow. These data types are used as base classes for the COM component definitions. The type information that the components provide are used to create COM definitions. The type *ComClass* is rarely used in a ProcSee application, but is used as a base class for other components, like *FlexGrid* control, *SpreadSheet* control, etc.

- ***ComLibrary*** – container for the other COM definitions. The other COM types will be children of this library.
- ***ComClass*** – definition of a COM component.
- ***ComInterface*** – definition of an interface supported by a component. It contains functions and properties used to access functionality in a component.
- ***ComEnum*** – enumeration definition. It contains integer constants.

To use a COM component in ProcSee an object of a COM definition must be created, i.e. objects of a type derived from ComClass or ComInterface. The following types are used as base classes for the COM objects in ProcSee.

- ***ComObject*** - object of a type derived from ComClass without a user interface.
- ***ComShape*** - object of a type derived from ComClass with a graphical user interface. This is a shape type used in pictures and classes.
- ***ComInterfaceObject*** - object of a type derived from ComInterface used to access functionality in a component.
- ***ComEventObject*** – object of a type derived from ComInterface used to get call-backs from a component.

17.5 Using COM from GED

This section will describe how to use COM components from the graphics editor GED. It will guide you through the steps that are necessary in order to use COM components in ProcSee.

Making COM support available in GED

In the graphics editor GED check that the *ComLibraries* heading is available from the main tree. This is default. If this option is available then COM support is enabled in GED. If not follow the instructions below to enable it, otherwise this section could be skipped.

From the Options menu in the main window in GED select View Settings. The following dialogue window will appear on the screen, see Figure 147. In this dialogue make sure that the *Show COM headings* is checked and press the OK button.

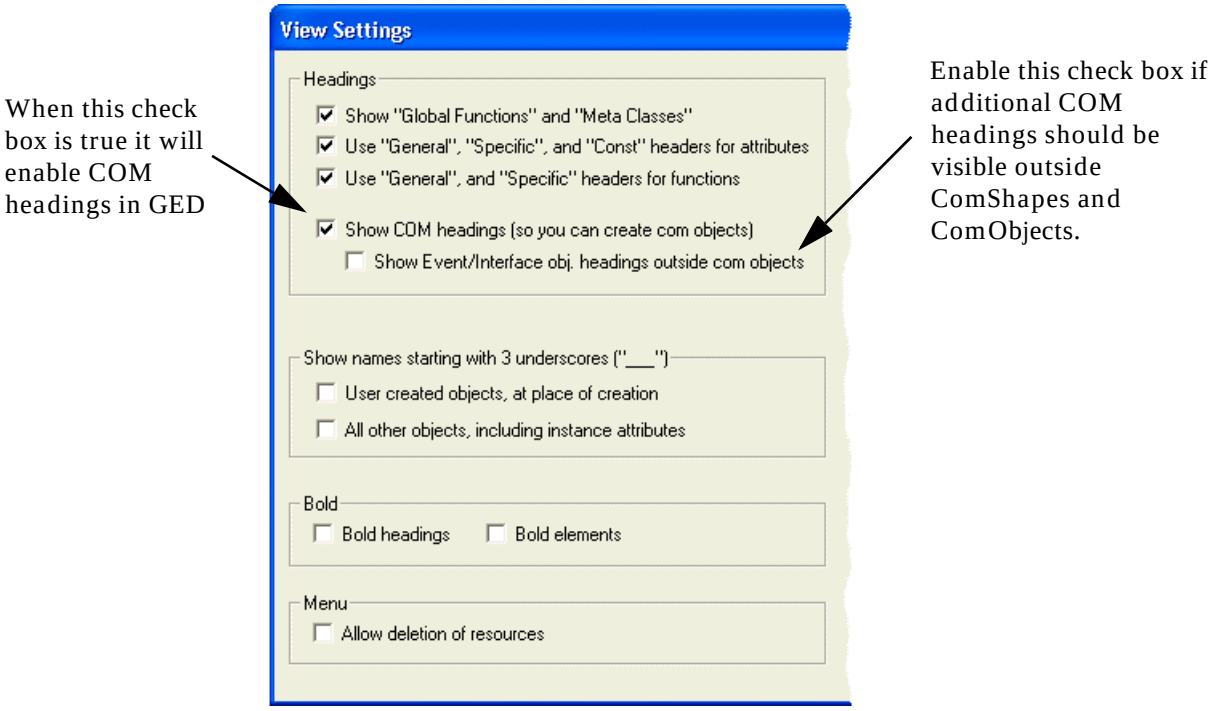


Figure 147 *This picture displays the View Settings window in the graphics editor GED.*

When *Show COM headings* is checked new headings in the tree view in the main window will be accessible from GED. GED is now ready to be used as the main editor for inserting COM definitions and using COM objects and COM shapes in ProcSee.

The option *Show Event/Interface obj. headings outside com objects* is normally not necessary to set. This option must be enabled if you want to create ComInterfaceObject and ComEventObject outside ComShapes and ComObjects. When this option is enabled GED will add ComInterfaceObjects and ComEventObjects headings in the tree view and in the picture/class properties window in the drawing editor.

How to insert COM libraries into the RTM

Before any COM component can be used in ProcSee they have to be accessible from the pTALK language. The graphics editor GED has support for inserting COM component defintions on-line. In the

tree view control in the main window in GED, select ComLibraries. Right click with the mouse button on the *ComLibraries* heading and select *New...* from the popup-menu. See Figure 148.

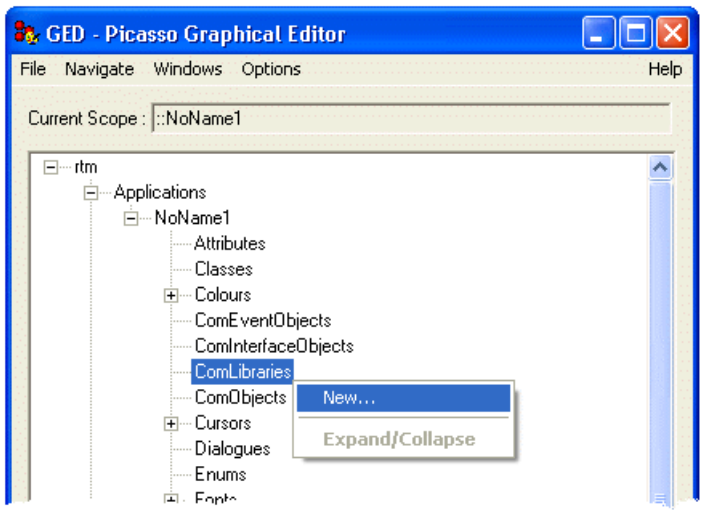


Figure 148 *This picture shows how to open the New COM library window from the main tree in GED.*

A *New Com Library* dialogue box will be opened and displayed on the screen, see Figure 149. New COM component definitions can be inserted into ProcSee from this window.

Notice that the number of components and what kind of components available on the computer will vary from computer to computer. It depends on a number of factors, that is, the operating system used, the programs installed and the type of components installed. Thus, it is very important to realize that COM component definitions inserted into ProcSee on one computer will not be available if the ProcSee application runs on another computer where the components are not installed. The COM components used in the ProcSee application must be installed on all the computers where it will be running.

The COM library window is divided into four parts. The upper part provides information about the library that is selected. That is the name of the library, the unique library identifier, the major and minor version number and the location where it is put on the harddisk, see Figure 150.

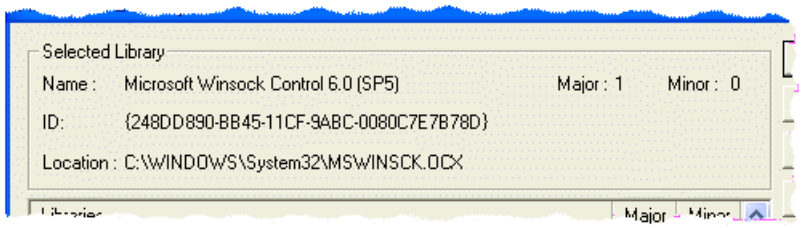


Figure 150 *This figure shows the selected library information provided by the New COM library window in the graphics editor GED.*

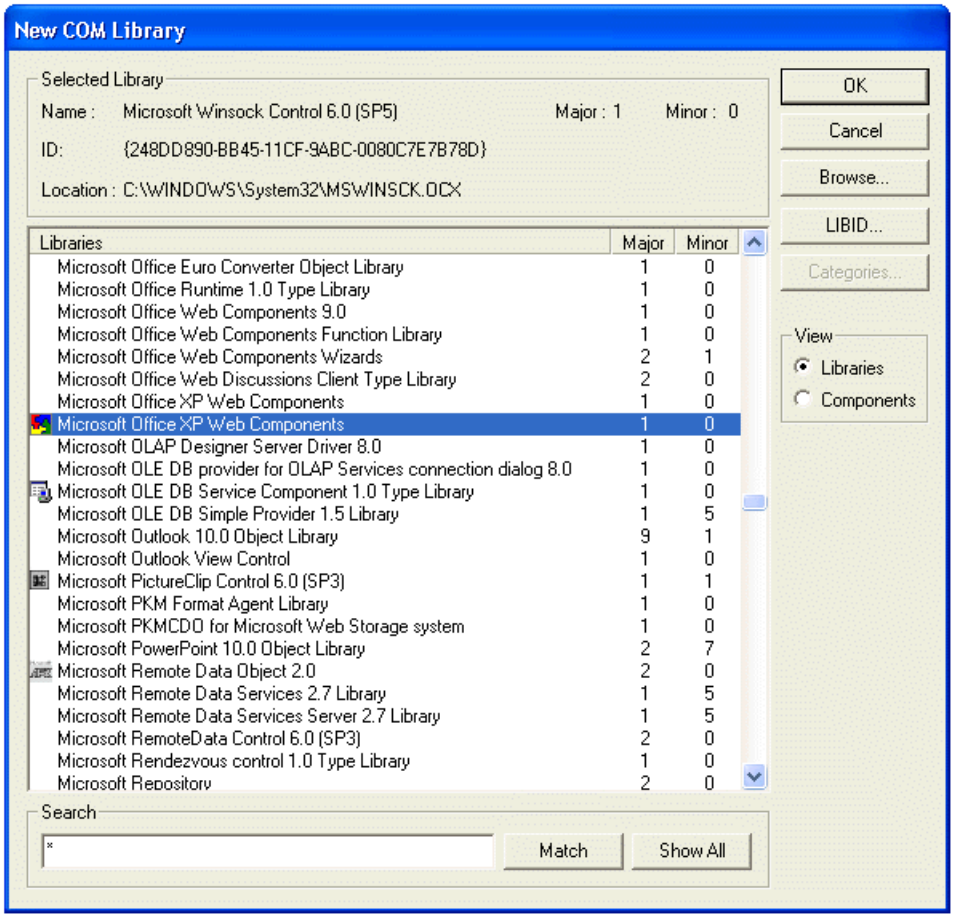


Figure 149 *This figure shows the New COM library window where COM component defintions can be inserted into ProcSee.*

The main part is the list of the components or libraries installed and registered on the computer. The list displays either COM libraries or COM components. To change this list from libraries to components or vice versa, select either Libraries or Components from the View group. See Figure 151.

Figure 151
These radio buttons are used to change between library view and component view.

The search part is used when searching for specific components or libraries. If for instance a button should be inserted into ProcSee, but the name of the component is unknown, then the search mechanism in GED is a powerful tool for getting just the right component. When typing something in the search input field, the list will scroll automatically to the first component or library in the list which matches the characters typed into the input field. A short beep will be audible if the input that is typed into the input field does not match any item in the list. The *Match* button will narrow

the list to be just the components or libraries that match the characters typed into the input field. Notice that using the asterisk (*) sign will match zero or several characters, and the question mark (?) will match exactly one character. For instance *Microsoft will match any component or library containing the word Microsoft. Pressing the *Show All* button will display all the components or libraries registered on the computer.

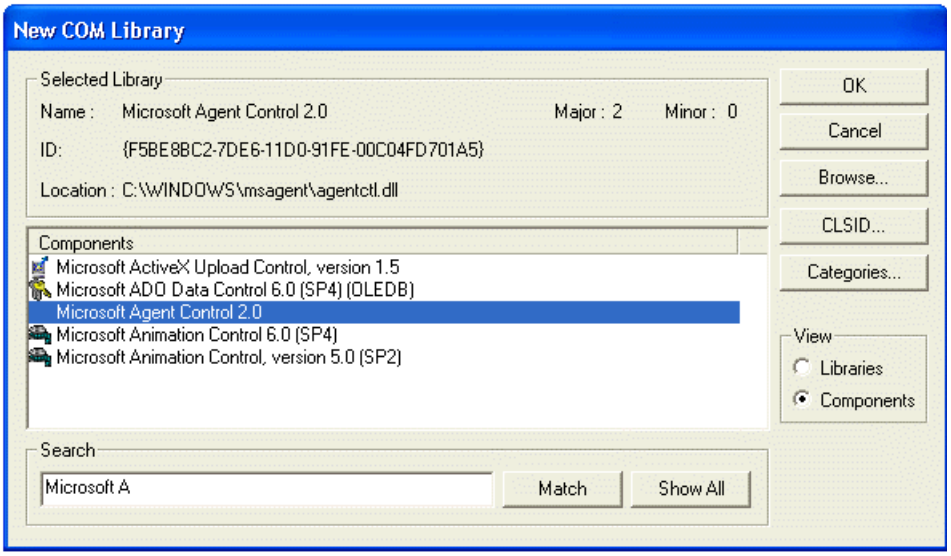


Figure 152 *This figure shows the search factilities in the New COM Library window. All components containing the string "Microsoft A" are displayed in the component list.*

The last part is the button row. The *OK* button accepts and inserts the COM library that is selected into ProcSee. The *Cancel* button rejects the selected COM library and closes the window.

The *Browse...* button opens a file dialogue box where one of the following files can be selected:

- type library file (tlb)
- dynamic link library file (dll)
- OLE custom control file (ocx)
- executable file (exe)

This feature is helpful when the location of the component on the disk is known and it is difficult to find it in the component or library list. Another reason for using this file dialogue box is that the infor-

mation about the components do not exist at the expected locations in the registry. That is, some components have not been registered properly in the Windows registry.

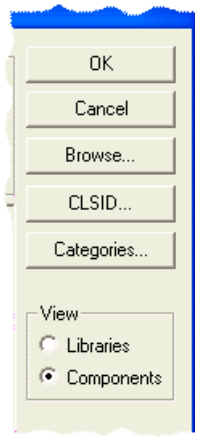


Figure 153
This figure show the button row in the New COM Library window.

Depending on whether *Libraries* or *Components* is selected in the View group, the name of the next button will vary. If *Components* is selected, pressing the *CLSID*... button will display a dialogue where either a program identifier can be entered (for instance OWC10.ChartSpace.10) or a class identifier (for instance {0002E556-0000-0000-C000-000000000046}). For more information about ProgId's and CLSID's, see Microsoft documentation.

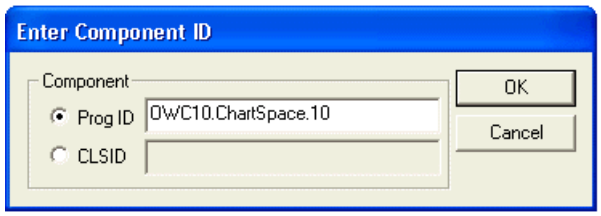


Figure 154
This window is used to manually type in either the component's ProgID or CLSID.

If *Libraries* is selected, pressing the *LIBID*... button will display a dialogue where the COM library information can be entered. The LIBID is the identifier that uniquely identifies the library. This LIBID together with the library major and minor version number will make the COM library accessible from GED.

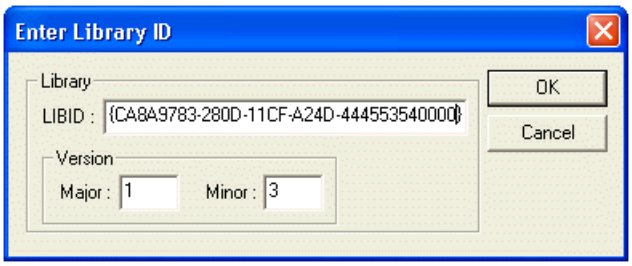


Figure 155
This window is used to manually type in the library identifier and major and minor version number.

The *Categories* button is enabled if *Components* is selected in the *View* group. Pressing this button will open a dialogue where it is possible to select among the different COM categories which are available on the computer. Almost all COM components that are installed on a computer are registered in a category. The category *Controls* is by

default selected. Most of the components with a graphical user interface are registered in this category. All components registered in the Automation Objects category are automation compatible components that should work in ProcSee.

Changing the categories will modify the list displayed in the list of components to select from. Select the categories and press the *OK* button.

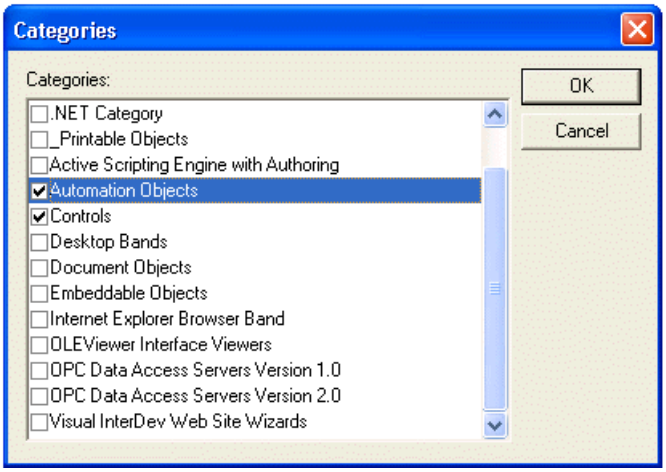


Figure 156 *This figure shows the Categories window where you select among the COM categories registered on your computer. Changing the categories will modify the components list.*

The list of available categories may vary from computer to computer. In Figure 156 the OPC Data Access Servers Version 1.0 is a category that is inserted when the OPC component is installed. So, depending on the COM components installed, the number of different categories provided by the computer may vary.

Notice that the same component can be registered in several categories. For instance, the spreadsheet component from Microsoft is registered in both the Controls category and the Automation Objects category.

Inserting ComShape

To insert a COM shape in a picture or class press the COM shape icon in the shape icon toolbar. This is normally a vertical toolbar located to the left in the graphics editor GED. Left click on the COM shape icon, as shown in Figure 157. This will open the *New ComShape* dialogue window.

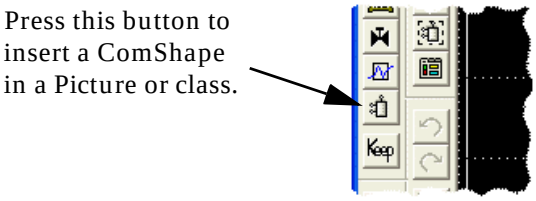


Figure 157 *This figure displays a section of the shape toolbar.*

The *New ComShape* window displays all the COM component definitions which have been inserted into the RTM and are available from the pTALK language. The COM component defintions are put into the different categories where they belong. The default category which is used for COM shapes are *Controls.* All COM components which have been registered in this category will be displayed in the list when *Controls* have been selected in the tab window. If a COM shape is registered in another category just click on that category in the tab window and GED will display those COM shapes. The components which are not registered in any of the categories will be displayed under the *Others* tab window.

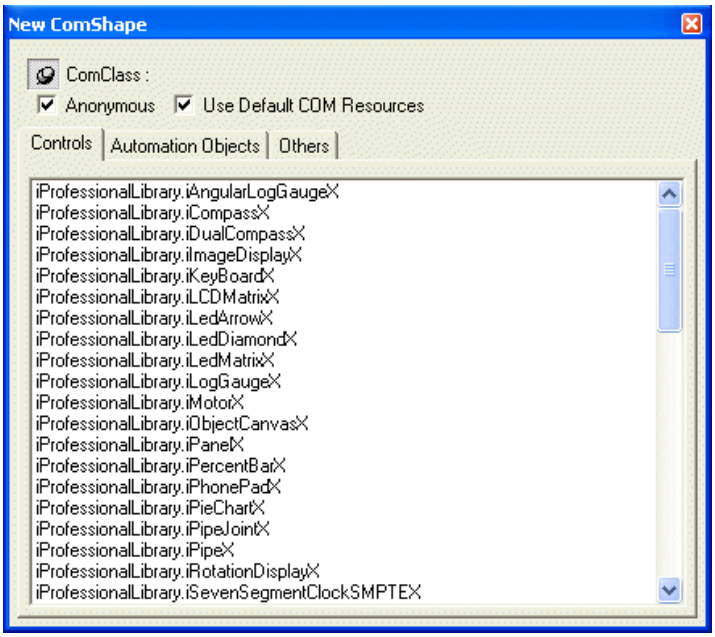


Figure 158 *This figure displays the New ComShape window which is used to insert new ComShapes in a picture or class in ProcSee.*

Select the COM component defintion that you want to insert into the RTM from the list. The currently selected COM component defintion is displayed to the right of the text *ComClass*. To insert the COM component that is selected into a picture or class, click with the left mouse button in the picture on the position where you want to place the COM shape.

The COM shape that is inserted will be displayed in an edit state. Some COM shapes do not look identical in edit state and test state. Figure 159 illustrates a COM component that has two different appearances in edit and test state. The left picture displays the component in test mode while the right picture displays the same component in edit mode.

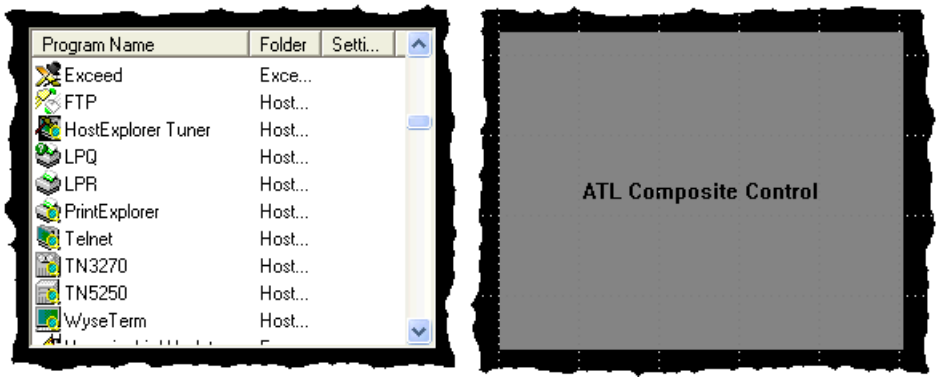


Figure 159 *This figure illustrates the difference between Microsoft Search Assistant component in edit and test state.*

An error message will be printed in the ComShape if the component fails to be initialized when it is inserted into ProcSee. A component can fail for several reasons. The most obvious reason is that the component does not have the interfaces required for a control with a graphical user interface, i.e. it is not compliant with ActiveX. An error message resembling the ComShape which is illustrated in Figure 160 will be displayed.

Figure 160
This figure shows a non-graphical component inserted into a picture as a ComShape.

The configuration data used by a ComShape can be saved and later restored when the picture or library is being reloaded. If and where the ComShape data is saved is controlled by the ComShape function *persist()*. Passing the value 0 to this function will ignore the ComShape's data when saving the picture or library. When passing the value 1 to the *persist()* function, the ComShape data will be saved in the same file as the library or picture file (this is the default behaviour). When the parameter passed to the *persist()* function is set to the value 2, the ComShape data will be saved in a separate file.

When passing the value 2 to the *persist()* function, it is very important to give the ComShape names and all the parent objects. A detailed explanation why it is important to give the COM shapes names and a description of the COM binary files are given in section 17.9 *"COM Binary files"* on page 299. It is possible to insert non anonymous COM shapes from the *New ComShape* window in the graphics editor GED. The component that is inserted will automatically be given a name if the *Anonymous* check box is unchecked. The name will be *comSH_#,* where # is a sequential number staring with the number 1.

If the *Use Default COM Resources* check box is checked then the COM shape that is inserted, will use the components default values for the ambient properties. The ambient properties which it try to use is, foreground colour, background colour and font. If this check box is unchecked then the RTM will try to use the colours and font set as default resources in the RTM. The default resources in the RTM can be changed from the *Graphics Attribute* window, see section 13.4 *"The View Menu"* on page 192.

Pressing the push-pin in the *New ComShape* window's upper left corner will keep the window in a mapped state.

A *ComShape* can be set in a user mode state. When a *ComShape* is in this state it may support in place editing. Not all components support this feature. To set a COM shape in this user mode state, double click on it with the left mouse button. A COM shape can only be set in a user mode state when the picture is in edit mode. Why do some components support this state? The answer to this question is that in place editing is more user friendly than setting the same properties by using interface functions and interface properties. Another reason is that a component in this state may support more property pages than it does in test or edit state. More about property pages in section *"ComShape property pages"* on page 278.

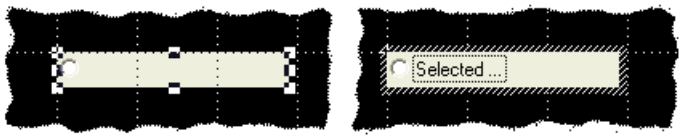


Figure 161 *This figure illustrates Microsoft Forms Option-Button in edit state and user mode state.*

When a *ComShape* is in edit state it can be edited just like any other shape in ProcSee. A *ComShape* in edit state is displayed with eight rectangular markers. This is similar to, for instance the rectangle shape in ProcSee. The markers can be used to resize the COM shape. If you press the mouse on one of the markers you can resize the object. Moving a COM shape is done by pressing the mouse on the object. You get an outline of the object that you can use to move it to another position in the picture.

When it is in a user mode state the eight rectangular markers are replaced with a hatch pattern that is displayed around the component. The two pictures in Figure 161 illustrates the difference between the two states. It is easy to see that a component is in user mode state. Notice that only one component in a picture can be in this state, simultaneously. In Figure 161 the picture to the right illustrates that text can be entered into the option-button when it is in a user mode state. The option-button component marks this with a rectangle with dotted lines. This is component specific behavior. It is not possible to enter text into this option-button when it is in edit state.

ComShape property pages

Property pages are used to display a component's configuration information to the user and to enable the user to modify that information. It can contain one or several property pages that can be selected from a tab window.

However, not all components support property pages. If they do then use it to set the state of the component. Most components support the same functionality through interface functions and interface properties, but it is often easier to use property pages to set the state.

To open the property pages window in the graphics editor GED click on the ComShape property pages button in the property toolbar. See Figure 162. This button is only enabled if a ComShape is selected in the picture.

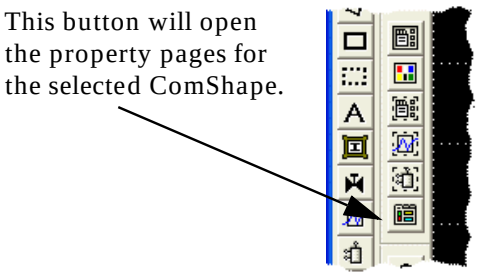


Figure 162 *This figure displays a section of the property toolbar in the drawing editor in GED.*

The property pages window is tailor-made to the component that it is associated with, i.e. the layout of the window and the number of tabs in the tab window is component specific. Figure 163 illustrates

this where the left picture is a property page window for a calendar control, while the right picture is a property page window for a grid control.

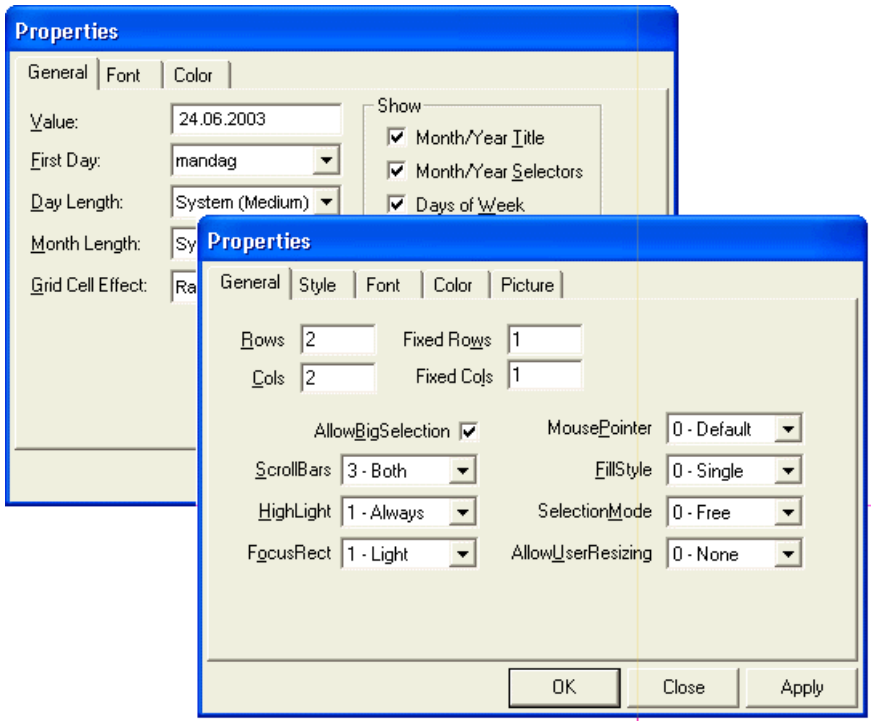


Figure 163 *This figure illustrates the difference between two property pages. The left picture is the property page for Microsoft Calendar control, while the right picture is the property page for Microsoft Flexgrid control.*

Some COM shapes have different property pages depending on whether the component is in edit state or user mode state. The user mode state will generally have more options than the edit state if the property pages are different in the two states. Figure 164 and Figure 165 illustrates those differences. Figure 164 illustrates the property pages for the Microsoft Spreadsheet in edit state. Figure 165 illustrates the same component when the component is in the user mode state. As can be seen from these figures the user mode state has more options which can be used to configure the component.

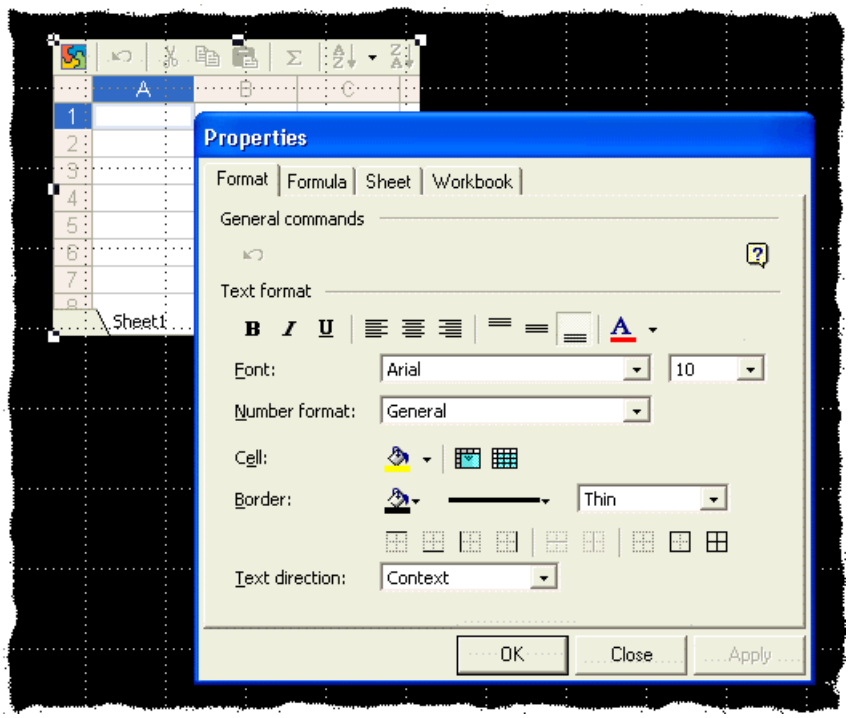


Figure 164 *This figure shows the property pages for the Microsoft SpreadSheet when the component is in edit state.*

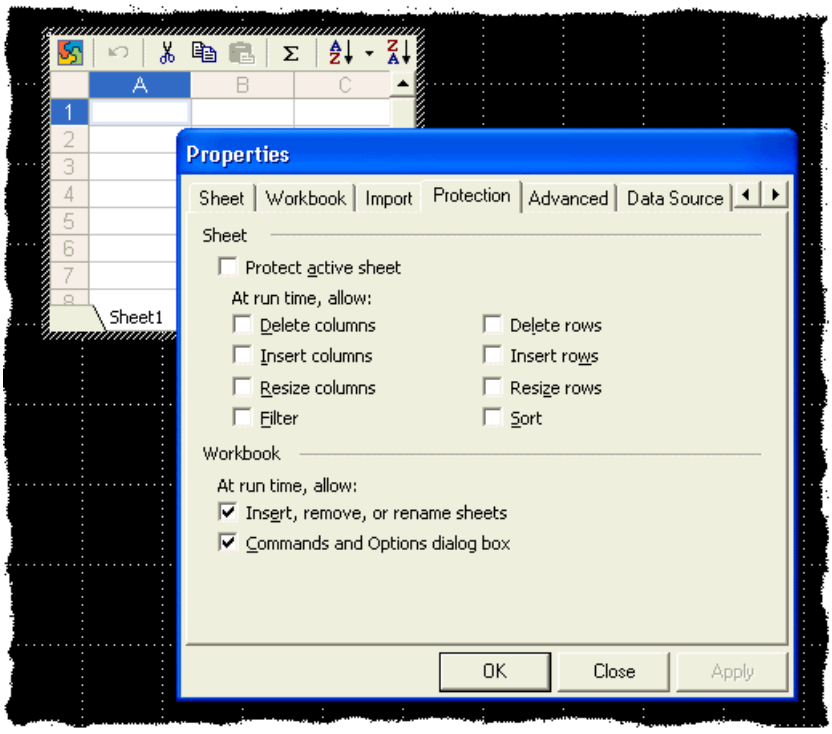


Figure 165 *This figure shows the property pages for the Microsoft SpreadSheet when the component is in user mode state.*

Selected Properties

To edit the properties, functions and dialogues of a single selected COM shape, open the *Selected Properties* window and the *Selected ComShape* window. These windows are opened by clicking on the buttons illustrated in Figure 166. These buttons are located to the left in the drawing editor in GED.

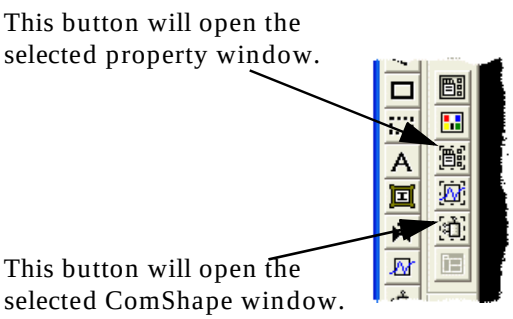


Figure 166 *This figure shows the property toolbar in the drawing editor in GED.*

The *Selected Properties* window which is used when editing COM shapes, is the same window used when editing any other shape in ProcSee. The *Selected Properties* window provides a list of all attributes, functions and dialogues which are common to all COM shapes. That is background and foreground colour, font, position, width and height, visibility, resizeable, scale and rotation.

Notice that rotation of COM shapes are not supported by any of the ActiveX control interfaces and thus they can not be rotated. They are rotated just like the text shape in ProcSee, i.e. only the position is rotated around its reference point.

The resizeable attribute controls whether the COM shape is allowed to resize itself or not. Setting this attribute to 1 will permit the COM shape to do so.

Figure 167 shows the attributes that are available when a push button is selected in the drawing editor in GED.

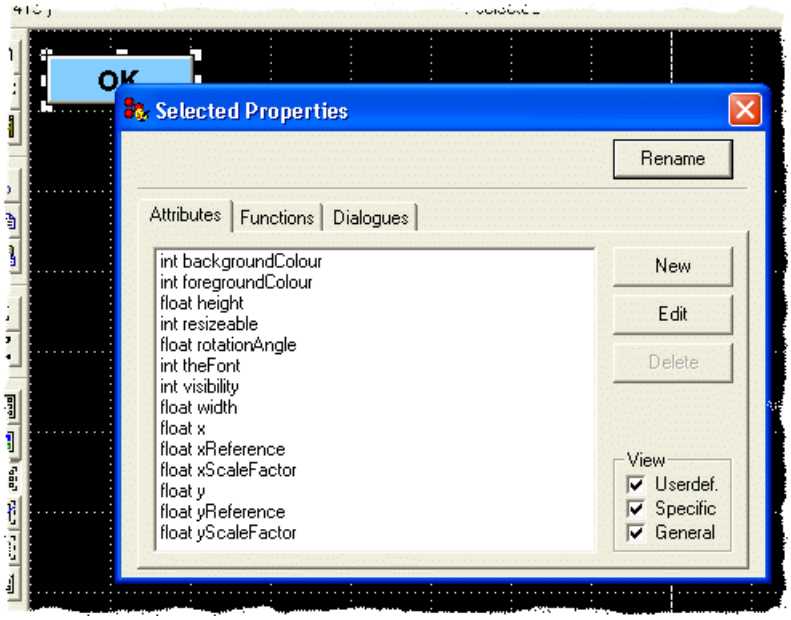


Figure 167 *This figure illustrates the seleced properties window for a ComShape.*

The *Selected ComShape* properties window is used together with the *Selected Properties* window to select the interfaces and event interfaces to edit. This window has the same manner of operation as the *Selected Trend* properties window in GED. Figure 168 shows the *Selected ComShape* window.

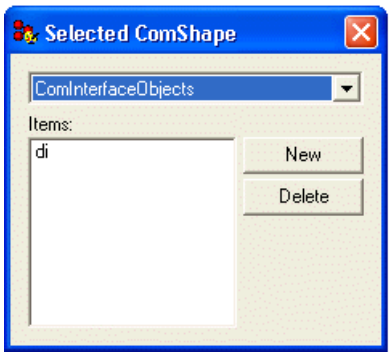


Figure 168
This figure shows the Selected ComShape window in the drawing editor in GED.

When the drop down list is clicked on with the left mouse button, located in the upper part of the window, a list with the items shown in Figure 169 will be displayed. The following items are available in the list, *Attributes*, *ComInterfaceObjects* and *ComEventObjects*.

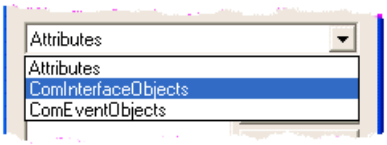


Figure 169
This figure shows the pull-down menu in the Selected ComShape window.

The items list is used to display all the items, i.e. *ComInterfaceObjects* and *ComEventObjects* that are created as children of the *ComShape*. The list will be empty if *Attributes* is selected in the drop down list. Selecting one of the items in this list will display all the attributes and functions provided by the selected interface in the *Selected Properties* window.

The default interface and default event interface will be given default names when being created as children of the *ComShape* and *ComObject*. The name of the default interface is *di* and the name of the default event interface is *dei*. These default names can be changed from the *p3RegAdmin* program. See Figure 170.

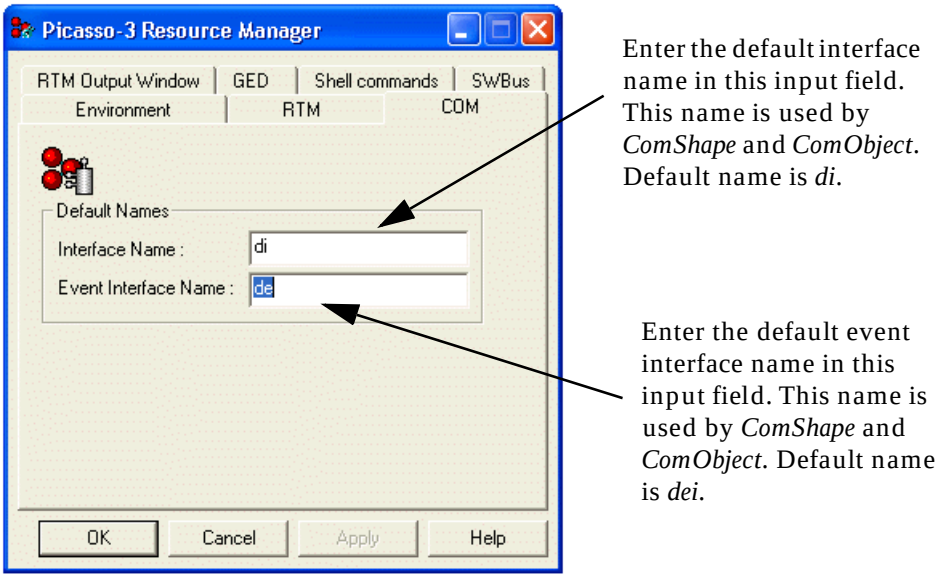


Figure 170 *This figure shows the COM tab in the ProcSee Resource Manager (p3RegAdmin) program.*

The button column in the *Selected ComShape* window will be enabled when *ComInterfaceObject* or *ComEventObject* is selected in the drop down list. To add a new interface or event interface click on the *New* button. This will open a window that displays all the interfaces that are available in the RTM. See Figure 171. Select the interface to create in the list and enter a name for the interface and press the *OK* button. The *ComInterfaceObject* will be created as a child of the *ComShape*. This interface will automatically be initialized.

To delete an interface select the interface to delete in the list and press the *Delete* button.

Interface properties (attributes in ProcSee) can be changed from the *Selected Properties* window in GED. In the *Select ComShape* window, select *ComInterfaceObject* from the drop down list and in the items list the interface to modify. The attributes for the interface will be displayed in the attributes list in the *Selected Properties* window. To modify one of these attributes select the attribute and press the *Edit* button. The *Attribute* editor will be opened with the specified

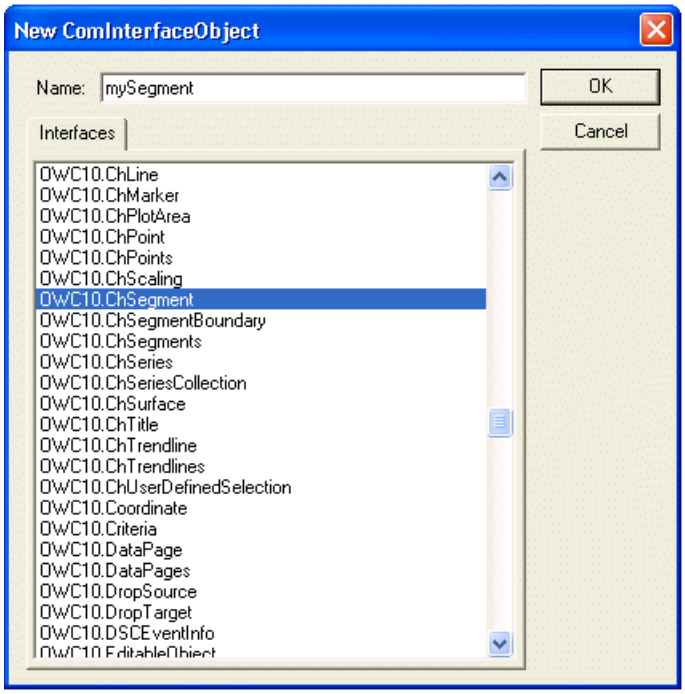


Figure 171 *This figure shows the New ComInterfaceObject window.*

attribute. Modify the attribute value as if it was an ordinary attribute and press the *Compile* button and *OK* button. See Figure 172.

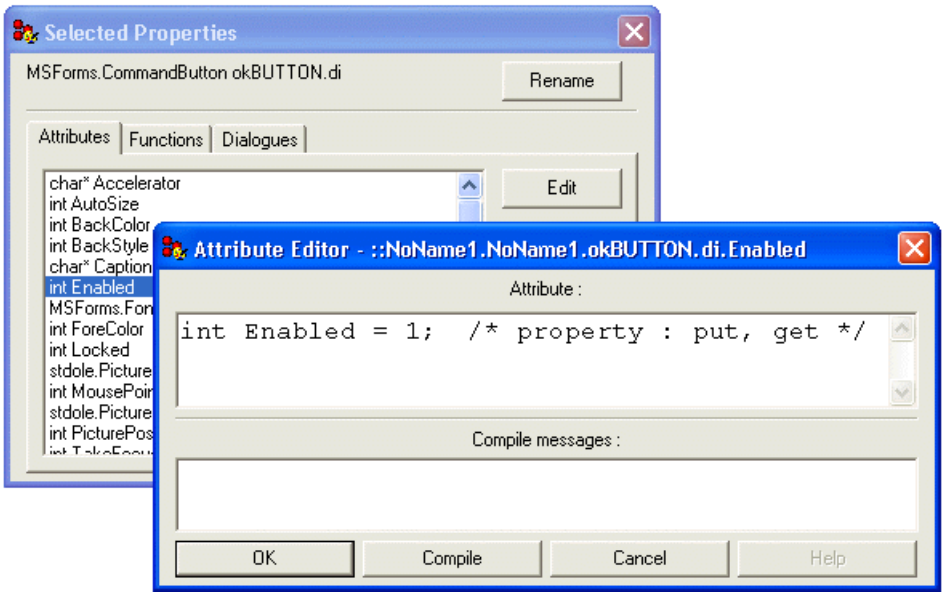


Figure 172 *This figure illustrates the use of the attribute editor to modify a component's property from the selected properties window.*

To add or edit an event interface function, select ComEventObjects from the drop down list in the *Selected ComShape* window. In the list of event interface objects select one of the items. All event functions which are implemented will be displayed when the *Functions* tab in

the *Selected Properties* window is selected. To edit an event function, select the function in the list and press the *Edit* button. Modify the function and press *Compile* and *OK*. Notice that you should not modify the function prototype. It is very important that the parameters and return value from the function exactly match what is expected from the COM component.

To implement a new event function, press the *New* button in the *Selected Properties* window. A function editor will be opened where you can select from a drop down list all event functions which are supported by the interface. Select one of the event functions in the list and implement the function body. Press *Compile* and *OK*. Figure 173 shows the function editor which is used when creating new event functions. In this figure is the *Click()* event function selected from the list of functions provided by the event interface. Notice that the event function will not be called if the picture is in edit mode. To test an event function, set the picture in test mode.

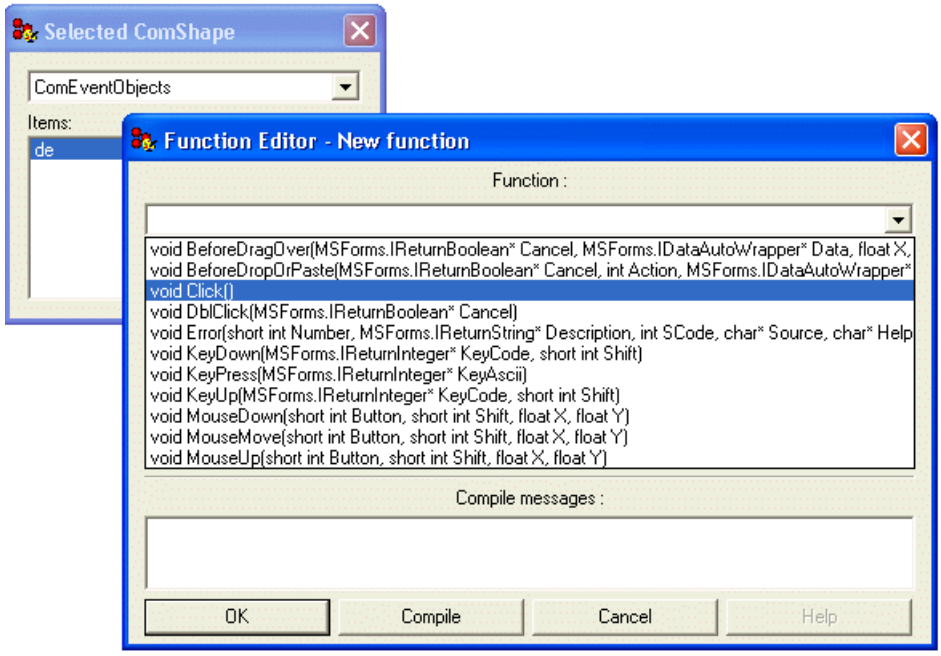


Figure 173 *This figure illustrates the use of the function editor to edit event functions in the drawing editor in GED.*

Picture/Class Properties

To create a ComObject in a picture, open the *Picture Properties* window in the drawing editor in GED. Select the *ComObjects* tab and press the *New* button. See Figure 174.

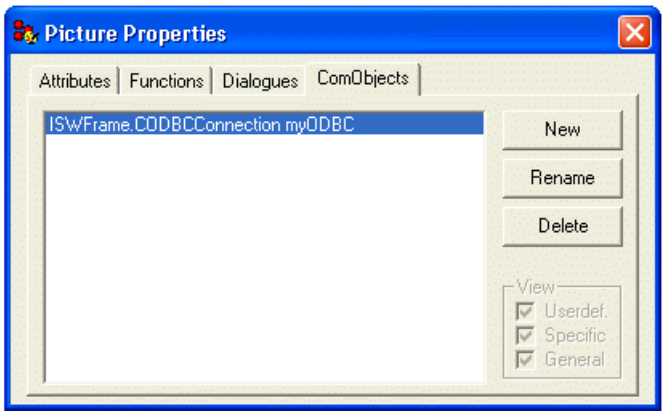


Figure 174 *This figure shows the Picture Properties window when a ComObject is selected.*

This action will open the *New ComObject* window. It displays all the COM component defintions inserted into the RTM which are available from the pTALK language. The COM component defintions are put into the different categories where they belong. The default category that is used for COM objects are *Automation Objects*. All COM component defintions which have been registered in this category will be displayed in this list when *Automation Objects* has been selected in the tab window. If a COM object is registered in another category just click on that category in the tab window and GED will display those COM component defintions. The components not registered in any of the categories will be displayed under the *Others* tab.

Select the COM component defintion that you want to insert into the RTM from the list. Enter a name for the COM component. These actions will enable the *OK* button. The COM object of the selected type will be inserted into the picture when the *OK* button is pressed.

How to create new *ComInterfaceObjects* and *ComEventObjects* are described in *section "Adding ComInterfaceObject and ComEventObject"* on page 288.

It is also possible to create *ComInterfaceObject*s and *ComEventObject*s from the *Picture Properties* window. These tabs are available when the option *Show Event/Interface obj. headings outside com objects* is enabled in the *View Settings* window. See Figure 176. How to enable this option, see section *"Making COM support available in GED"* on page 268. Press the *New* button to create either a new *ComInterfaceObject* or a new *ComEventObject*. This action will open the *New ComInterfaceObject* window. This is described in section *"Selected Properties"* on page 281.

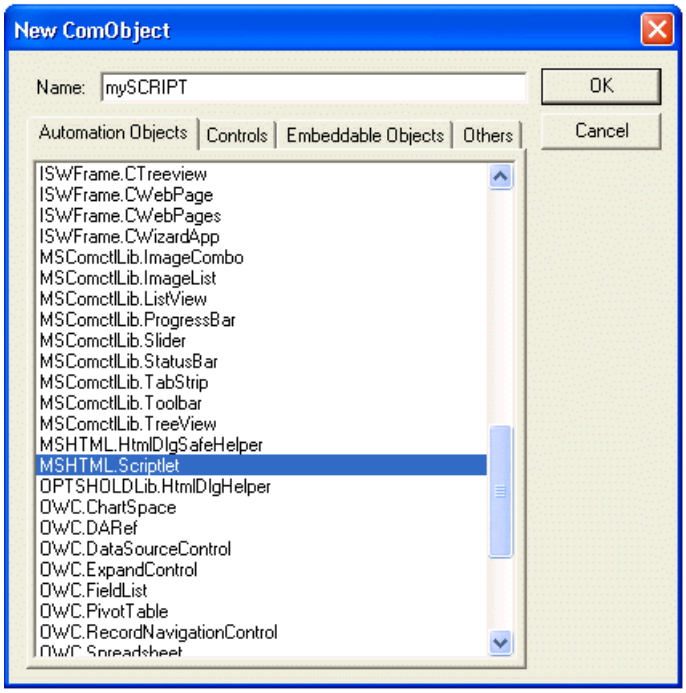


Figure 175 *This figure shows the New ComObject window where you can select com objects to insert into* ProcSee.

The *ComInterfaceObject* will not be initialized if it is not created as a child of a *ComShape* or a *ComObject*. To initialize a *ComInterfaceObject* instance run the pTALK function *queryInterface(…)* or use the assignment operator. This function is either a member of the *ComObject* or *ComShape* or any of the associated interfaces. The *ComInterfaceObject* instance is passed as argument to this function.

The function *comAdvise(…)* must be called when adding user implemented call-back functions to an event interface which is not created as a child of a *ComShape* or a *ComObject*. This function accepts two arguments, the first parameter is a reference to the event interface and the second is a pointer to an interface. To add a user implemented call-back function to this event interface, use the tree view in the main window in GED. Find the *ComEventObject* in the tree and select new from the *Functions* heading under the event object. For more information about creating event interface functions, see section *"Selected Properties"* on page 281.

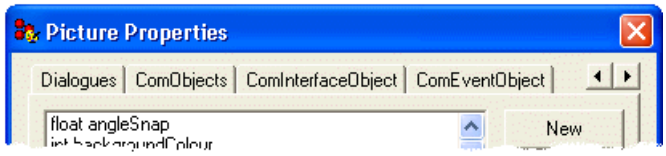


Figure 176 *This figure shows the extra tabs which are available when the show COM interface/event headings is enabled in the View Settings in GED.*

Adding new ComObject in application or library

To add a *ComObject* in an application or library select the *ComObjects* heading in the tree view list in the main window in GED. Click with the right mouse button on this item. A pull down menu will be displayed. From this window select *New Object*. See Figure 177.

Figure 177
This figure shows the pop-up menu which is displayed when the right mouse button is clicked on the ComObjects heading in the main tree view in GED.

This action will display the *New ComObject* window. How to create a new *ComObject* instance is described in detail in section *"Picture/ Class Properties"* on page 286.

Adding ComInterfaceObject and ComEventObject

To add a new *ComInterfaceObject* or a new *ComEventObject* in the application or the library, navigate to the *ComInterfaceObjects* heading or *ComEventObjects* heading in the tree view list in the main window in GED.

Click with the right mouse button on the item. This action will display a pull-down menu. Select *New...* from this menu. The *New ComInterfaceObject* window will be displayed on the screen.

For more information about how to add these kind of interfaces, read section *"Selected Properties"* on page 281.

The *ComInterfaceObjects* and *ComEventObjects* items are available when the option *Show Event/Interface obj. headings outside com objects* is enabled in the *View Settings* window. For more information, see section *"Making COM support available in GED"* on page 268. However, it is recommended that *ComInterfaceObjects* and *ComEventObjects* are created as children of *ComShape* or *ComObject*.

The update function onUpdate

Defining a function called *onUpdate()* as a child of a *ComShape* or a *ComObject* has a special meaning. ProcSee will on every update look for this function in the application, library, COM objects, COM shapes, instances, groups and trend. When the *onUpdate()* function is found it will be executed. Notice that this function will be called before the ordinary picture update.

This function can be used to modify a COM object on every update. For instance, if several cells in a grid component should be updated with new data on every update, then update the component in the *onUpdate()* function. An example is given below.

```
comShape GridLib.Grid theGrid
{
   comInterfaceObject GridLib.IGrid di
   {
      ...
   }

   function ´void onUpdate()
   {
      di.Update( 0, 0, Value[0] );
      di.Update( 0, 1, Value[1] );
      di.Update( 1, 0, Value[2] );
      di.Update( 1, 1, Value[3] );
   }´
}
```

Notice that interface attributes can have dynamic expressions just like any other attribute. It will be evaluated on every update and the COM component will be updated with the updated values.

17.6 COM definitions in the symbol table

Before a COM component can be used in ProcSee it must be inserted into the symbol table. All components must be inserted into ProcSee before they can be available from the pTALK language. This is just like any other type, resource or class used in ProcSee. They must be known before they can be used. The graphics editor GED is used to insert new COM component defintions and to make them accessible from the pTALK language in ProcSee. All components on the machine are displayed in a list in GED where users can select components to insert into the RTM.

When a COM component defintion is inserted into ProcSee, everything which the component has references to, will be created in the symbol-table in addition to the component itself. That is, interfaces, enumerations, libraries, components, functions and properties.

All available information in the type library will be used when creating the COM definitions in the symbol-table in ProcSee. The type library contains enough information to create complete types so that they can be used in the pTALK language. For instance, function names, argument names and argument types are specified in the type library. The data type *any* is used in ProcSee to specify a data type of any type (VARIANT data type in COM). So, what happens when a COM component is inserted? Well, the COM library that the component belongs to will first be created in the symbol table. Next, all the interfaces, enumerations and components available in the type library will be created as children of this library. Due to this fact, all components exposed by a type library will be inserted when inserting a single component from the library. For instance inserting

the component Microsoft Forms CommandButton will insert all the other components in the library as well. That is ScrollBar, CheckBox, etc.

COM Library

A COM library is a container for the COM classes, interfaces and enumerations. These types will be created as children of the COM library in the symbol table. The COM library does not contain any functionality. The main purpose with the COM library is to organize COM in a hierarchy, like it is organized in the type library. This will avoid name conflicts when several components are inserted into ProcSee from different vendors. COM components from different vendors may have used same names on interfaces, components or enumerations.

When inserted into ProcSee, the COM library will be given the same name in the symbol-table as specified in the type library. This is the default name. It can later be renamed either from the graphics editor GED or by using the pTALK function rename(…). This name is part of the full qualified name that must be used to access enumerations, interfaces and COM classes in ProcSee.

A library identifier (LIBID) together with a major and minor version number is used to uniquely identify a COM library in ProcSee. This information is used when the library is reloaded from file. For more information, see the “Configuration Language" section in the Reference Manual.

ComLibrary is the name of the meta-class that is used in ProcSee. All COM libraries inserted into ProcSee are derived from this base class. It does not contain any functions except for the functions derived from the meta-class Object.

COM Class

The COM class describes a component which is inserted into ProcSee. The COM class will be inserted into the symbol-table as a child of the COM library the component belongs to.

The COM class contains information about the interfaces and event interfaces which is supported by the component. Almost every component available on the market has something that is called default interfaces. These interfaces will automatically be created when an instance of one of the COM classes are inserted into ProcSee. A component has normally one default interface and one default event interface.

The COM class information is available from the graphics editor GED and can be displayed using the view button when selecting a COM class from the tree view in the main window in GED. A list of

the interfaces and event interfaces supported by the component will be displayed. This is very useful since it can be used to get information about the interfaces supported by a component.

The component identifier (CLSID) is used to uniquely identify a COM class in ProcSee. This information is used when the component is reloaded from file. For more information, see the “Configuration Language" section in the Reference Manual.

When inserted into ProcSee, the COM class will be given the same name in the symbol-table as specified in the type library. This is the default name. It can later be renamed either from the graphics editor GED or by using the pTALK function rename(…).

ComClass is the name of the meta-class which is used in ProcSee. All COM classes inserted into ProcSee are derived from this base class. It does not contain any functions except for the functions derived from the meta-class Object.

The name of the ComClass is used when inserting ComShapes or ComObjects in ProcSee. For more information about ComShapes and ComObjects see section *"COM Shape"* on page 294 and section *"COM Object"* on page 294. A full qualified name must be used when creating these objects. That is the name of the COM library plus the name of the class. The graphics editor GED or the pTALK functions newComShape(…) or newComObject(…) are used to create instances of the COM classes in the symbol-table.

COM Interface

The COM interface definition describes the functions and properties which can be used to access a component. It is important to stress the fact that a COM interface is just a definition. It can not be used to access a component. To get access to a component’s functionality either a ComInterfaceObject or an interface pointer of a COM interface type must be used. For more information, see section *"COM Interface Object"* on page 295 and section *"COM Interface Pointers"* on page 297.

The COM interface will be inserted into the symbol-table as a child of the COM library the interface belongs to. A COM interface can describe both an interface that is used to access a component and an event interface that is used to get callbacks from a component.

An interface contains functions and properties. COM functions are accessed just like any other function in pTALK. A function has input and output parameters and it may have a return value. Properties are defined as attributes in ProcSee. They can be accessed on both sides of the assignment operator. Notice that some properties can be read-only or write-only attributes.

However, some properties will be converted to functions in ProcSee. Properties which have more than one argument, will automatically be converted to functions. For instance an array property using an index to access correct element in the array will become a function. Since a property can be used on both sides of the assingment operator (lvalue and rvalue), three different functions will be created with different postfix. Three, because COM supports both put and put reference properties. The postfix notation used by ProcSee is:

- *property*_G – get property
- *property*_P – put property
- *property*_PR – put reference property

The following is an example of an array property (called Level) specified in a type library.

```
[id(1), propget]
HRESULT Level([in] int ix, [out, retval] VARIANT* value );
[id(1), propput]
HRESULT Level([in] int ix, [in], VARIANT value );
[id(1), propputref)]
HRESULT Level([in] int ix, [in], VARIANT* value );
```

The Level property will be converted to these functions in ProcSee.

```
void Level_P(  int ix, any value );  // put
void Level_PR( int ix, any* value ); // put reference
any  Level_G(  int ix );             // get
```

The interface identifier (IID) is used to uniquely identify a COM interface in ProcSee. This interface identifier is used when the interface is reloaded from file. For more information, see the “Configuration Language" section in the Reference Manual.

When inserted into ProcSee, the COM interface will be given the same name in the symbol-table as specified in the type library. This is the default name. It can later be renamed either from the graphics editor GED or by using the pTALK function rename(…).

ComInterface is the name of the meta-class that is used in ProcSee. All the interface classes, which are inserted into ProcSee, will have *ComInterface* as the base class. This meta-class does not contain any functions except for the functions derived from the meta-class Object.

COM Enumeration

The COM enumeration type defines a set of integer constants which can be used to initialize function parameters and properties when accessing functionality in a COM component. An enumeration contains a set of enumerators (fields) where each enumerator is assigned a constant integer value. To access an enumerator value a full qualified name must be used.

MSForms.fmAlignment.fmAlignmentLeft is an example of a full qualified name. This name must be used in pTALK to get the value of the enumerator *fmAlignmentLeft* which is defined in the enumeration *fmAlignment* in the COM library *MSForms.*

The COM enumeration will be inserted into the symbol-table as a child of the COM library the enumeration belongs to. All enumerators will be children of the enumeration. An example is given below.

```
comLibrary liquidLIB
{
   comEnum eLiquid
   {
      eCoffe     = 0x1;
      eThee      = 0x2;
      eChocolate = 0x3;
   };
};
```

The enumeration identifier (EID) is used to uniquely identify a COM enumeration in ProcSee. It is used when the enumeration is reloaded from file. For more information, see the “Configuration Language" section in the Reference Manual.

When inserted into ProcSee, the COM enumeration will be given the same name in the symbol-table as specified in the type library. This is the default name. It can later be renamed either from the graphics editor GED or by using the pTALK function rename(...). However, some enumerations do not have the unique identifier (EID) as described above. These enumerations can not be renamed because the enumerations can not be uniquely identified when they are being reloaded from file.

17.7 Using COM in ProcSee

The description in this chapter has so far been focused on the COM type definitions. It is important to have the knowledge about a COM component before it is used. COM components can in worst-case consist of several interfaces with hundreds of functions and properties. Those components can be very difficult to use and the documentation are very often required. However, other COM components are small and can be quite easy to use. For instance, a push button is an example of a simple component, while for instance Microsoft Spreadsheet is an example of the opposite.

This section will describe the ProcSee types that are used to create and access functionality in a COM component. So far, the focus has mainly been on the COM definitions. The question is now, how can COM components be created and used in ProcSee? This section will give you the answer.

New code has been added to the graphics editor GED to support the new COM functionality. It should be quite easy to create COM components and interfaces in ProcSee. Using GED is in most cases the preferred way when creating COM components in ProcSee. However, it is still possible to create COM components and interfaces with pTALK functions (newComShape(…), newComObject(…), newComInterfaceObject(…), newComEventObject(…)). For more information about these pTALK functions, see the chapter about pTALK functions in the ProcSee Reference Manual.

COM Object

A COM object in ProcSee is an instance of a COM class that does not have a graphical user interface. Before a COM object can be created the definition of the component has to be inserted into the symbol-table. That is a *ComClass* derived data type. The name of the COM class is one of the arguments used when creating the component.

The COM object is an instance of a COM component and it contains a pointer to the actual object. However, it does not have any functionality. All access to the object is through interfaces. When inserted the default interface and the default event interface supported by the component will be created and initialized in ProcSee.

The data type in ProcSee is ComObject. It has one function and that is *queryInterface(…)*. This function is used to initialize interfaces supported by the component. The argument to this function is an object of a COM interface type or a COM interface pointer. The attribute *perist* controls whether to save the data for the ComObject or not and where to save it, that is in a separate file or in the application, library or picture file (default).

A COM object can be created in an application, library, class or picture.

An OPC server connection and a database connection are examples of COM objects without a graphical user interface.

COM Shape

A COM shape in ProcSee is a component with a graphical user interface (also known as ActiveX controls). Graphical user interface components can for instance be used in ProcSee to create applications with more Windows look-alike user interfaces. Examples of such components are menus, push-buttons, check-boxes, radio-buttons etc. Other graphical user interface components like spreadsheets, word processors, charts etc. can also be used.

Before a COM shape can be created the definition of the component has to be inserted into the symbol-table. COM components that have a graphical user interface can be used in ProcSee only if they support certain interfaces provided by ActiveX controls. Those in-

terfaces will not be described in this document since they are out of scope of this User's Guide. A list of the interfaces accessed from ProcSee is given in section *"User interface controls"* on page 301.

The name of the COM class is one of the arguments to the pTALK function used when creating components in ProcSee.

The COM shape is an instance of a COM component and it contains a pointer to the actual object. However, it does not have any functionality. All access to the object is through interfaces. When inserted the default interface and the default event interface supported by the component will be created and initialized in ProcSee.

The data type in ProcSee is ComShape. This data type contains functions and attributes. For a detailed desription, see "ProcSee Reference Manual".

A COM shape can be created in a class or picture.

COM Interface Object

To get access to the functionality that a COM component provides an interface object must be created and initialized. An interface object is either created from the graphics editor GED or by using the pTALK function *newComInterfaceObject(…)*. One of the arguments to this pTALK function is a COM interface class. For more information about COM interface classes, see section *"COM Interface"* on page 291.

When a COM interface object is created it can not be used before it has been initialized. All COM objects, COM interface objects and COM interface pointers have a function that is called *queryInterface(…)*. This function initializes a COM interface object or a COM interface pointer. The *queryInterface(…)* function has one argument and that is a pointer to a COM interface object or a COM interface pointer.

COM interface pointers will be described in detail in section *"COM Interface Pointers"* on page 297. If the *queryInterface(…)* function succeeds, then the COM interface object can be used to access and manipulate the functionality that the component provides through the interface. The function will fail if the interface which is requested is not supported by the component.

When creating a COM shape or a COM object the default interface will be created as well. This default interface can be used right away without any initialization since it will be automatically initialized by ProcSee.

A COM interface object has generally several functions and properties. These function and properties can be used just like any function and attribute in ProcSee. The following example will call the function *statement(...)* through a *mySQL* interface object pointer.

```
mySQL->statement( "select * from Alarm where Time > 71245325" );
```

The data type in ProcSee is ComInterfaceObject. It has one predefined function and that is *queryInterface(…)*. This function is used to initialize interfaces supported by the component. The argument to this function is a pointer to a COM interface type or a COM interface pointer. The ComInterfaceObject is created based on a ComInterface derived class type. The ComInterfaceObject will have the same set of functions and properties (attributes) as the COM interface definition type it is based on.

A COM interface object can be created in an application, library, class or picture. However, a COM interface object is normally created as a child of a COM shape or COM object. It is not necessary to initialize the COM interface object with *queryInterface(…)* when it is created as a child of a component. ProcSee will automatically initialize the interface.

COM Event Object

Interfaces are used to access and manipulate functionality in a COM component. This is single communication path from ProcSee to the COM component. However, events are provided by COM components for communication in the opposite direction, which is from components to ProcSee. ProcSee can respond to events in an object, provided by a COM shape or COM object. Another name of events is call-back functions.

When creating a COM shape or a COM object the default event interface will be created as well. This default event interface can be used right away without any initialization since it will be automatically initialized by ProcSee.

The data type in ProcSee is ComEventObject. All event objects are derived from this type. The ComEventObject is created based on a ComInterface derived class type. The ComEventObject will have the same set of functions as the COM interface definition. A COM event object has functions like an ordinary COM interface object. The difference is that COM event objects call pTALK functions in ProcSee and these functions must be implemented by the application developer.

For instance creating a picture containing a push-button that should respond to button clicks is quite easy. Let's say that the push-button has a default event interface of type *IPushButtonEvent* that is called *dei*. From the graphics editor GED select *dei* from the *Selected ComShape* dialogue (This dialogue is described in section *"Selected Properties"* on page 281). Open the function editor from the *Selected Properties* dialogue. For event functions, GED provides a dropdown list with all the functions supported by the event interface.

From this list select one of the functions, for instance *void OnClick(int buttonNo)*, and write the source code. This user implemented pTALK function will be called each time the push-button is clicked.

A COM event object can be created in an application, library, class or picture. However, a COM event object is normally created as a child of a COM shape or a COM object.

When a COM event object is created it can not be used before it has been initialized. pTALK functions are provided by ProcSee to initialize and uninitialize event objects. These functions are called *comAdvise(…)* and *comUnadvise(…). comAdvise(…)* must be called to initialize a COM event object. If the event object is not required anymore, *comUnadvise(…)* should be called to uninitialize it. When it is created as a child of a COM component it is not necessary to initialize the event object with the pTALK function *comAdvise(…)*. ProcSee will automatically initialize the event interface.

COM Interface Pointers

A COM interface pointer is a pointer attribute in ProcSee of a COM interface definition type. COM interface pointers are often used as local variables in functions and dialogues. It can also be created like any other attribute in ProcSee with the attribute editor in the graphics editor GED. The difference is that the attribute is of a COM interface type.

The following example initializes a local variable of type *MSForms.ICommandButton* that is used to change the button text.

```
void InitButton()
{
   MSForms.ICommandButton* ib = 0;

   // QuitButton is the name of the COM shape
   QuitButton.queryInterface( &ib );

   // Change the caption
   ib->Caption = "QUIT";
   ib = 0;
}
```

When an interface pointer has been initialized it can be used just like a COM interface object. It has full access to the functionality provided by the component through the interface. It can manipulate the component with the functions and properties that the interface provides.

Setting an interface pointer to nil (value 0) will release the interface. The interface pointer will also be released when the local variable goes out of scope.

COM interface pointers are normally used in functions and dialogues, but they can appear anywhere where normal attributes can be created. That is in applications, libraries, classes or pictures.

COM interface pointers must be initialized before they can be used. The function *queryInterface(…)* must be called to get an interface of the requested type, either from a COM component, COM interface object or another COM interface pointer.

An interface can also be initialized through functions that return a pointer to an interface. Notice that the interface will be set to nil (value 0) if the requested interface is not supported by the component.

17.8 Predefined COM Data Types in ProcSee

ProcSee provides a set of predefined data types that are used with COM. The following section will describe these data types.

IUnknown

The IUnknown data type is used when a function or property returns a pointer to an IUnknown interface. It is normally used as a temporary placeholder for interfaces. Sometimes is it necessary to save an interface if it should be used as an argument to another function, for instance in another interface.

The function *queryInterface(…)* is the only function that is provided by this interface.

IDispatch

The IDispatch data type is used in the same way as IUnknown. The difference is that it is used where an IDispatch pointer is required. It is normally used as a temporary placeholder for interfaces of this type. For more information, see section *"IUnknown"* on page 298.

IEnumAny

The IEnumAny interface is used to iterate over objects that a component provides. An object can be of any type, for instance interfaces, integers, floating points etc. Before an IEnumAny object can be used it has to be initialized. The property _NewEnum provided by an interface is a frequently used name that is used in the initialization of an enumerator.

To use an enumerator an attribute of type IEnumAny pointer must be created. An enumerator is normally used in functions and dialogues, but they can appear anywhere where normal attributes can be created. That is in applications, libraries, classes or pictures.

To enumerate through object in a list four interface functions are provided by the IEnumAny interface. A short description of these functions is given next.

- int clone(IEnumAny* i) - gets an exact copy of the enumeration.

- int next(any* p) – gets the next object in the enumeration.
- int reset() – resets the enumeration to point to the first object in the list.
- int skip(int n) – skips the next n objects in the list.

The following example creates and initializes an enumeration that is used to iterate over the elements in a list control.

```
void Iterate()
{
   // iListBox points to a list box control
   IEnumAny* enumList = 0;
   enumList = iListBox->_NewEnum;
   if ( !enumList ) return;  // Failed

   // item points to an item in the list box
   IListItem* item = 0;
   while ( enumList->next( &item ) > 0 )
   {
      // Do something with all selected items
      // and unselect
      if ( item->Selected )
      {
         item->Selected = 0;
         ...
      }
   }

   enumList = 0;
}
```

17.9 COM Binary files

The function *persist()*, which is a method in the classes ComShape and ComObject, controls whether to save the data for the objects in a separate file, in the (application, library, picture) binary file or ignore the data completly. This section described the COM binary file used when the ComShape or ComObject has the persist property set to the value 2. The persist property is set using the ComShape and ComObject function *persist()*.

A separate binary file will be created when an application, library or picture is saved containing one or several named *ComObjects* or *ComShapes* which have the persist property set to the value 2. When saving a file from ProcSee the name of this COM binary file will get the same name as the ProcSee binary file with another extension. The following list gives the extensions used for the different type of files that are saved from ProcSee.

- Application (pctx) – pcco
- Library (plib) – plco
- Picture (ppic) – ppco

All COM specific properties will be saved on this COM binary file. It is very important to realize that only objects that have names will be saved. Anonymous objects will not be saved. This is the main reason for having the *Anonymous* option in the *New ComShape* window in the drawing editor. For more information, see section *"Inserting ComShape"* on page 275. If you have instances containing *ComShapes* or *ComObjects*, it is important to give those instances names as well. The *ComShape* or *ComObject* will not be saved if the instances are anonymous.

If you have for instance a spreadsheet in a picture that has formulas, different colours on different cells etc., then the spreadsheet should probably look exactly like that the next time you reload the picture from file. It should not be necessary to initialize the spreadsheet every time it is reloaded. Or if you have a push-button in a picture which has set the text, then the text should be saved. It should not be necessary to set the text each time the picture is reloaded. This kind of COM specific data is saved on the COM binary file.

Notice that there is no text document file (Tdoc) associated with the COM binary file that you can run the ProcSee compiler (pcc) on to produce a binary file.

The COM binary files are internally organized in a strict hierarchy as it is in the symbol-table in ProcSee. Instances containing *ComShapes* or *ComObjects* will produce nodes, while *ComShapes* and *ComObjects* will be leafs containing object data.

17.10 Create your own COM components

Two different kinds of COM components are supported by ProcSee. One type is COM components with a graphical user interface. These kinds of controls can be used as graphical user interface components in pictures in ProcSee. They must comply with the rules which exist for interfaces supported by ActiveX components. A push-button is an example of this type of component. The other type of COM components which are supported in ProcSee are components with a non-graphical user interface. An example of this type of component is a database component.

Type information must be provided by the component through the ITypeLib interface. Components that do not support this interface can not be used in ProcSee. The information provided by the type library is used to declare COM libraries, classes, interfaces, enumerations, properties and functions in the symbol-table in ProcSee.

Any COM component which should be used in ProcSee must support the IDispatch interface. All arguments to interface functions and return values, including properties, must be OLE Automation compatible. Only pure automation interfaces or dual interfaces are

supported. These interfaces are derived from IDispatch. Custom interfaces are not supported by ProcSee, i.e. interfaces derived directly from IUnknown.

Event interfaces, which are used in the communication from the component to ProcSee, must follow an even stricter rule. Only pure automation event interfaces are supported. Dual and custom event interfaces are not supported by ProcSee.

User interface controls

These types of COM controls, also known as ActiveX controls, must follow some rules. There are some interfaces which are required to be implemented and others that can be omitted. The following is a list of interfaces that ProcSee supports. Notice that not all of these interfaces are required to be implemented in the controls.

- IAdviseSinkEx
- IConnectionPoint
- IConnectionPointContainer
- IOleControl
- IOleInPlaceObject
- IOleObject
- IOleWindow
- IPersistStream
- IPersistStreamInit
- IPropertyNotifySink
- IQuickActivate
- IDispatch
- IViewObject
- IViewObject2
- ITypeLib
- ISupportErrorInfo
- IErrorInfo

ProcSee also supports ambient properties which are used by a control so that it can assume something about the look and feel of its surrounding environment. The ambient properties are managed and exposed by the container. At run-time, the control requests information about the ambient properties. Not all ambient properties are supported by ProcSee. Supported properties are listed below.

- Auto clip
- Message reflect

- Locale ID
- Support mnemonics
- Display as default
- Show grab handles
- Show hatching
- User interface dead
- User mode
- Foreground colour
- Background colour
- Appearance
- Font
- Right to left
- Char set
- Code page
- Display name
- Palette
- Scale units
- Text align
- Top to bottom

Background and foreground colour, font, user mode, etc. are typically examples of ambient properties used by a control.

It is possible to set the foreground colour, background colour and font from the graphics attribute window in the drawing editor in GED. The control must support one or several of the following ambient properties:

- Foreground colour- DISPID_AMBIENT_FORECOLOR
- Backgournd colour- DISPID_AMBIENT_BACKCOLOR
- Font - DISPID_AMBIENT_FONT

Non-graphical components

Components without a user interface are much simpler compared to user interface controls. There are relative few interfaces which ought to be supported by these components. The following is a list of interfaces that ProcSee supports. Notice that not all of these interfaces are required to be implemented in the components.

- IConnectionPoint
- IConnectionPointContainer

- IPersistStream
- IPersistStreamInit
- IDispatch
- ITypeLib
- ISupportErrorInfo
- IErrorInfo

18 Drag and Drop

This chapter gives an introduction to the drag and drop concept in ProcSee. Drag and drop allows users to pick up an object from one part of the display and move it to a new location with the mouse.

18.1 Introduction

Drag and drop is a feature available in ProcSee. On Microsoft Windows and X-Windows the drag and drop protocol is supported by almost every known application, from word processors, spreadsheets, database programs, etc. Users world wide are used to move information or data from one program to another without using the keyboard.

The drag and drop facility allows users to pick up an object from one part of the display and move it to a new location. It is just a shortcut for the copy-paste functionality. Instead of using copy-paste it is sometimes much easier to select for example some text in a word processor, click with the mouse on the selected text, move the mouse while the mouse button is down and drop the data in another application. The text will then be copied or moved to the target application. When the copy-paste functionality is used the same text needs to be selected. Then the text must be copied to the clipboard using copy from the menu or using control-C. In the target application the mouse must be moved to the point where the text should be inserted. To paste the text the control-V key or paste from the menu must be selected. The text will then be copied from the clipboard to the application. The drag and drop operation eliminates the need for menus and is most of the times quicker than the copy-paste sequence.

Another feature when drag and drop is used is the visual feedback the users get while moving data from one application to another. Different cursors can for example be used when moving or copying data. This is very common for applications running on Microsoft Windows platforms. The ability to create an image of the data or the data type while using drag and drop is a feature that gives the users more visual feedback of the information passed between the applications.

Drag and drop is not restricted to applications exchanging information. It can also be used to transfer data within the same application or in the same picture. In its simplest form it can be used to send data between shapes in the same picture.

18.2 Using drag and drop in ProcSee

There are several steps involved when using the drag and drop feature in ProcSee. First of all the following question needs to be answered. Is the picture or shape a drop target or a drag source, or both? Using ProcSee as a drag source is quite simple, while using ProcSee as a drop target is more complex and involves more steps. For a detailed description of these steps, see section 18.3 *"Drag source"* on page 306 and section 18.4 *"Drop target"* on page 308.

What kind of objects can be drag source or drop target in ProcSee? The answer to this question is quite simple, that is **Picture**, **Window** and any object derived from **Shape**, i.e. **Rectangle**, **Circle**, **Instance**, etc.

The classes **DnDSource** and **DnDTarget** are the objects involved when using ProcSee as drag source or drop target. These objects can not be created in ProcSee the usual way, but are instead return values from and are only valid within the action part of drag and drop dialogue triggers. Using **DnDSource** or **DnDTarget** involves several function calls.

A simpler, but full functional variant, are the functions **dndSimpleDrag(...)** and **dndSimpleDrop(...)**. These functions give access to the same drag and drop protocol but are much easier to use compared to **DnDSource** and **DnDTarget**. The disadvantage when using these functions are that they do not support all the functionality supported by the classes **DnDSource** and **DnDTarget**.

18.3 Drag source

In ProcSee the drag and drop loop can only be started in the action part of the dialogue trigger **dndBegin(int button)**. A pointer to a **DnDSource** object is returned either from the **dndBegin(...)** dialogue trigger or from the function **dndSource()** and is valid only in the action part of this trigger.

As explained in section 18.2 *"Using drag and drop in ProcSee"* on page 306 there are two different ways to start a drag operation, a simple variant using **dndSimpleDrag(...)** and a more complex variant using the class **DnDSource**. The question is then, when do I need to use the more advanced **DnDSource** class? The answer to this question is quite obvious. First, when you need to add several data types to the drag source. Second, when you want to use a semi transparent image while dragging. Last but not least when you want to control the cursors to use during the drag and drop operation.

Sometimes is it necessary to add several data types to the drag source, for example plain text and some kind of user defined data. Different applications may need different types of data. A text editor for example, like *WordPad* may prefer formatted text (*Rich Text Format*, *RTF*) rather than plain text. *MFC text input field* on the other hand, which is a much simpler editor, does not recognize the *RTF* format. If formatting of text is essential and *WordPad* and *MFC text input field* are supported by the drag source then both *RTF* and plain text should be added. Note that it is always up to the target application to request the most suitable data supported by the drag source.

How is it possible to add several data types to the drag source? The answer to this question is obvious. Just call the object **DnDSource** method **setData(...)** several times with different data and type names. The **setData(...)** function takes two arguments. The first argument is the data itself and the last argument is the type name. The last argument can be omitted, meaning that the default type name for the data to be added is used. However, only character strings are supported in current version of ProcSee. On Microsoft Windows the default type for a character string is CF_TEXT. When using the **DnDSource** class the data must be added before the drag and drop loop is entered.

EXAMPLE

```
{
   DnDSource* dndS = dndSource();
   …
   dndS->setData( "RECT 10, 10, 100, 100", "SHAPE_INFO" );
   dndS->setData( "Plain Text" );
    …
}
```

The drag source controls the operations available during a drag and drop session. In ProcSee the default is *copy*, but it can be changed to *copy*, *move*, *link* or a combination of these. The operation is controlled by the *action* argument in the function **dndSimpleDrag(...)** or the **DnDSource** method **drag(...)**.

The following guidelines should be followed when using drag and drop in ProcSee. This applies to both Microsoft Windows and to X-Windows on linux and unix platforms. If the control key is down a *copy* operation is assumed. If the modifier keys shift and control are down a *link* operation is assumed. A *move* operation is assumed if no modifier keys are pressed.

Still, the actual operation chosen depends on both the source and the destination. The applications that are controlling the source and the destination may have specified that only some of the possible operations are allowed. If several actions have been added for a drag and drop operation, the end user can change it at any time during the drag by pressing appropriate modifier keys. The function **dndSimpleDrag(...)** and the **DnDSource** method **drag(...)** return the status of the drag and drop operation (The status flags are defined in the enumeration **DnDFlag**). The data at the drag source must be deleted when a *move* operation has been completed.

Instead of using default cursors, individual cursors can be set for different drag operations. The **DnDSource** method **setCursor(...)** is used to set the cursor. Different cursors can be set for the operations *move*, *copy* or *link*. This function must be called before the drag loop is entered.

A semi transparent bitmap used during the drag operation can be set on the drag source before the drag and drop loop is entered. A shape in a ProcSee picture or an arbitrary bitmap loaded from file can be used as an image. Use the **DnDSource** method **setBitmap(...)** to set the bitmap.

The drag and drop loop is entered either by calling **dndSimpleDrag(...)** or the **DnDSource** object method **drag(...)**. When the drag operation starts a modal loop is entered which will not end till the data is dropped or the drag operation is cancelled. The drag and drop operation can be cancelled either by pressing the escape key or dropping the data in a non active drop zone. The return values from these functions are the result of the drag and drop operation. If the drag and drop operation was cancelled a value less than zero is returned, otherwise a value from the enumeration **DnDFlags**.

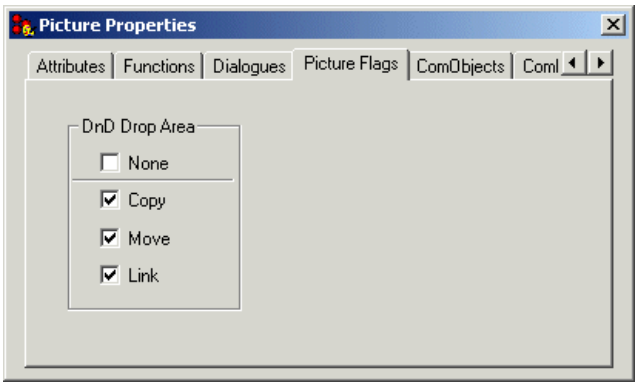
18.4 Drop target

The drop target handles the receiving portion of the drag and drop operation. Before a **Picture**, **Window** or a shape derived from the class **Shape** can be used as a drop target, it must be activated as an active drop target. This can be done either from the graphics editor GED or using the functions **dndActivate(...)**, **setShapeFlags(...)**, **setPictureFlags(...)** or **setWindowFlags(...)**. For more information about these functions, see the ProcSee reference manual. When de-

signing pictures in ProcSee drop targets will normally be activated using the built-in functionality in GED. In this user's guide only GED will be used to activate drop zones.

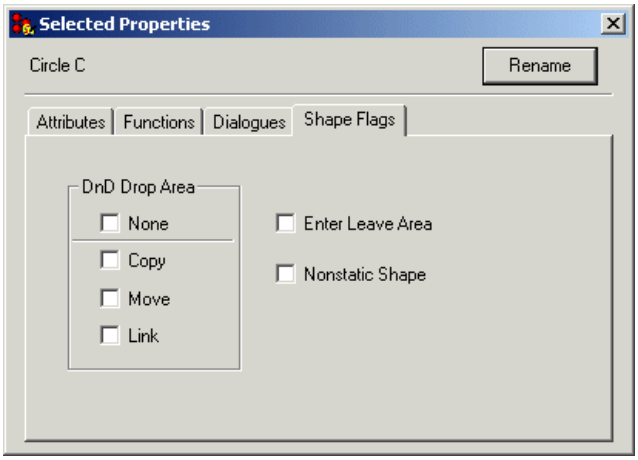
The question is now, how can we from the graphics editor GED activate a drop zone? The answer to this question is from the selected properties dialogue box and the picture/class properties dialogue box. To activate a picture as a drop zone, select the tab *"Picture Flags"* in the picture properties dialogue box. From this tab select the drop target operations supported by the picture, i.e. *none*, *copy*, *move*, *link* or a combination, see Figure 178.

Figure 178
This figure displays the Picture Flags tab available in the Picture Properties dialogue box. From this dialog the picture can be activated as a drop zone for drag and drop operations.



To activate a shape as a drop target, select the *"Shape Flags"* tab in the selected properties dialogue box. To activate a shape as a drop target select *none*, *copy*, *move*, *link* or a combination, see Figure 179.

Figure 179
This figure displays the Shape Flags tab available in the Selected Properties dialogue box. A shape can be activated as a drop zone for drag and drop operations from this dialogue.



ProcSee supports a simple function for accepting the drop, **dndSimpleDrop(...)** and a more complex variant using methods of the class **DnDTarget**. The question is then, when do I need to use the more advanced **DnDTarget** class? The answer to this question is when you need to check if the data supported by the source can be accepted by the target. The cursor can for example be changed if the target does not recognize the data. To change the cursor the drop effect must be changed using the **DnDTarget** class **setDropEffect(...)**, see next paragraph for a detailed description. It is also possible in the

target dialogue to check which mouse buttons are down and which modifier keys are down. This kind of functionality are not available in the function **dndSimpleDrop(...)**.

The drop target can change the cursor used during a drag and drop operation. Use the **DnDTarget** method **setDropEffect(...)** to change the cursor. Legal drop effects are *copy*, *move*, *link* or *none*. If *none* is used it indicates that this is not a legal drop zone for the data supported by the drag source. It is not possible to set the drop effect to a type not supported by the source. For example, if the source has set the drop operations to *"copy | link"* you can not change the drop effect to *"move"*. Before changing the drop effect use the **DnDTarget** methods **isCopy()**, **isMove()** or **isLink()** to check the drop operations supported by the drag source.

To get an object of type **DnDTarget** use the dialogue triggers **dndDrop()**, **dndEnter()**, **dndLeave()** or **dndMotion()**. These dialogue functions plus the function **dndTarget()** return a pointer to a **DnDTarget** object. Notice that objects of the class **DnDTarget** can be acquired and are valid only in the body of these dialogue triggers.

To get the data from a drag and drop operation use the function **dndSimpleDrop(...)** or the **DnDTarget** method **getData(...)**. The argument *data* is a pointer to the variable and the argument *typeName* is the type of the data. For more information about type names, see section 18.3 *"Drag source"* on page 306. The source code example below illustrates how to get the data when the target accepts both *"SHAPE_INFO"* and plain text as type name.

EXAMPLE

```
dndDrop()
{
   DnDTarget* dndT = dndTarget();

   char* value;
   if ( dndT->getData( &value, "SHAPE_INFO" ) )
       do-something-with-the-shape-info-data
   else if ( dndT->getData( &value ) )
       do-something-with-the-plain-text
   else
   {
       dndT->setDropEffect( "none" ); // Unsupported data
       return;
   }
   dndT->setDropEffect( "copy" );     // Copied the data
}
```

Notice that it is possible to return a value from the drop target to the drag source. That was demonstrated in the example above with the **DnDTarget** method **setDropEffect(...)**. The drop effect can only be returned to the drag source when a drop has been performed, i.e. in the body of the dialogue trigger **dndDrop()**. Legal drop effect val-

ues are *none*, *copy*, *move* and *link*. The default is *copy*. If **dndSimpleDrop(...)** is used then the second argument *dropEffect* is returned to the drag source.

18.5 Simple example

Let us start with a simple example that copies some text from a text shape in ProcSee to a rectangle in the same picture. The rectangle just prints the text to the output window. Start the graphics editor GED and create a new picture, then follow the steps below.

First, create the drag source. Add a text shape in the picture and call the shape *T*. To be able to type in some text add the following dialogues. The text entered will be used during the drag and drop operation.

EXAMPLE

```
buttonDoubleClicked(1)
{
   edit( 1, 200 );
}
```

This dialogue will set the text in input mode when it is being double clicked using the left mouse button. The next dialogue is fired when the return key is pressed. It copies the edit buffer into the text shape and terminates the text input mode.

EXAMPLE

```
keyPressed(0xFF0D)
{
   theText = buffer();
   edit( 0 );
}
```

Now, we have added functionality for entering text into the text shape. Next, we need to add the dialogue **dndBegin(0)**, which will start the drag and drop operation.

EXAMPLE

```
dndBegin(0)
{
   dndSimpleDrag( theText );
}
```

The function **dndSimpleDrag(...)** adds the text entered in the text shape to the drag source and starts the modal drag and drop loop. This function will not terminate till either the escape key is pressed to cancel the operation or the data is dropped somewhere on the display.

Now, we are finished with the drag source. Let us continue with the drop target. Create a new rectangle shape in the picture. Open the selected properties dialogue box and select the *"Shape Flags"* tab. From this tab select *copy*. The rectangle shape has now become an active target for a drag and drop operation.

Add the dialogue **dndDrop()** and add the following code. This code will display the data dropped in the rectangle in the output window.

EXAMPLE

```
dndDrop()
{
   char* data;
   if ( dndSimpleDrop( &data ) )
      printf( "Data = \"%s\"", data );
}
```

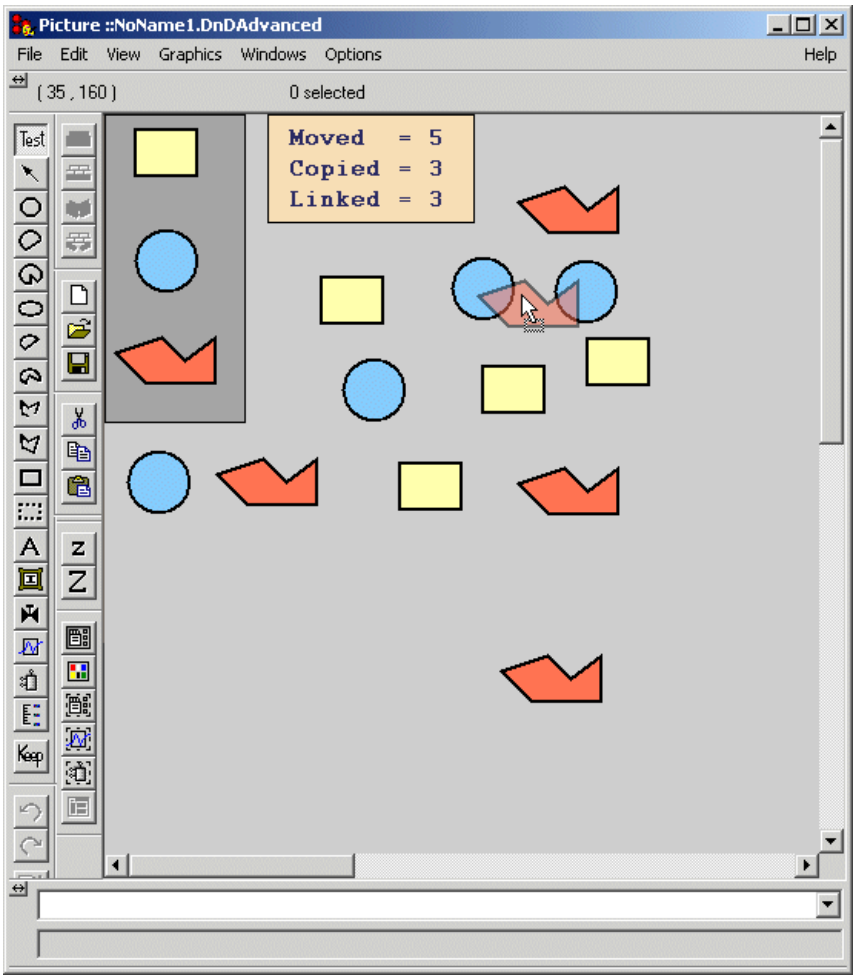
This example is completed. Now, test the drag and drop functionality you have just added. First, double click with the left mouse button on the text shape. Type in some text and press the return key. Start the move operation by pressing one of the mouse buttons on the text shape and move the data into the rectangle. Release the mouse button when the mouse is above the rectangle. Open the output window and you will see that the text you entered is displayed in the window.

18.6 Advanced example

In this example a ProcSee picture will be created where three different types of shapes can be copied using drag and drop functionality. The picture containing the shapes is an active drop zone for the drag and drop operation. New shapes equal in size and shape will be created at the position where they are dropped. In this example the classes **DnDSource** and **DnDTarget** are used. It also demonstrates the use of type names. The type names are used in the **dndDrop()** dialogue to determine what kind of shape to create. See Figure 180 for a view of the final picture. Notice that this is not a complete example.

Let us start to look at the source code which starts the drag and drop operation. The source code below illustrates the drag source for the rectangle shape. There are a few things in this source code which needs an explanation. First, the argument to the method **setBit-**

Figure 180
This figure displays the drag and drop example developed in this section. The drag sources are located in upper left corner. Information about the number of moves, copies and links which have been completed are displayed to the right.



map(...) may seem a little bit awkward. It uses the @ operator and the pTALK function **fullName()** to create a pointer. What is the result of this expression? Well, the result is a **Shape** pointer pointing to the rectangle itself. The last argument to **setBitmap(...)** is the transparency colour used by the image. The same colour is used as the picture's background colour.

There are two calls to the method **setData(...)**. The first call is recognized by the drop target that will be created later in this example. The first argument to this method is information about the rectangle shape, i.e. width, height, colour and line width. The next argument is the type name *"RECT_INFO"* which indicate that the data contains rectangle information. The next **setData(...)** call is just for oth-

er drop zones accepting plain text, like a word processor etc. Open the selected properties dialogue box and add the following dialogue for the rectangle shape.

EXAMPLE

```
dndBegin(0)
{
   DnDSource* dndS = dndSource();

   dndS->setBitmap( (Shape@fullName()), 0, 0, grey80 );
   dndS->setData( "40|30|lightYellow|2", "RECT_INFO" );
   dndS->setData( "RECTANGLE" );

   int result = dndS->drag( "copy | move | link" );
   Dropped( result );
}
```

The result of the drag and drop operation is returned in the attribute *result*. This attribute is an argument to the user defined pTALK function *Dropped(…)*. This function has functionality for checking what kind of drag and drop operation that has been completed. It uses the predefined enumeration **DnDFlags** to check the type.

EXAMPLE

```
void Dropped( int result )
{
   if ( result <= 0 ) return;

   if ( result & DnDFlags.move ) …;
   if ( result & DnDFlags.copy ) …;
   if ( result & DnDFlags.link ) …;
}
```

The same dialogue triggers as illustrated in the source code above are added for the circle shape and the polygon shape. The only difference is in the arguments to the method **setData(…)**.

EXAMPLE

```
…
  dndS->setData( "20|lightSkyBlue|2", "CIRCLE_INFO" );
  dndS->setData( "CIRCLE" );
…
```

This code is used by the circle shape. The type name *RECT_INFO* is replaced with *CIRCLE_INFO* and the data is circle information, i.e. radius, colour and line width. The source code shown below is used by the polygon shape. The data contains five polygon points, plus colour and line width.

EXAMPLE

```
…
    dndS->setData( "-30|10|-10|30|35|30|35|0|15|15|coral|2",
                   "POLYGON_INFO" );
    dndS->setData( "POLYGON" );
…
```

Now, we are finished with the drag source. The next step is to create the drop target. In the example we develop, the picture will be a drop target. The picture must first be activated as an active drop zone. Open the picture properties dialogue box and select the *"Picture Flags"* tab. Select the check boxes *copy*, *move* and *link*, see Figure 178.

The picture has now become a drop target for the drag source. The next step is to create the dialogue trigger **dndDrop()** as shown in the source code below. This dialogue will be fired when the drag and drop object is dropped in the picture.

EXAMPLE

```
dndDrop()
{
    DnDTarget* dndT = dndTarget();

    char* data;
    if ( dndT->getData( &data, "RECT_INFO" ) )
       CreateRect( data );
    else if ( dndT->getData( &data, "CIRCLE_INFO" ) )
       CreateCircle( data );
    else if ( dndT->getData( &data, "POLYGON_INFO" ) )
       CreatePolygon( data );
    else
       dndT->setDropEffect( "none" );
}
```

This source code illustrates the use of type names. It accepts type names of type *RECT_INFO*, *CIRCLE_INFO* and *POLYGON_INFO*. This is the same type names which where specified in the **dndBegin(…)** dialogue triggers created for the drag source shapes. The method **getData(…)** returns false if the type name passed as argument is not supported by the drag source. In this example different shapes can be created based on the type names. The source code for the *CreateRect(…)* function is illustrated below. The only thing to point out in this code is that the pTALK function **strfield(…)** is used

to get the shape properties. The functions *CreateCircle(…)* and *CreatePolygon(…)* are analogues to *CreateRect(…)* and will therefore not be illustrated in this example.

EXAMPLE

```
void CreateRect( char* rInfo )
{
   float w = atof( strfield( rInfo, 0, '|' ) );
   float h = atof( strfield( rInfo, 1, '|' ) );
   Rectangle* r = newRectangle( fullName(), 0,
                                cursorX(), cursorY(), w, h );
   r->foregroundColour      = black;
   r->lineWidth             = atof( strfield( rInfo, 3, '|' ) );
   r->foregroundFillColour := strfield( rInfo, 2, '|' );
}
```

To test the drag and drop feature click with the mouse on one of the three shapes, keep the mouse button down and move the mouse. While you are dragging, a semi transparent bitmap of the shape will be displayed. Move the mouse to the position where you want to put the shape and release the button. An exact copy of the shape will be created at the cursor position where you released the mouse.

18.7 Drag and drop on X-Windows

Drag and drop is not implemented in the X-Windows versions of ProcSee. One reason for this is the fact that there are no standard implementation of the drag and drop protocol on X-Windows. However, two commonly used drag and drop protocols have been investigated and one or both of these will probably be implemented in a future release of ProcSee.

PART 3

Developing Applications

19 Application Code Basics

This chapter deals with the basic mechanisms used to connect application code to the ProcSee Run-Time Manager.

19.1 A Small Example

The simplest application program code required to connect to ProcSee is shown in the example in this section.

It consists of include directives for header files, a pair of 'callback' routines, an initialisation function and a main loop. These are described below.

header files

The directive

```
#include <api/api.h>
```

includes the C header file **api.h**. This file is part of ProcSee, and contains data types, constants, and function prototypes. They must be included wherever the API is used.

connection callbacks

The C functions ***connectCB()*** and ***disconnectCB()*** are connection callbacks. They are user supplied functions that will be called by the API to inform the application code about changes in the connection with ProcSee.

The ***connectCB()*** function is called whenever the program establishes contact with the Run-Time Manager (RTM). This situation occurs when the RTM starts or when the program is first started.

The routine ***disconnectCB()*** is called whenever the program loses contact with the RTM. This situation occurs when the Run-Time Manager stops or crashes. This routine should be used to define appropriate actions for coping with these situations.

EXAMPLE

```
/* C */

#include <api/api.h>


void connectCB( status, rtmName )
int32 status;
char* rtmName;
{
   printf( "Connection with RTM <%s> established \n", rtmName );
}


void disconnectCB( status, rtmName )
int32 status;
char* rtmName;
{
   printf( "Connection with RTM <%s> lost \n", rtmName );
}


main()
{
   PfInitialize( "A", "P", "rtm", NULL, 0, 0, connectCB, disconnectCB );
   PfMainLoop();
}
```

The *status* and *rtmName* parameters can be used to distinguish between the different situations and different RTMs in case the program is connected to several RTMs (see chapter 22 *"Extended API features"* on page 349).

PfInitialize()

The API function ***PfInitialize()*** is called to connect to the Run-Time Manager. If the RTM is not running, the API will periodically attempt to connect until connection is established. ***PfInitialize()*** returns error and prints a message if it fails to register with the Control Server.

The routine takes several arguments. In the example, ***PfInitialize()*** is used to connect to an instance of the RTM called **"rtm"**. The process presents itself under the name **"P"**, and it tries to make contact with an application context, within the RTM called **"A"**.

PfMainLoop()

The ***PfMainLoop()*** routine is the heart of almost every program that communicates with ProcSee. Most of the execution time will be spent within this routine, which is responsible for calling the user defined connection callbacks (in this case: ***connectCB()*** and ***disconnectCB()***) and communicating with the RTM.

The functionality of the most frequently used API routines will be explained in more detail in later sections. The next section describes how to compile and link a typical application program.

19.2 Compiling and Linking your Program

When the program code has been completed, it needs to be compiled and linked before it can be run. The actual commands and their arguments vary between different machines and operating systems, and for information on this, please read *"Part3 - Manual pages - Application Programmers Interface Manual pages - Compiling and Linking"* in the ProcSee *Reference Manual*.

19.3 Handshaking with the Run-Time Manager

The API has the sole responsibility for carrying out the handshaking procedures with the ProcSee Run-Time Manager. This includes both connection and disconnection operations. When the API is referred to in the following subsections, the API function ***PfMainLoop()*** is really being referred to.

Connecting

The API can be in several states, or stages, in its relation to the control server and the Run-Time Manager.

Control problems

The initial state, in some cases, might be the situation where ***PfInitialize()*** detects that the application program can not register with the Control Server. This might happen if no Control Server is running or if another process with the same name is already registered with the Control Server. In such cases ***PfInitialize()*** prints an error message and returns a suitable error code. As no contact to other processes can be established without the Control Server running, the Control Server should be started once and never be terminated.

RTM not running

If ***PfInitialize()*** succeeded in registering with the Control Server, the API will check for the presence of a Run-Time Manager with the name *rtmName*. If it is available, ***connectCB()*** will be called and the connection is established. If the Run-Time Manager is not running, the API will start to poll to see if the RTM has become available.

RTM starts

When the Run-Time Manager starts the API will respond by performing the necessary handshaking with the Run-Time Manager and eventually call the ***connectCB()*** function. The connection is then established.

The value of the *status* argument to ***connectCB()*** is always **PfCrtmStart**.

Disconnecting

The application program might loose contact with the Run-Time Manager for several reasons. Each case must be handled.

RTM stops

If the Run-Time Manager stops, the API will immediately be informed of this event by the SWBus, and will respond by calling ***disconnectCB()***. If the RTM was stopped gracefully, the *status* argument will have the value **PfCrtmStop**. If the RTM crashed, the *status* argument will have the value **PfCrtmStop | PfCrtmCrash**.

The API is then back at the second stage in the connection process, where it has registered with the Control Server but has no contact with the RTM. When the Run-Time Manager has been restarted, the user defined connect callback routine will be called with a **PfCrtmStart** *status* argument.

19.4 The Main Loop

A ProcSee application program is not the kind of program that initializes, performs a series of sequential tasks, and then exits. It is *event driven*.

An event-driven program performs tasks as a result of events that occurs. Events may be generated internally from within a program, or externally as messages sent to it. It responds to each event according to a description that is partly predefined and partly user defined.

In the previous section, examples of ***PfMainLoop()***'s predefined operations were given, specifically the operations carried out to connect to and disconnect from the Run-Time Manager. This section focuses on some common user defined operations.

Periodic Actions

PfMainLoop() can be instructed to perform certain actions at predefined intervals. This is done by calling ***PfSetProcessHandler()***. In the example below, ***PfMainLoop()*** is instructed to call the function ***periodicCB1()*** every 5 seconds, and the function ***periodicCB2()*** once a second.

The function ***PfSetProcessHandler()*** takes a pointer to a function and an interval given in milliseconds, and returns a unique identifier given to this periodic action. Several periodic actions can be active simultaneously, and will be called by ***PfMainLoop()*** at their predefined times. The periodic callback function must return the constant **OK**, otherwise the periodic action will be cancelled. The function ***periodicCB2()*** in the example cancels itself the 10th time it is called.

A periodic action can also be cancelled by calling ***PfRemoveProcessHandler()***. This routine takes the unique identifier for the action, returned by ***PfSetProcessHandler()***, as an argument and cancels the corresponding action.

EXAMPLE

```
/* C */

#include <api/api.h>


int32 periodicCB1( id )
int32 id;
{
   printf( "CB1 is called\n" );
   return OK;
}

int32 periodicCB2( id )
int32 id;
{
   static int counter = 0;

   printf( "CB2 is called\n" );
   if ( ++counter == 10 )
      return !OK;
   else
      return OK;
}

main()
{
   int32 id1, id2;

   PfInitialize( ... );

   id1 = PfSetProcessHandler( periodicCB1, 5000 );
   id2 = PfSetProcessHandler( periodicCB2, 1000 );

   PfMainLoop();
}
```

Listening to other Input Sources

PfAddInput()

The API's ***PfMainLoop()*** is specialized to listen for incoming messages on the channels used by the SWBus. Sometimes it is necessary to communicate on other channels. Therefore, the ***PfMainLoop()*** can be configured, by calling a routine called ***PfAddInput(),*** to listen to channels other than the SWBus channels..

In the example code below, which only works on UNIX platforms, the ***PfMainLoop()*** has been configured to listen to terminal input.

EXAMPLE

```
/* C */

#include <stdio.h>
#include <api/api.h>


int32 kbdHandler( id )
int32 id;
{
   char character;

   read( STDIN_FILENO, &character, 1 );
     :
     :
   return OK;
}


main()
{
   PfInitialize( ... );

   PfAddInput( PfCKeyboardFd, PfCReadCondition, keyboardCB );

   PfMainLoop();
}
```

PfAddInput() takes a UNIX file number as its first argument. The second argument in this case is set to **PfCReadCondition**, which configures ***PfMainLoop()*** to *read* from the specified channel. The last argument is a pointer to a callback routine that will be called whenever there is something to read from the channel.

A channel in UNIX can be described as either a *socket (file) number* or a file *descriptor mask*. The file number is simply an integer identifying one channel, while a file descriptor mask is a bit-mask identifying several channels. If a channel has the file number `3`, then the corresponding mask is `1<<3`, i.e. `8`. The mask `10` represents two file numbers, namely `1<<3` (`8`) and `1<<1` (`2`). Refer to the description of ***select(2)*** in the UNIX man(ual) pages for a more detailed explanation of file descriptor masks.

Note that both ***PfAddInput()*** and ***PfSetProcessHandler()*** can be called before ***PfMainLoop()*** is entered. These routines are the only API functions that can be called between ***PfInitialize()*** and ***PfMainLoop()***, because they do not rely on an established connection with a Run-Time Manager.

19.5 Summary

It might appear strange that an application program that spends most of its execution time within a library routine, can do anything useful. The programmer might think that when ***PfMainLoop()*** is called, he or she loses control of the program execution, but this is not the case.

The programmer's control lies in instructing the main loop what to do in certain situations. The programmer can tell it to call his or her routines at user defined intervals. He or she can also instruct it to call user defined routines whenever a message arrives from an external source. The main loop routine uses these descriptions to carry out its tasks.

In the following chapters, it will become even clearer that the control lies, as it should, in the hands of the programmer.

20 Variables

Variables are central to ProcSee because they connect application data to the user interface. Variables are owned by the application process and are made available to the Run-Time Manager for use in pictures.

20.1 Variables are owned by Processes

Bear in mind whenever dealing with variables that *they are owned by application programs (processes) and not by ProcSee*.

Variables are an application's data. An application program is normally written in C or C++ and is a complete program, without user interface code.

20.2 Creating a Variable

A variable is created by calling the API function ***PfCreateVar()***. ProcSee needs to know the *name*, *type*, and *value* of the variable. Below is an example where a program creates an integer variable '**Temperature**'.

Note the following facts:

- The call to ***PfCreateVar()*** is placed in the connect callback function, so it takes place only when the program has established contact with the Run-Time Manager.
- ***PfCreateVar()*** takes the name[1] of the variable as first argument, which is the name by which the variable is known in ProcSee.

EXAMPLE: CREATING VARIABLES

```
/* C */

#include <api/api.h>

int32 temp_id;
int   temp;


void connectCB( status, rtmName )
int32 status;
char* rtmName;
{
   temp_id = PfCreateVar( "temperature", PfCInt, NULL, 0, &temp );
   :
   PfFlushCreateVar();
}


main()
{
   PfInitialize( ..., connectCB, ... );

   PfMainLoop();
}
```

- The type is given as second argument, in this case **PfCInt**. If the type is **PfCRecord**, the name of the record type is given as third parameter. In this case it is **NULL.**
- The fourth argument is ignored.
- A pointer to the C variable '**temp**' is given as last argument.
- ***PfCreateVar()*** returns an integer, which is a unique index identifying the variable. The id will be used in later API calls to reference the variable.
- ***PfFlushCreateVar()*** has to be called when all variables are created. While ***PfCreateVar()*** merely creates the variable in the API, ***PfFlushCreateVar()*** is responsible for creating the variable in the RTM.

After the call to ***PfFlushCreateVar()***, the Run-Time Manager will be able to access the C variable '**temp**'. Now that the API knows the type and value (given through the pointer **&temp**), it will send the data to the Run-Time Manager together with the name "**temperature**". The variable value can then be accessed in pTALK under the name "**temperature**".

1 The name of the variable can contain any characters, but if the name is a reserved keyword or contains other characters than letters, underscore or digits, or start with a digit, pTALK needs a set of '$'-characters around the name. See "Identifiers" section in the Reference Manual.

The table below lists the data types that are available.

Constant	Description
PfCUnsignedChar	8 bit unsigned character (0-255)
PfCShortInt	16 bit signed integer
PfCUnsignedShortInt	16 bit unsigned integer
PfCInt	32 bit signed integer
PfCUnsignedInt	32 bit unsigned integer
PfCFloat	32 bit floating point
PfCDouble	64 bit floating point
PfCRecord	user defined record type

Creating Arrays

A separate API call ***PfCreateArray()*** is used to create array variables. It works exactly as ***PfCreateVar()***, except that it takes one additional argument, namely the size of the array. The following example would create an array of 10 floating points:

```
PfCreateArray("arr",PfCFloat,10,NULL,0,&arr);
```

20.3 Changing Variable Values

When a C variable changes, the API could take one of two approaches to detect the change:

1) It could periodically scan all variable values and compare them with stored values, to identify any values that have changed.

2) It could rely on the application program to inform the API whenever changes occur.

The ProcSee API uses the second approach, since constantly polling all variable values would be much too expensive. Consequently there are separate API functions for informing the API about changes in variable values. The most frequently used functions for this are ***PfPut()***, ***PfSend()*** and ***PfUpdateValuesId()***. The example below illustrates the use of ***PfUpdateValuesId()*** which takes an array variable identifier as argument.

PfCreateVar()

Let's look at each part in turn, and start with the connect callback function. ***PfCreateVar()*** is used to create three variables, namely **radians**, **cosValue** and **sinValue**. These are the names under which the C variables **rv**, **cv**, and **sv** are made available to the Run-Time Manager. Note that the variables are created only when a working

EXAMPLE: CREATING AND CHANGING VARIABLES

```
/* C */

#include <math.h>
#include <api/api.h>

int32  rv_id, cv_id, sv_id;
float  rv, cv, sv;
int32  indices[3];
int    index;
bool   hasConnection = false;

void connectCB( status, rtmName )
int32 status;
char* rtmName;
{
   rv_id = PfCreateVar( "radians",  PfCFloat, NULL, true, &rv );
   cv_id = PfCreateVar( "cosValue", PfCFloat, NULL, true, &cv );
   sv_id = PfCreateVar( "sinValue", PfCFloat, NULL, true, &sv );
   PfFlushCreateVar();
   hasConnection = true;
}

void disconnectCB( status, rtmName )
int32 status;
char* rtmName;
{
   hasConnection = false;
}

int32 step( id )
int32 id;
{
    index = 0;
    rv += 3.14/30;    indices[index++] = rv_id;
    cv = cos( rv );   indices[index++] = cv_id;
    sv = sin( rv );   indices[index++] = sv_id;

    if ( hasConnection )
    {
       PfUpdateValuesId( indices, index );
       PfFlush();
    }

    return OK;
}

main()
{
   PfInitialize( ..., connectCB, disconnectCB );
   PfSetProcessHandler( &step, 1000 );
   PfMainLoop();
}
```

connection with the Run-Time Manager has been established. ***PfFlushCreateVar()*** is called when all variables are created. A global flag **hasConnection** is set to true to indicate a working connection.

PfSetProcessHandler() The program is a small simulator that produces cosine and sine curves. The **rv** variable is the angle given in radians, and the variables **cv** and **sv** are the current cosine and sine values, respectively. A periodic function is set up to update these values every second.

PfUpdateValuesId() Each second, the radians value is incremented by 1/60th of a circle, and the new cosine and sine values are calculated. These values are put directly into the C variables **rv**, **cv**, and **sv**. At the same time, the identifiers of all changed variables are appended to an array, in this case called **indices**. ***PfUpdateValuesId()*** is called to put the updated values into a buffer for later transfer to the RTM. This function takes as arguments the array of indices together with the number of indices. All information necessary to collect and send the values was established when ***PfCreateVar()*** was called. Note that the values are only sent to ProcSee if there is a working connection.

PfFlush() ***PfFlush()*** transfers the values to the RTM and cause the Run-Time Manager to update the user interface according to the new data values.

20.4 Record Variables and Types

So far variables of predefined simple datatypes have been introduced but it is often necessary to construct complex user defined datatypes, i.e. records or structs. This section looks at how such a data type can be defined, and how to create variables of the type once it has been defined.

There are two ways of defining record types. The simplest and recommended way is reading a database file (.pdat file) by ***PfReadScript()***. However, three dedicated API functions for creating record types are also available. Both ways are described below.

Defining Record Types Using a Database File

This way of defining record types is simple and flexible, and it is the recommended way because it enables the application developer to reuse the database file (.pdat file) from the RTM. Putting the record definitions into a seperate data base file, to be read by the RTM as well as the application program ensures consistency in record definitions. In addition it is easy to compare record definitions from a database file with the C struct definitions in the application program. The example below shows how to define a record type using a database file.

- A separate file, here named SystemBase.pdat, is created according to the ProcSee database file format described in detail in the Reference Manual. This file should be reused in the RTM, by the database definition in the application resource file as shown in the example. This way one can be certain that there are no inconsistencies in record definitions between the API and the RTM.
- The database file should contain the definitions of all record types used by the application. In the example a single record called "**Valve**" is defined. The initial value for the "**State**" field of any variable of type "**Valve**" is set to 1.

EXAMPLE: DEFINING RECORD TYPES USING A DATABASE FILE

```
/* SystemBase.pdat */

struct Valve
{
   int32 State = 1;
   unsigned char Name[28];
};


/* System.Tdoc */

application System
{
   :
   database = "SystemBase.pdat";
   :
}


/* C */

#include <api/api.h>

void connectCB( status, rtmName )
int   status;
char* rtmName;
{
   PfReadScript( "SystemBase.pdat" );
   :
}


main()
{
   PfInitialize( ..., connectCB, ... );

   PfMainLoop();
}
```

- ***PfReadScript()*** should be called from the connect callback function to define the records once a working connection is established.

Defining Record Types Using API Functions

Another way to define record types is by calling the dedicated API functions ***PfBeginRecord()***, ***PfAddField()***, and ***PfEndRecord()***. An example is provided below.

- The record definition is placed in the connect callback function because the record definition is only meaningful when a working connection with the Run-Time Manager exists.
- ***PfBeginRecord()*** simply starts a new record definition, in this case of a record type called "**Valve**".
- ***PfAddField()*** adds a new field to the record type which is given as the first argument. The second argument is the name of the field, "**State**", and the fourth argument is the type of the field,

EXAMPLE: DEFINING RECORD TYPES USING API FUNCTIONS

```
/* C */

#include <api/api.h>


void connectCB( status, msg )
int   status;
char* msg;
{
   if ( status == PfCrtmStart )
   {
      int  initState = 1;
      PfBeginRecord( "Valve" );
      PfAddField( "Valve", "State", 0, PfCInt, NULL, &initState );
      PfAddField( "Valve", "Name", 28, PfCUnsignedChar, NULL, NULL );
      PfEndRecord( "Valve" );
         :
   }
         :
}


main()
{
   PfInitialize( ..., connectCB, ... );

   PfMainLoop();
}
```

PfCInt. The third argument should be `0` if the field is a single value, otherwise it should indicate the number of elements in an array.

- If the type argument has the value **PfCRecord**, the fifth argument should be the name of another record type. In this way it is possible to nest records, i.e. have records containing fields of other record types. The last argument can contain a pointer to an initial value, but this is only applicable in a few cases. Refer to the reference manual for further information.
- The record definition is complete when a call is made to ***PfEndRecord()***.

When ***PfEndRecord()*** is called, the complete record definition is sent to the Run-Time Manager. The RTM stores the definition as a data type that is available globally within the application.

Creating Record Variables

As soon as the record type definition is available to both the API and the Run-Time Manager, the application code can start to create record variables. This is done using ***PfCreateVar()*** in exactly the same manner as for variables of simple types.

In the example, ***PfCreateVar()*** is used to create a variable named "**v10**" of type "**Valve**". Note that the C type **VALVE** is the corresponding C struct. It has a layout similar to the record type "**Valve**"

EXAMPLE: CREATING RECORD VARIABLES

```
/* C */

#include <api/api.h>

typedef struct
{
   int  State;
   char Namme[28];
} VALVE;

VALVE  v10;


void connectCB( status, rtmName )
int32 status;
char* rtmName;
{
   /* Define records using PfReadScript() or PfBeginRecord/PfEndRecord */
   PfBeginRecord( "Valve" );
      :
   PfEndRecord( "Valve" );

   PfCreateVar( "v10", PfCRecord, "Valve", true, &v10 );
      :
   PfFlushCreateVar();
      :
}


main()
{
   PfInitialize( ..., connectCB, ... );

   PfMainLoop();
}
```

declared to ProcSee. Basically, the calls to ***PfReadScript()*** or ***PfBeginRecord()*** etc. are how the application code tells ProcSee about its data types. The calls to ***PfCreateVar()*** are how the application code informs ProcSee about its data, i.e. makes the data available to the Run-Time Manager.

20.5 Changing Variables from pTALK

We have now explained how you can make your application variables available to the Run-Time Manager, and how you inform the API about changes to the values. The variables can be referred to by name in any pTALK expression as part of the user interface logic. But what happens when a variable is *changed* from pTALK? The new value will be sent from the Run-Time Manager to the API. The RTM sends pairs of integer identifiers and values, so the API has enough information to insert the new value into the application code's C variable.

21 Commands and Messages

This chapter describes methods of communication between the Run-Time Manager and application programs using commands and messages. The difference between synchronous and asynchronous message passing is explained.

21.1 Motivation

Commands and messages are needed in several situations; here are some examples:

- From a dialogue in the user interface, a message should be sent to a simulator program to start, freeze, or stop the simulation.
- From a dialogue in the user interface, a database server should be told to load or save some data.
- An alarm program should send a message to ProcSee in order to display an alarm list.
- When a certain event occurs, for example that a picture is displayed, a message should be broadcasted to all programs to inform them about the event.

These are all tasks that can be effectively handled using the mechanisms of commands and messages that are explained in this chapter.

21.2 Synchronous versus Asynchronous

When dealing with commands and messages, it is important that the difference between *synchronous* and *asynchronous* message passing is understood. The examples below use people communicating by letter and phone in order to explain the difference.

Synchronous Message Passing

If a message is passed synchronously, the sender has to wait until the message has been received and accepted. This is similar to giving messages by telephone. The sender has to dial the number, wait until the receiver answers, pass on the message and be sure it's understood. In this way the sender is able to get an immediate answer to the request.

Figure 181

Giving messages by telephone is an example of synchronous message passing.

Synchronous message passing can also be called "on-line communication", and means to

- Send a message and stay tuned.
- Wait until the message is confirmed and an answer is given.

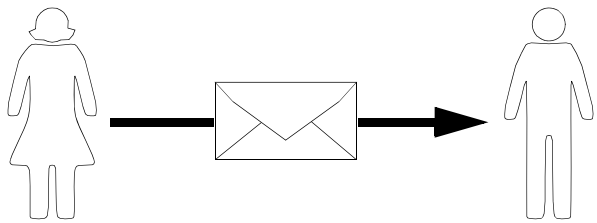
Normally, waiting for answer implies that the application becomes idle, i.e. it does nothing until the answer is received. This is not so in ProcSee. While waiting for an answer from the RTM, the application program enters an internal loop, just like the one used by ***PfMainLoop()***. Thus, while waiting, the periodic functions set up by ***PfSetProcessHandler()*** and callback functions set up by ***PfAddInput()*** are still being called. This way, the API attempts to utilize the CPU to the maximum extent, to ensure steady execution times for the periodic functions and to reduce the likelihood of communication deadlocks.

Asynchronous Message Passing

When a message is passed asynchronously, the sender does not wait for a response. This is similar to sending a letter. The sender posts the letter and then does other things.

Figure 182

Giving messages by letter is an example of asynchronous message passing.



Asynchronous message passing can also be called "off-line communication", and means to

- Send the message and forget about it, at least for a while.
- The receiver reads the message when it is convenient to do so.
- Confirmation must be sent as a separate message, i.e. as a new letter.

Sending a letter is not necessarily a better mode of communication than calling somebody. What is best depends on the situation, and the same is true when messages are to be sent between computer programs. Both synchronous and asynchronous messages are frequently used when two or more computer programs communicate with each other.

21.3 Sending Commands to RTM

Often the application program wants to send commands to the Run-Time Manager, i.e. instruct RTM to execute a certain action. There is a dedicated API call for this, ***PfExecute()***. A command is expressed using pTALK.

SYNOPSIS

```
char* PfExecute( char* scopeName, char* commandString, ... );
```

The argument *commandString* should be a valid pTALK compound statement or expression. Optional arguments can be supplied and used in conjunction with C's 'printf' style `%` formatting options in the command string. The argument *scopeName* indicates the scope in which the command should be executed.

EXAMPLE

```
/* C */

float newSnap;
   :
PfExecute( NULL, "{ canvas.xSnap = %f; }", newSnap );
```

PfExecute() is a synchronous routine, which means that your program waits while the RTM handles the request. Calling ***PfExecute()*** is like making a phone call to the Run-Time Manager to ask for a task to be performed and then waiting until the task's completion has been confirmed.

Using the Run-Time Manager for Calculations

If the command is a valid pTALK expression, then result of that expression is returned as a character string by ***PfExecute()***. Possible uses for this feature include:

- Querying or setting the value of any arithmetic or string object within the Run-Time Manager by application programs.
- Performing interactive or interpreted calls to the Run-Time Manager, using it as a run-time pTALK source code compiler.

Figure 183 shows what happens when ***PfExecute()*** is called to evaluate an expression.

21.4 Sending Commands from the RTM

While ***PfExecute()*** is used to issue a command to the RTM, one is also likely to go the other way, i.e. to send commands from the RTM to the application program. The mechanism used for this is *Remote Function Calls*.

Remote function calls enables the software developer to call C functions in application programs from pTALK.

Declaring a Function to the RTM

A C routine must be declared to the Run-Time Manager before it can be called from pTALK. The API routine ***PfRegisterFunction()*** is used to do that. The following information needs to be given to ProcSee:

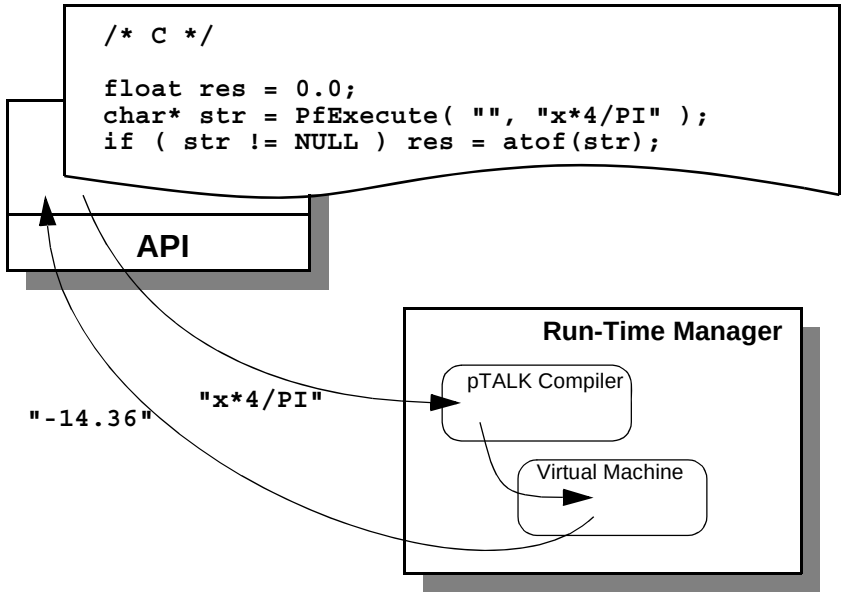
- The name of the function
- A pointer to the C function

Figure 183

*When **PfExecute()** is called, the source code string is sent to RTM.*

RTM compiles the string into binary code that can be executed by the Virtual Machine.

The Virtual Machine executes the code, and the result, if any, is returned as a text string to the program.



- A list of formal arguments

function signature Hence, the complete *signature* of the function (name and formal arguments) is given to ProcSee. This information is passed over to the Run-Time Manager, while the function pointer is stored internally in API.

formal arguments The formal arguments are given as an array of structures containing the type and size of each formal argument. The type code **PfCInt** together with a size of `1` indicates the type `int`. The type code **PfC-**

Float together with a size of **4** indicates the type **`float[4]`**, while the type code **PfCUnsignedChar** together with a size of **0** indicates the type **`char*`**.

SYNOPSIS

```
typedef int32 (*PfTFunction)( int32, void* );

typedef struct
{
   int32 dtype;     /* PfCInt, PfCUnsignedChar, PfCFloat, etc. */
   int32 size;      /* 0 = pointer, 1 = simple, >1 = array     */
} PfTArg;


int32 PfRegisterFunction( char* name, PfTFunction f,
                          int32 argc, PfTArg argv[]);
```

EXAMPLE

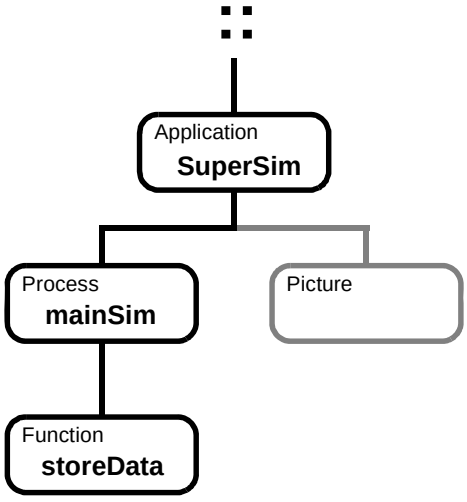
```
/* C */

/*  Declare the function
 *        void storeData( int, float[4], char* );
 */

int32 store( int numberOfArguments, void* arguments ) { ... }

   :
PfTArg formals[] = { {PfCInt,1}, {PfCFloat,4}, {PfCUnsignedChar,0} };
   :
PfRegisterFunction( "storeData", store, 3, formals );
```

The function becomes a member of the process that declares it. The symbol table in the figure below, shows that the function is a child of the process.

Figure 184

A C function that is declared to the Run-Time Manager, becomes a member (child) of the process that declared it.

In this example, storeData is a function and a member of the process mainSim. The process is part of the application SuperSim.



Remotely Calling a C Function

When a C function is available to the Run-Time Manager, it can be called from pTALK as if it were written in pTALK itself.

Unpacking Actual Arguments

Since C is a compiled language, it is not possible to remotely call a function with a variable number of arguments. All remotely declared functions must have the same *physical* signature although they might have different *logical* signatures.

logical signature The logical signature is the signature of the function as the RTM sees it, for example as **`void storeData(int,float[4], char*)`**. The Run-Time Manager sees **`storeData`** as a void function taking an int, an array of 4 floating point numbers and a character string.

physical signature The physical signature must be **`int32 store(int32,void*)`** for all functions that are to be called remotely. The first physical argument holds the actual number of logical arguments passed, and the second physical argument is a pointer to a data block containing all the logical arguments. The data block holds each argument's actual size and type in addition to the value itself. The API function ***PfGetFuncArg()*** helps retrieving the actual data.

The difference between logical and physical arguments might seem confusing initially, but the example in Figure 185 should make things clearer.

21.5 Broadcasting Messages

PfExecute() and remote function calls are used to issue commands to a single receiver, and the program that issues the command also decides who is to receive it. *Broadcast messages* are different from this in two respects:

- The receiver itself decides what kinds of messages it will receive. This is done by *subscribing* to one or more message *categories*.
- The receiver also decides the *scopes* (parts of the user interface hierarchy) from which to receive messages.
- A message is sent to all subscribers of that message's category and scope.

A useful analogy for this is radio broadcasting. Scopes are like radio stations, while different categories are like radio shows.

Figure 185

When calling a remote function, the actual arguments are passed from the Run-Time Manager to API.

The C routine is then called with two physical arguments, the first holding the number of logical arguments, and the second pointing to a data block holding the logical arguments.

The logical arguments are extracted in the C routine by calling PfGetFuncArg. PfGetFuncArg returns a pointer to the data block of the argument , increments the data pointer to be positioned for the next argument and delivers the actual type and size in its two last arguments.

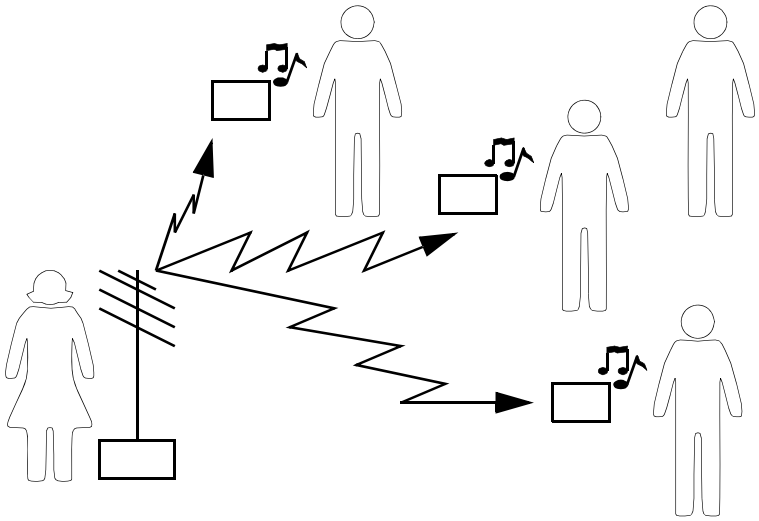
```
// pTALK
   :
mainSim.storeData( 666, v, "String" );
   :
```

```
/* C */

int32 store( int32 numArgs, void* data )
{
   int32 size, type;
   void* theData;
   int32 intVal;
   float floatVals[4];
   char  stringVal[32];
   int   i;

   if ( numArgs != 3 )
      return !OK;

   /* Extract first argument */
   theData = PfGetFuncArg( &data, &type, &size );
   if ( type != PfCInt || size != 1 )
      return !OK;
   intVal = *(int32*)theData;

   /* Extract second argument */
   theData = PfGetFuncArg( &data, &type, &size );
   if ( type != PfCFloat || size != 4 )
      return !OK;
   for ( i = 0; i < 4; i++ )
      floatVals[i] = ((float*)theData)[i];

   /* Extract third argument */
   theData = PfGetFuncArg( &data, &type, &size );
   if ( type != PfCUnsignedChar && size > 32 )
      return !OK;
   strcpy( stringVal, (char*)theData );

   :
   return OK;
}
    :
    :

PfTArg formals[] =
{
   {PfCInt,1},
   {PfCFloat,4}
   {PfCUnsignedChar,0}
};

PfRegisterFunction("storeData",store,3,formals);
```

Figure 186

Providing information using radio transmission is an example of asynchronous message passing using broadcasting.

The receiver can choose from different channels and programs.



A ProcSee broadcast message has three characteristics:

Category 31 different message categories are available, 16 of which are predefined, and 15 of which are user defined.

Scope All broadcast messages come from somewhere within the user interface hierarchy. The scope is the full name of the component that initiated the message.

Text The message itself is a text string.

Message Subscription

The API routine ***PfDiagnosticsSubscribe()*** is used to subscribe to different message categories.

SYNOPSIS

```
typedef int32 (*PfTCallback)( int32, void* );

int32 PfDiagnosticsSubscribe( char* scope, int32 categories,
                              PfTCallback f );
```

The argument *scope* indicates the part of the user interface for which the subscription is valid. The argument *categories* is a 32 bit signed bitmask that indicates which categories the subscription involves. If

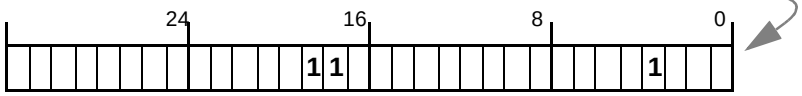
bit number **n** is set, then message category **n** is involved. The last argument *f* is the C function that should be called whenever a message satisfying the required categories and scope is received.

cancel subscription If either the argument *categories* is negative, or the function pointer *f* is **NULL**, then the subscription to those categories will be cancelled.

It is possible to broaden or narrow an existing subscription, and only one subscription will be registered for a given scope. Subsequent calls to ***PfDiagnosticsSubscribe()*** for the same scope change the existing subscription rather than create a new one. The new subscription will be made broader if it is given a positive category mask and narrower if it is given a negative category mask or a **NULL** callback function.

EXAMPLE

```
/* C */

int32 handler( int32 requestId, void* msg ) { ... }

    :
    :
int32 mask = (1<<3) | (1<<17) | (1<<18);

PfDiagnosticsSubscribe( "::SuperSim", mask, &handler );
```




```
/* Unsubscription, alternative I, negative category mask */
PfDiagnosticsSubscribe( "::SuperSim", -mask, &handler );

/* Unsubscription, alternative II, NULL pointer callback */
PfDiagnosticsSubscribe( "::SuperSim", mask, NULL );
```

Receiving a Message

When a message is received by an application program, the callback routine for that category of message is activated. The example below shows a broadcast message callback routine that prints out the message category, text, and scope. The first argument *requestId* is

not used, while the second argument *msg* contains the message. The function ***PfGetDiagnosticsMessage()*** has been provided to unpack the category, scope, and message text, respectively.

EXAMPLE

```
/* C */

int32 MyMessageLogger( int32 requestId, void* msg )
{
   int32  category;
   char*  scope;
   char*  message;
   PfGetDiagnosticsMessage( msg, &category, &scope, &message );
   printf( "Category : %d\n", category );
   printf( "Scope    : %s\n", scope    );
   printf( "Text     : %s\n", message  );

   return OK;
}
```

A callback routine must return the value **OK** after processing a message. If any other value is returned, then it is interpreted as a request to cancel the corresponding subscription, and no more messages will be received.

Broadcasting a Message

As mentioned earlier, some message categories (0-15) are predefined and will be broadcasted automatically by the Run-Time Manager. The categories 16-30 can be used freely and be given any meaning by the application developer. A message can be broadcasted using the pTALK routine ***message()***.

SYNOPSIS

```
void  message( int category, char* messageText, ... );
```

The *category* argument gives the category in the form `1<<n`, where `n` is the category, and the *messageText* argument is a character string containing the message itself. For convenience ***message()*** accepts additional arguments in a manner similar to the C function ***printf()***, enabling the application developer to construct message text using `%` formatting options without using the C function ***sprintf()***.

implicit scope

It is not necessary to explicitly give the scope as an argument as it is automatically derived as the scope from which the ***message()*** routine is called.

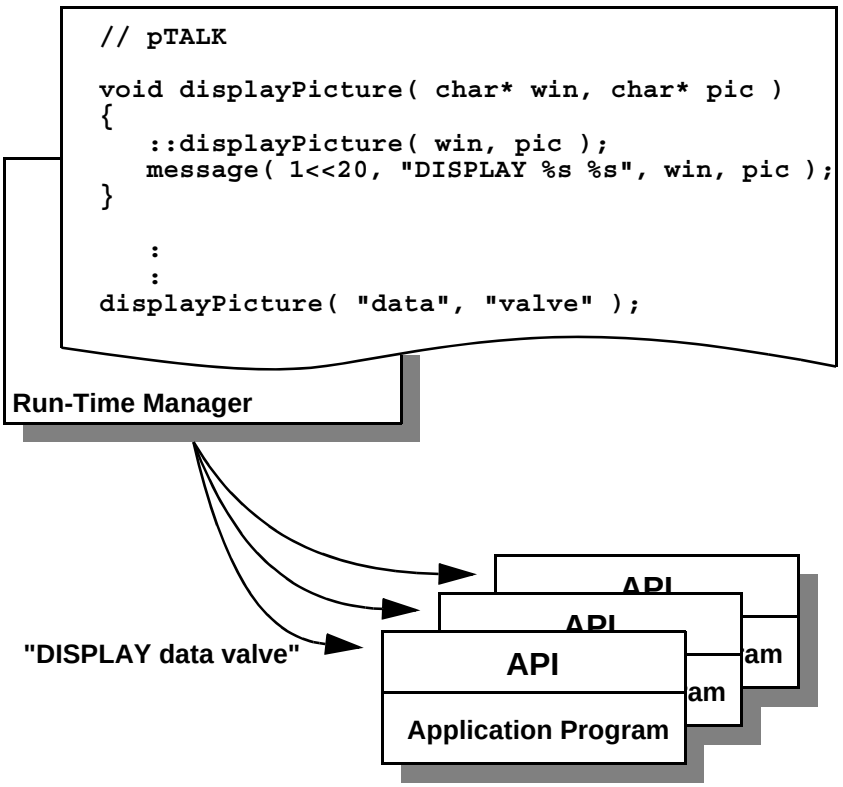
Figure 187 shows how to overload (change the behaviour of) the standard pTALK function ***displayPicture()***, to make it also broadcast a message about what it is doing.

Figure 187

The behaviour of the standard ***pTALK*** *function* ***displayPicture()*** *is overloaded (redefined) to also broadcast a message about which picture is displayed in which window.*

Message category 20 is chosen for this kind of messages in this example, and the message text starts with the keyword DISPLAY to further identify the message type.

Note how the redefined function calls the original function by applying the **::** *scope operator.*



Message() is an asynchronous routine, which means that messages are queued up internally in the Run-Time Manager and are then passed on to the application programs at some later point in time. In practice, messages are broadcasted when RTM becomes idle, i.e. when no events are waiting to be handled.

Message Categories

The table below shows the predefined message categories. C constants for these categories can be found in the header file **<P3Misc.h>**.

category	mask	C constant	description
0	0x1	PfCCompilationErrors	Error message from pTALK compiler
1	0x2	PfCCompilationWarnings	Warning message from pTALK compiler

21.6 Remote Function Call versus Broadcasting

Both remote function calls and message broadcasting are used to direct information from the Run-Time Manager to application programs. Each mechanism has its own distinct characteristics. Below is a table that compares the two mechanisms. It can be used to select the most appropriate method for a task.

Remote Function Call	Message Broadcasting
Only one programs receives the call	Zero, one, or many programs receive the message
The caller decides who is the receiver	Each program decides which messages to receive.
Any number of arguments of any type can be passed	Predefined arguments *category*, *scope* and *text* are passed
Synchronous	Asynchronous

22 Extended API features

This chapter describes methods for how to connect to more than one RTM and how to integrate OSF/Motif applications with the ProcSee Run-Time Manager.

22.1 Communicating with several RTMs

Establishing connections

To make an application process communicate with several RTMs, ***PfInitialize()*** must be called once for each RTM. The rtmName argument to ***PfInitialize()*** must match the name of the RTMs as specified by the -n option when the RTMs are started.

The example below shows how an application process is connected to two RTMs, an RTM called "Logger" working as an external trend logger and an RTM called "Rtm1" displaying pictures.

```
main()
{
   :
   PfInitialize("System",NULL,"Logger",NULL,0,0,connectCB,disconnectCB);
   PfInitialize("System",NULL,"Rtm1",  NULL,0,0,connectCB,disconnectCB);
   :
   PfMainLoop();
   :
   PfClose();
}
```

This code assumes that the two RTMs are started with the -n option as shown below.

```
% rtm -n Logger -r Logger.pctx

% rtm -n Rtm1   -r System.pctx
```

Note that the *applName* and *processName* arguments to ***PfInitialize()*** must be identical for all RTMs involved in a system. One can not connect to an application "A" in rtm "Rtm1" and to application "B" in rtm "Logger" at the same time. In order to produce resource files (.pctx-files) for applications in different RTMs, the -f option to pcc is very useful. See the man pages for pcc.

In the example above, the ***connectCB()*** and ***disconnectCB()*** functions were used for both RTMs, but one could have used one function for the trendlogger and a different function for the display RTM. However, those functions usually creates records and variables which shall be available in both RTMs, so using the same functions are natural. The second argument to ***connectCB()*** and ***disconnectCB()*** is a character string containing the name of the actual RTM to which connection is established or lost, so the routines can take appropriate action depending on the rtmName.

By the way, there is no requirement to establish connections to all RTMs at once, ***PfInitialize()*** may be called anywhere in the application code, not necessarily from the main-routine.

Which RTMs do the API functions talk to?

The behaviour of the API functions when communicating with several RTMs is based on the notation of the "set of current RTMs". All API functions use this set. The default rules for this set are as follows: Inside ***connectCB()*** and ***disconnectCB()***, the set of current RTMs consists of the single RTM to which connection is established or lost. Anywhere else, the current set of RTMs consists of all the connected RTMs for which ***connectCB()*** have finished.

RTMs might be added to or withdrawn from the set of current RTMs using the API functions ***PfAddRtmToCurrent()*** and ***PfWithdrawRtmFromCurrent()***. Refer to their man pages for details. The id for an RTM can be obtained by using ***PfId()***, e.g. rtmId = PfId(PfLocalProcess,"rtm1").

What about return values from API functions?

Return values from API functions can be split into three categories.

1) Functions returning the value of apiError.
2) Functions returning an id to be used later.
3) Functions returning a result from an RTM.

The global variable apiError is set by all API functions to reflect the error status from the function. What is the error status from a function talking to several RTMs? It is OK if the function worked properly in all RTMs. It is the error status of the first failing RTM if the function failed for one or several RTMs. The definition of the term "first RTM" is implementation dependent.

Functions returning an id to be used in later API function returns local ids, so there should be no problems for these functions. For instance, PfCreateVar returns an id, and if the same variable is later created in another RTM, the local variable will be reused, and the id returned will be identical.

Some functions returning a result from an RTM require that the set of current RTMs consists of a single RTM. Others return the result from the first RTM. Details are found at the man pages.

22.2 Integrating OSF/Motif

Ignoring OSF/Motif when designing ProcSee was impossible as Motif was the preferred interface style within the UNIX world. Early tests however showed that integrating X Intrinsics and the Motif widget set directly into the Run-Time Manager was not a good solution. Motif is really a "closed" concept in the sense that it demands full control of its user interface components, which made it difficult to combine them with ProcSee's own graphical objects. ProcSee wasn't meant to be a tool for developing push-buttons and menus either; there are literally dozens of good tools available for doing this. The solution was based on the following facts:

- Few applications actually mix widgets and graphics; most are organized as an "administrative" shell with menus, push-buttons and scrollbars, i.e. widgets, and all the graphics in a dedicated window (drawing area or canvas) somewhere in the centre.
- X-Windows is unique in the way it handles most of its resources. Windows are publicly available so several programs can simultaneously access the same windows on the screen. Although one application A created and hence *owns* a window, another application B may well draw graphics in it and receive operator events from it.

22.3 The Concept

A concept was chosen that, in contrast to the alternative of full integration, gives some very important benefits:

- Almost any commercially available tool for building Motif applications can be used in conjunction with ProcSee.
- The integration concept is not limited to OSF/Motif; it holds for other toolkits as well, like Sun's Open Windows, and even plain Xlib programs.

The integration concept is schematically shown in Figure 188. The Motif part of the interface is managed by a separate program which can be "hand coded" or generated by some Motif interface builder. The ProcSee API library is linked with this process to enable communication with the Run-Time Manager. The Motif process should

contain no more than the code necessary to produce the interface, as the application code is separated in another process.

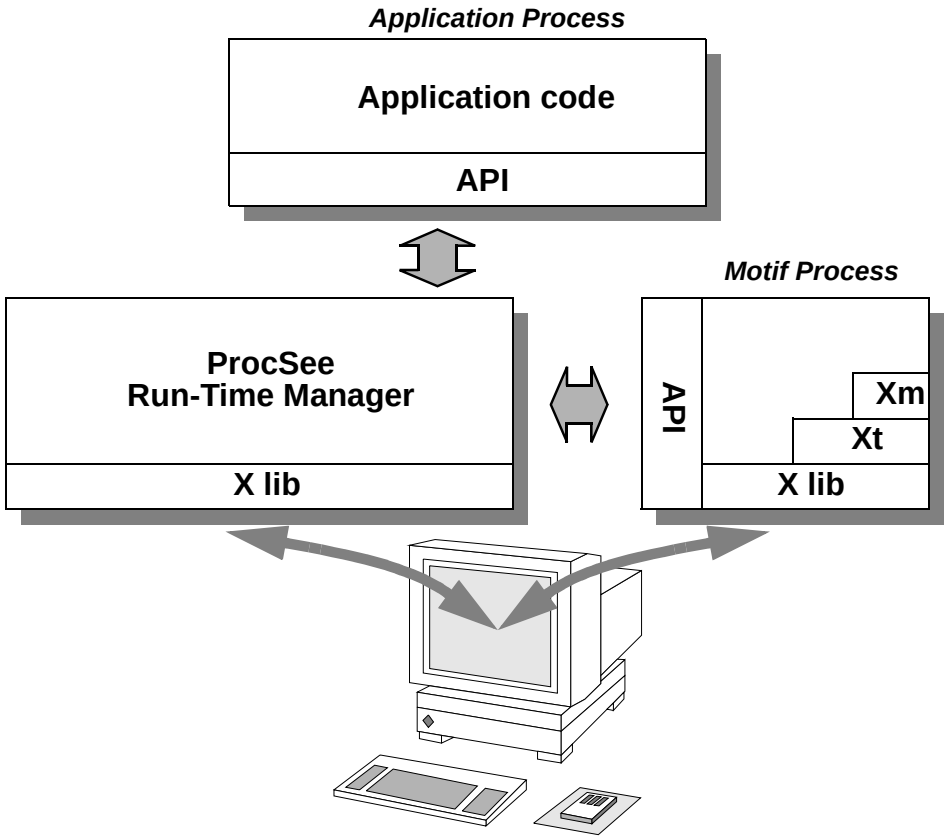


Figure 188 *The Motif part is separated from the ProcSee part of the user interface. The application is still a separate process to ensure full separation of the user interface from the application code. In this figure the computer icon also represents the X server.*

The Motif process performs some handshaking upon start-up:

- It connects to the Run-Time Manager by a call to ***PfInitialize***. RTM registers the Motif process as it does with any process that connects.
- It must instruct Xt to listen for input on the socket(s) used by the API in order to handle incoming messages from the RTM.
- It "exports" the canvas window(s) to RTM by calling ***PfExportWindow***. RTM will register the window much like its own windows, but the window is flagged as imported. RTM will subscribe to all the usual X events in order to obtain operator input in the window.

Because of a lot of special handling inside the Xt Intrinsics main loop used by Motif, a Motif application has to use the standard Xt Intrinsics main loop instead of the ProcSee API main loop. This leads to a rather complicated piece of code to instruct this main loop to listen to the right sockets. To make this connection easier, a head-

er file p3xapi.h and a part of the api library has the necessary code. To activate the right code, the application has to use ***PfXtInitialize*** instead of ***PfInitialize***. The ***PfXtInitialize*** sets up some callback routines that makes the ***XtAppMainLoop*** react to the messages from the RTM, and also makes the API periodic routines work as expected. After this initialization, a normal ***PfInitialize*** is executed as part of the ***PfXtInitialize*** function. To clean up afterwards, the ***`PfXtClose`*** should be used instead of the ***`PfClose`*** function. ***PfXtClose*** executes the ***PfClose*** function, and does some cleanup for the Xt motif main loop.

22.4 Motif integration example

To give an idea of what code is needed to create a ProcSee application with Motif integration, this section describes some of the code needed to create the application shown in Figure 189.

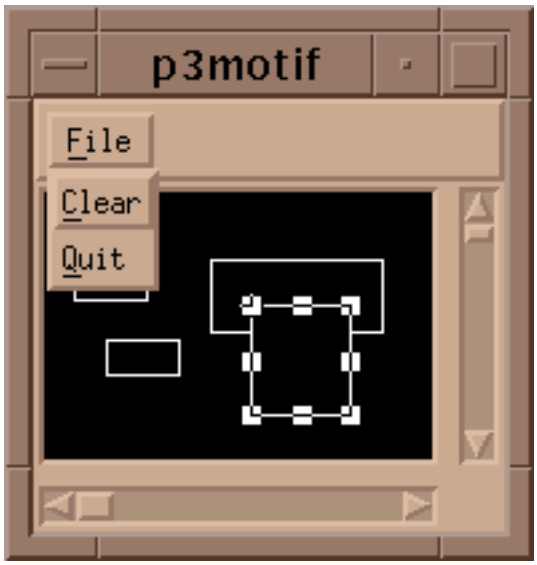


Figure 189 *The p3motif example application.*

This program contains some simple Motif menus and an drawing area used by ProcSee. Only the ProcSee interfacing parts of the program will be described, assuming that the programmer know Motif already.

In addition to the Motif and X header files the p3xapi.h should be included:

Example:

```
#include <api/p3xapi.h>
```

Code for ProcSee connect and disconnect events are needed. The connect function exports the drawing area, and loads and displays a picture. In this example the window is also put into draw rectangles mode:

Example (continued):

```
bool hasContact = FALSE;

void p3connect(status, rtmName)
int32 status;
char *rtmName;
{
   printf("Connected to rtm ('%s')\n", rtmName);

   PfExportWindow("testWin",NULL,XtWindow(drawArea),
                  WOpColor|WOpZoomPan,wopCB,drawArea);

   PfExecute("", "{ load(\"NoName.ppic\"); }");

   PfExecute("","{ displayPicture(\"testWin\",\"NoName\"); }");

   PfExecute("","{ editWindow(\"testWin\",%d,\"\"); }",pRectangleClass);

   hasContact = TRUE;
}
```

The disconnect function only adjusts the connection status:

Example (continued):

```
void p3disconnect(status, rtmName)
int32 status;
char *rtmName;
{
   if (status & PfCrtmStop )
      printf("Lost contact with rtm ('%s'): (%s) \n",
             rtmName, status & PfCrtmCrash ? "crash" : "stop");
   hasContact = FALSE;
}
```

A function to let ProcSee change motif settings:

Example (continued):

```
int32 wopCB(a,b)
   PfT a;
   char *b;
{
   Arg                av[10];
   int                ac;
   Widget             w;
   Widget             scrolledW;
   pWindowOperation*  op;

   w = (Widget)a;
   op = (pWindowOperation*)b;


   switch (op->any.operation)
   {
```

Code to set the window colour:

Example (continued):

```
      case WOpBackgroundColor:
      case WOpForegroundColor:
      case WOpColor:
        ac=0;
        XtSetArg(av[ac], XmNbackground, op->color.background); ac++;
        XtSetValues(w, av, ac);
        break;
```

Code to set window size, adjust the scrollbars, and possible code for a lot of other cases...

Example (continued):

```
      case WOpZoomPan:

        break;


      case WOpResize:
        break;
    }

    return OK;
}
```

Functions called by the Motif menus and buttons contain code that executes ProcSee actions:

Example (continued):

```
void file_cb(w, item_no)
   Widget w;
   int item_no;
{
   switch (item_no)
   {
    case Clear_ItemNo:
```

Test if contact, then select all shapes:

Example (continued):

```
        if (hasContact)
           PfExecute("","{ NoName.selectAllIn(1,0,0,0,0); cut(); }");
        break;
     case Quit_ItemNo:
```

When the user quits, the window(s) should be released:

Example (continued):

```
        PfReleaseWindow("testWin");
```

And PfXtClose called to clean up the p3xapi:

Example (continued):

```
        PfXtClose();
        exit(0);
        break;
     default:
        break;
   }
}
```

When a window is exported, only ProcSee can listen to some of the events (Button pressed can not be sent to more than one application at the time. Exposure to Motif will lead to flicker and sometimes areas without ProcSee shapes.):

Example (continued):

```
/* Make sure only rtm listens to the ButtonPressed and ... events */
void removeBadEvents(aW)
   Widget aW;
{
   long  event_mask;

   event_mask = (long)XtBuildEventMask(aW);
   event_mask = event_mask & ~ExposureMask;
   event_mask = event_mask & ~ButtonPressMask;
   event_mask = event_mask & ~ButtonReleaseMask;

   XSelectInput(XtDisplay(aW), XtWindow(aW), event_mask );
   XSync(XtDisplay(aW), False );
}
```

The main function contains code that builds the Motif part of the application, and the call to PfXtInitialize that initializes the ProcSee api and connects to the ProcSee RTM:

Example (continued):

```
main(argc,argv)
   int argc;
   char *argv[];
{
   Widget main_w, menubar, widget;
   XtAppContext app;
   XmString file, quit, clear;
```

After the code to create the motif window and the menus, the drawing area is created:

Example (continued):

```
    drawArea = XtVaCreateManagedWidget("Working", xmDrawingAreaWidgetClass,
                                       main_w, NULL);
```

After the widgets have been realized and the bad events have been removed from the drawing area, the PfXtInitialize is called:

Example (continued):

```
    XtRealizeWidget(toplevel);
    removeBadEvents(drawArea);

    PfXtInitialize(app,"NoName","p3motif","rtm",NULL,100,TRUE,
                   p3connect,p3disconnect);
```

Then the Motif main loop is called:

Example (continued):

```
    XtAppMainLoop(app);
}
```

The complete example is found in **$PROCSEE_DIR/examples**. The code can be compiled as follows:

```
cc p3motif.c -o p3motif -L$PROCSEE_DIR/lib/$ARCH -
lProcSeeApiC -lXm -lXt -lX11
```

To run this example, the RTM has to be started as:

```
rtm -r $PROCSEE_DIR/etc/$ARCH/NoName.pctx &
```

Then the example program can be started:

```
p3motif &
```

It is now possible to draw ProcSee rectangles in the drawing editor window, and use the Motif menus to control the process.

The source code for the Motif example can be found in APPENDIX C - motif integration example on page 367.

APPENDIX A - MyApp.Tdoc

This appendix contains a sample application resource file in document format (Tdoc).

```
/*
 * ProcSee CONFIGURATION LANGUAGE FILE
 *
 *
 * Application name: MyApp
 *
 *
 * Documented by  : kjellab
 * Host           : polaris
 * Date           : 14/02-95 at 15:45:40
 *
 *
 * Generate with:
 *    pcc MyApp.Tdoc
 *
 */


application MyApp
{

  database = "./MainDb.pdat";


  process secondaryDb
  {
    database = "./secondaryDb.pdat";
  }

  use "./libraries/Components.plib";

  int MyAttribute = 0;
  int F1 = 65470;


  function `void constructor()
  {
     // Initialize attribute

     MyAttribute = 0;

     // Load and display the start up picture

     displayPicture("winMain", "./pictures/MyStartPic.ppic" );
```

```
    winMain.title("This is my start up picture");
}`

dialogue
{
  event = `keyPressed( F1 )`;
  action = statement `{ displayPicture("winMain", "MyStartPic" ); }`;
}

colour colour0 { red = 0; green = 0; blue = 0; }
colour colour1 { red = 65535; green = 65535; blue = 65535; }
colour colour2 { red = 65535; green = 0; blue = 0; }
colour colour3 { red = 0; green = 65535; blue = 0; }
colour colour4 { red = 0; green = 0; blue = 65535; }
colour colour5 { red = 65535; green = 65535; blue = 0; }
colour colour6 { red = 65535; green = 0; blue = 65535; }
colour colour7 { red = 23130; green = 23130; blue = 23130; }
colour colour8 { red = 32639; green = 32639; blue = 32639; }
colour colour9 { red = 37779; green = 37779; blue = 37779; }

font font0
{ name = "-adobe-helvetica-bold-r-normal--12-120-75-75-p-70-iso8859-1"; }
font font1
{ name = "-adobe-helvetica-medium-r-normal--14-140-75-75-p-77-iso8859-1"; }
font font2
{ name = "-adobe-helvetica-bold-r-normal--14-140-75-75-p-82-iso8859-1"; }
font font3
{ name = "-adobe-helvetica-medium-r-normal-*-10-*-*-*-*-*-*-*"; }
font font4
{ name = "-adobe-helvetica-bold-r-normal--18-180-75-75-p-103-iso8859-1"; }
font font5
{ name = "-adobe-helvetica-bold-r-normal-*-10-*-*-*-*-*-*-*"; }
font font6
{ name = "-adobe-helvetica-bold-r-normal--24-240-75-75-p-138-iso8859-1"; }
font font7
{ name = "-adobe-helvetica-bold-r-normal--12-120-75-75-p-70-iso8859-1"; }
font font8
{ name = "-adobe-helvetica-bold-r-normal--14-140-75-75-p-82-iso8859-1"; }
font font9
{ name = "-adobe-helvetica-bold-r-normal--18-180-75-75-p-103-iso8859-1"; }
font font10
{ name = "-adobe-helvetica-bold-r-normal--24-240-75-75-p-138-iso8859-1"; }
font font11
{ name = "-adobe-helvetica-bold-o-normal--12-120-75-75-p-69-iso8859-1"; }
font font12
{ name = "-adobe-helvetica-bold-o-normal--14-140-75-75-p-82-iso8859-1"; }

pattern DefaultPattern1 { patternBitmap = "p3.internal.patt1"; }
pattern DefaultPattern2 { patternBitmap = "p3.internal.patt2"; }
pattern DefaultPattern3 { patternBitmap = "p3.internal.patt3"; }

linestyle DefaultStyle1
{
  numberOfElements = 1;
  element[0] = 1;
}

linestyle DefaultStyle2
{
  numberOfElements = 1;
  element[0] = 2;
}

linestyle DefaultStyle3
{
  numberOfElements = 1;
  element[0] = 4;
}
```

```
  linestyle DefaultStyle4
  {
    numberOfElements = 1;
    element[0] = 8;
  }

  linestyle DefaultStyle5
  {
    numberOfElements = 1;
    element[0] = 12;
  }

  linestyle DefaultStyle6
  {
    numberOfElements = 1;
    element[0] = 16;
  }

  linestyle DefaultStyle7
  {
    numberOfElements = 1;
    element[0] = 24;
  }

  display mainDisplay
  {
    connection = "";
    width = 1280;
    height = 1024;
    resizeMode = 0;
  }

  window winMain
  {
    parent = "mainDisplay";
    x = 0;
    y = 0;
    width  = 1142;
    height = 870;
    backgroundColour = -1;
  }

}
```

APPENDIX B - Trend diagram with ruler

This appendix contains excerpts of a picture document file showing the use of the trend ruler in a trend diagram. This example relates to the illustration given in Figure 121 on page 204.

```
trend T
{
  trLog = "TrLog";
  x = 35;
  y = 44;
  width = 400;
  height = 200;

  visibility = 1;
  timespan = 240;
  orientation = 0;
  scroll = 0;
  foregroundFillColour = 16;
  backgroundFillColour = -1;
  foregroundColour = 1;
  backgroundColour = -1;
  lineWidth = 0;
  linePattern = 1;
  fillPattern = 1;

    trendPlot Value1Crv
    {
      variable = `Var1`;
      presentation = 0;
      upperLimit = 500;
      lowerLimit = 0;
      visibility = 1;
      logFrequency = 10;
      foregroundFillColour = 1;
      backgroundFillColour = -1;
      foregroundColour = 4;
      backgroundColour = -1;
      lineWidth = 0;
      linePattern = 1;
      fillPattern = 1;
      autoScale = 0;
      upperBand = 1;
      lowerBand = 1;
    }

    trendPlot Value2Crv
    {
      variable = `Var2`;
      presentation = 0;
      upperLimit = 500;
      lowerLimit = 0;
      visibility = 1;
      logFrequency = 10;
      foregroundFillColour = 1;
      backgroundFillColour = -1;
      foregroundColour = 4;
      backgroundColour = -1;
      lineWidth = 0;
      linePattern = 1;
      fillPattern = 1;
```

```
      autoScale = 0;
      upperBand = 1;
      lowerBand = 1;
    }


    trendLabels TimeTag1
    {
      offset = 20;
      alignment = 0;
      interval = 120;

      theFont = 2;
      format = "%H:%M:%S";
      visibility = 1;

      foregroundColour = 1;

      backgroundColour = -1;

    }  // end of trLabel


     trendRuler Ruler
     {
       visibility = 0;
       foregroundColour = 26;
       backgroundColour = -1;
       lineWidth = 1;
       linePattern = 1;
       rulerIsSelected = 1;
     } // end of trendRuler

     dialogue
     {
       event = `shapeEntered()`;
       action = statement `{
                   Ruler.visibility = 1;
                   thePicture().theWindow().update();
                 }`;
     }

     dialogue
     {
       event = `shapeLeft()`;
       action = statement `{
                   Ruler.visibility = 0;
                   thePicture().theWindow().update();
                 }`;
     }

     dialogue
     {
       event = `cursorMoved()`;
       action = statement `{
                   Ruler.moveRuler();
                   thePicture().theWindow().update();
                 }`;
     }

}  // end of trend T


//
// The Time text
//

  text Time
```

```
  {
    x = 700;
    y = 50;
    format = "%-19s";
   theText = `cftime(timeStr, "%x", T.Value1Crv.getRulerTime(0)), timeStr`;
    visibility = `T.Ruler.visibility`;
    foregroundColour = 1;
    backgroundColour = 41;
    theFont = 8;
  }

//
// The Value1 text
//

  text Value1Txt
  {
    x = 700;
    y = 100;
    format = "%-10.2f";
    theText = `T.Value1Crv.getRulerValue(0)`;
    visibility = `T.Ruler.visibility`;
    foregroundColour = 1;
    backgroundColour = 41;
    theFont = 8;
  }

//
// The Value2 text
//

  text Value2Txt
  {
    x = 700;
    y = 150;
    format = "%-10.2f";
    theText = `T.Value2Crv.getRulerValue(0)`;
    visibility = `T.Ruler.visibility`;
    foregroundColour = 1;
    backgroundColour = 41;
    theFont = 8;
  }
```

APPENDIX C - motif integration example

This appendix contains an example on how OSF/Motif program can be integrated with ProcSee.

```
/* A simple motif program that displays a ProcSee picture in a window   */
/* The picture is put in draw rectangles mode, allowing the user to place */
/* as many rectangles as he/she wants in the ProcSee window.            */


/*
* sccsid: @(#)examples/p3motif.c  106.2  96/02/12 22:10:08 HRP
*/

#include <Xm/MainW.h>
#include <Xm/DrawingA.h>
#include <Xm/ScrolledW.h>
#include <stdlib.h>

#include <api/p3xapi.h>


#define Clear_ItemNo 0
#define Quit_ItemNo 1


Widget toplevel, drawArea;

bool hasContact = FALSE;


Widget get_widget(w,resource)
   Widget w;
   char *resource;
{
   Arg al[1];
   Widget value;

   XtSetArg(al[0], resource, &value);
   XtGetValues(w, al, 1);

   return value;
}


void file_cb(w, item_no)
   Widget w;
   int item_no;
{
   switch (item_no)
   {
    case Clear_ItemNo:
      if (hasContact)
 PfExecute("","{ NoName.selectAllIn(1,0,0,0,0); cut(); }");
      break;
    case Quit_ItemNo:
      PfReleaseWindow("testWin");
      PfXtClose();
      exit(0);
      break;
    default:
```

```
      break;
   }
}


int32 wopCB(a,b)
   PfTReqId a;
   char *b;
{
   Arg               av[10];
   int               ac;
   Widget            w;
   Widget            scrolledW;
   pWindowOperation* op;

   w = (Widget)a;
   op = (pWindowOperation*)b;

   printf("wopCB(%d,%p)\n",a,b);


   switch (op->any.operation)
   {
    case WOpZoomPan:
      printf("WindowOp: WOpZoomPan(w = %d, h=%d, x=%d, y=%d)\n",
     op->zoomPan.width, op->zoomPan.height,
     op->zoomPan.x, op->zoomPan.y);
      ac = 0;
      XtSetArg(av[ac], XmNwidth,  op->zoomPan.width);  ac++;
      XtSetArg(av[ac], XmNheight, op->zoomPan.height); ac++;
      XtSetArg(av[ac], XmNx,      op->zoomPan.x);      ac++;
      XtSetArg(av[ac], XmNy,      op->zoomPan.y);      ac++;
      XtSetValues(w, av, ac);

      /* Check if parent to parent widget of
 drawing area is a scrolled window. */
      scrolledW = XtParent(XtParent(w));

      if (XtIsSubclass(scrolledW, xmScrolledWindowWidgetClass))
      {
 {
    Widget sbw;
    int    value,
    slider_size,
    increment,
    page_inc,
    minPos,
    maxPos;

    sbw = get_widget(scrolledW, XmNhorizontalScrollBar);
    /* Get minPos, maxPos and slider_size */
    XtVaGetValues(sbw, XmNminimum, &minPos, XmNmaximum,
  &maxPos, NULL);
    XmScrollBarGetValues(sbw, &value, &slider_size,
 &increment, &page_inc);

    value = -(op->zoomPan.x);

    if (value < minPos)
       value = minPos;

    if ((value + slider_size) > maxPos)
       value = maxPos;

    XmScrollBarSetValues(sbw, value, slider_size, increment,
 page_inc, True);
 }
 {
```

```
   Widget sbw;
   int    value,
   slider_size,
   increment,
   page_inc,
   minPos,
   maxPos;

   sbw = get_widget(scrolledW, XmNverticalScrollBar);
   /* Get minPos, maxPos and slider_size */
   XtVaGetValues(sbw, XmNminimum, &minPos, XmNmaximum,
 &maxPos, NULL);
   XmScrollBarGetValues(sbw, &value, &slider_size,
&increment, &page_inc);

   value = -(op->zoomPan.y);

   if (value < minPos)
      value = minPos;

   if ((value + slider_size) > maxPos)
      value = maxPos - slider_size;

   XmScrollBarSetValues(sbw, value, slider_size, increment,
page_inc, True);
}
     }
     break;

   case WOpBackgroundColor:
   case WOpForegroundColor:
   case WOpColor:
     printf("WindowOp: WOpColor\n");
     printf(" background: %d\n", op->color.background);
     ac=0;
     XtSetArg(av[ac], XmNbackground, op->color.background); ac++;
     XtSetValues(w, av, ac);
     break;

   case WOpResize:
     printf("resize( %d, %d );",
    op->resize.width, op->resize.height);
     break;

   case WOpMove:
     printf("move( %d, %d );\n",
    op->move.x, op->move.y);
     break;

   case WOpMap:
      printf("map();\n");
      break;

   case WOpUnmap:
      printf("unmap();\n");
      break;

   case WOpIconify:
      printf("iconify();\n");
      break;

   case WOpClose:
      printf("close();\n");
      break;

   case WOpRaise:
      printf("raise();\n");
      break;
```

```
    case WOpLower:
       printf("lower();\n");
       break;

    default:
      printf("WindowOp: Unknown\n");
      break;
   }

   return OK;
}


void p3connect(status, rtmName)
int32 status;
char *rtmName;
{
   printf("Connected to rtm ('%s')\n", rtmName);

   PfExportWindow("testWin",NULL,XtWindow(drawArea),
                    WOpColor|WOpZoomPan,wopCB,drawArea);

   PfExecute("", "{ load(\"NoName.ppic\"); }");

   PfExecute("","{ displayPicture(\"testWin\",\"NoName\"); }");

   PfExecute("","{ editWindow(\"testWin\",%d,\"\"); }",pRectangleClass);

   hasContact = TRUE;
}


void p3disconnect(status, rtmName)
int32 status;
char *rtmName;
{
   if (status & PfCrtmStop )
      printf("Lost contact with rtm ('%s'): (%s) \n",
             rtmName, status & PfCrtmCrash ? "crash" : "stop");
   hasContact = FALSE;
}


/* Make sure only rtm listens to the ButtonPressed and ... events */
void removeBadEvents(aW)
   Widget aW;
{
   long  event_mask;

   event_mask = (long)XtBuildEventMask(aW);
   event_mask = event_mask & ~ExposureMask;
   event_mask = event_mask & ~ButtonPressMask;
   event_mask = event_mask & ~ButtonReleaseMask;

   XSelectInput(XtDisplay(aW), XtWindow(aW), event_mask );
   XSync(XtDisplay(aW), False );
}


main(argc,argv)
   int argc;
   char *argv[];
{
   Widget main_w, menubar, widget;
   XtAppContext app;
   XmString file, quit, clear;
```

```
    Arg arglist[5];
    int n;

    printf("p3motif example program started.\n");

    n = 0;
    XtSetArg(arglist[n], XmNallowShellResize, True);  n++;
    toplevel = XtAppInitialize(&app, "p3motif", NULL, 0,
       &argc, argv, NULL, arglist, n);

    printf("toplevel=%d\n",toplevel);

    main_w = XtVaCreateManagedWidget("main_window", xmMainWindowWidgetClass,
     toplevel,
     XmNscrollBarDisplayPolicy, XmAS_NEEDED,
     XmNscrollingPolicy, XmAUTOMATIC,
     NULL);

    /* A simple menu */
    file = XmStringCreateSimple("File");
    menubar = XmVaCreateSimpleMenuBar(main_w, "menubar",
      XmVaCASCADEBUTTON, file, 'F',
      NULL);
    XmStringFree(file);


    /* The file menu */
    clear = XmStringCreateSimple("Clear");
    quit = XmStringCreateSimple("Quit");
    XmVaCreateSimplePulldownMenu(menubar, "file_menu", 0, file_cb,
XmVaPUSHBUTTON, clear, 'C', NULL, NULL,
XmVaPUSHBUTTON, quit, 'Q', NULL, NULL,
NULL);
    XmStringFree(clear);
    XmStringFree(quit);


    XtManageChild(menubar);


    drawArea = XtVaCreateManagedWidget("Working", xmDrawingAreaWidgetClass,
       main_w,
       NULL);

    XtVaSetValues(main_w,
 XmNmenuBar, menubar,
 XmNworkWindow, drawArea,
 NULL);

    XtRealizeWidget(toplevel);
    removeBadEvents(drawArea);


    PfXtInitialize(app,"NoName","p3motif","rtm",NULL,100,TRUE,
  p3connect,p3disconnect);

    XtAppMainLoop(app);
}
```

Index

Symbols

.pctx 10, **26**, 50, **79**, 80
.pdat 26, 331
.plib 26
.ppic 26
.ptrv 240
.Tdoc 26, **44**, 53, **79**, 80, 138, 182, 359
$ARCH 10
$PROCSEE_DIR 10

Numerics

32 bit identifiers 40

A

Accept button 125, 126
action
  periodic 322
adjust
  scrollbar 355
Advanced Editing Settings... 227
advanced window settings 125
Align... 186
alignment
  trend 202, 203
  window 186, 187
Analyse menu 142
  Check All 142
  Check Current 142
  Install 142
API **19**, 20, 24–25, 28
appendLogBuffer **261**, 262
application 17, 18, 28, **46**
  active 168
  code 19, 319, 349
  create 30
  data 21, 26, 32, 74
  data connect 74
  database 25
  declare 31
  document (.Tdoc) 53
  document file 138
  level 21, 175
  lifecycle 18
  load 49
  name 37, 44, 50, 51, 79, 80
  new 51
  process 25
  program 20, 21, 25, 320, 321, 322
  resource file 193
  save 51
  save as 30, 52, 53
applName
  PfInitialize 350
appName 10
Arc Chord 221
Arc Slice 221
argument
  by value 67
  formal 339
  pass 67
  PfCReadCondition 324
arithmetic
  expression 58
array
  create 329
  indices 331
assignment
  bitwise 74
  dynamic 32, 75
  expression 60
  regular 32
attribute 80, 144, **144**

class 23
compile 146
create 145
current value 232
delete 147
dynamic value 194
edit 147
editor window 147
Fill Foreground 195
graphic 192
interval 202
new 145, 146
offset 202, 203
timespan 200
trend 200
user defined 177
value 193
Automation 267
autoscale
trend 201

B

Back 189
background
colour 229
Background Colour
window 229
binary
files 26
binding
dynamic 74, 75
bitwise
assignmen 74
operation 74
blink
colour 86
interval 86
blink colour
define 86
break
loop 64
statement 64
broadcast
message 343, 345, 347
browser
class 219
button
Cancel 147
Compile 147, 149
Delete 147
Edit 141, 150, 155
Focus 134
View 143

C

callbacks
connection 319
cancel
button 147
new picture 168, 171
print 183
cancelLogBuffer **261**, 262
category
message 343, 346
change
motif settings 355
Charset 91
Check All 142
Circle 214
Circle Arc 215
class 19, 165, **173**–176
attribute 23
behaviour 23
concept 22
coordinate system 176
create 31, **174**
definition 26
delete 176
edit 175
graphic 133
instance 20, 23, 219
library 23, **173**
new 135, 136, **174**
pushbutton 76
reuse 23, **173**
standards 23
template 173
Class Browser 219
Clear 186
close
drawing editor 184
graphics editor 184
code
application 319, 349
coexistence 24
colour 80

background 229
blink 86
create 83
define 30
editor 83, **85**, **87**, **88**
Graphic Attributes... 195
HLS 84
RGB 84
systems 84
tuning 183
View menu 83
Colour Editor
Add blink 86
Delete blink 86
COM 265
Automation 267
COM Interface Pointers 297
ComClass **290**
ComEnum **292**
ComEventObject **296**
ComInterface **291**
ComInterfaceObject **295**
ComLibrary **290**
ComObject **294**
ComShape **294**
ComShape property pages 278
Inserting ComShape 275
onUpdate 288
Picture / Class Properties 286
Selected Properties 281
COM Types 268
ComEnum 268
ComEventObject 268
ComInterface 268
ComInterfaceObject 268
ComObject 268
ComShape 268
COM types
ComLibrary 268
ComClass 268, **290**
ComEnum 268, **292**
ComEventObject 268, **296**
ComInterface 268, **291**
ComInterfaceObject 268, **295**
ComLibrary 268, **290**
command
print 183
recall 233
command line field
drawing editor 232
comment
multi-line 56
nesting 56
single-line 56
communication 24
methods 335
ComObject 268, **294**
compile
attribute 146
button 147, 149
dialogue 155
function 149
compound
statement 62
ComShape 268, **294**
ComShape property pages 278
conditional
statement 63
connect 321
connectCB 319, 320, 321, 350
connection
callbacks 319
establish 349
constant
expression 57
constructor 155–157, 173
continue
loop 64
statement 64
control 29
SWBus 29
Control Server
problems 321
coordinate system
class 176
screen 166
world **166**, 167
Copy 185
cord
arc 221
create
array 329
attribute 145
class 31, 174
data file 141
function 148

library 31, 133
new dialogue 152
new picture 168
create a dialogue 152
crosshair
cursor 225
crosshair size
cursor 225
current value
attribute 232
cursor
create 110
crosshair 225
crosshair size 225
define 80
editor 110
View menu 110
cursor type 125
Cut 185

D

data
sharing 24
data file
create 141
edit 141
database 138
application 25
file format (.pdat) 27
interface 20
network 22
process 44
server 17, 335
system 26
datatypes
PfCDouble 329
PfCFloat 329
PfCInt 329
PfCRecord 329
PfCShortInt 329
PfCUnsignedChar 329
PfCUnsignedInt 329
PfCUnsignedShortInt 329
declaration
statement 62
declare
application 31
function to RTM 338
Decoration 124
decoration 125
default process 18
define
blink colour 86
colour 30
cursor 80
font 30
window 30, 120
definition
trendLog 240
delete
attribute 147
button 147
class 176
dialogue 155
function 150
windows 121
delete a dialogue 155
delete a function 150
delete an attribute 147
dentifier 56
depthPosition
trend grid 203
destructor 156, 173
diagram
trend 204, 363
dialogue
action 152
compile 155
delete 155
edit 155
key window 154
new 152
trigger 152
user defined 177
Dialogue Area 215
dialogues 21
disconnect 321
disconnectCB 319, 320, 322, 350
displayPicture 345, 346
Distribute... 187
distribution
window 188
division
integer 58
Document 182
doubleIt 68

do-while
statement 64
draw
rectangle 194
drawing editor 165, ??–171, 174–176, 179
close 184
command line field 232
different cursors 211
message line 211
message window 232
test 179
dummy user program 233
dynamic
assignment 32, 75
binding 74, 75
graphics 43

E

easy testing 43
Edit
button 155
edit
attribute 147
button 141, 150
class 175
data file 141
dialogue 155
File menu 142
font viewer 89
function 150
picture **171**
windows 121
edit a dialogue 155
edit a function 150
edit an attribute 147
Edit menu **185**
Align... 186
Back 189
Clear 186
Copy 185
Cut 185
Distribute... 187
Front 189
Group 188
Paste 186
Rotate +45° 189
Rotate -45° 190
Scale 1.1 190
Scale 1 / 1.1 190
Select All 186
Ungroup 188
Unselect All 186
Editing Options
window 223
Editing... 223
editor
colour 83, **85**, **87**, **88**
COM 265
cursor 110
drawing 165, 179
linestyle 108
pattern 95
resource 43
Ellipse 216
Ellipse Arc 216
e-mail addresses 10
Encoding 91
equality
testing 71
establish
connection 349
execute 76
executeAsync 76
existing
pdat file 142
expression 60
aritmetic 58
assignment 60
constant 57
operator 57
statement 62
type 57
value 57
external trend logger **241**, 241

F

Family 91
file
binary 26
descriptor mask 324
new 51
socket number 324
Tdoc 27
File menu 49, 51, 133, 168, 170, 175, **181**
Document 53, 182
Edit 142

Insert 175
Insert As... 175
New
Application... 51
Library... **133**, 133, **134**, **136**
Picture 181
Open
Application... 49
Picture... 181
Print... 182
Remove 142
Save (needed) 51, 52, 136, 182
Save As... 51, 136, 168, 170, **182**
Fill Foreground
attribute 195
filter
process 26
program 25
focus 30, 46, 135
button 134
window 46, 134
font 80
define 30
font viewer 89, 90, 92
edit 89
new 89
for
statement 69
formal
argument 339
format
Tdoc 28
Foundry 91
Front 189
fullname 35
lookup mechanism 36
Functio 124
function **66**, **148**
appendLogBuffer **261**, 262
cancelLogBuffer **261**, 262
compile 149
connectCB 319, 320, 350
create 148
declare 338
delete 150
disconnectCB 319, 320, 350
displayPicture 345, 346
doubleIt 68
edit 150
execute 76
executeAsync 76
getLogBuffer **263**, 264
new 148
newLogBuffer **261**, 262
open 76
periodicCB1 322
periodicCB2 322
PfAddField 332, **332**
PfAddInput 323, 324, 336
PfAddRtmToCurrent 350
PfBeginRecord 332, **332**
PfCreateArray 329
PfCreateVar **327**–329, 331, 333
PfCRecord 333
PfDeleteProcessHandler 322
PfDiagnosticsSubscribe 343, 344
PfEndRecord 332, **333**
PfExecute 338, 339, 341, 354, 356
PfExportWindow **352**, 354
PfFlush 331
PfFlushCreateVar 328, 330
PfGetDiagnosticsMessage 345
PfGetFuncArg 342
PfId 350
PfInitialize **320**, 324, 349–353, 357
PfMainLoop **320**, 321, 322–325, 336
PfReadScript **331**, 332, 334
PfRegisterFunction 338, 342
PfReleaseWindow 356
PfSetProcessHandler **322**–324, 330, 336
PfTArg 342
PfUpdateValuesId 329
PfWithdrawRtmFromCurrent 350
PfXtClose 356
PfXtInitialize **353**
picture **172**, 173
power **66**, 66
recursive **68**
remote call 341
remove 173
signature 339
square **66**, 67
strftime(3C) 202
tr 258
user defined 177

void 67
void constructor 173, 176
void destructor 173, 176

G

GED 18, 27, 28, 43, 44
drawing editor 179
Focus 135
graphics editor 46
main window 83, 84, 93, 95, 108, 110, 121, 122
ged 29
GED Focus
Remove 137
GED's main window
View menu 83
getLogBuffer **263**, 264
GIS 26
graphic attributes menu 192
Graphic Attributes...
window **192**, 195
graphic class 133
graphic elements 165
Graphic menu
Circle 214
Circle Arc 215
constant line width 211
Dialogue Area 215
Ellipse 216
Ellipse Arc 216
Image 217
Instance 219
Keep 220
Line 217
Polygon 218
Rectangle 217
RoundRect 218
Select 214
Test 213
Text 218
Trend 219
graphic shape
move 225
rotationAngle 199
graphics
dynamic 43
graphics editor 18, 20–23, 26, 28, 43, 44, 179, 183, 205
Close 184
GED 46
Graphics menu 211, **212**, 231
grid
trend 202, 253
Group 188
grouping 58
GUI 9

H

handler
process 330
header file 319
historic trend 204
HLS (colour) 84

I

I-beam
icon 10
icon
explanation 10
I-beam 10
if
statement 63
if-else
statement 63
ignore
OSF/Motif 351
Image 217
implicit scope 345
include 319
indices
array 331
info
printer 183
input sources 323
Insert 175
Insert As... 175
Inserting ComShape 275
Install 142
instance **165**, 173, 176, **219**
class 20, 23, 219
tool 219
Instantiate Class 219
integer
division 58
integration
motif 353

interface
database 20
internal trend logger 240
interval 202
testing 70
iterative process 20

K

Keep 220
key selection menu 154
Key Window 154
keyPressed
trigger
pop-up menu 154
KeyRelease
trigger
pop-up menu 154
keysym 151
keywords 63

L

lCycle
value 262
library 80, 133
add resources 134
class 23, **173**
create 31, 133
new 133, 134, 136
new class 135
remove 137
save 136
symbolic names 135
Line 217
linestyle
create 108
editor 108
View menu 108
load
picture 171
location value 60
logFrequency
trend 201
logical
operator 59
signature 341
lookup
run-time 76
lookup mechanism
fullname 36
name 36
loop **64**
break 64
continue 64
lowerBand
trend 201
lowerLimit
trend 201
lvalue 60

M

main loop
motif 358
main menu 51
main window
GED 83, 84, 93, 95, 108, 110, 121, 122
menu
Analyse 142
Edit **185**
File 49, 51, 133, 168, 170, 175, **181**
graphic attributes 192
Graphics 211, **212**, 231
key selection 154
main 51
New 51
Open 49
Options 176, 215, 216, **221**, 223, 229
Selected Properties 201, 202, 203
Selected Trend 200
tools 185, 211
Trend Properties 200, 201, 202, 203
View 83, 134, 145, 155, 168–176, **192**, 218
Windows 137, **220**
zoom buttons 210
message 345, 346
broadcast 343, 345
category 343
receive 344
scope 343
subscribe 343
text 343
unsubscribe 344
message broadcast 347
message category 346
message line
drawing editor 211

message window
drawing editor 232
motif
integration 353
main loop 358
program 367
motif settings
change 355
move
graphic shape 225
multi-line
comment 56

N

name
lookup mechanism 36
picture 172
names 35
special 57
Navigate 235
nesting
comment 56
new
application 51
attribute 145, 146
class 135, 136, 174
dialogue 152
file 51
font viewer 89
function 148
library 133, 134, 136
pdat file 141
picture 168
trend 201, 202, 203
window 121, 122
windows 121
New menu 51
New Picture 181
new picture
cancel 168, 171
create 168
New Process 140
newLogBuffer **261**, 262
nework
database 22
noOfLines
trend grid 203

O

offset 202, 203
on-line
compilation 20
onUpdate 288
open 76
picture **171**
Pictures 171
Open menu 49
Open Picture... 181
operation
bitwise 74
window 172
operator
logical 59
relational 59
operators
special assignment 61
option menu
Charset 91
Encoding 91
Family 91
Foundry 91
Pixels 91
Slant 91
Spacing 91
Weight 91
Width 91
Options menu 176, 215, 216, **221**, 223, 229
Advanced Editing Settings... 227
Arc chord 221
Arc Slice 221
Background... 229
Editing... 223
Picture Settings... 222
OSF/Motif
ignore 351
program 367

P

p3sim
utility program 233
p3xapi.h 353
pass
argument 67
Paste 186
path 168
pattern 82

create 95
editor 95
View menu 95
pcc 27, 28
pdat file
existing 142
new 141
periodic actions 322
periodicCB1 322
periodicCB2 322
PfAddField 332, **332**
PfAddInput **323**, 324, 336
PfAddRtmToCurrent 350
PfBeginRecord 332, **332**, 334
PfCDouble 329
PfCFloat **329**, 330, 339
PfCInt 328, **329**, 333, 339
PfClose 353
PfCReadCondition 324
PfCreateArray 329
PfCreateVar **327**–329, 331, 333
PfCRecord **328**, 329, 333, 334
PfCrtmCrash 322
PfCrtmStart 322
PfCrtmStop 322
PfCShortInt 329
PfCUnsignedChar **329**, 333, 340
PfCUnsignedInt 329
PfCUnsignedShortInt 329
PfDeleteProcessHandler 322
PfDiagnosticsSubscribe 343, 344
PfEndRecord 332, **333**
PfExecute 338, 339, 341, 354, 356
PfExportWindow **352**, 354
PfFlush 244, 331
PfFlushCreateVar **328**, 330
PfGetDiagnosticsMessage 345
PfGetFuncArg 342
PfId 350
PfInitialize **320**, 321, 324, 349–353
PfMainLoop **320**, 321, 322–325, 336
PfPut 329
PfReadScript **331**, 332, 334
PfRegisterFunction 137, 338, 342
PfReleaseWindow 356
PfSend 329
PfSetProcessHandler **322**–324, 330, 336
PfTArg 342
PfUpdateValuesId **329**, 331
PfWithdrawRtmFromCurrent 350
PfXtClose 353, 356
PfXtInitialize **353**, 357
physical
signature 341
picture
constructor 173
coordinate system 222
destructor 173
edit **171**
load 171
name 172
new 168
open **171**
print 183
remove **172**
rotationAngle 196
standard function 173
standard functions **172**
title 172
update modes 223
Picture Properties... **195**
window 197
Picture Settings
window 222
Picture Settings... 222
Picture / Class Properties 286
Pixels 91
Polygon 218
power **66**, 66
precedence 58
presentation
trend 201
primitive shape 165
print
cancel 183
command 183
picture 183
resource style 117
Print... 182
printer
colour tuning 183
info 183
setting 183
printf 345
problems
Control Server 321

procedure 66
  remote 25, 347
Process
  Editor 138, 141
  New 140
  Viewer 138, 143
process 17, **137**, 320
  data server 25
  database 17, 44
  database file 26
  default 18
  filter 26
  handler **322**–324, 330
  iteractive 20
  record 138
  remote function 46
  variable 39, 46, **144**, 157, 231
process variable
  View 143
processName
  PfInitialize 350
ProcSee
  class 19, 23
  Configuration Language 237
  Reference Manual 13
  support 10
  Tutorial 13
PROCSEE_DIR 10
program
  filter 25
  motif 367
  OFS / Motif 367
  PfUpdateValuesId 331
  user 45
pTALK 10, 21–25, 28, 43, **55**, 58–63, 65–72, 75–77, 148–151, 156, 157, 172, 194, 201, 214, 231, 328, 334, 337–342, 345, 346
  statement 232
push button
  class 76

R

read value 60
recall
  command 233
receive
  message 344
record
  process 138
record type
  PfAddField 332, **332**
  PfBeginRecord 332, **332**
  PfCRecord 333
  PfEndRecord 332, **333**
record variable
  PfBeginRecord 334
  PfCreateVar 333
  PfReadScript **331**, 334
Rectangle 217
rectangle
  draw 194
recursive
  function **68**
regular
  assignment 32
relational
  operator 59
remote
  procedure 25, 347
Remote Function Calls 338
Remove
  File menu 142
remove 173
  library 137
  picture **172**
Remove button 137
rename
  trend 201, 202, 203
reserved
  word 63
Reset 210
resource editor 43
resource styles 117
resources
  colour 80
  cursor 80
  font 80
  library 134
  linestyle 80
  pattern 80
  window 80
return value
  from API function 350
reuse
  class 173

RGB (colour) 84
root
  scope 35
Rotate +45.0° 189
Rotate -45.0° 190
rotationAngle
  graphic shape 199
  picture 196
RoundRect 218
RTM
  for calculations 338
  stops 322
rtm 19–21, 24, 26, 27, 29, 45, 47, 48, 319, 320, 322, 333–334, 337, 346
  commands 338
  not running 321
  starts 321
  status 320
  stops 322
rtmName 320
ruler
  trend 204, 252, 363
run-time
  creation 22
  lookup 76
rvalue 60

S

save
  application 51
  library 136
Save (needed) 136, 182
Save As... 51, 136, 168, 170, **182**
Scale 1.1 190
Scale 1 / 1.1 190
scope 35
  message 343
  root 35
screen
  coordinate system 166
scrollbar
  adjust 355
Select 214
Select All 186
select(2)
  UNIX manual 324
Selected Properties 281
Selected Properties menu 201, 202, 203
Selected Properties...
  window 198
Selected properties... 198
Selected Trend menu 200
Selected Trend... 200
server
  database 335
set
  window colour 355
  window size 355
Set Focus 135
setting
  printer 183
shape 173
  constant line width 211
  primitive 165
Shape Flags 199
shifting bits 73
signature
  function 339
  logical 341
  physical 341
single
  statement 232
single-line
  comment 56
Slant 91
slice
  arc 221
snap 224
source
  input 323
source code
  motif
    motif
      source code 358
Spacing 91
special names 57
sprintf 345
square **66**, 67
state argument
  PfCrtmCrash 322
  PfCrtmStart 322
  PfCrtmStopp 322
statement 61
  break 64
  compound 62
  continue 64

contitional 63
declaration 62
do-while 64
expression 62
for 69
if-else 63
pTALK 232
single 232
switch 70
while 64, 70
strftime(3C) 202
Structs 143
subscribe
message 343
support
ProcSee 10
SWBus
control 29
switch
statement 70
symbolic names
library 135
symboltable **33**

T

Tdoc
documentation language 26
file 27
file format 26
format 28
template
class 173
test
drawing editor 179
testing
easy 43
equality 71
interval 70
Text 218
text
message 343
time
label 246
time label
trend 202, 254
timeInterval
trend grid 203
timeOffset
trend grid 203
timespan
trend 200
title
picture 172
tool
Circle 214
Circle Arc 215
Dialogue Area 215
Ellipse 216
Ellipse Arc 216
Image 217
instance 219
Keep 220
Line 217
Polygon 218
Rectangle 217
RoundRect 218
Select 214
Test 213
Text 218
Trend 219
tools menu 185, 211
tr 258
trend **200**–204, 219, 244
alignment 202, 203
appendLogBuffer **261**, 262
attribute 200, 260
autoScale 201
cancelLogBuffer **261**, 262
diagram 204, 237, 239, ??–253, 261, 363
externalLogger 243
getLogBuffer **263**, 264
grid 247, 253
depthPosition 253
noOfLines 254
timeInterval 254
timeOffset 254
historic 204
internalLogger **240**, 248, 251, 262
interval 202
log 260
logFrequency 201, 250
logger 237, **239**–241, 244–??, 244, ??–254
configuration 238
external 241
internal 240

lowerBand 201
lowerLimit 201, **250**, 251
new 201, 202, 203
newLogBuffer **261**, 262
offset 202, 203
orientation 200
presentation 201, 247, **250**, 251, **258**
rename 201, 202, 203
ruler 202, 204, 247, **252**, 363
rulerIsSelected 252
scroll 200
shape **247**–??
time label 202, 254
alignment 255
format 255
interval 255
offset 255
theFont 255
timeMaster 244
timespan 200
timeTickIntvl 250
timeVariable 244
trLog 200
updateMode 200
upperBand 201
upperLimit 201, **250**, 251
variable 201, **245**, 245, 250, 259
trend grid 202
depthPosition 203
noOfLines 203
timeInterval 203
timeOffset 203
Trend Properties
menu 201
Trend Properties menu 200, 202, 203
trendLog
definition 240
trendPlot 251
trendTime 244
trigger 152
pop-up menu 153
keyPressed 154
KeyRelease 154
trLog 201
attribute 201
trVar 260

U

UIMS 9, 17
Ungroup 188
uniqueness 35
Unselect All 186
unsubscribe
message 344
Update 192
update
variable 24
update modes
picture 223
upperBand
trend 201
upperLimit
trend 201
user
program 45
user program
dummi 233
utility program
p3sim 233

V

value
attribute 193
lCycle 262
location 60
read 60
variable 328
variable
attribute 144
PfBeginRecord 334
process 144
trend 201, 245
update 24
value 328
View
button 143
process variable 143
View Menu 218
View menu **83**, 134, 145, 155, 168–176, **192**
Attributes 144, **145**, **146**, 147
Delete 147
Edit 147
New 145
Classes **174**, 176
Delete 176

New 174
Open 175
Colours **83**
Create 83
Edit 83
Cursors
Create 110
Dialogues **152**, 155
Delete 155
Edit 155
New 152
Functions **148**, 150
Create 148
Delete 150
Edit 150
Graphic Attributes... **192**
Linestyles
Create 108
Patterns
Create 95
Picture properties... **195**
Pictures 168
Edit 171
New 168
Open 171
Remove 172
Selected properties... 198
Selected Trend... 200
Update 192
Windows 121
Delete 121
Edit 121
New 121
Zoom... 210
view menu
windows
edit 121
Viewer
Process 143
void
constructor 173, 176
destructor 173, 176
function 67

W

Weight 91
while
statement 64, 70
widget 357
Width 91
window 80
alignment 186, 187
Background Colour 229
Class Browser 219
decoration 125
define 30, 120
Delete Window 121
distribution 188
Edit Window 121
Editing Options 190, 223
Focus 134
focus 46
GED Focus 135
Graphic Attributes... 195
Instantiate Class 219
New Process 140
New Window 121
operation 172
Picture Properties... 197
Picture Settings 222
Selected Properties... 198
window colour
set 355
Window Editor 121, 122, 123, 124
Accept 123
advanced settings 125
attributes 124
cursor type 125
decoration 125
function 125
Parent 124
Restore 123
Type 123
window size
set 355
Windows menu 137, **220**
Focus Window
Remove 137
word
reserved 63
world
coordinate system 166, 167

X

XtAppMainLoop 353

Z

Zoom 192, **210**
zoom buttons menu 210
Zoom In 210, **210**
Zoom Out 210, **211**
Zoom... 210