Introduction to PxPlus

A Hands-On Training Guide for Developers

Notice

The information presented in this document is intended solely for training purposes. PVX Plus Technologies Ltd. may make changes to the information in this document at any time without notice.

Copyright © 2025 PVX Plus Technologies Ltd. (Ontario, Canada)
https://home.pvxplus.com

All rights reserved. Reproduction in whole or in part without permission is strictly prohibited.

PVX Plus, PVX Plus logos, and the PVX Plus product and service names mentioned herein are registered trademarks or trademarks of PVX Plus Technologies Ltd. or its affiliated entities.

All other products referred to in this document are trademarks or registered trademarks of their respective trademark holders.

Version 1.0
ISBN: 978-980-18-4638-3

Acknowledgement

PVX Plus Technologies Ltd. would like to express our sincere thanks and appreciation to Jean Hendrickx for his hard work, enthusiasm and support. As a developer, his eagerness to spread the word about the power and versatility of the PxPlus programming language led him to creating the first edition of this book. We are grateful to Jean for sharing it with us and are excited to use it as an educational tool to begin teaching others about the PxPlus environment.

Table of Contents

1. Introduction ... 8
PxPlus Development Suite ... 9
Introduction to Webster+ ... 12
Additional Resources ... 14
How to Obtain and Install PxPlus ... 15
2. PxPlus IDE: Integrated Development Environment ... 16
IDE Components ... 18
3. Graphical Applications Design ... 20
Introduction to Graphical Interfaces ... 21
NOMADS Panel (Screen) Designers ... 23
What does it mean to "Create a Panel"? ... 24
Exercise: Designing the First Panel ... 27
NOMADS Control Properties ... 52
Dynamic vs. Fixed Properties and Values ... 62
NOMADS Reserved Variables ... 64
4. Introduction to Files and Tables ... 66
Exercise: Defining a Table ... 71
5. Automatic Tools: File Maintenance and Query Manager ... 82
NOMADS File Maintenance ... 82
NOMADS Query Definition ... 96
Connecting Panels ... 106
6. Panels: Look and Feel ... 109
Exercise: Defining a Panel with Look and Feel ... 109
Custom Title Bars ... 113
Exercise: Adding a Custom Title Bar ... 114
Aids to Panel Editing and Design ... 122
Using the Bulk Edit Utility ... 132
7. NOMADS: Using Panel Controls ... 134
Understanding the Use of .CTL Variables in NOMADS ... 146
DROP_BOX Control: Drop-Down List Boxes ... 150
Numeric and Text (Alphanumeric) Variables ... 152
Some Literal Functions and Text Expressions or Substrings ... 155
Exercise: Manual Creation of a CRUD Panel ... 161

Using Folders on Panels ... 179
GRID Control ... 183
Exercise: Creating a GRID Control ... 184
GRID Object Properties ... 189
Exercise: Defining a Table with Sample Data ... 192
8. Introduction to Data Sources ... 195
Defining a View ... 204
Exercise: Defining a View ... 204
9. The Report Writer ... 214
Exercise: Using the Report Wizard ... 215
Exercise: Defining a Calculated Field (Numeric) ... 219
10. BUTTON, CHECK_BOX, RADIO_BUTTON: A Deeper Explanation ... 232
BUTTON Control: Other Functions and Attributes ... 233
RADIO_BUTTON Control: Other Functions and Attributes ... 235
Exercise: Creating RADIO_BUTTON Controls ... 238
CHECK_BOX Control: Other Functions and Attributes ... 241
Exercise: Creating CHECK_BOX Controls ... 242
Using a RADIO_BUTTON or CHECK_BOX Control in a Program ... 243
11. Files and Tables: An In-Depth View ... 245
Types of Files in PxPlus ... 247
READ/WRITE Clauses and Operations ... 250
Using a Table with an Embedded Data Dictionary ... 252
Using the Structure SELECT FROM..NEXT RECORD ... 253
Erasing Records from a Table ... 256
How to Use PxPlus Special Devices ... 257
Using Special Access TAGS ... 273
Using List-Type Controls (LIST_BOX, DROP_BOX) ... 276
TREE_VIEW Control ... 287
Exercise: Creating a TREE_VIEW Control ... 287
GRID Control ... 291
Exercise: Defining a Grid Control ... 291
12. Elements of Programming: Building Programs ... 300
Creating a PxPlus Program ... 300
Routines and Branching ... 303
Loops and Repetitive Cycles ... 305

PxPlus Program Components 311
Introduction to Language Directives (Commands) 313
Using the Built-In Functions 332
System Variables 340
Introduction to Mnemonics 344
Using User Parameters in PxPlus 346
13. NOMADS: Other Functions and Tools 351
Themes and Visual Classes 351
Exercise: Creating a Visual Class 353
Exercise: Applying a Visual Class 354
Data Classes 357
Exercise: Creating a Data Class 358
User-Defined Controls (CTLs) 361
Using Data Dictionary Fields 363
Exercise: Creating a Data Element Control 364
Drag and Drop Functionality 365
Exercise: Using Drag and Drop 365
Dependency Definitions 368
Floating User Tips (Floating Help) 371
Defining a MENU_BAR and POPUP MENU 374
Exercise: Defining a MENU_BAR 375
Exercise: Defining a Menu Link 377
Exercise: Assigning a Popup Menu 383
MULTI_LINE Control: Calendar and Spinner Control Query Types 387
Exercise: Defining a Calendar Control 390
MULTI_LINE Control: Other Query Types 392
What are Link Files? 398
Exercise: Creating a Link File 399
Embedded Procedures (Inside Tables) 402
START_UP: Automatic Execution Program 405
PxPlus Error Handling 406
NOMADS: Using Embedded Panels 409
Exercise: Creating an Embedded Panel 410
Introduction to Object-Oriented Programming (OOP) 414
Exercise: Object-Oriented Programming 415

Using the LIKE Command to Inherit Other Classes ... 421
The Report Designer: An In-Depth View ... 423
Exercise: Creating a Report with Report Designer ... 426
Exercise: Defining a Group Function ... 430
Exercise: Creating a Report (from the beginning) ... 446
NOMADS Security System ... 450
Exercise: Using NOMADS Security ... 453
Using Passwords to Protect Programs and Tables ... 456
14. The Final Steps ... 459
Publishing Our System ... 459
PxPlus Working in Web Mode ... 461
PxPlus Working in Network Mode ... 462
Handling Programs and Files ... 468
Fixing Errors, Debugging and Testing Programs ... 470
15. Appendix ... 474
Error Codes in PxPlus ... 474
Examples of PxPlus Programs and Useful Routines ... 475
Data Manipulation Routines ... 476
A Complete Routine for Table/File Handling ... 482
Exercise: Sorting Data in a Table ... 482
How to Read a Table from Last to First Record (Reverse Order) ... 488
16. Conclusion ... 489

1. Introduction

Businesses and companies must track a large amount of information: purchases, sales, customers, products, transactions, dates and quantities, etc. The success or failure of a commercial or business management application depends on the approach given to the processing of information, the data it records, the controls and guidelines it applies to users, and how versatile, simple and fast the information can be obtained with the registered data. Although there are many excellent languages today, some offer a series of tools that facilitate this management and allow the programmer to obtain the best control over them, preserving integrity while protecting sensitive information from unauthorized eyes.

With more than 40 years of history, PxPlus has evolved to be able to provide its users with a complete set of tools to save, sort, calculate, filter, modify and display that information, while taking care of these tasks in a transparent, fast and safe way and offering compatibility and flexibility to allow integration with other tools. PxPlus users appreciate how complete and robust the language has become. Each year, a new version of PxPlus is released to ensure the language meets all modern day enhancements and security protocols.

PxPlus is a language created for developing commercial applications with an emphasis on data manipulation (amounts, dates, quantities, etc.) and with a focus on managing large volumes of data. It has evolved to include an ecosystem that manages, stores and processes large volumes of data in a simple and efficient way, along with other tools for the manipulation and presentation of that information (Data Entry, Queries, Reports, etc.).

This book has a rather modest intention: Try to teach you quickly and directly the main aspects of this tool so that you can be productive in a short time. To do this, it has been necessary to "overlook" basic aspects, which are not necessary to start working. We will do our best so that, once we are fully using the tool, we can incorporate other concepts and strengthen the theoretical framework and foundations of the language. Details on the more advanced toolsets will be made available in future versions of the book.

To better understand and take advantage of this book, it is recommended to have basic programming concepts, know about system analysis and design, and, preferably, have knowledge of some other programming language or database management.

PxPlus is a tool that is compatible with different environments (Windows, Linux and Web) and offers several "presentations" or tool configurations. For now, we are going to work with one environment, MS Windows and a PxPlus Web version, but most likely you can follow the content of this book with any other.

We hope you enjoy learning about this powerful language and toolset - built from a strong history and evolving with a bright future!

PxPlus Development Suite

The PxPlus Development Suite is a programming language and toolset. Its purpose is to allow software developers to easily build, enhance and maintain business applications.

The PxPlus Development Suite includes the following features:

- Cross-platform (Windows/Linux/Mac)
- NOMADS toolset for developing application screens (Windows and Web-based user interfaces)
- Data Dictionary with Data Classes
- PxPlus data files (optimized for business data)
- Built-in access external databases (i.e. MySQL, SQL Server, any via ODBC)
- Program editors for creating PxPlus programs (including Visual Studio Code Extension)
- Allows external applications to access PxPlus data via the PxPlus SQL ODBC Driver
- Can run stand alone, client/server architecture or in a browser
- Report Writer for designing and generating custom reports
- Query System for generating and displaying lists of data records for selection
- Can create and modify PDFs
- Comprehensive Web support: FTP/SFTP/TCP/SSL/TLS/Web Services
- Google Workspace integration
- PxPlus IDE, a project-based, integrated PxPlus development environment

Let's look at some of the main features.

PxPlus Language

The **PxPlus Language** itself is handled by the ***pxplus.exe*** program. PxPlus programs are executed using the ***pxplus.exe*** program. If a developer has written a program in PxPlus, the user needs PxPlus installed to run the program.

To write and modify programs, PxPlus has three program editors: the Visual Studio Code Extension, Integrated Toolkit (*IT) and ED+.

- **Visual Studio Code Extension** is the most recent editor integration.

  Refer to the tutorial **How to Set Up and Use the PxPlus Visual Studio Code Extension** in the PxPlus Help documentation.

- **Integrated Toolkit (*IT)** is the original program editor.

- **ED+** is a program editor that supports running on the Web and provides additional features, such as autocomplete.

Data Dictionary

The **Data Dictionary** is a program that is used to create and modify PxPlus data files, including external databases. It allows you to define what fields a file has, the type of data in each field, the length of each field and how that data is validated and formatted.

You can also define the file's keys, which determine how data is sorted in the file. **Example:** You may define the Customer Number field as a key as generally you do lookups based on the Customer Number.

The data dictionary allows you to define data classes, which are a common set of attributes for fields.

Example: A Currency data class can be used for fields that store a value of money. These fields will all be the same across your application so you can define the class once and use it when creating new files.

With a data dictionary, you can also define file maintenance screens, links between files, queries based on files, views based on files, and import/export to external databases.

NOMADS

NOMADS is a toolset used for creating Windows and Web-based user interfaces for your programs. The main component of this is the drag and drop panel (screen) designer, which allows you to place buttons, text fields, and other controls onto screens and then define how they look and behave. Generally, you specify what program runs when, for example, a button is pressed.

iNomads

iNomads is a toolset used for creating a Web user interface for your programs. It can take screens that were built using NOMADS and run them via a Web browser. This is great for taking an existing application built with NOMADS and bringing it over to the Web without having to build a new user interface.

The look and feel of iNomads can be changed using iNomads templates, which gives you control of how the Web version of your application looks.

Webster+

Webster+ is a toolset used for creating a true Web user interface for your programs. It allows you to create HTML Web pages that can hook into PxPlus programs, file maintenance, queries, the data dictionary, etc.

The Webster+ system includes a user management/login system, a menu/navigation system, administrative tools, file browser, program editor, and HTML editor.

WindX

PxPlus can run in a client/server architecture, which we call **WindX**. This is where the application is on a central server, and you have client computers that connect to it and run the application on the server. This allows them to have one license with many users. The setup on each client is minimal and involves just pointing it at the server.

It also allows the server to be running on Linux; however, the clients can be Windows and can use a Windows user interface. Any user interface commands are passed on to the Windows client to execute locally.

The modern version of WindX is called Simple Client Server (PxPlus Simple CS) and SSH.

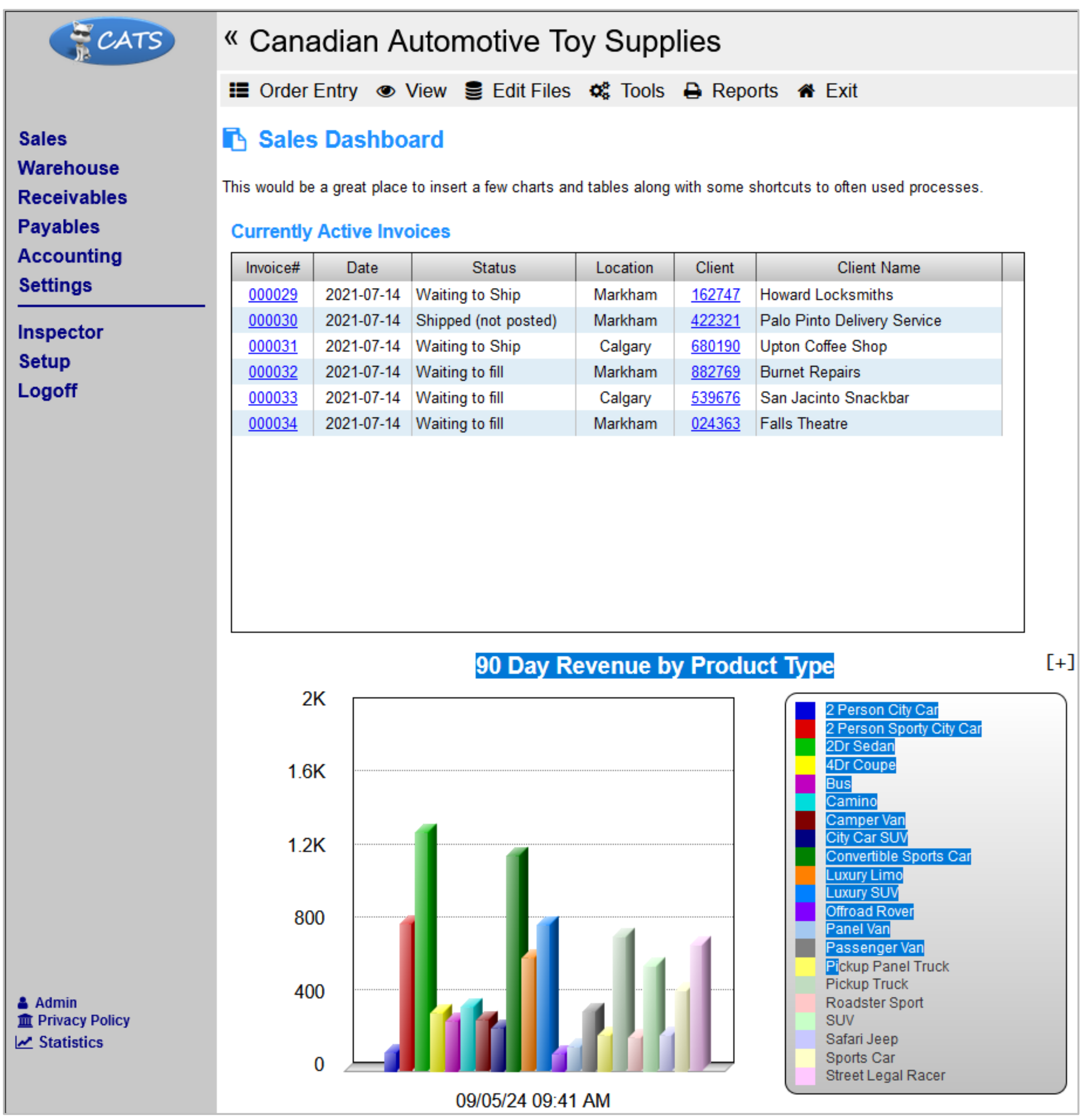
Some legacy versions of WindX are also supported: NTHost/NTSlave, Application Server and Telnet.

With SSH, you do not need to run a server. You just use SSH to connect to the remote machine and run PxPlus. PxPlus will detect this and handle the routing of user interface commands.

Introduction to Webster+

Webster+ is a toolset designed to create browser-only applications. It makes the generation of Web pages and the setup of Web sites easier. It uses a standard REST-style interface where every exchange between the workstation/browser and the host is standalone. Webster+ comes complete with setup/configuration tools, a menu system, a security system, etc.

Example: Webster+ HTML page



Invoice#	Date	Status	Location	Client	Client Name
000029	2021-07-14	Waiting to Ship	Markham	162747	Howard Locksmiths
000030	2021-07-14	Shipped (not posted)	Markham	422321	Palo Pinto Delivery Service
000031	2021-07-14	Waiting to Ship	Calgary	680190	Upton Coffee Shop
000032	2021-07-14	Waiting to fill	Markham	882769	Burnet Repairs
000033	2021-07-14	Waiting to fill	Calgary	539676	San Jacinto Snackbar
000034	2021-07-14	Waiting to fill	Markham	024363	Falls Theatre

Webster+ provides the following functionality:

- Ability to create dynamic HTML 5 compatible pages through the use of **Short Codes**
- Site management tools to handle setup/configuration, user registration, password control and security
- Optional two-step authorization using SMS or Email verifications
- Menu system for selection of panels and security
- Integration with **NOMADS Query** and **Report Writer**
- Integration with **File Maintenance Generator**, which can be used to create Webster+ HTML pages
- Generic and custom templates for page rendering
- **Apache**, **EZWeb** and **IIS** compatible
- Provides ***both*** full page loads or Ajax-style page updates

Use these links to access additional Webster+ information:

Source	Description
PxPlus on the Web	Opens a page on the PVX Plus Knowledge Center called **PxPlus on the Web**, which includes an overview of Webster+.
Webster+ Video Series	"How To" tutorial videos that include setting up Webster+, creating a Webster+ based file maintenance page, using queries and running filtered reports under Webster+.
Test Drive Webster+	Webster+ Demo application that you can use for practice and experimentation.

Additional Resources

Use these links to access additional PxPlus training materials, tutorials and "How To" videos:

Source	Description
PVX Plus Knowledge Center	Resource center for additional PxPlus learning on various topics, including: • PxPlus Language • PxPlus IDE • Data Dictionary and Data Classes • Databases • NOMADS • ODBC • Query Subsystem and Report Writer • Security • Thin Client/WindX
PxPlus User Forum	Forum for interacting with the PxPlus community to allow you to share knowledge and code samples, exchange ideas, ask for help and feedback
PxPlus Online Help	PxPlus online documentation
How To Tutorials	Step-by-step instructions on how to use some of the development tools provided in PxPlus

How to Obtain and Install PxPlus

PxPlus is a product developed by **PVX Plus Technologies Ltd.**, a Canadian company with extensive experience in the development of programming languages. The language has been evolving (including changing name, appearance and owners) but maintaining backward compatibility. The language has a rich history and will continue to evolve with new enhancements to meet ever-changing needs for years to come!

PxPlus operates under an intellectual property license (it is not open source or free). It is activated with a license key and sold at the user level. A Demo version is available for a 30-day trial with a maximum 5,000 records. The Continuous Care Program provides access to all the latest features, enhancements and security protocols.

PxPlus can be downloaded from the PVX Plus **Downloads** page.

Different versions of this product are available: a Base version, a Professional license, and a Web license, which is the most complete of all. For a complete list of the components included in each version, visit the PVX Plus **Development Environment** page.

To learn more about how to install PxPlus and how PxPlus works in multi-user and client-server environments, refer to the section **The Final Steps** within this book for information on how to configure PxPlus in those environments.

Note: For the purposes of this book, the installed license is a Demo mode of PxPlus on an MS Windows machine.

It is recommended that you read the book and practice, explore and experiment with the information presented here so that knowledge is consolidated in a faster and more efficient way. It is also possible to simply read the book without having a computer, but the maximum potential will be obtained by having both items on hand.

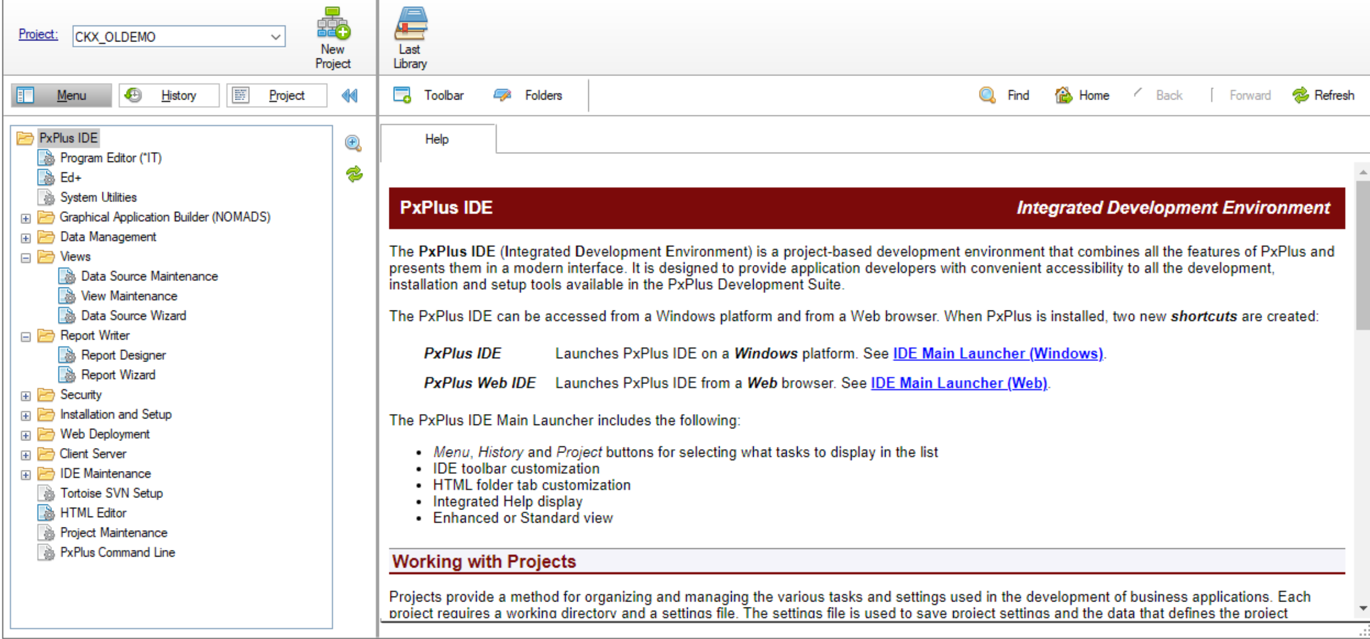
For licensing questions, please contact **sales@pvxplus.com**.

2. PxPlus IDE: Integrated Development Environment

PxPlus groups most of its functions in a convenient menu that offers an **Integrated Development Environment** (or **IDE**), which is an environment that offers a series of tools and facilities from the design and creation of a simple table to the creation, organization and maintenance of projects, such as payroll or medical appointment control.

The PxPlus IDE can be launched on a Windows platform, and there is also a version of the PxPlus IDE that runs in a Web browser called the **PxPlus Web IDE**.

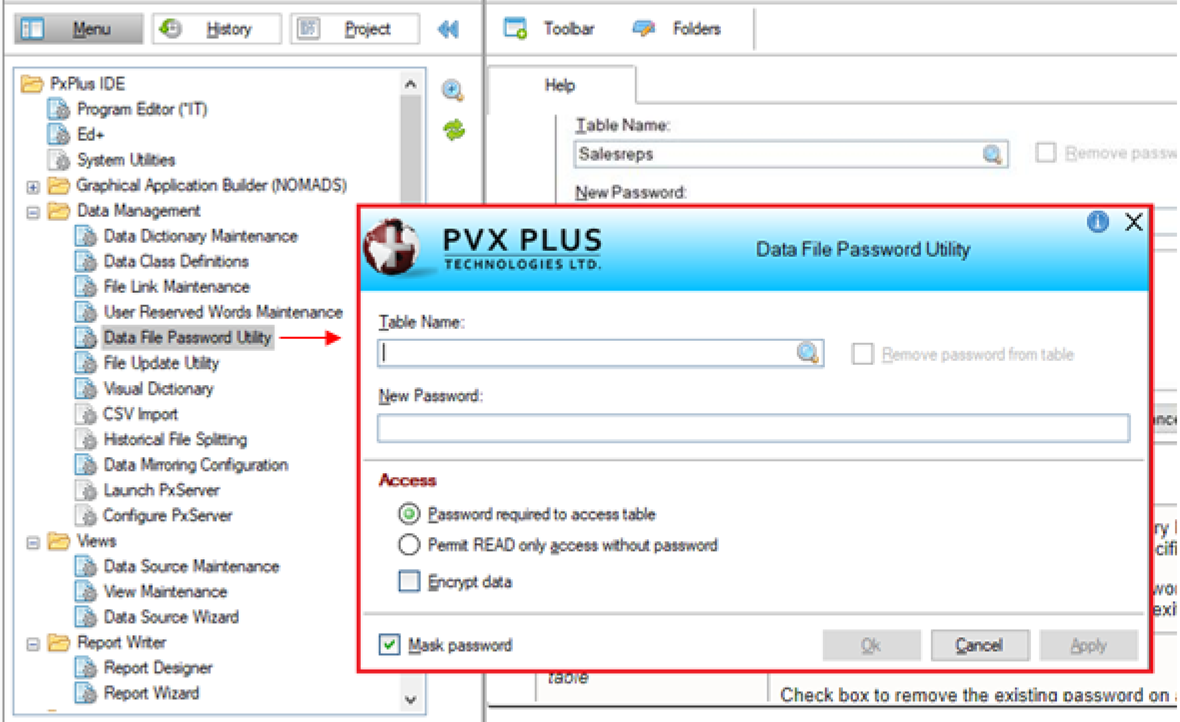
Example: This shows the PxPlus IDE on a Windows platform:



It has two main parts:

- The left side shows an expandable tree menu with access to all the various tasks of the PxPlus environment.
- The right side shows information regarding a selected option. If we double click on an option in the tree menu, a window will open with that tool or function. An example is shown on the next page.

Example: When the **Data File Password Utility** task is selected from the tree menu, the task window is launched. The Help information for this task is displayed on the right.



It is important to note that some options are "repeated" to offer a logical way of operating, and in other cases, to group different functions in the same section. In its Help system, PxPlus offers a summary of the different ways to RUN or CALL that tool.

In some cases, it is also possible to have two tools that offer similar functionality either because they are compatible with each other or because they have evolved to offer the same functionality. An example of this is the Report Designer and Report Wizard.

IDE Components

The PxPlus IDE is dynamic, depending on several factors. The number and order of the tools shown can change, and it is also configurable. For now, we will only look at the main components without delving into details.

Projects

Projects provide a method for organizing and managing the various tasks (such as objects, programs files) and settings used in PxPlus for the development of business applications. Projects make it easy to access all the related tasks through the PxPlus IDE.

Project data is stored in files outside of the PxPlus installation and therefore is not affected when PxPlus is updated.

The IDE **Projects** menu provides options for creating, editing, deleting, copying and maintaining projects.

Refer to these tutorials in the PxPlus Help documentation: **How to Create and Edit a Project**, **How to Copy a Project** and **How to Delete a Project**.

Program Editors: Create and Modify PxPlus Programs

PxPlus offers several program editors, such as the **Visual Studio Code Extension**, the **Integrated Toolkit (*IT)** and the **ED+ Program Editor**.

Refer to the tutorial **How to Set Up and Use the PxPlus Visual Studio Code Extension** in the PxPlus Help documentation.

Graphic Utilities: File Maintenance, Settings, Various Tools

This set of tools allows you to manipulate files, programs and directories (create, modify, view, etc.), manage images, fonts and colors, among others.

NOMADS, a RAD Environment: Graphic Design of Panels and Windows

NOMADS is a graphical application designer, a tool to visually create (without having to program anything) shapes, panels or windows. We could say that this tool serves to create the interactive part of the applications. It offers a large number of tools and functions that we will see in the next chapters.

Toolkit Themes: Enhance the PxPlus Development Suite Experience

Toolkit Themes are an easy way to enhance the user experience of the PxPlus development suite. The **Toolkit Theme** option allows you to apply one of the predefined PxPlus themes: **Dark*, **Light* or **None*. Toolkit themes are applied on a project by project basis. This means that if you have more than one project, a different toolkit theme can be set for each project.

Refer to the tutorial **How to Set a Toolkit Theme** in the PxPlus Help documentation.

You can also create your own copy of the PxPlus themes and modify them for use in your application.

Refer to the tutorial **How to Copy a Theme** in the PxPlus Help documentation.

Data Management: Modeling and Maintenance of Tables, Files and Databases

Perhaps this is the first tool to know. Here we can define the storage structure of our data, sometimes called data definition or modeling. It is where we define and create the way our information will be stored. This can be from a simple table to several databases and their relationships. PxPlus allows you to go at your own pace, as we will see very soon. A popular concept is the Data Dictionary because it is where the data, its characteristics, structures, relationships, etc. are defined.

Views: Generation and Maintenance of Views (Data Query)

Once the storage structures are defined, PxPlus allows you to "show" them automatically (without having to program even a line of code) and establish data, relationships, ordering, filtering, etc. Once these Views are defined, they can be used transparently in your systems.

Report Writer: Report Generator and Graphic Designer

An important requirement in many commercial applications is the dumping of information in an unalterable format, such as paper or a readable file, such as PDF. PxPlus allows you to draw complete reports, again without the need for programming, and then call or execute them from our application.

Note: When the full PxPlus suite is installed, about 20 icons and tools are installed. We will see some of them in the course of this book. For now, let's focus our attention on the "essentials".

Unlike other languages, the logical start of PxPlus is not to go to the language. Depending on our needs, we can start by defining a new project or go directly to defining a table or database. We won't see "compilers", "linkers" or anything like that either. Once you have learned the simple - but very efficient and complete - approach of PxPlus, you will appreciate the way the environment works.

You will also see that programming or writing code is one of the many facets of application production. For now, let's continue getting to know the tool a little more.

3. Graphical Applications Design

Application Analysis and Design – A Summary

When we focus on the development of an application, we go through the initial step of knowing the problem we must solve and, at that moment of analysis, we must begin the definition of the storage structures. Then, we must make the panels or forms that allow the user to interact with the system. Normally, there are loading or maintenance panels (to enter information), query panels (to see the content of the loaded data), a calculation or update process and one or more reports.

An outline of what the development of an application would be like in PxPlus would be similar to:

1. Create a **Library** to group our work and keep it organized.
2. Use the **Data Dictionary** to define files (tables, databases, storage structures), its elements (fields), add indexes (keys) and establish the structures to store our data.
3. Through **NOMADS**, draw the data-loading forms, create the system menus and establish the relationships between them. Each panel or form incorporates controls in the form of buttons, text, text entry boxes, lists, drop-down lists, etc. These controls are defined with actions based on events. **Example:** An event would be pressing a button and the associated action could be calling another panel or printing a report.

   Likewise, these controls have properties and attributes to fine tune the behavior and appearance, such as colors, font, size and position, in addition to other attributes.
4. Through NOMADS or Views, we define the forms for **Queries** and **Views**. A Query is a prefixed query where the information is extracted, sorted and filtered automatically by NOMADS by specifying our criteria without having to create a program for it. For a View, a data source is originally defined, which can be anything as simple as a table, to an object containing multiple tables in distributed databases.
5. With the **Report Writer**, we can define the output reports, which we can take to paper, view them on the screen (in the Viewer) or take them to a portable format (such as PDF).

Introduction to Graphical Interfaces

One of the most important (although often overlooked) components when creating graphical applications is the presentation - the user interface, which is what the user handles and interacts with. An unattractive presentation with many cluttered options will not be as well received as well as a clean, orderly and logically grouped work environment.

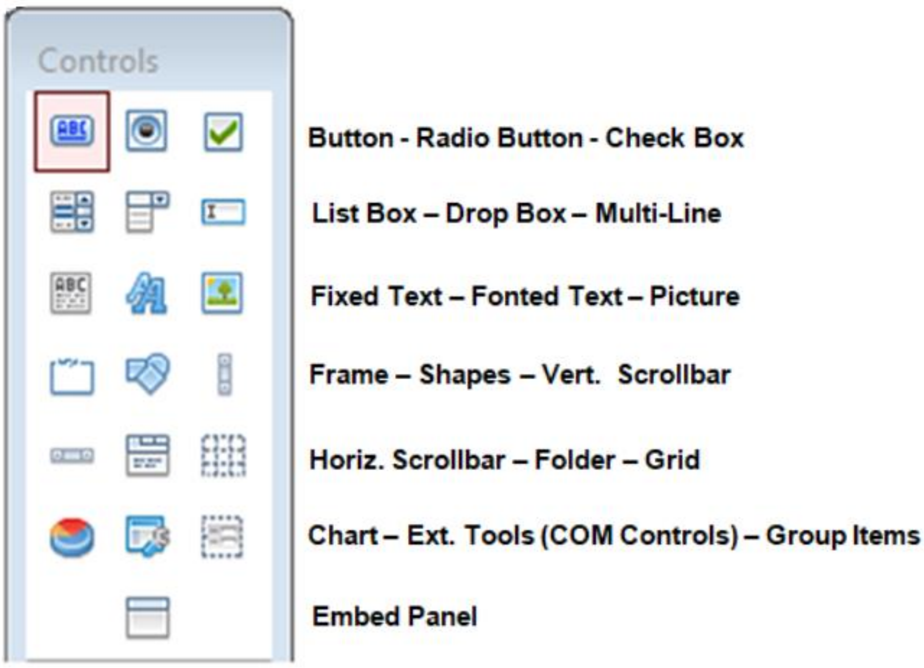
That design must go beyond simply placing a series of controls on a panel or window. The correct use of the controls, typography, color selection, functionality and attractive grouping will make part of the difference in user acceptance.

PxPlus offers the right tool for designing graphical applications called **NOMADS** (**N**on-procedural **O**bject **M**odule **A**pplication **D**evelopment **S**ystem) where you can graphically and intuitively create panels, windows or shapes without having to write a line of code.

A panel or form in PxPlus is part of a library, which is a repository of panels and objects composed of regular panels, queries, file maintenance and more. Any program can call panels from different libraries.

Within a panel, there will be a series of controls: buttons, text entry boxes, selection boxes, images, etc. Behind an action as simple as dragging a button hides a powerful and flexible object-oriented interface (OOP). PxPlus allows the user to ignore the events that he/she does not need and only use those that are needed. That is, if only one of the 20 functions or attributes of a control is needed, we can focus on it and ignore the others.

By default, NOMADS offers the necessary **Controls** to cover most programmers' needs, but it is possible to add new controls or modify existing ones. We will look at this in more detail later.



Behind many of these controls, there are variations: various types of lists, various forms, etc.

For now, let's briefly review the main functions of each control:

BUTTON	Executes an action when clicked with the mouse or when the ENTER key is pressed.
RADIO BUTTON	Allows you to select one and only one option among several, and all of them are mutually exclusive. **Example:** Cash/Visa/MasterCard
CHECK BOX	It is an indicator that can be turned On or Off, known as a flag, and indicates an optional condition. **Example:** Include Freight?
LIST BOX	Allows you to display a set of data and optionally select any of them. It has several formats or presentations.
DROP BOX	Similar to a LIST BOX but takes up less space on the screen, and clicking it opens or displays the content for selection.
MULTI-LINE	Used to enter information.
FIXED TEXT **FONTED TEXT**	Shows text on the screen. The main difference is that FIXED TEXT is sensitive to mouse clicks, and FONTED TEXT is much more versatile and complete.
PICTURE	Shows an image or logo on the panel.
FRAME	Square-shaped decoration. It can have or simulate a 3D, Recessed or Raised look.
SHAPES	Geometric shapes, such as Circle, Polygon, Line, etc.
VERT. SCROLLBAR	Controls vertical scrolling.
HORIZ. SCROLLBAR	Controls horizontal scrolling.
FOLDER	Groups several panels to give them the appearance of folders or tabs.
GRID	Spreadsheet-type control with multiple cells. It is perhaps the most complete control.
CHART	Presentation of bar graphs, lines, columns, etc.
EXT. TOOLS	Allows you to incorporate objects or controls from other applications or manufacturers.
GROUP ITEMS	Used to select multiple controls. You can also click to select the first one and then [**Shift Click**] to select the next ones.
EMBED PANEL	Allows you to define a space to insert an external panel automatically; that is, at execution time, PxPlus will embed the other panel (and its functionality) in the panel that is being executed.

Refer to **Types of Controls** in the PxPlus Help documentation.

NOMADS Panel (Screen) Designers

The NOMADS panel (screen) designer is the primary work area for drawing, defining and maintaining the graphical user interface components. The panel designer comes in three different versions or styles:

- **Folder Style** *(Used throughout this book)*: This presents a window with a layered file folder format and tabbed sub-panels to display the design properties of each panel component.

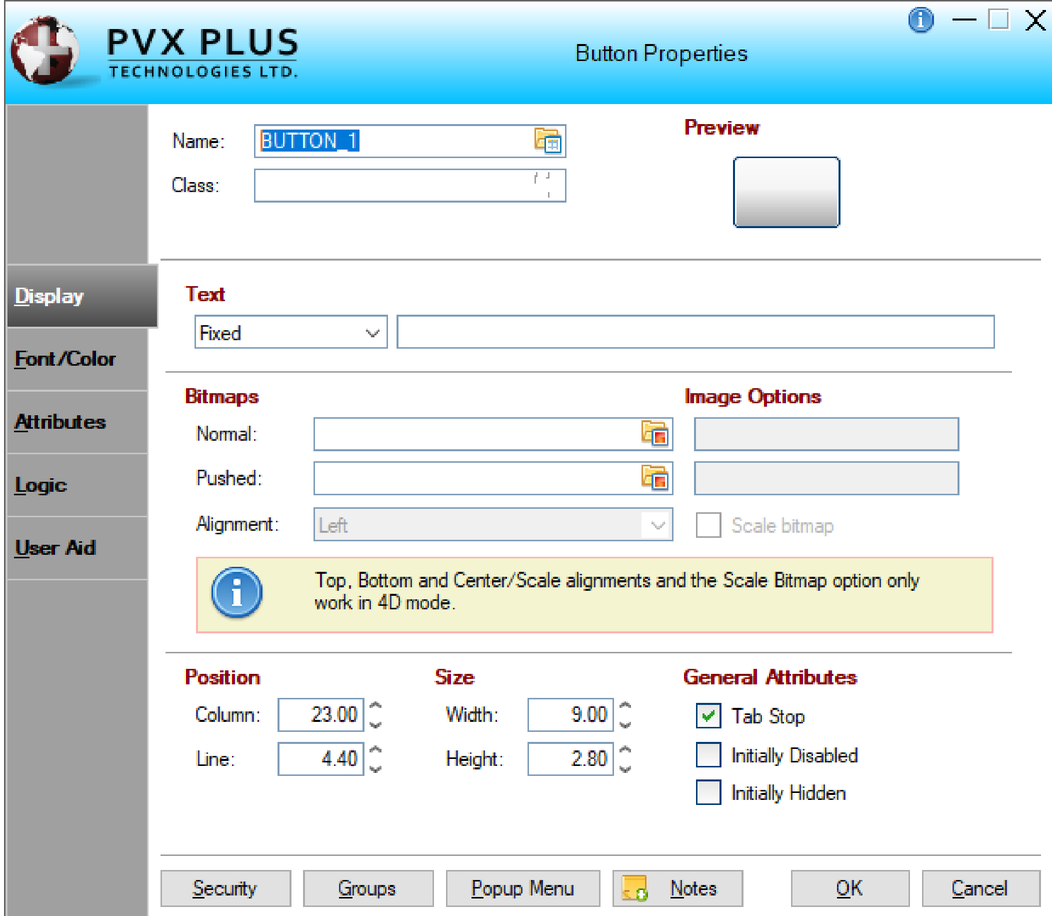
- **Nomads+ Toolbar**: This consists of two main components: the NOMADS+ Toolbar (comprised of a menu bar, controls toolbar, ribbon bar and a control grid), and a separate Design panel.

- **Property Sheets** *(Legacy designer)*

What does it mean to "Create a Panel"?

Once you have the form or panel, NOMADS allows you to drag and paste controls (Buttons, List Boxes, Multi-Lines, etc.) in the position and size we want. Once each control is created, NOMADS opens a properties window where we can specify a large amount of information, from entering the name of the control (**Example**: EXIT_BUTTON), its size and position, to associating a logic or event. (**Example**: If the button is pressed, then run the CALCULATOR application.)

To create a new control, we must first select the control in the **Controls** box on the left. Position the mouse pointer inside the panel where we want to place the control and then drag the mouse and draw the outline of the control. Upon completion, a panel with the control properties will open, similar to this:



Some controls do not require renaming, and NOMADS will assign a name automatically.

Example:

IMAGE_1, IMAGE_2, etc.

For other controls, it is convenient (and even important) that the names are appropriate, since in our programs and routines, we must use those names.

Example:

IF BOX_TEXT01$=CODE$ THEN PERFORM "INVENTORY;ENTRY"

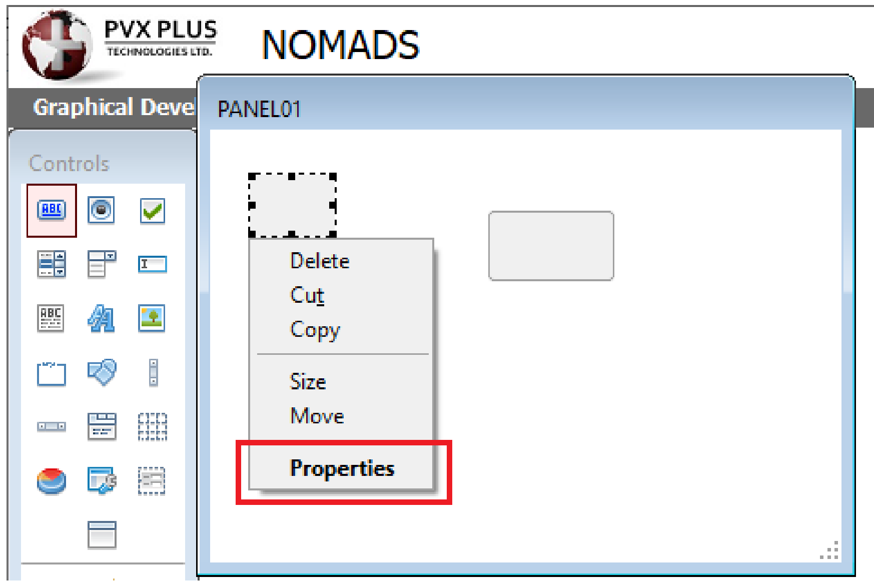
Note: Each control ***must*** have a ***unique*** name. It is possible to change the name of a control after it has been created.

In reality, internally the system identifies the controls by a numeric CTL (**C**on**T**ro**L**) value, which will allow precise identification. The value of the control will be its name followed by the prefix ".CTL".

Example:

If a control is called **BUTTON_EXIT**, its control value will be the variable **BUTTON_EXIT.CTL**. Operations and comparisons can be performed on it, such as **Disable BUTTON_EXIT.CTL** to disable the control. These operations will be seen in detail later.

Once the controls are positioned on the panel, we can make adjustments to each control simply by double clicking or by selecting the control and right clicking, which opens a menu where we select the [**Properties**] option.



This action will bring back the control's properties window.

A great feature in NOMADS is that the control properties window is the same for all controls, sharing a similar structure and operation for all controls. Although not all controls will have the same properties or functions, the common properties will be the same across all controls.

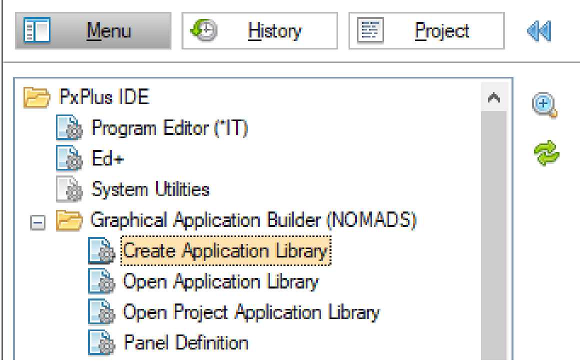
Example:

On the left, we have the two tabs (**Display** and **Font/Color**) from the properties window of the Fixed/Fonted Text control. On the right, we have the same two tabs, plus more, from the properties window of the Grid control.

Exercise: Designing the First Panel

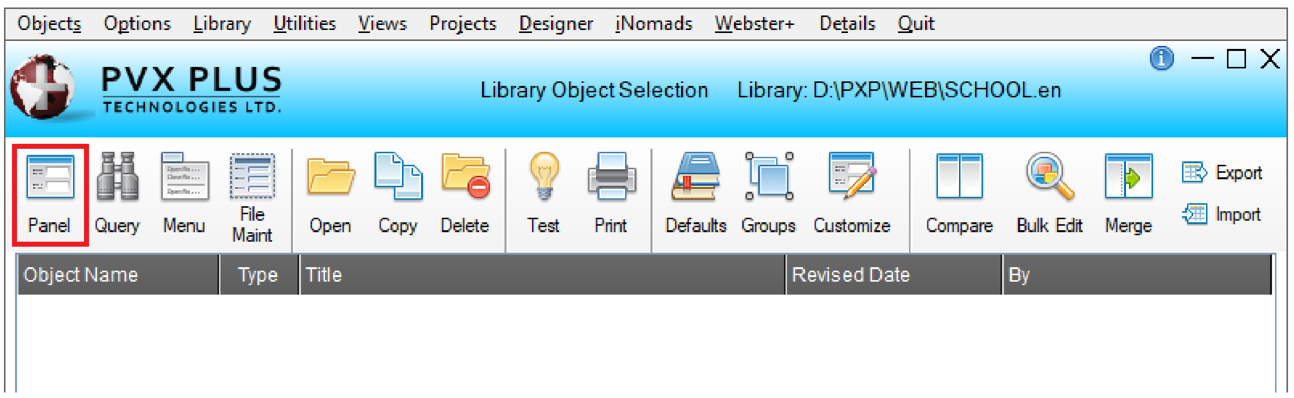
We are going to create a panel for the entry of Student data. Basically, this panel will have the labels of the fields or elements (we will only use Code, Name, Age), the corresponding text entry boxes (Multi-Lines) for entering the information and a couple of buttons to save the information and to finish.

From the main menu of the PxPlus IDE, we open the **Graphical Application Builder (NOMADS)** category and double click on the **Create Application Library** task:



We are going to create an application library called **SCHOOL**. By default, NOMADS will assign the extension **.EN**; that is, the full name will be **SCHOOL.EN**.

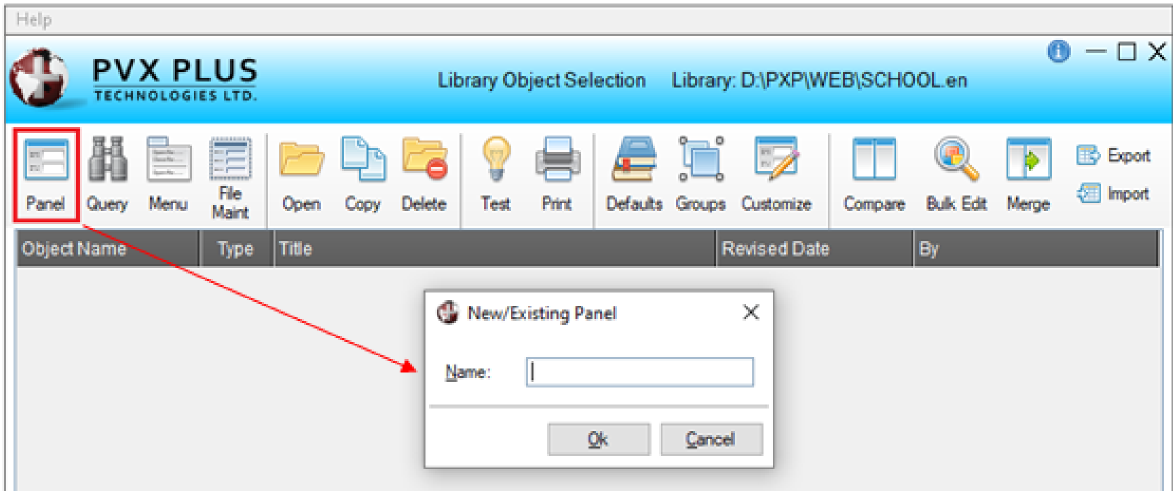
The NOMADS **Library Object Selection** window will display, similar to this:



As this is a new library, we will not have any panels defined. As we create them, they will appear at the bottom of the panel.

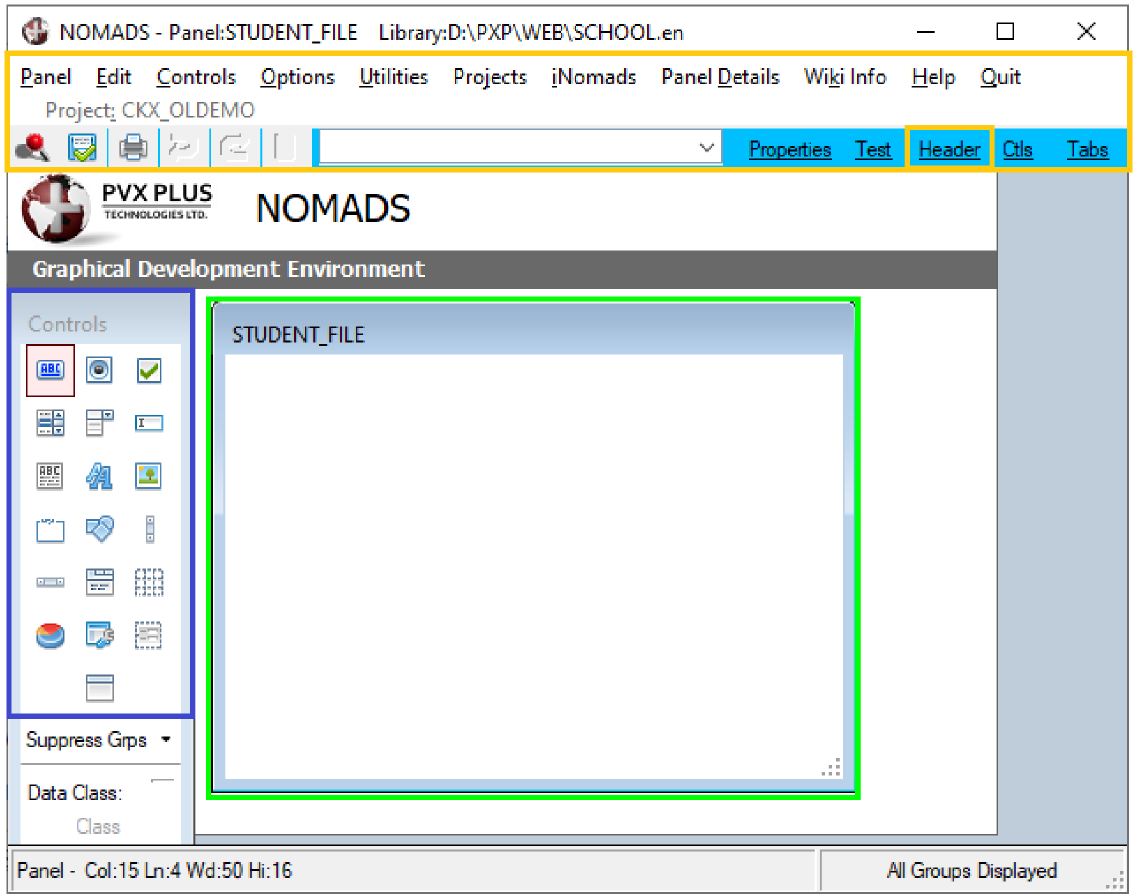
For now, we must click the [**Panel**] button (marked in red above) at the top left of the **Library Object Selection** window.

A dialog box will appear for entering the name of a panel, which may be new or already exist. As this library is new, there are no panels.



We enter the panel name: **STUDENT_FILE**.

We are now in the NOMADS **Panel Designer** window (shown below). We have a top menu (marked in yellow), a panel with controls (marked in blue), and an editing area (marked in green) where the panel we are creating (**STUDENT_FILE**) is located.

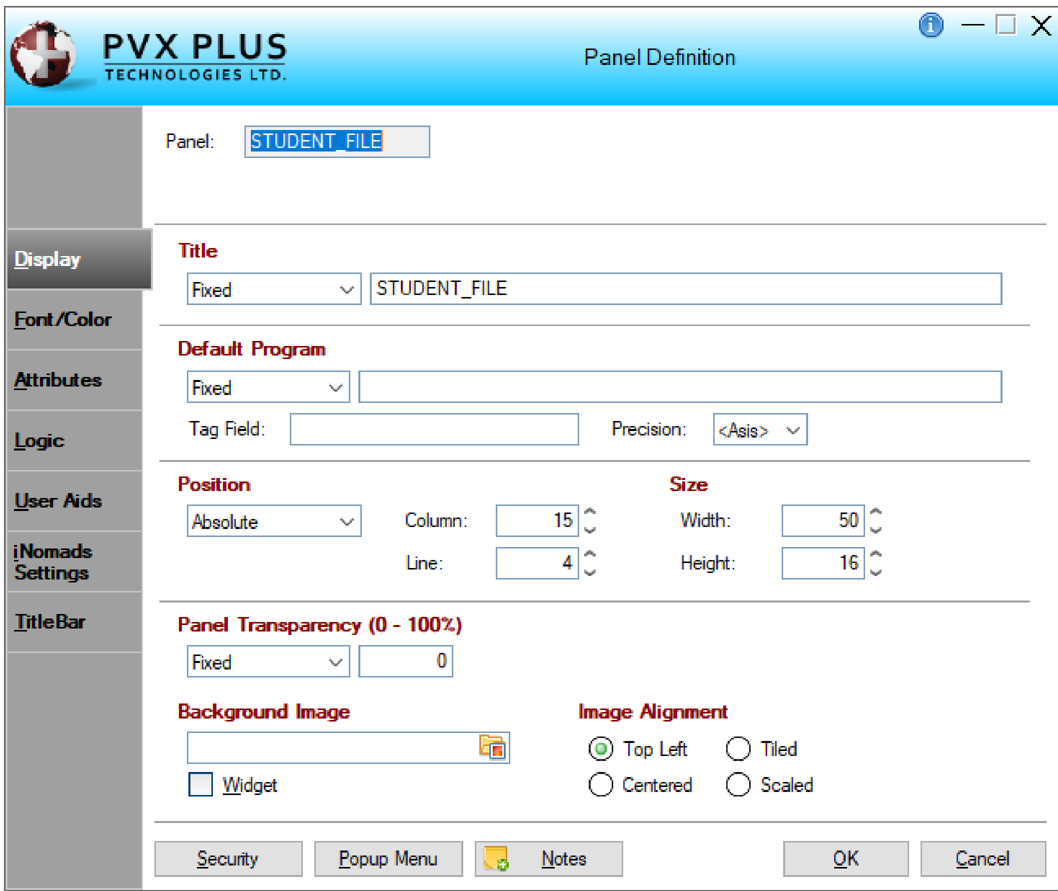


Note: These colored boxes are for ***reference purposes only*** and are ***not*** part of the NOMADS environment.

The first thing we must do is select the [**Header**] option located in the top menu (marked in yellow). This opens the **Panel Definition** window, which is used to modify the properties of the panel itself; that is, the window or shape, its colors, dimensions, attributes, etc.

Remember that ***all*** screens for modifying the properties of a control in NOMADS use a similar structure - a series of options or tabs on the left side: **Display**, **Font/Color**, **Attributes**, **Logic**, **User Aids**, **iNomads Settings** and **TitleBar**.

Note: Not all controls have the same properties, and each control will display unique properties, such as the last two, **iNomads Settings** and **TitleBar**.



The first tab, **Display**, allows you to adjust or change the following values:

Title: Title of the panel (in this case, **STUDENT_FILE**).

Default Program: We can define a program that will be in charge of managing the logic of this panel, and thus, when we need to define a routine, we only have to specify the name of the routine (**Example:** ROUTINE) instead of specifying both (**Example:** PROGRAM; ROUTINE).

Position/Size: Window location, coordinates and size. It is possible to modify the latter by dragging the edges of the window.

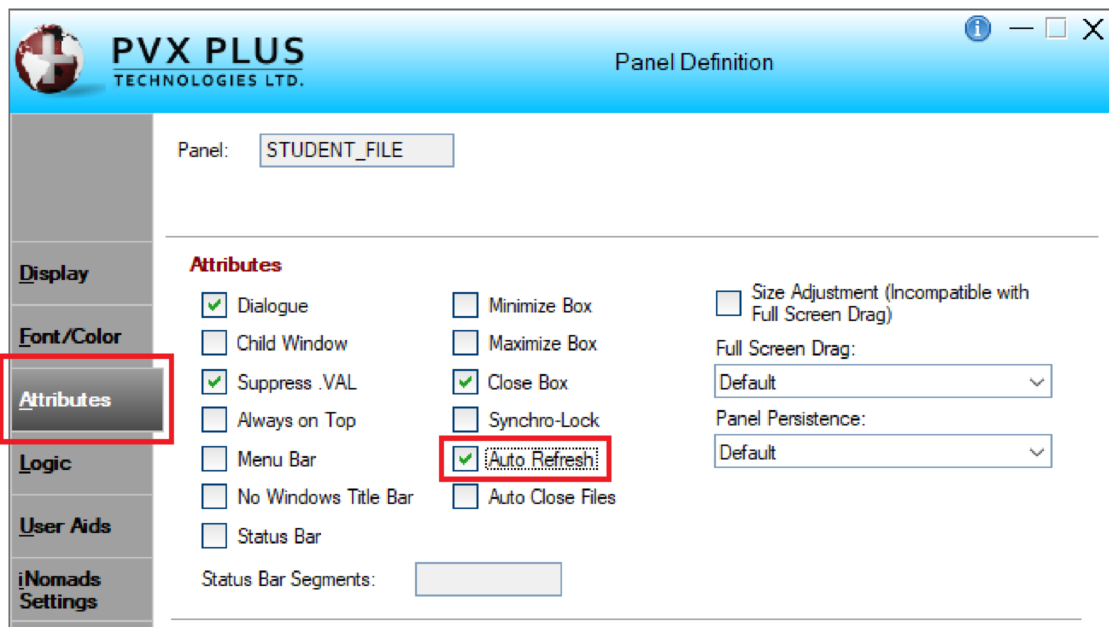
Panel Transparency: We can define the transparency factor of the window so that it obscures (or not) the image behind it.

Background Image: Define an image as the background of the panel/shape.

Image Alignment: Image location (*Top Left*, *Centered*, *Tiled* or *Scaled*).

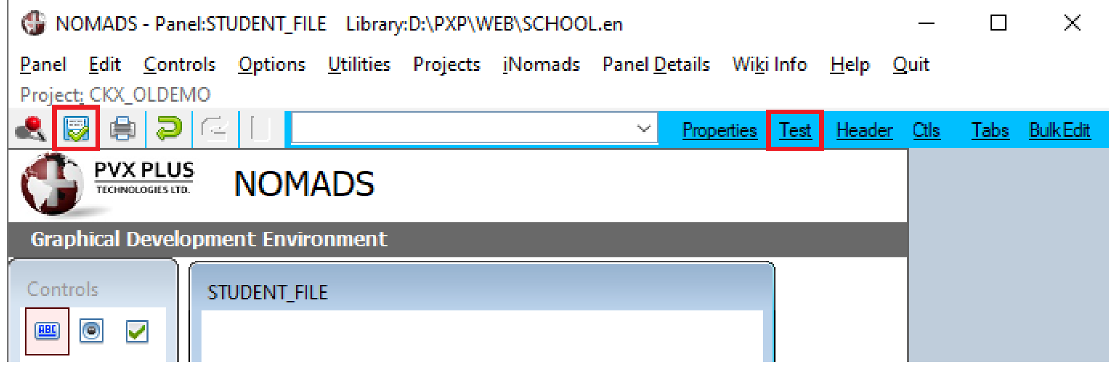
Widget: Indicates if the panel we are defining will be a widget or applet.

We are going to ignore all this, and go to the **Attributes** tab to activate the [**Auto Refresh**] option. This allows NOMADS to automatically update control values without having to redraw the panel.



After the [**Auto Refresh**] option is selected, click the [**OK**] button to return to the designer screen.

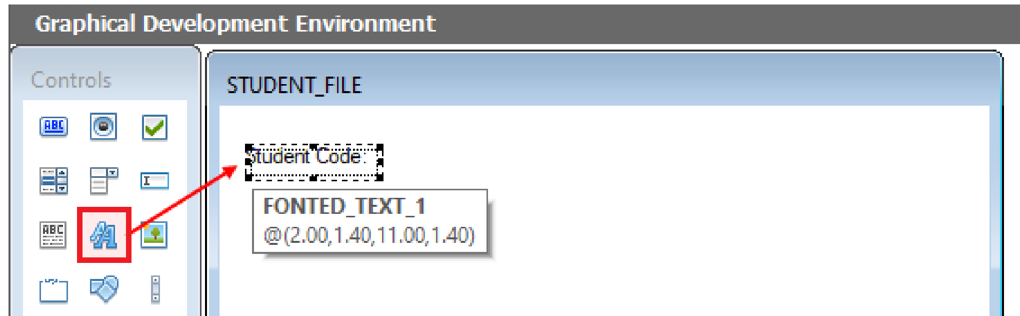
Notice the two buttons located at the top of the NOMADS panel designer: one shows a picture of a diskette, and the other is the [**Test**] button.



The diskette button is the **Save** button. It allows us to save our work so that we can guarantee that it is not lost in the event of an electrical failure or human error. It is also possible to save our work by selecting the options [**Panel**] -> [**Save**] in the top menu or by pressing [**Ctrl S**].

The [**Test**] button allows us to run our panel to test its operation. We can also do it by selecting the options [**Panel**] -> [**Test**] in the top menu.

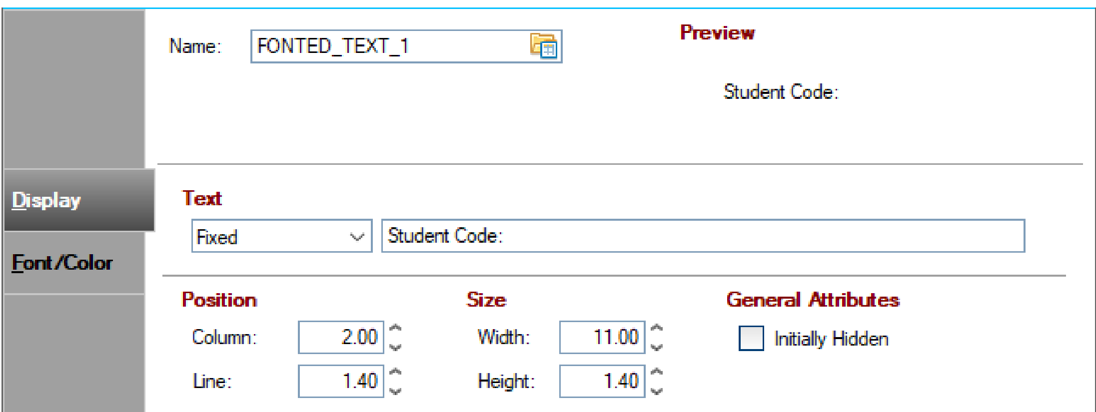
We are going to place our first control in the panel. In the **Controls** box on the left, click the [**Fonted Text**] control (blue double A). Draw a box in the upper left part of the panel. Hold down the left mouse button, drag the mouse to create a box of the desired size and then release the mouse. In the **Fonted Text Properties** window, enter the text **Student Code:** (including the colon).



Don't worry if the appearance of your control is not exactly the same as shown above. We will refine the details later. These controls are actually simple labels. They will not affect the operation of the program at all if, for example, you entered "Code:" instead of "Student Code:" or any other text. These text controls are used to display information on the screen and do not affect the behavior of the program in any way.

Note: You may notice that the mouse control is not as precise as you would like and that it is positioned on a preset grid. This may be happening, but we will see more later.

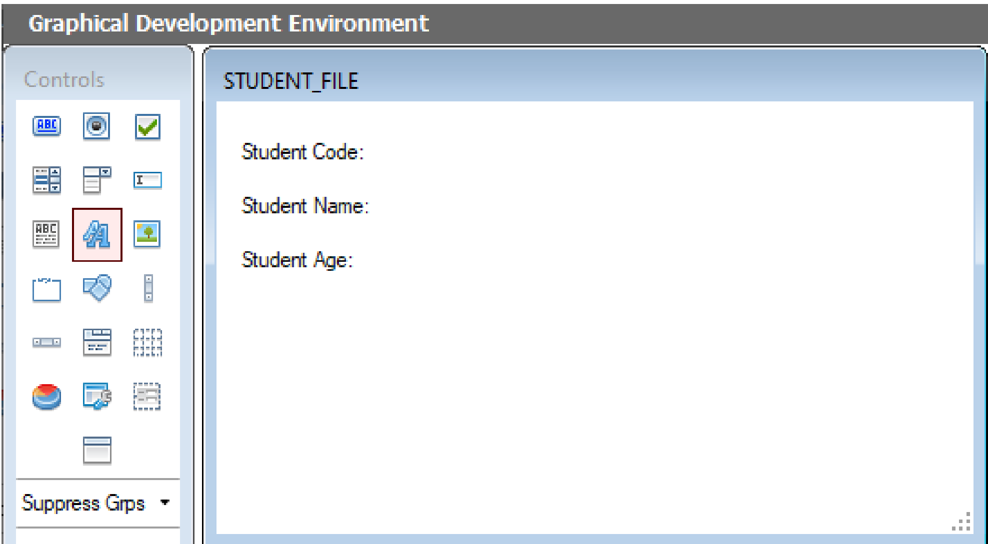
Note: If you want to correct the entered text, just double click on it and the control properties window will appear:



In this first panel, we can see the position and size of the control. It is possible that not all of the entered text will display. In that case, you will have to adjust the width of the control.

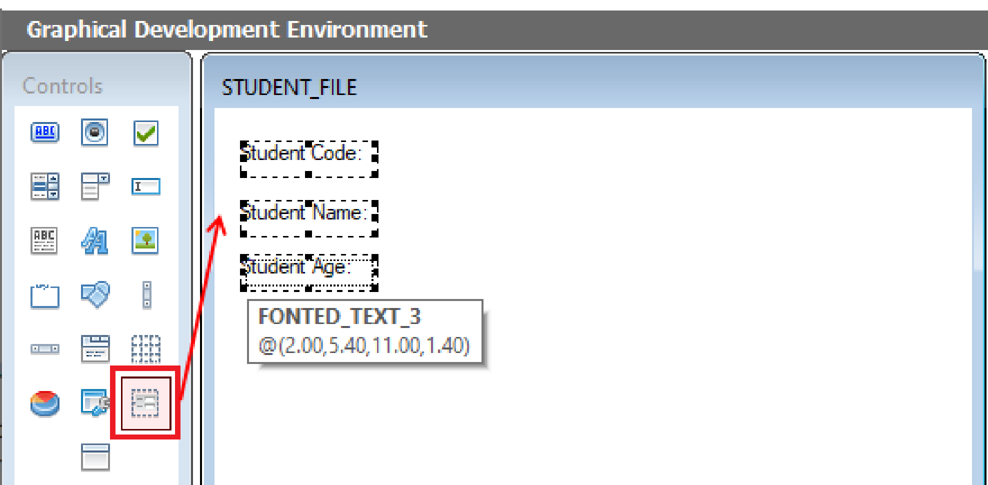
Tip: Don't waste time now making adjustments to place the control in an orderly manner or aligned with the others. We will see later how to do this quickly and efficiently.

For now, let's repeat this process for two other labels, **Student Name** and **Student Age**.



Note: Remember that these are simple labels. Take your time and familiarize yourself with the working environment and the mechanics of the program.

Tip: You can set up or align multiple controls simultaneously by selecting them all at once (or by selecting the first one and then using [**Shift Click**] to select the next ones). You can also click on the [**Group Items**] option in the **Controls** box and then use the mouse to draw a box that encompasses all the controls. Once all the controls have been marked, they will appear like this:

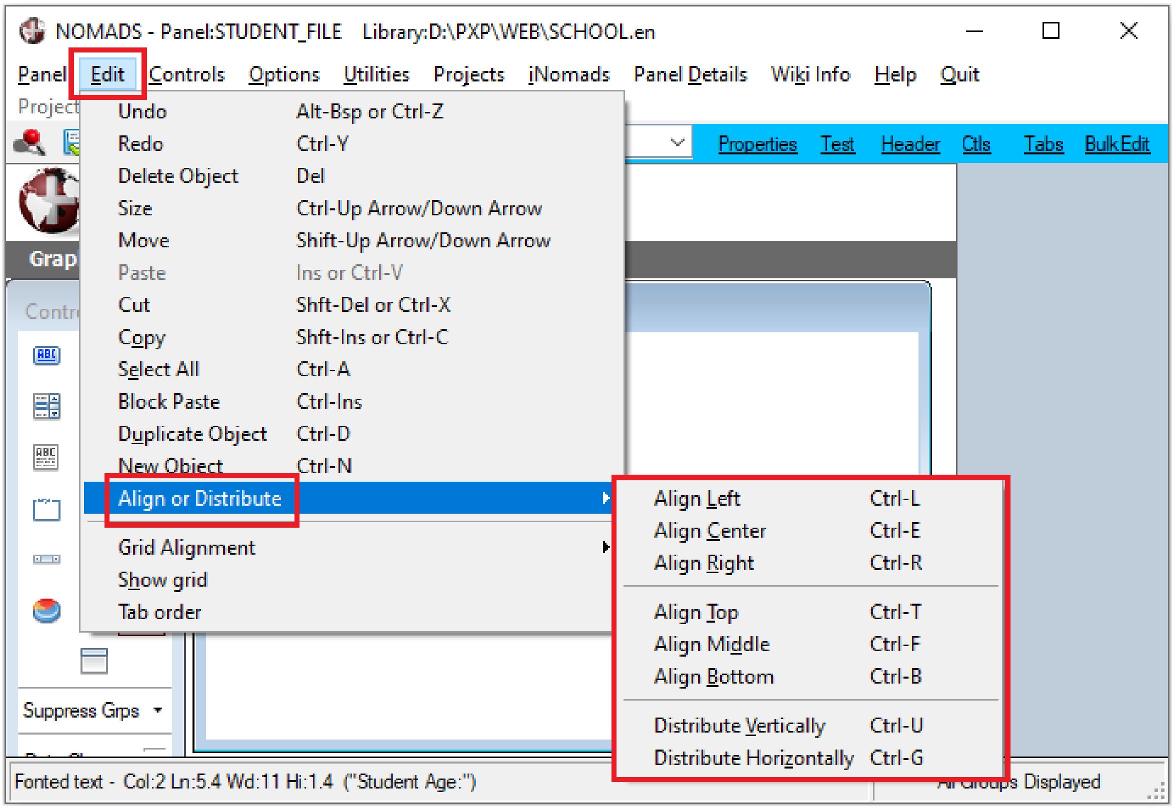


Once all the controls have been selected, you can select [**Edit**] -> [**Align or Distribute**] from the top menu bar and select one of the options.

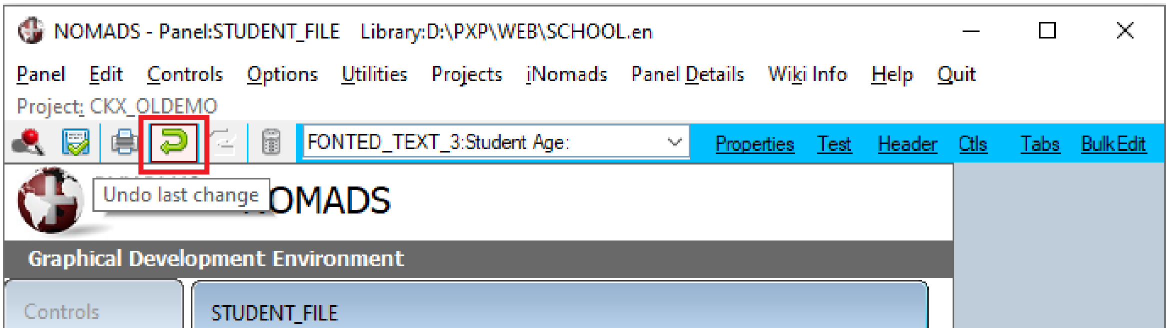
The first three options align the text or control to the left [**Align Left**], center [**Align Center**] or right [**Align Right**] of the panel (according to the size of the controls involved).

The other options align the controls at the top [**Align Top**], in the center [**Align Middle**] or at the bottom of the panel [**Align Bottom**].

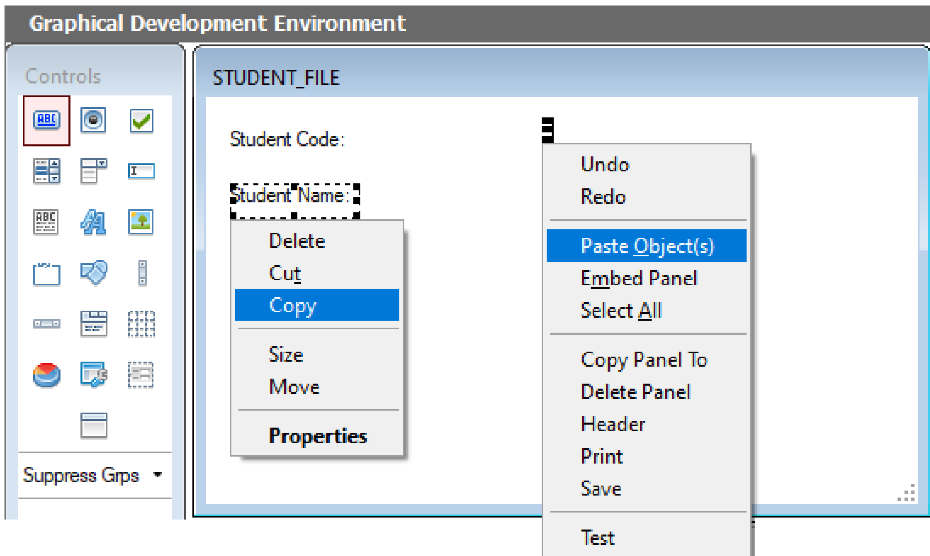
The last two options distribute the controls equally in vertical space [**Distribute Vertically**] or horizontally [**Distribute Horizontally**].



Note: If you make a mistake or the program does not do what you want, you can select the Undo button (green arrow located in the top toolbar). You can also select [**Edit**] -> [**Undo**] from the top menu bar or press [**Ctrl Z**].



Another way to speed up our work is by using the context menu (by right clicking). First, click on the control to select it. Then, right click to bring up the context menu and select the [**Copy**] option. Then, we go to another part of the panel, right click and select the [**Paste**] option.



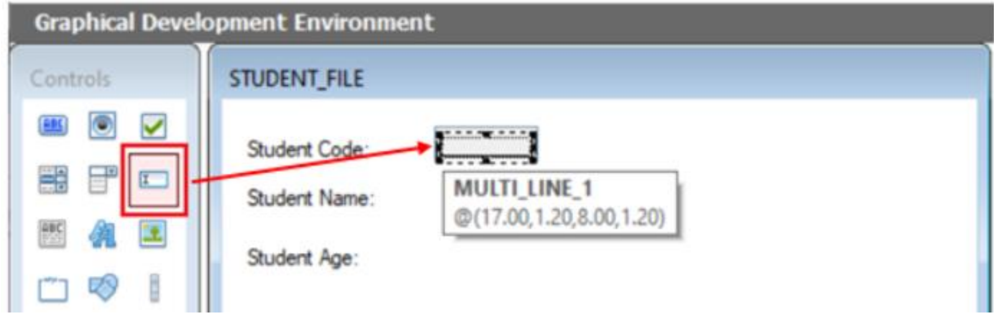
Tip: Once the control is in memory, you can paste the object multiple times by right clicking and selecting the [**Paste**] option to make multiple instances of that control.

Important Note: Whenever we do something such as [**Copy**] and [**Paste**] a control, the copied control "inherits" all the attributes of the original object, with the new control being a new "instance" of the control in question. Fortunately, for many beginners, it is not necessary to know object-oriented programming to use NOMADS or PxPlus but using it and knowing it will allow you to get more out of it and be more efficient.

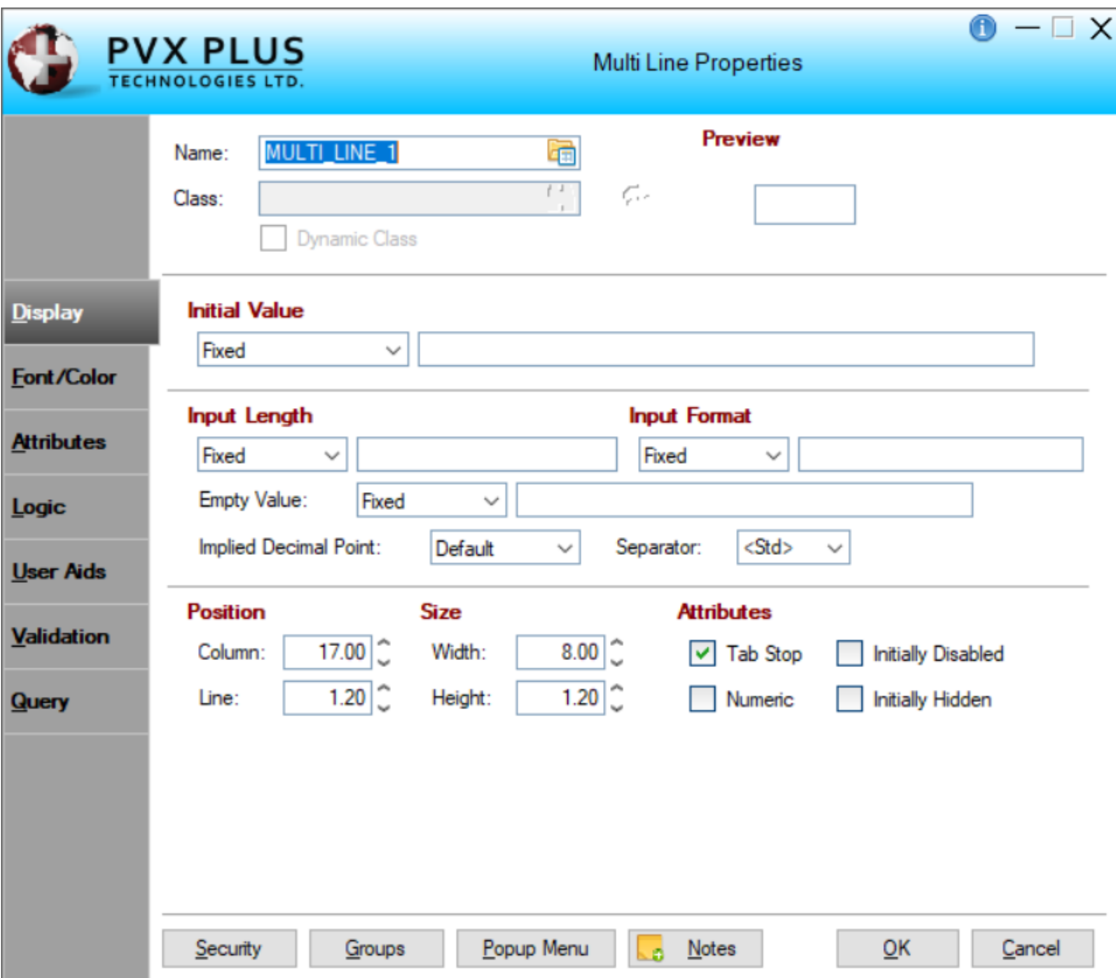
Let's continue with the exercise.

Once the Text controls (labels) have been created, we are going to create the Multi-Line controls (text entry boxes) that will be used to enter the information. Unlike the first controls, it is important to give meaningful names to the fields because we will use the same names later to reference the entered data.

Select the [**Multi-Line**] control in the **Controls** box and draw a box similar to the one shown below:



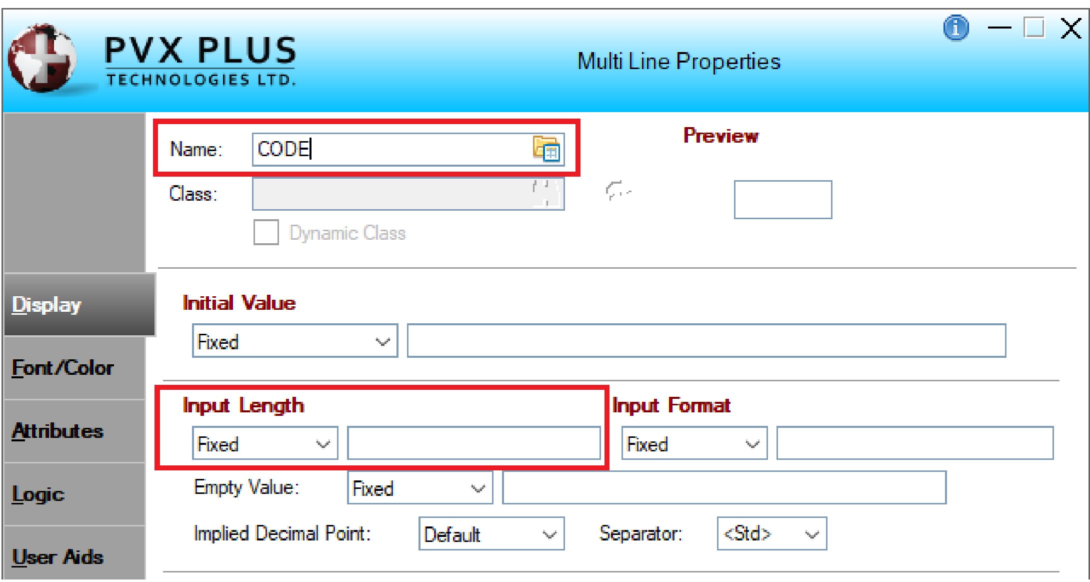
Once we finish drawing the control, the **Multi-Line Properties** window opens:



The first thing we must do is change the [**Name**] field at the top from **MULTI_LINE_1** to **CODE**. We

see that the **MULTI_LINE** control type shares many properties with the other controls, but it includes a **Validation** tab and a **Query** tab.

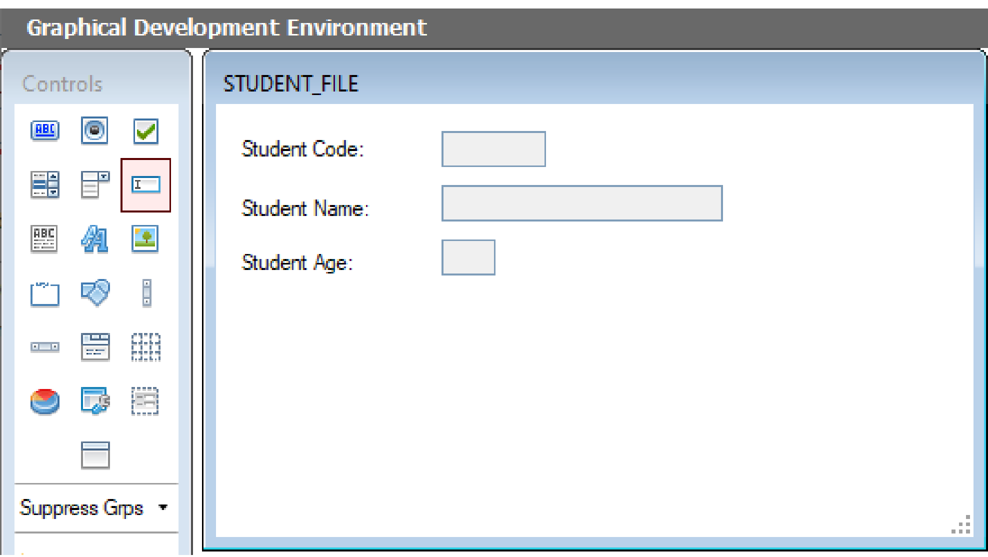
Perhaps the other part of the **MULTI_LINE** control properties to highlight at this moment is the one regarding the [**Input Length**] that will limit keyboard entry to the amount specified in that field.



Let's apply the Copy and Paste method (via the right click/context menu) to add two more Multi-Line controls for **NAME** and **AGE**. We must also change the [**Name**] field in the properties window for each control.

Using the mouse, we can click on the control and drag the right part of it to dynamically change its size and make it more in line with what was initially specified; that is, make the **NAME** field a little longer and the **AGE** field shorter.

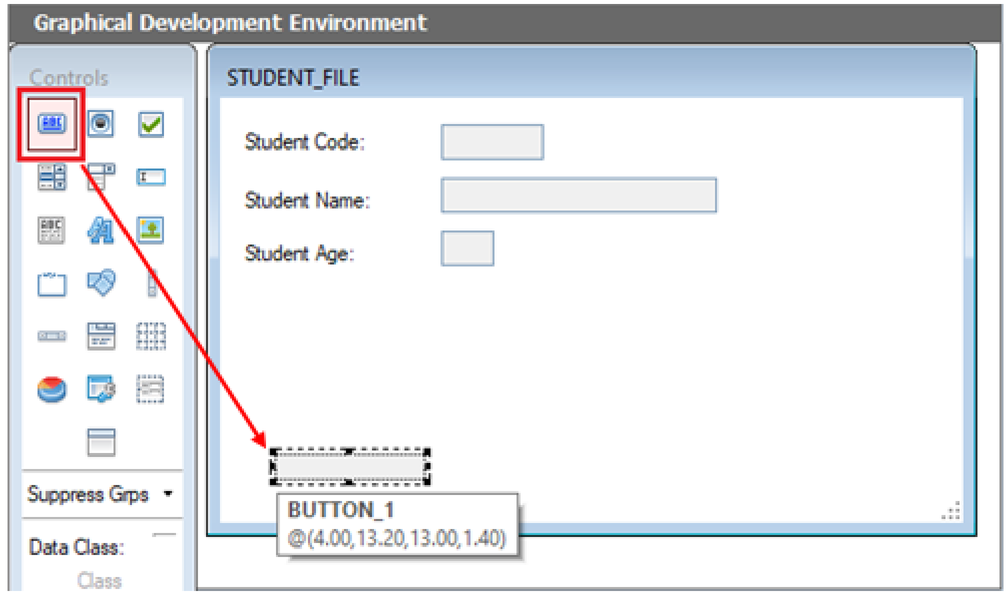
Our panel should look similar to the one shown below:



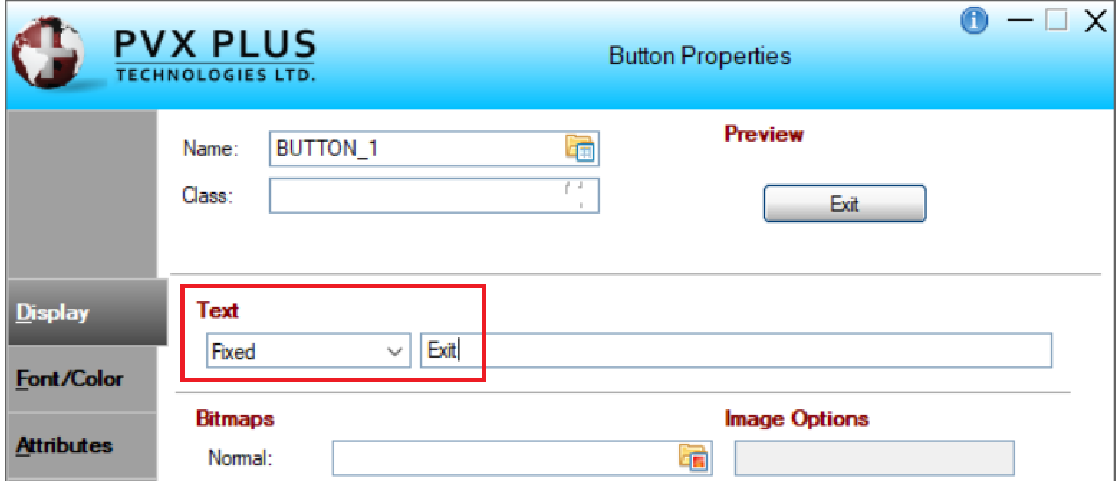
Let's not worry about the functionality of the panel at this point. Let's enjoy the process of creating our first graphical panel in NOMADS!

To complete the panel, we are going to create a couple of buttons: an **Exit** button and a **Save** button to record the information entered from the keyboard.

Select the [**Button**] control in the **Controls** box and draw a box:

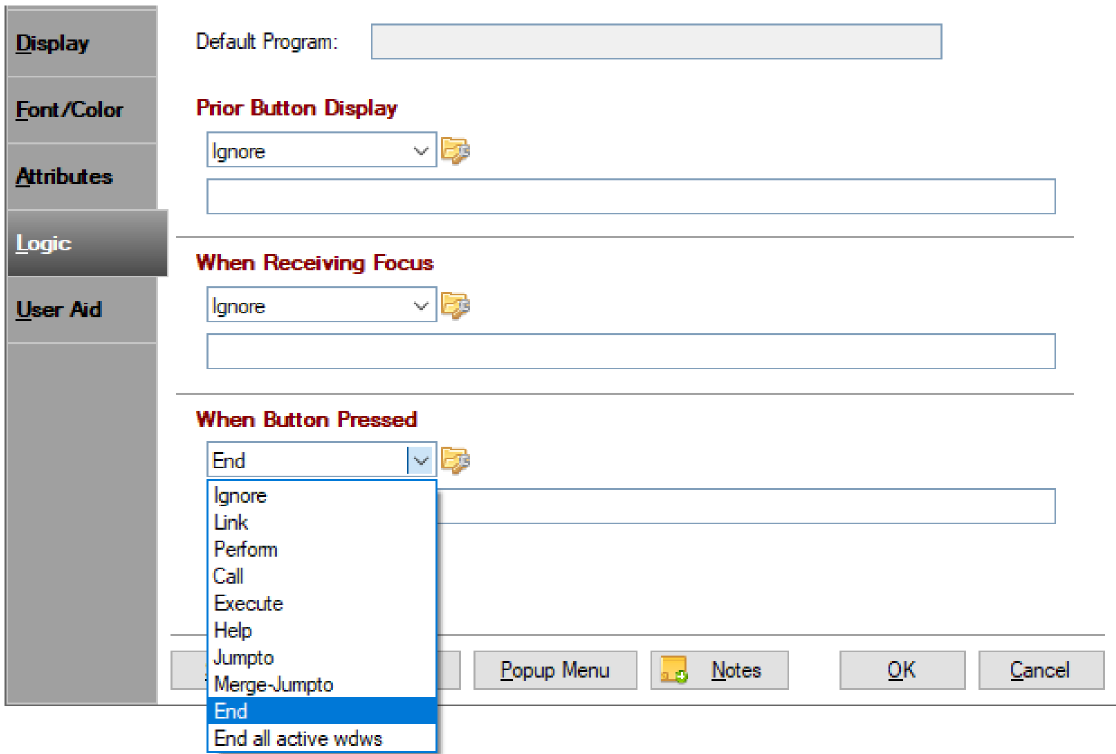


When we finish drawing the control, the **Button Properties** window will display where we will enter the word **Exit** for the button [**Text**], as shown below:



The [**Text**] field of the button is its content or description and could optionally be accompanied by an image or icon (which can change when the button is pressed). For now, let's just change the text of the button, and then move on to the **Logic** tab located in the left sidebar.

We can see that the Button control type can have three "events" or interaction situations: **Prior Button Display**, **When Receiving Focus** or **When Button Pressed**:



The first event, **Prior Button Display**, allows you to perform an action ***before*** drawing the control on the panel. **Example:** If there is no printer connected, the Print button may not be displayed.

The second event, **When Receiving Focus**, is when the button receives focus or processing attention (a right click, tab stop, or other instance was made).

The third event, **When Button Pressed**, which is the one in question, will execute some action when the button is clicked (or pressed with the finger if it is a device with a sensitive screen).

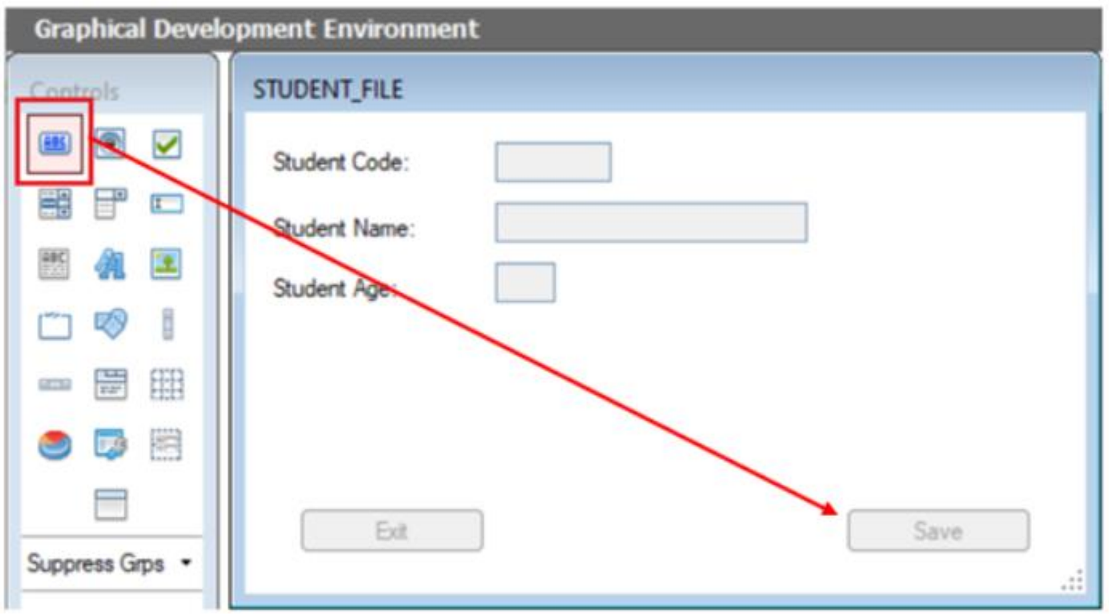
We can see that there is a series of actions, from doing nothing (**Ignore**), executing a routine (**Perform**) or ending the panel (**End**).

For now, let's ignore the other options and select **End** so that when the button is clicked, the panel ends its execution. Click the [**OK**] button to save these changes and exit the properties window.

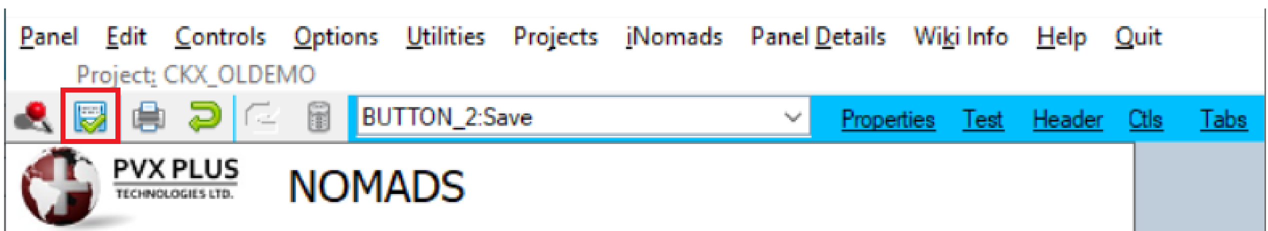
We are going to define the second button (without associating any logic to it for now).

Select the [**Button**] control again in the **Controls** box, and then use the mouse to draw the button on the panel.

In the **Button Properties** window, enter **Save** for the button [**Text**].

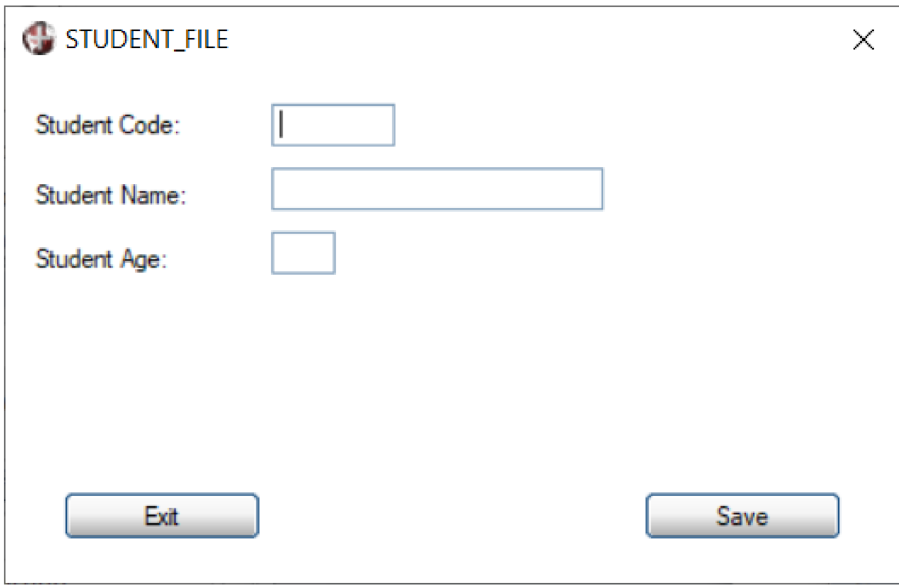


It is a good time to save our panel to preserve all the changes made. To do this, we can click on the diskette icon in the top menu or select [**Panel**] -> [**Save**] from the menu bar or press [**Ctrl S**].



We are going to carry out the first execution or test of our panel. It is important to ***save the panel before testing it***. To do this, click on the [**Test**] option in the top menu.

The result should be similar to the panel shown below:



We can enter text in the Multi-Line controls, and clicking the **Exit** button will close the panel. The **Save** button does nothing for now.

Let's review these steps.

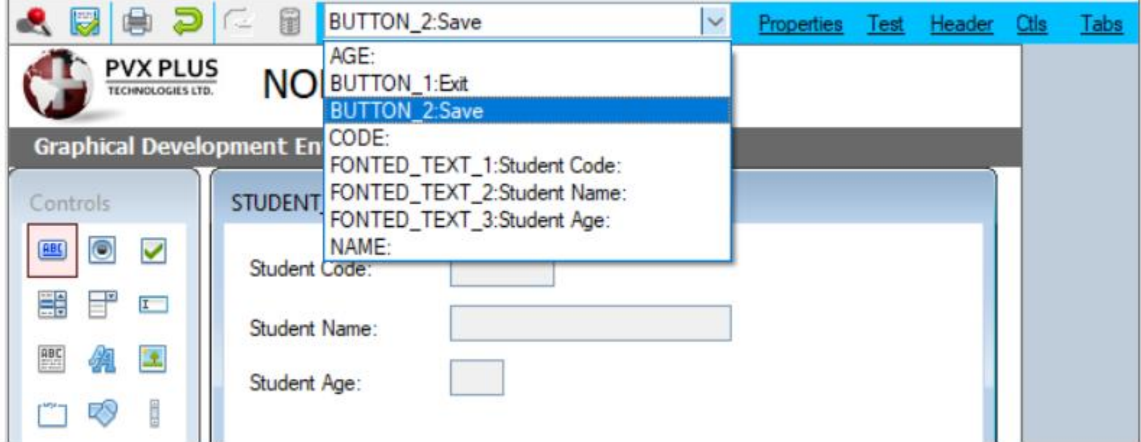
- The panel was created within NOMADS. We entered a name for the library (which is a group of panels), and then we entered a name for the panel (**STUDENT_FILE**). The Panel Designer window appeared, and we selected the type of controls to create.
- After drawing a control, the properties window for the control appeared. In our panel, we have Text controls for the field labels. We also have text entry boxes (**MULTI_LINES**) to allow text entry by keyboard.
- Finally, we have two Button controls, **Save** and **Exit**.

It is important to remember that the panel properties are modified by selecting the [**Header**] option. We must select the [**Auto Refresh**] attribute so that NOMADS updates the controls every time there is a change.

Remember that each control must have a unique name. NOMADS will automatically assign these names, but we can change it in the properties window of every control.

Internally, not a single line of code or program has been generated/written. NOMADS saves the objects and their positions in a library file and will create them upon execution.

At the top, there is a drop box where we can see all the controls that make up the panel. Our panel is made up of the following controls:



Let's remember that, in some controls (such as the Multi-Line), it is convenient to know the name of the control, since the system will use a variable with that name to refer to the value of the control.

Example: If the text entry box is called **MULTI_LINE_1**, the variable **MULTI_LINE_1$** will be used to store the contents of the text entry box.

At the program level, we can make comparisons, such as:

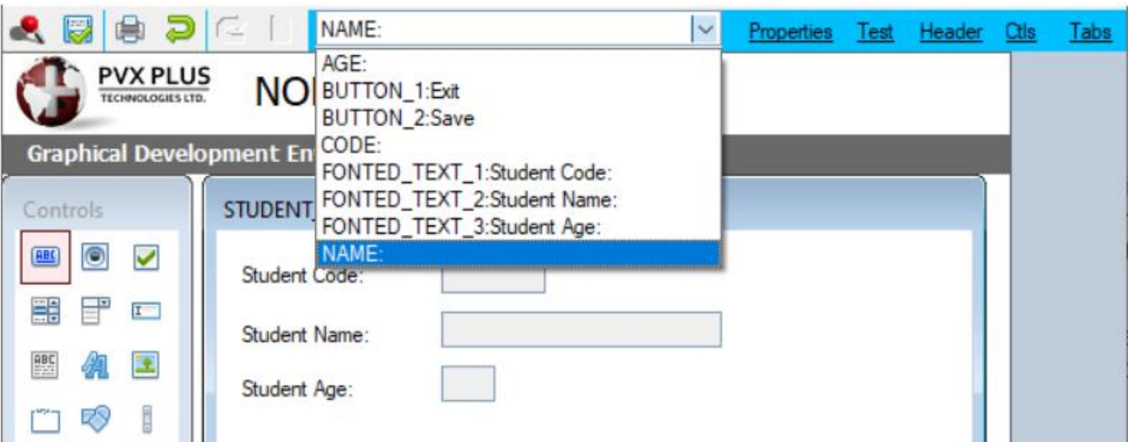
```
If MULTI_LINE_1$="" then MSGBOX "Empty"
```

Although this is a simple panel, it is easy to get confused by not being able to relate each field to the name of the control. We assume that **MULTI_LINE_1** should be the Student **CODE** and **MULTI_LINE_2** should be the Student **NAME**. When creating the control, it is preferable to assign it a more mnemonic name related to its content.

In our panel example, let's check that the first Multi-Line control is called **CODE**, the second is called **NAME**, and the third is called **AGE**. If this is not the case, we can change it by double clicking on each control and changing the name.

Remember to save the panel by clicking the [**Save**] button (diskette icon) or by selecting the [**Panel**] -> [**Save**] option in the top menu bar.

Another way we can verify the control names is by clicking again on the drop-down box in the top menu. Notice that the Multi-Line control names no longer appear and were replaced by the new names: **CODE**, **AGE** and **NAME**:



Important Note: For the purposes of this exercise, the order in which the controls appear in the drop-down box does not matter.

Right now, we have a panel called **STUDENT_FILE** with the [**Auto Refresh**] attribute activated. It has three labels, three text entry boxes and two buttons. The only "intelligence" that our panel has is that we have changed the properties of the **Save** button. We double clicked on the **Save** button, selected the **Logic** tab, and selected **End** in the **When Button Pressed** event drop-down box.

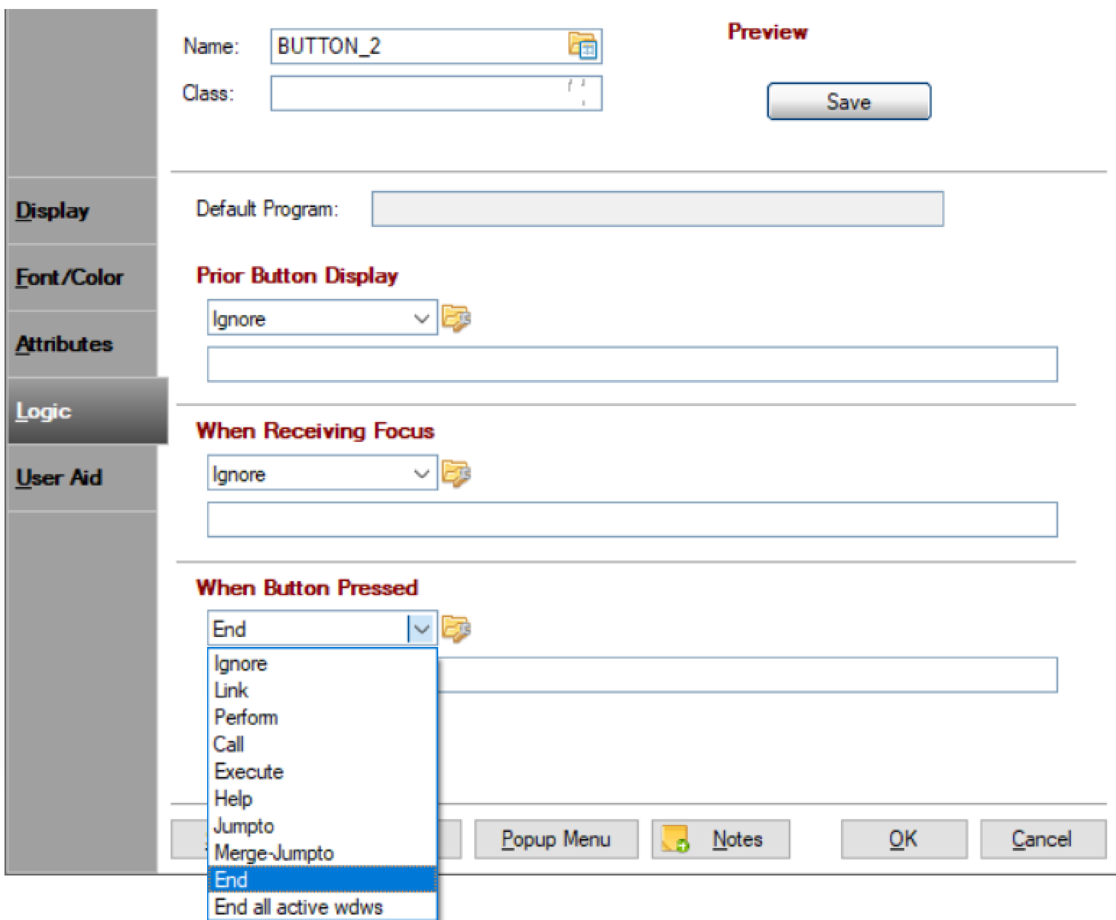
If we execute the [**Test**] option for our panel and enter information in each text entry box using the keyboard, at the program level, we will have three variables with information: **CODE$** will contain the data entered in the **CODE** text entry box, **NAME$** will be the variable for the **NAME** text entry box and **AGE** will have the information from the **AGE** text entry box.

Note: In PxPlus, variables that end ***with*** the **$** (*dollar sign*) indicate that the content is ***alphanumeric***, while variables ***without*** the **$** (*dollar sign*) can only contain ***numbers***. Let's not divert our attention to those details now. We will clarify these points later.

Remember that the Button control has three associated events: **Prior Button Display**, **When Receiving Focus** and **When Button Pressed**.

> **Prior Button Display**: Before displaying the button, it will execute the selected action ***before drawing the button***.
>
> **When Receiving Focus**: It will execute the selected action when the button receives focus.
>
> **When Button Pressed**: It will execute the selected action when we click on the button or press it.

Important Note: It is a good time to check if the logic associated with the **Exit** button works. Save the panel by clicking the diskette icon or by selecting [**Panel**] -> [**Save**] from the top menu bar. Click the [**Test**] button in the top menu bar. When you click the **Exit** button on the panel, the panel must end and return control to NOMADS.



When the drop-down box for the **When Button Pressed** event is selected, the possible actions are:

Ignore: As the name implies, it will not take any action when this event occurs.

Link: Allows you to connect to another panel (from the same library or another).

Perform/Call: Calls or executes a routine or program. It will be one of the most used options in this part of the book. You can pass arguments with variables.

Execute: Directly executes a PxPlus language command.

Help: Invokes the online Help subsystem.

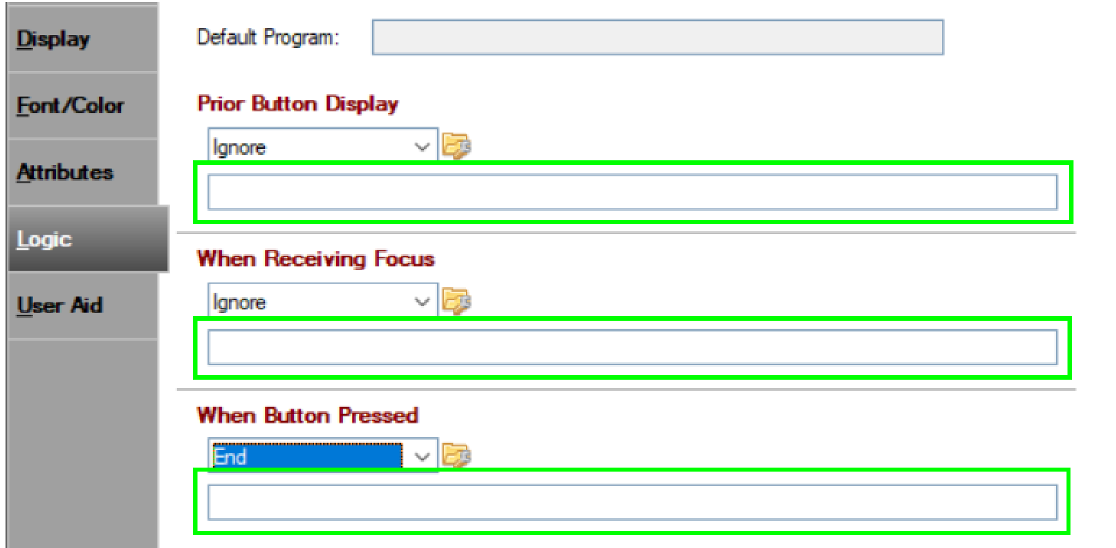
Jumpto: Calls or executes a routine or program. The difference with Perform/Call is that with this option, the entire work environment (variables and other memory structures) is passed.

Merge-Jumpto: Used to run concurrent panels, meaning both will be present and active (it is possible to do this with multiple panels).

End: Ends the execution of the current panel. If the panel was called from another panel, the control or flow of the program will be returned to the previous panel or parent panel (as it is also called).

End all active wdws: As the name indicates, it ends the execution of all panels or windows and exits execution.

Some of the actions in the drop-down list (in all three events) have, in addition to the drop-down box, a space for an argument (marked in green).



Seeing the [**Default Program**] text entry box first and seeing that some actions ask for a program name, we could save work and be more efficient if we put the program name at the top. For now, let's continue with the explanation.

This argument can be a panel name (options are **Link**, **Jumpto** and **Merge-Jumpto**) and could be in any of these formats:

"Panel_Name" ***or*** "Panel_Name", "Library_Name"

If only the panel name is specified, it is assumed that the panel belongs to the same library as the parent panel.

If the argument is a program (options: **Perform/Call**), the format could be:

"Label" ***or*** "Program;Label"

The tag refers to a routine within a program, such as UPDATE_CODE or CHECK_BALANCES or any valid name. If no program is specified, the name of the default program (assigned in the first text entry box [**Default Program**]) will be searched; otherwise, both "Label" and "Program" are specified. The routine with that name will be searched in the specified program. Later, we will see practical examples with this.

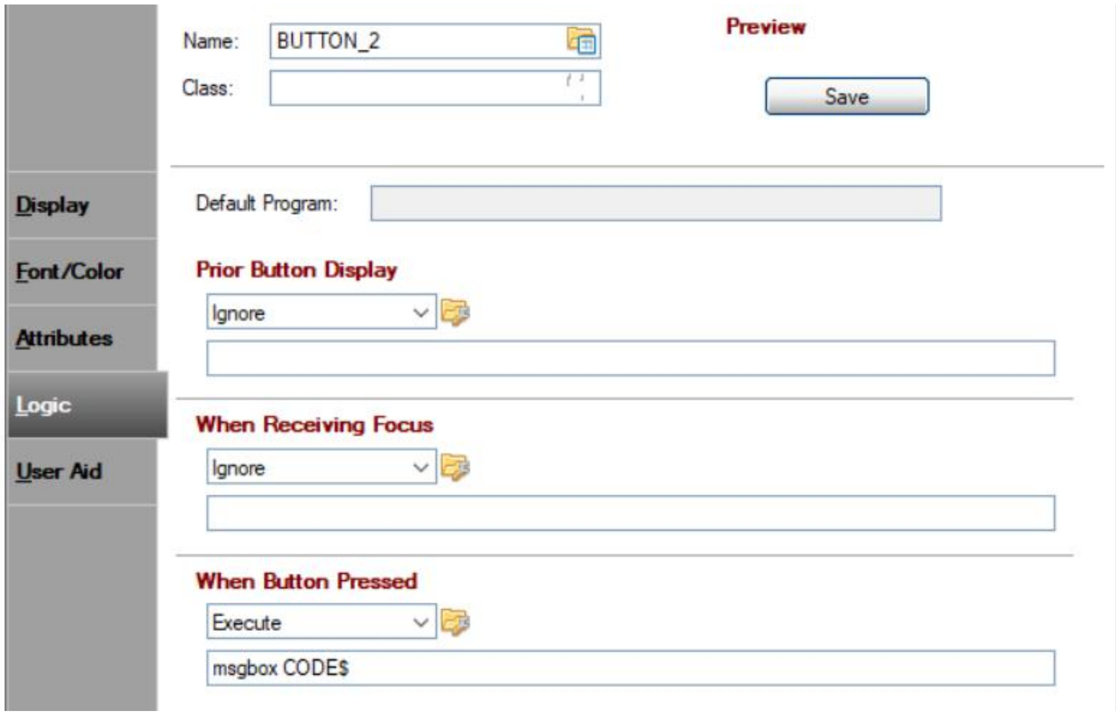
Let's start interacting a little more with the information that our panel contains. Let's see how we can review the content that a user enters in our form. To do this, we are going to modify the logic associated with the **Save** button so that when it is selected, a message shows on the screen with the entered data.

As we only want to show the information or the content of a variable (text entry box), the simplest option would be to make a program or routine that shows it, or better yet, directly executes the command. For the latter, we will select the action **Execute**.

We are going to double click on the **Save** button to open the **Button Properties** window, and then, select the **Logic** tab. Select the drop-down box associated with the **When Button Pressed** event and select **Execute**. In the argument, we will enter our first PxPlus command:

```
msgbox CODE$
```

The panel will look like this:



Basically, we are telling PxPlus (through the NOMADS tool) that when the BUTTON_2 button (which has the text **Save**) is pressed, a command (**msgbox** CODE$) must be executed (**Execute**).

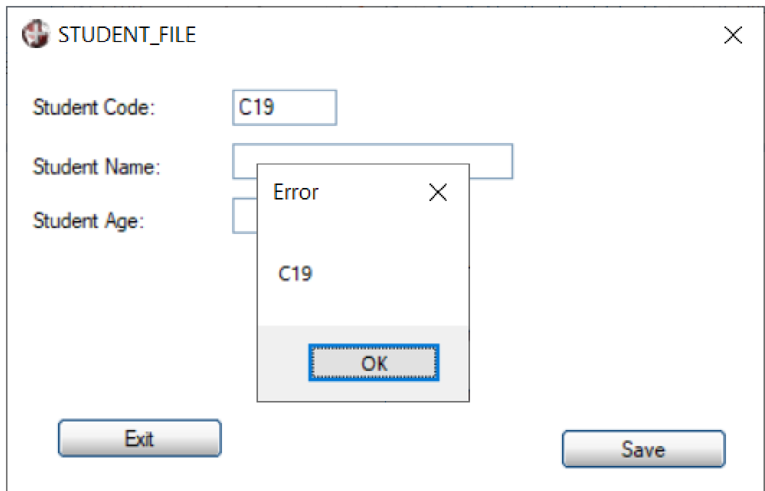
PxPlus is a very complete language. Let's not worry about the commands or their syntax. We will see them as we need them, and in a future section, we will have a more complete treatment of the PxPlus instructions, their use and syntax.

We must finish the panel, and save it.

Note: Remember that you can save the panel by clicking the diskette icon, by selecting the [**Panel**] -> [**Save**] options in the top menu bar or by pressing the key combination [**Ctrl S**].

Once we have saved our panel, we are going to select the [**Test**] button and enter some data in the **CODE** multi-line. Then, we are going to select the **Save** button.

We should get a result similar to the one shown below:



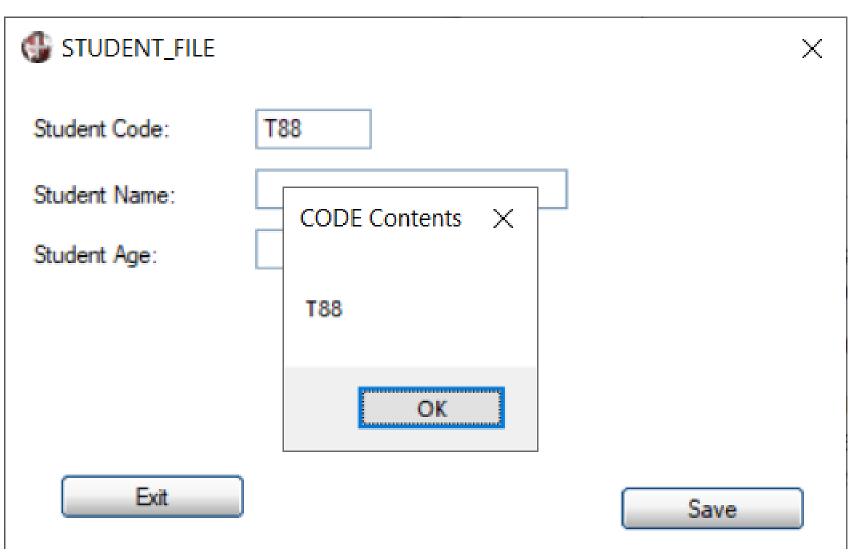
Note that there is no error but just the default value for the **msgbox** command title. We can change it by simply adding a title to the command:

```
msgbox CODE$,"CODE Contents"
```

To make the change, we end the panel by selecting the **Exit** button, double click on the **Save** button, select the **Logic** tab and change the command.

Then, we save our work again by clicking the diskette icon, by selecting the [**Panel**] -> [**Save**] options in the top menu bar or by pressing the key combination [**Ctrl S**].

Select the [**Test**] button again. The result after the change will be similar to the one shown below:



Instead of showing the content of only the **CODE** field, we could modify the command so that it shows the content of both fields. In PxPlus (and in the **Execute** argument), we can concatenate commands using the **;** (*semi-colon*):

But, we will see very quickly that starting to concatenate commands at this moment will complicate our way of programming. Therefore, we must make two important changes:

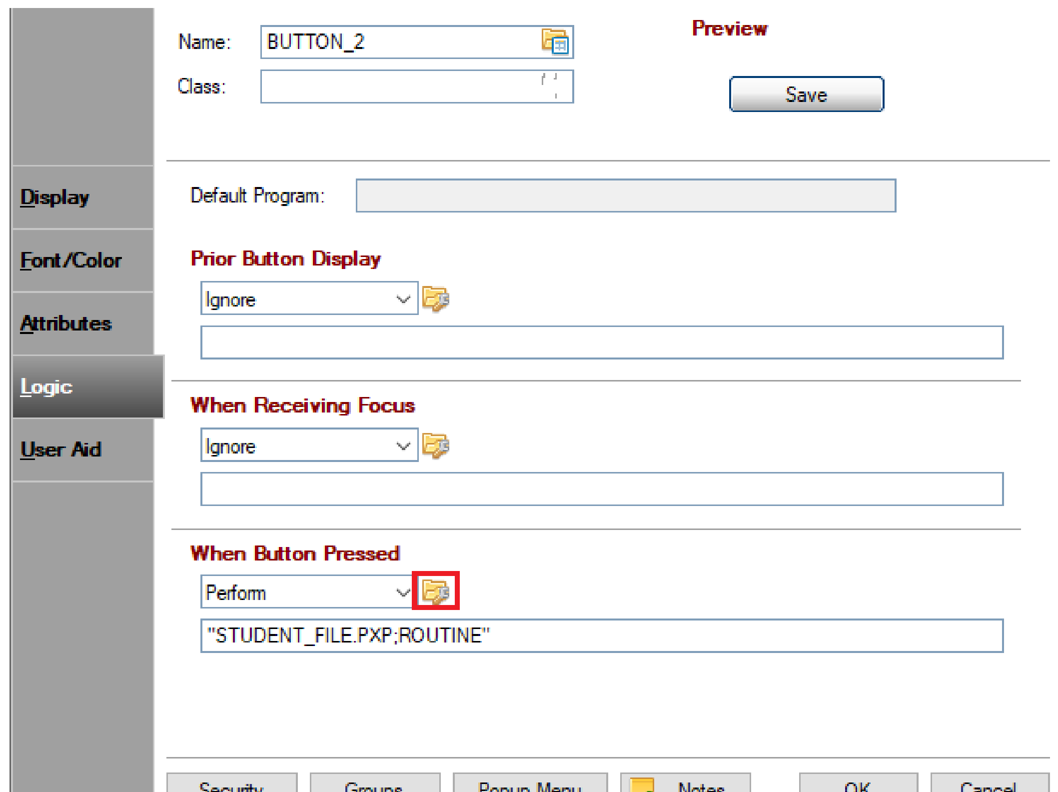
1. Change the action **Execute** to **Perform** in the drop-down list.
2. Change the **msgbox** command to a program name in the argument box, something similar to: "STUDENT_FILE.PXP;ROUTINE".

Note: PxPlus does not place restrictions on the program name. It can be in upper or lowercase, with or without an extension in the name. You must define your standard, remembering that some operating systems make distinctions between names in lower and uppercase, so "FILE.PXP" and "file.pxp" will be two different programs. Likewise, remember that the extension is optional. "NAME", "name.txt" or "NAME.PRG" are valid, but remember that for other applications, filename extensions also have a purpose. For the purposes of this book, we will use the ***.pxp*** or ***.PXP*** extension to refer to programs.

Remember to finish running the panel before you can make any modifications. In case of any doubt, the [**Esc**] key allows you to finish the execution within the NOMADS designer.

To make the proposed change, you must double click on the **Save** button, select the **Logic** tab, modify the drop-down list to select **Perform** and then enter the name of the program in quotes. ***Do not select the*** [**OK**] ***button yet***.

With the proposed changes, the result will look similar to the one shown below:



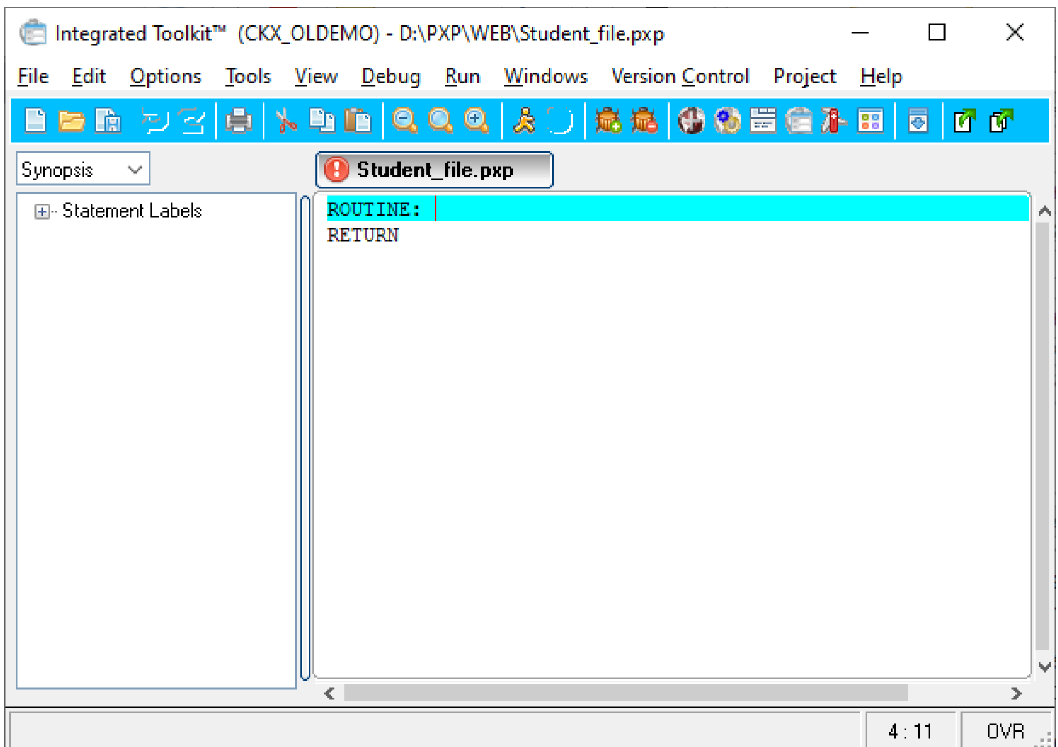
Note: The program name, as well as any literal value in PxPlus, ***must*** be enclosed in *double quotes* (**Example:** "program").

With this change, we are telling PxPlus (through NOMADS) that when the **Save** button is selected, it executes the "ROUTINE" found within the "**STUDENT_FILE.PXP**" program.

This program does not exist yet, so let's create our first PxPlus program!

There are many ways to do it. For now, the simplest way is to select the icon (with the yellow folder and tool) located to the right of the action to be executed (marked in red above).

When you do this, a new window will open (independent of the others), and a program editor will be displayed. The presentation may vary, but it should look something like this:



Tip: It is normal in this type of concurrent window environment to "lose track" because some windows may overlap or be hidden or minimized. It is recommended that you familiarize yourself with it, because once you master it, your productivity will be much greater; for example, leaving the program editor window (there can be several) open at all times.

A PxPlus routine basically has the following structure:

```
Label:
return
```

But, it can also be of the form:

```
Label:
exit
```

Both functionalities are valid. For now, we are going to focus on our program.

Important Note: PxPlus does not impose syntactic restrictions on whether the commands must be in lower or uppercase; both are allowed. There are adjustable parameters to change the behavior of the language. We will use upper and lowercase interchangeably.

Let's modify our program, opening a blank line, pressing [**Enter**] and entering the following line of code:

```
If code$="" then msgbox "The code is empty","Warning";exit
```

Note: Depending on the editor configuration, the appearance and formatting of the lines will change. They could have colors, line numbers, be in upper or lowercase, be indented, etc.

This first statement is conditional. It basically compares the contents of the **CODE** text entry box with a null value, represented with "" (*double quotes* twice). In case the comparison is correct, the **MSGBOX** command is executed and then the "exit" command.

Note: Whenever you are inside a PxPlus routine or program that is being executed from NOMADS, you can return to the panel using the "exit" command.

Important Note: PxPlus allows the direct execution of programs that were not necessarily designed or work with NOMADS, hence the limitation.

So far, our program is a simple validation that returns control of the program to NOMADS in case the code is blank.

Let's do something that validates that the three fields or elements (**CODE**, **NAME**, **AGE**) contain information:

```
If code$="" then msgbox "The code is empty","Notice";exit
If name$="" then msgbox "The name is empty","Notice";exit
If age$="" then msgbox "Age is empty","Notice";exit
```

An alternative solution (of the many that exist) could be to perform all the checks in a single statement:

```
If code$="" or name$="" or age$="" then msgbox "Some of the fields are blank", "Notice"; exit
```

This instruction compares three conditions, and if ***any*** of them are met, the program ends. It would be the equivalent to asking:

Is this condition or this other condition or this other one met?

If you have some programming experience, you probably already have a way to do this more efficiently. For now, let's continue.

Let's paste the above code into our program. It should look like this in the editor:

```
Synopsis                 Student_file.pxp
Statement Labels
Variables

ROUTINE:
 if CODE$="" \
  then msgbox "The code is empty","Notice";
       exit
 if NAME$="" \
  then msgbox "The name is empty","Notice";
       exit
 if AGE$="" \
  then msgbox "Age is empty","Notice";
       exit
 return
```

Note: It is worth noting that the NOMADS panel we are using and this small program are completely separate elements. That is, we can use the ROUTINE in any other panel, as well as use the STUDENT_FILE panel in another program. PxPlus does not impose limitations in this sense.

This is a good time to practice and experiment. It's okay if you "mess up" something. You can delete the panel or program or simply create a new one.

Important Note: We can instruct NOMADS that, when it returns to the panel, it takes the process focus to a particular control. The syntax is **NEXT_ID=CONTROL_NAME.CTL**. We can modify the routine so that in each case, it returns to the appropriate place:

```
If code$="" then msgbox "The blank CODE", "Notice"; next_id=code.ctl; exit
```

To summarize:

This is a good time to summarize everything covered in this chapter.

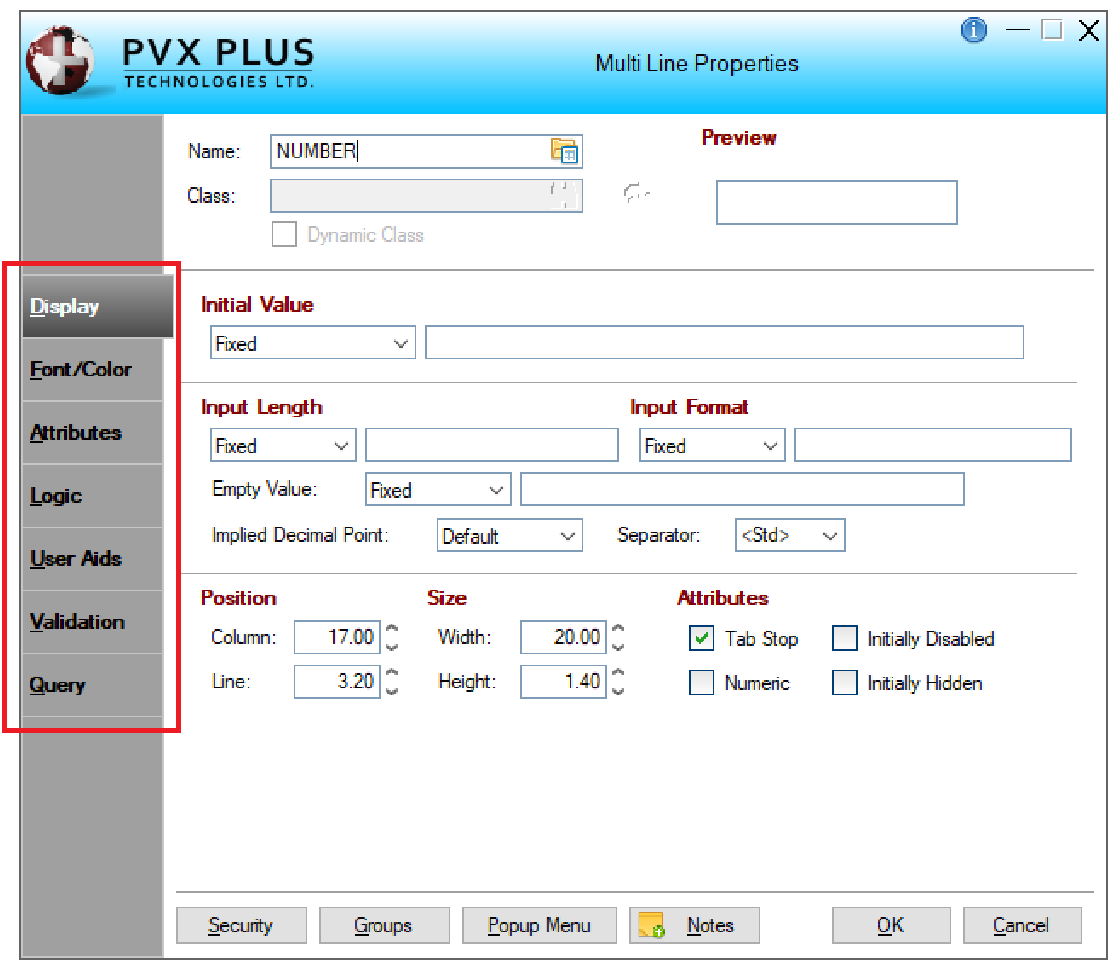
- NOMADS is a graphic panel designer and is part of PxPlus.
- A library is a set of panels; these in turn are forms or windows.
- Within a panel, we will have controls or objects, such as buttons, text, text entry boxes, etc.
- We must activate the [**Auto Refresh**] attribute so that the changes are reflected automatically.
- It is convenient to save our work [**Ctrl S**] before executing it [**Test**].
- Each control must have a unique name. If it is a **MULTI_LINE** control (text entry box), there will be a variable with the same name for its content, adding the **$** sign (CODE$ or NAME$).
- To modify a control, we can double click on it, and a window will appear with the object's properties, such as name, position, size, font, color, etc.
- The properties of the panel or window are modified through the [**Header**] option in the top menu.
- Controls can have associated routines or actions. This is in the properties, **Logic** tab. To associate a routine to a program, we use the **PERFORM** action, along with the name of the program in question.
- We return to NOMADS (from the program) with the commands, **EXIT** or **RETURN**.
- The **MULTI_LINE** controls have a tab called **Validation** where it is possible to perform validation (and formatting) routines for their content.
- The special variable **NEXT_ID** allows you to transfer the flow or focus to a particular control; we must make NEXT_ID=CONTROL_NAME.CTL.

NOMADS Control Properties

Each control in NOMADS has a series of attributes, many of which are common between the different controls. Let's see, in summary, the main aspects of each tab.

When we double click a control in a NOMADS panel, the properties window for that control opens.

Example: Double clicking on a **MULTI_LINE** control opens the **Multi-Line Properties** window:



Depending on the control, there will be fewer or more properties (and some may appear exclusive to some controls). The properties on the **Display** and **Font/Color** tabs are common for almost all panels. In the other tabs, we will see the main aspects of each one:

> **Display:** Location and size of the control, including X and Y coordinates, horizontal and vertical size, legends, etc.
>
> **Font/Color:** Font size and type (letter), color and background color.
>
> **Attributes:** Different attributes of each control. In some cases, it is possible to change the type or appearance of the control (as in the case of buttons, which have several styles), its initial state, behavior, blocking, etc.

Logic: Some controls may have actions associated with them, such as buttons, text entry boxes, lists, grids, and graphics, among others. On this tab, it is possible to associate these actions with predetermined events. **Example:** If the user selects an entry from the list, the XYZ routine is executed.

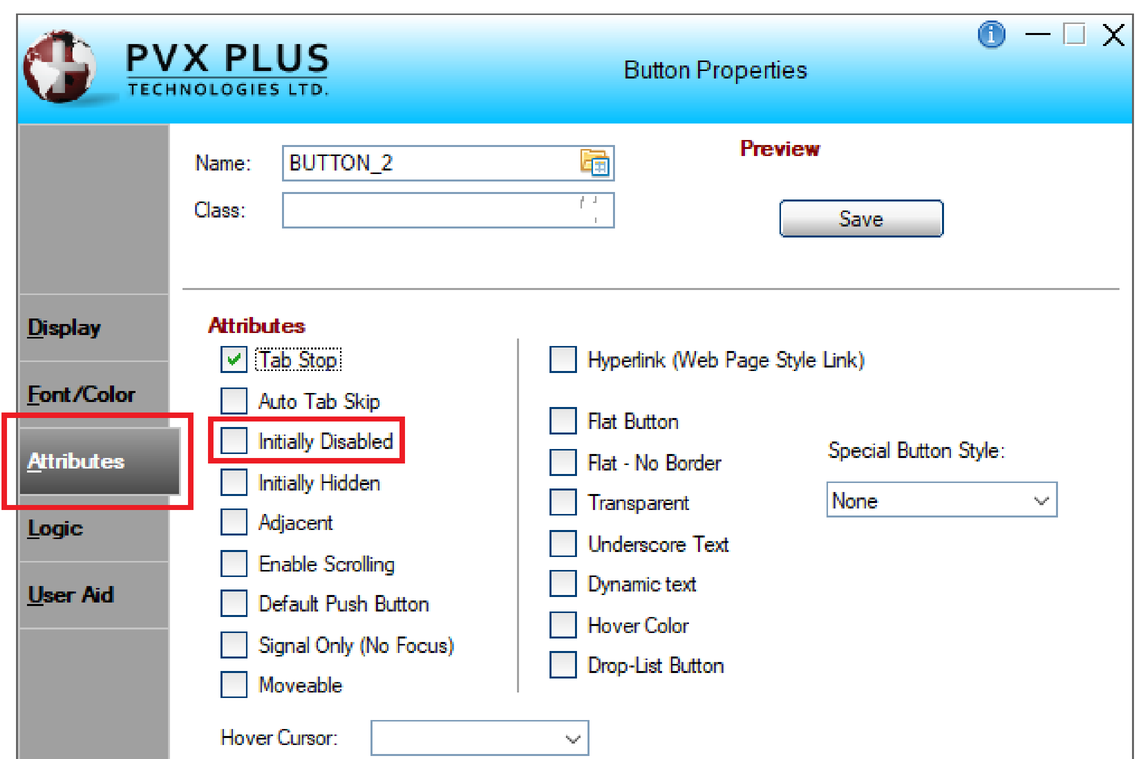
User Aid: This tab allows you to define online help or floating tips in some controls.

Validation: The **MULTI_LINE** controls have a tab to place a special program (VALIDATOR) that will be responsible for carrying out the corresponding validations, and another program (FORMATTER) that can modify the presentation or format of the content (such as adding hyphens to a telephone number).

Values: LIST type controls (**LIST_BOX** and **DROP_BOX**, among others) have a properties tab to assign initial values to the content of the list or have equivalent values. **Example:** Let the MEDIUM option be equal to 1 and the LARGE option be equal to 2.

It is not the intention of this book to discuss all the values and functions of each tab (some tabs for a control may have more than 20 elements); however, if further explanation is needed, then it will be included.

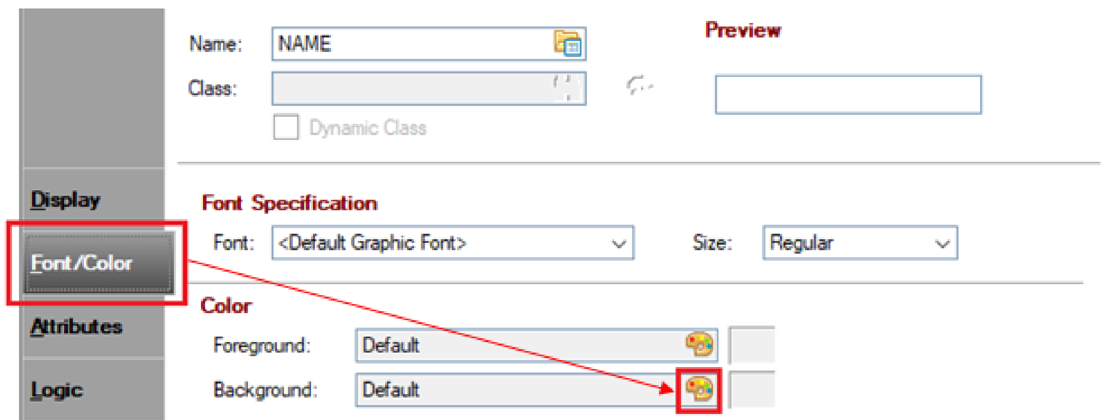
If, for example, we have no data uploaded (such as at the beginning of a panel run), we could have the **Save** button disabled. To do this, we double click on the button to open its properties window, go to the **Attributes** tab and select the [**Initially Disabled**] check box (marked in red below).



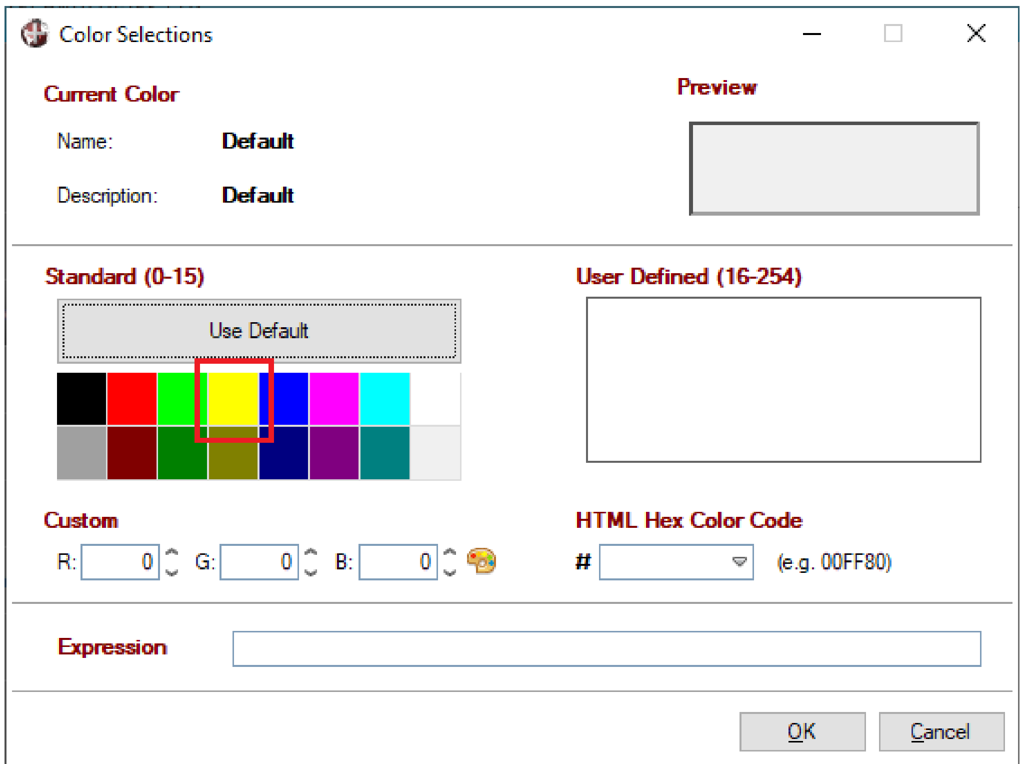
Click the [**OK**] button to accept the changes made to the **Save** button.

Let's take the opportunity to also change the background color of the Multi-Line control, **NAME**, and then we will see both changes.

For this second change, double click on the **NAME** control. In the **Multi-Line Properties** window, select the **Font/Color** tab. Click on the paint palette icon located to the right of the [**Background**] option.



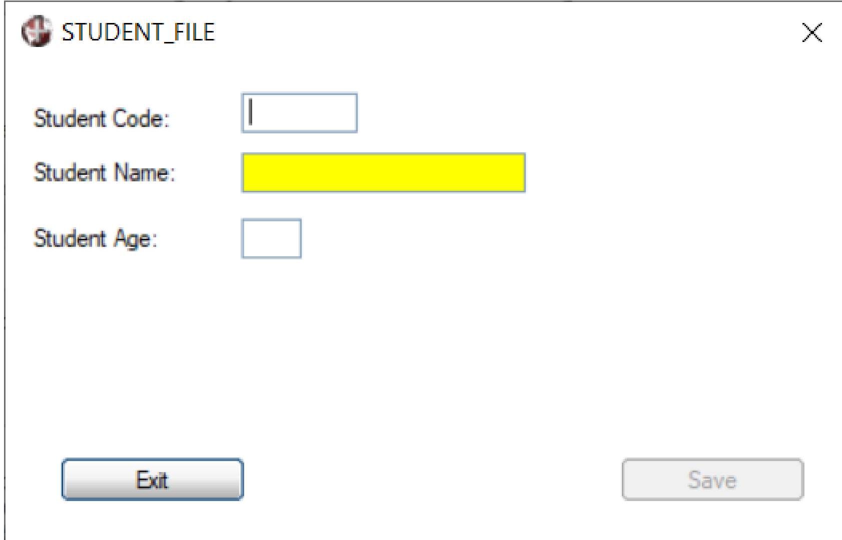
The **Color Selections** window displays. Select the Yellow color and click the [**OK**] button.



Save the panel by selecting [**Panel**] -> [**Save**] in the top menu bar and then run the panel by selecting the [**Test**] option.

We can clearly see the difference between the **Exit** button (enabled) and the **Save** button (disabled).

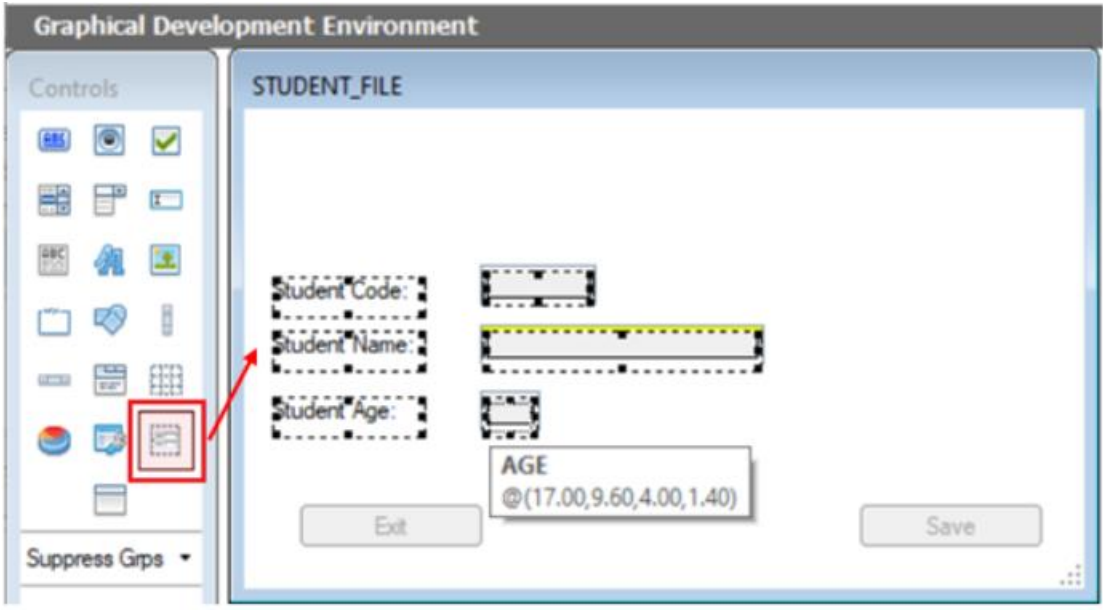
We can also see the Yellow background color for the **NAME** control:



Let's take the opportunity to decorate our panel a little.

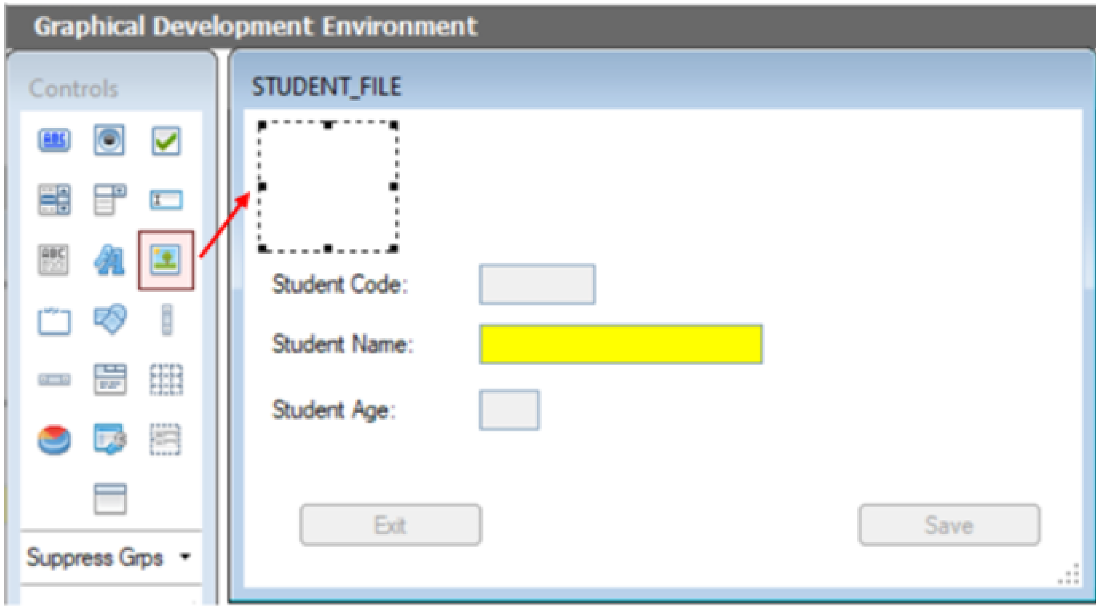
By using multiple selection (**Shift Click**) or the [**Group Items**] option from the **Controls** box, we select all the upper controls (all except the buttons). Once selected, we can move them down simply by dragging the mouse or by pressing the cursor keys.

Let's leave enough room to include an image and a title in large letters.



We proceed to create an image and text in large letters as a title. Select the [**Picture**] option from the **Controls** box, and then draw a control in the upper left part of the panel.

Note: If you make a mistake, you can correct the last action (or several actions) by using the Undo button (green arrow in the top menu) or by pressing the [**Ctrl Z**] keys.

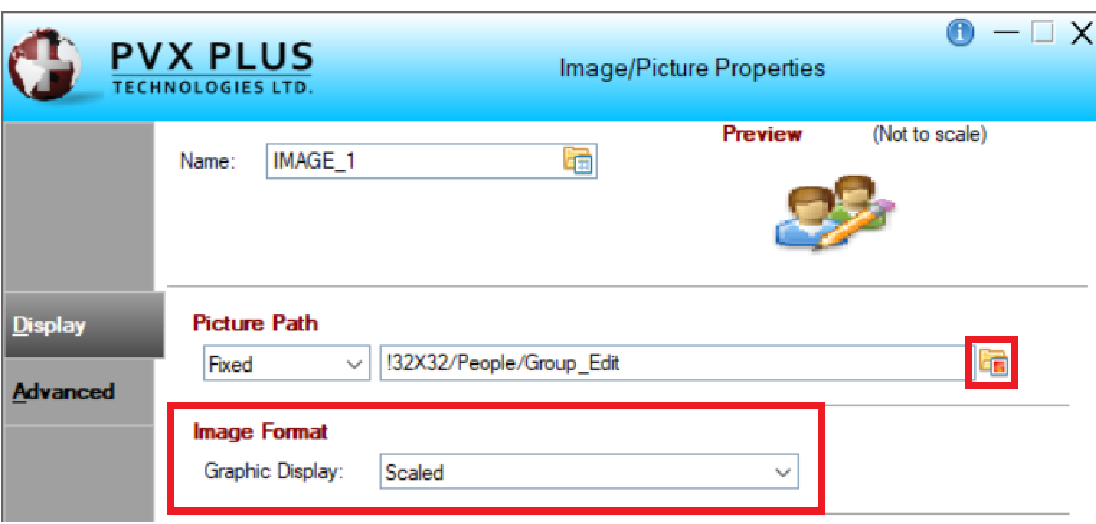


The **Image/Picture Properties** window displays. We will define an image in the upper left part of the panel and use one of the internal images.

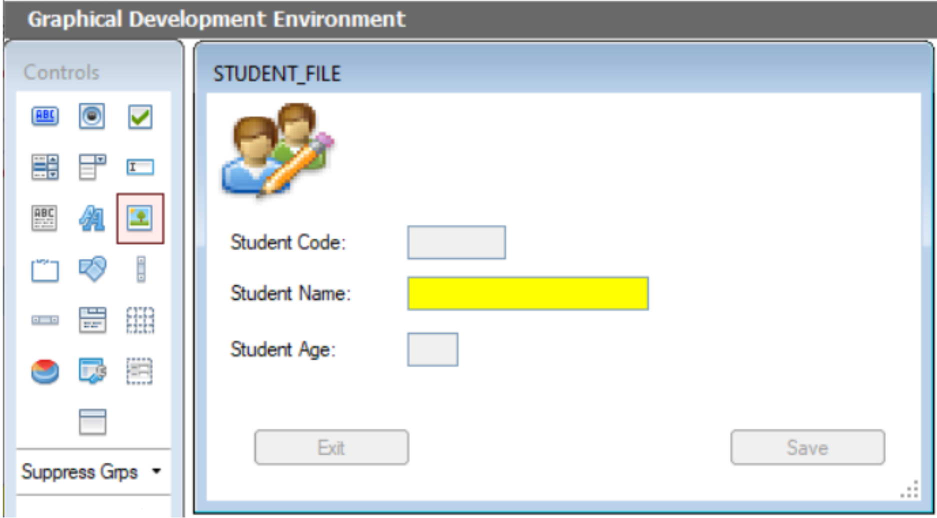
For [**Picture Path**], select the icon on the right (yellow folder with an image) to open the **Bitmaps** window, which displays a selection of predesigned icons.

You can select any image. To select the image in our example, click the **+** (*plus*) sign beside **Images Library** (located in the list on the left side) and then click the **+** (*plus*) sign beside 32x32. Select the "People" category from the list. Search for Group_Edit. Click on the image, and then click the [**OK**] button. The [**Picture Path**] now shows !32X32/People/Group_Edit.

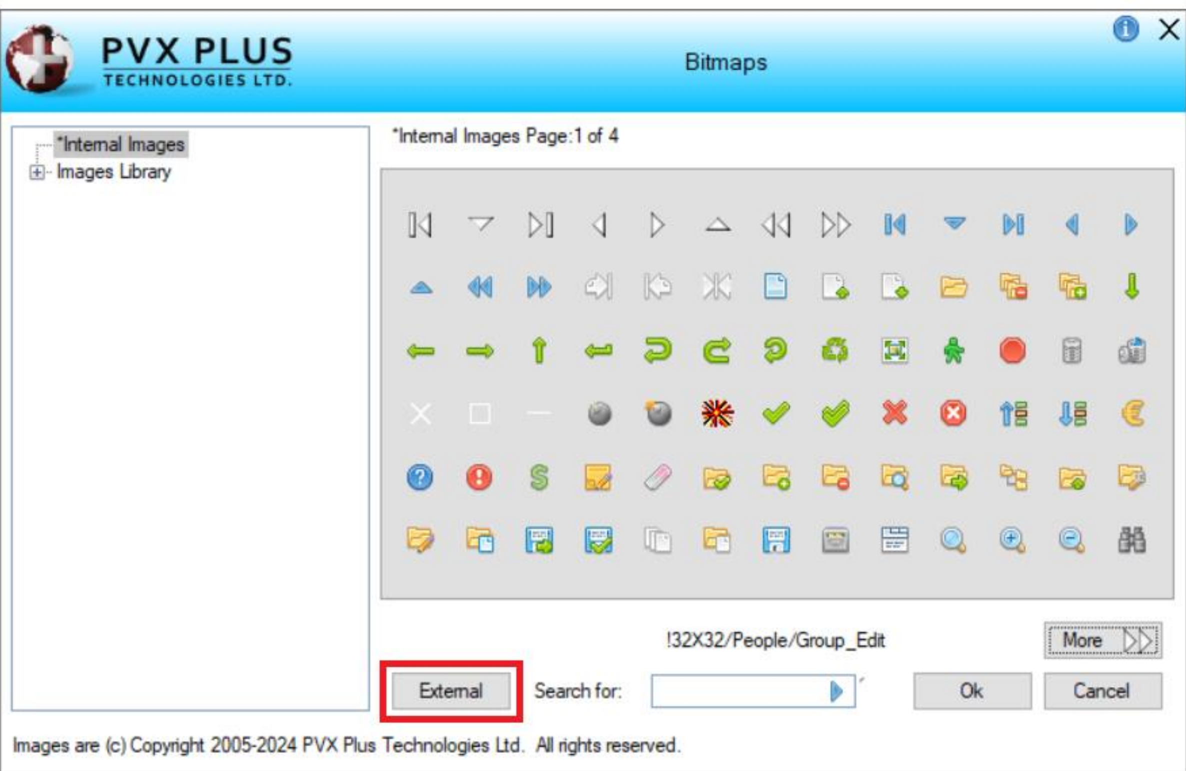
To make the image full size, select **Scaled** from the [**Graphic Display**] drop-down list.



The panel will look similar to the one shown below:



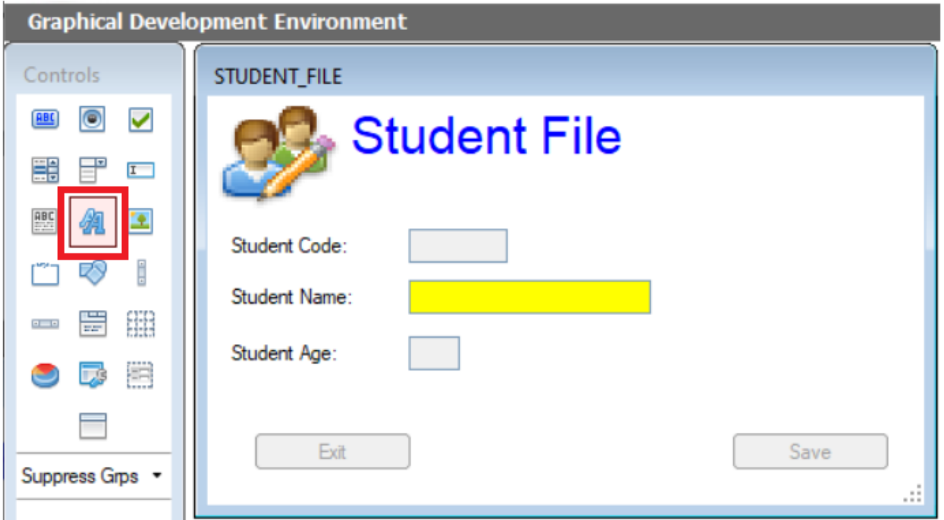
You can also select any compatible image you have on your computer by selecting the [**External**] button on the **Bitmaps** window, as shown below:



Now, select the [**Fonted Text**] control in the **Controls** box and draw a control large enough to place the title next to the image. The **Fonted Text Properties** window displays.

On the **Display** tab, in the [**Text**] field, enter **Student File**. On the **Font/Color** tab, select the Arial [**Font**] with a [**Size**] of 30 Points. For the [**Foreground Color**], select Light Blue.

The panel will look similar to the one shown below:

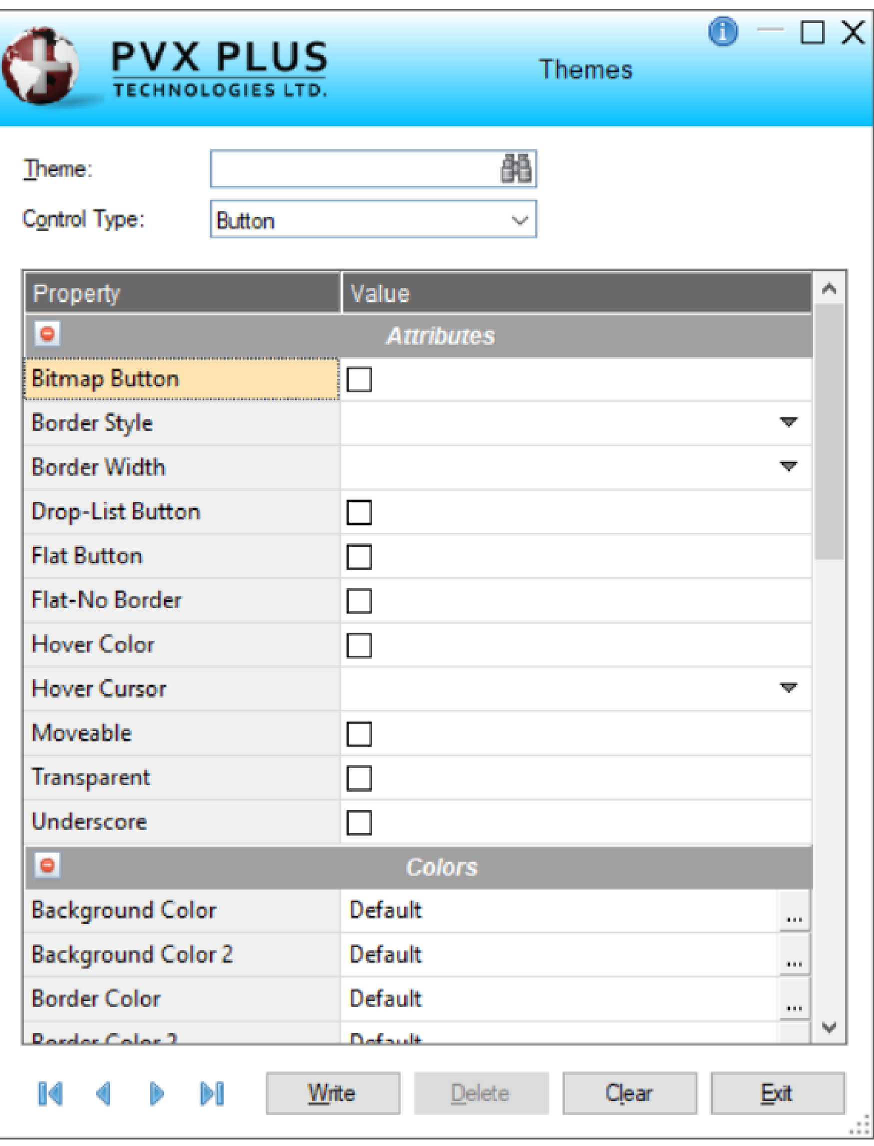


This little exercise demonstrates how easy it is to use NOMADS to design our panels so that they look attractive and modern.

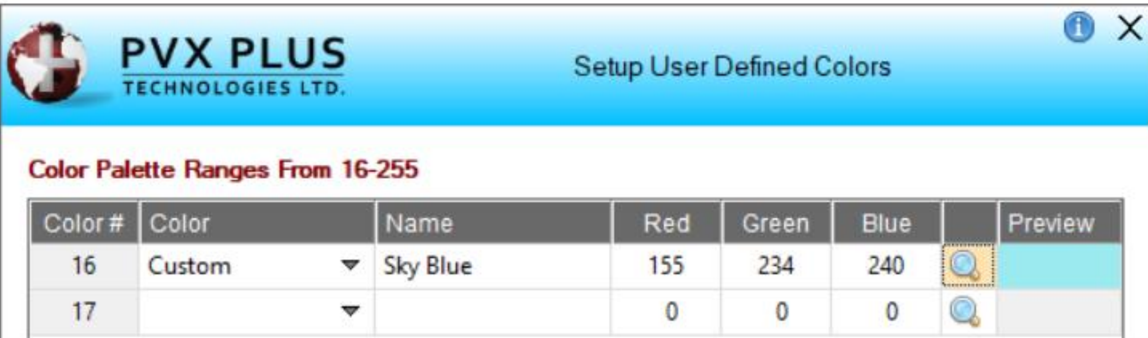
When you have a complete library, the idea is to define a **Theme**, which will give a modern and uniform appearance to all the controls and panels. This tool allows you to define color combinations and attributes for each control, including fonts.

PxPlus also offers **Visual Classes**, which are specifications of colors and attributes for a particular control (**Example**: Buttons).

Refer to [Themes]() and [Visual Classes]() in the PxPlus Help documentation.

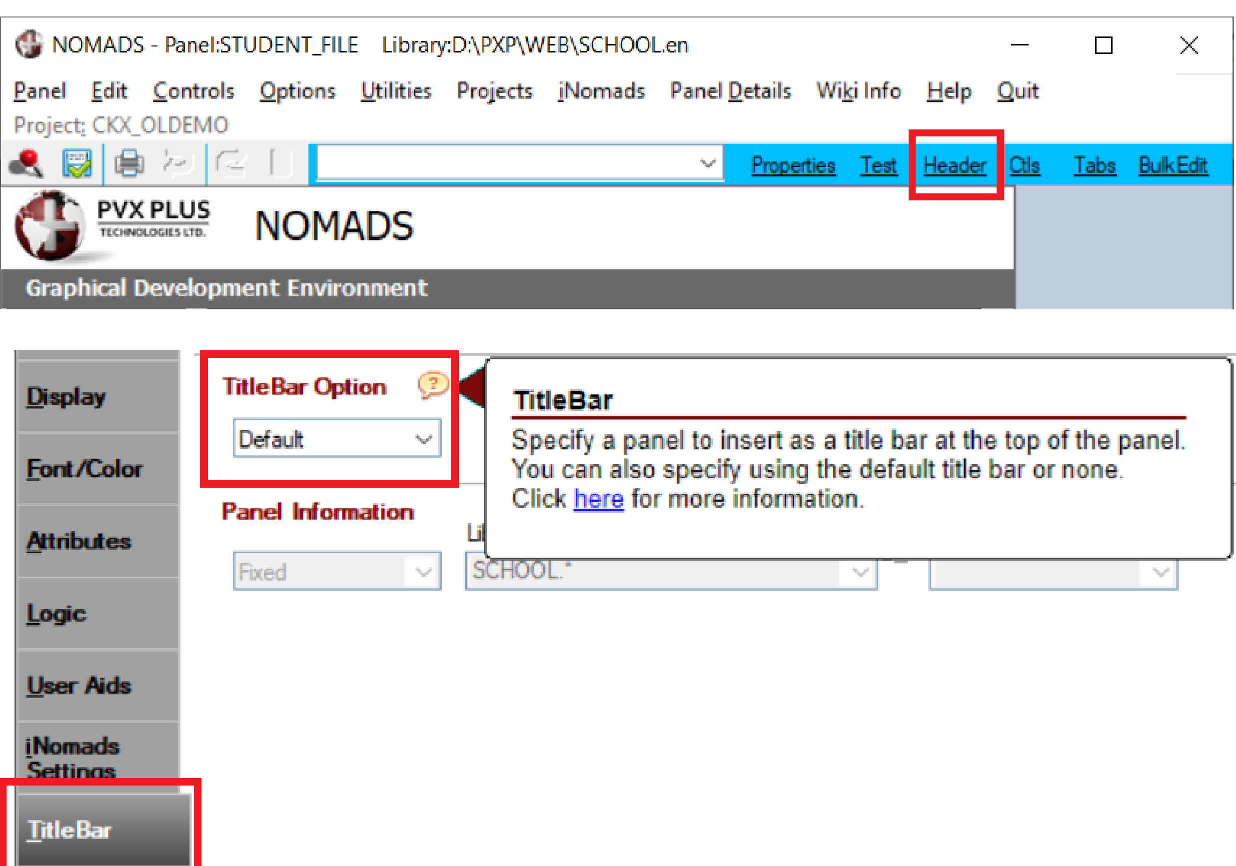


You can create your own colors beyond the eight basic colors and their eight variants by using the **Setup User Defined Colors** tool:



Refer to **Setup User Defined Colors** in the PxPlus Help documentation.

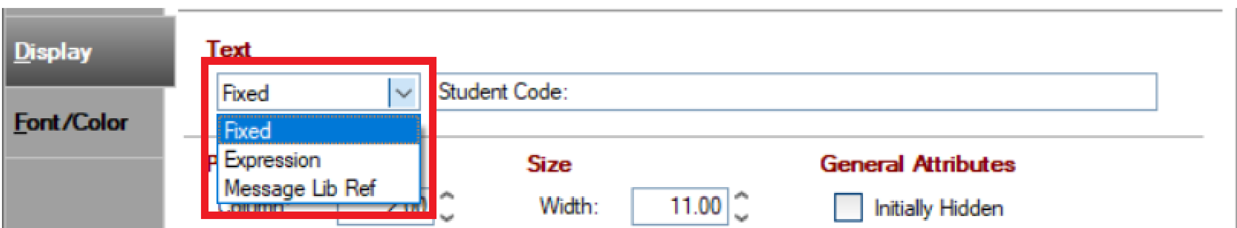
We can also define a header for the panel by selecting the [**Header**] option at the top. This opens the **Panel Definition** window. Select the **TitleBar** tab and then select the [**TitleBar Option**].



NOMADS offers a large number of tools to enhance our work and make it look more attractive and professional. We will look at other options as we progress through this book.

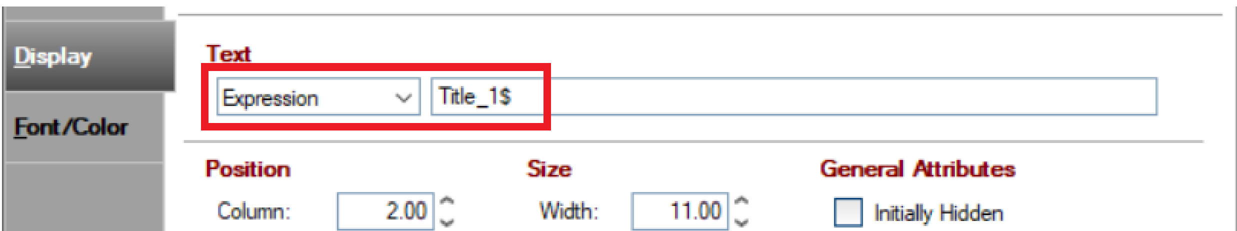
Dynamic vs. Fixed Properties and Values

You may have noticed that, when entering text in many of the NOMADS argument fields, such as in the [**Text**] property of a **Fonted Text** control, there is a drop-down list where there are several alternatives, such as **Fixed** and **Expression**:

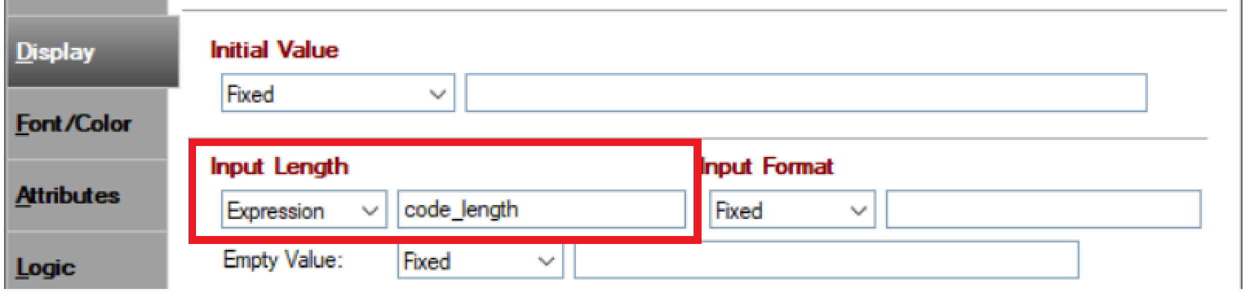


For the controls that have this as an alternative, it is possible to select **Expression**, and instead of entering a **Fixed** value, you can enter a variable or language expression (with information or a valid value).

For example, in a panel, dynamic titles could be placed. When the panel is executed, NOMADS evaluates the content of the variable **TITLE** and places it as the content of the control.



Suppose we have a panel that allows two types of products to be loaded, one with a length of 5 and the other with a length of 8. By modifying the [**Input Length**] field, it is possible to perform the validation:



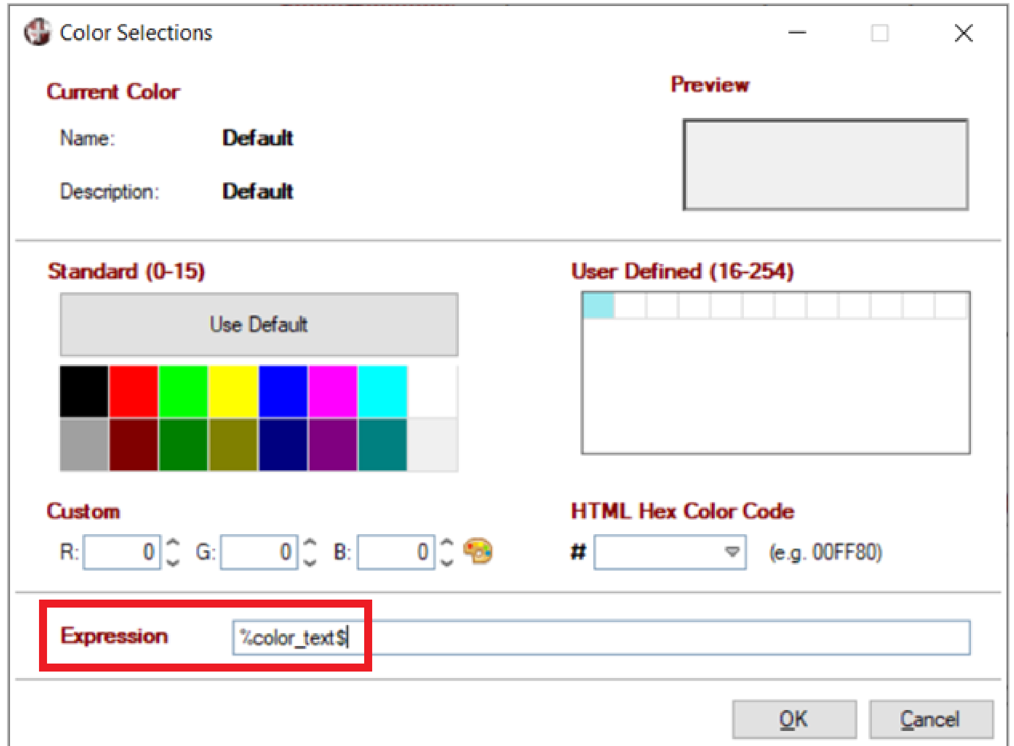
Depending on the type of product to be loaded, we modify the code_length variable with the value 5 or 8 to guarantee a correct entry.

In general, we can take advantage of the use of global variables denoted with the **%** (*percent sign*) in front of the name **%global_variable** to prefix values in all our panels.

Example:

```
start_routine:
%color_text$="RGB:64 128 128"
%background_color$="RGB:64 100 162" exit
```

Then, we place these values in the panels:



It is just an example. There are other ways to apply tones and colors to our panels and libraries. We just want you to see the possibilities. Even if you want some title or other value set at the beginning of the program and that remains throughout the session, we must use global variables.

Important Note: PxPlus distinguishes as ***separate*** variables **NAME** (local variable, numeric), **NAME$** (local variable, alphanumeric), **%NAME** (global variable, numeric) and **%NAME$** (global variable, alphanumeric), but **NAME**, **Name** and **name** if they refer to the same variable.

To summarize:

Usually, a NOMADS parameter or argument (generally anywhere where a text or numeric value can be entered) can be replaced with a variable of the same type.

An advantage of PxPlus is that all components start from the same common base, use the same language, the same variables and information structures. Communication between processes and tools is transparent at all times, based on the same language and using the same environment. In fact, PxPlus tools are developed in the PxPlus language!

NOMADS Reserved Variables

To facilitate many tasks for the programmer and give him/her even more complete control over the information and processing flow, NOMADS has a series of preset variables that will have valuable information about our panels, their controls and their behavior. There are three groups: ***Reserved*** variables, ***Control*** variables and ***Global*** variables.

For now, we will not make distinctions in these three categories. We are going to mention some of them and look at their function and use.

Refer to **NOMADS Variables** in the PxPlus Help documentation.

Note: There is another section that is very important, which is the functionality of NOMADS as an object. You will see that there are dozens (perhaps more than a hundred?) properties of the %NOMADS object to control and know in greater detail the operation of NOMADS. For now, let's look at the main variables and their use.

Control Variables

In these variables, we must replace **OBJ** with the name of the control we are dealing with.

> **OBJ** and **OBJ$:** This variable will have the value of the control, **OBJ**, if the control has a numeric value or **OBJ$** if it is alphanumeric. This only happens if the **Suppress.VAL** attribute was turned Off (see the next variable).
>
> **OBJ.VAL** and **OBJ.VAL$:** If the **Suppress.VAL** attribute has not been turned Off, the previous variables will be replaced by **OBJ.VAL** (numeric content) or **OBJ.VAL$** (alphanumeric content).
>
> **OBJ.CTL:** NUMERIC identifier of the object. Internally, all controls have a unique numeric identifier. It is known as the **CTL** (**C**on**T**ro**L**) value.
>
> **OBJ.TAG$:** Property or tag available to the user at the time of creating the control. For example, if our control is called NOTE_CODE, the variables would be NOTE_CODE$, NOTE_CODE.VAL$, NOTE_CODE.CTL, and NOTE_CODE.TAG$.

Reserved Variables

The variables in this list can be used to understand or alter the operation of NOMADS. We will look at some of the most relevant ones:

> **_OEM$:** Expanded information about the key pressed, including mouse status.
>
> **ARG*x*$:** List of arguments passed between panels or programs. If you want to restrict access to certain information, you can use via the **CALL** command up to twenty (20) arguments. NOMADS will replace each of them with the variables ARG1$, ARG2$, etc. until the number of arguments sent is complete (maximum 20).
>
> **CHANGE_FLG:** This variable will have a value other than zero. ***If any control*** on the panel changes its value or state, NOMADS does not reinitialize it. If you wish, you must do so.

CMD_STR$: Programmatic flow modification logic. You can use it to end a panel from your program (CMD_STR$="END") or to determine which panel will be executed next (CMD_STR$="Jpanelname"). For the full syntax, refer to **CMD_STR$** under **NOMADS Reserved Variables** in the PxPlus Help documentation.

DISP_CMD$: Set of instructions to be executed after displaying a panel.

EXIT_CMD$: Set of instructions to be executed before executing an EXIT.

ID: Identifier (CTL Value) of the current control.

ID$: Name of the current control.

NEXT_ID: Identifier (CTL Value) of the control that will receive focus next.

PRIME_KEY$: In a query, this variable brings the value of the primary key.

REFRESH_FLG: This flag is used to turn On the [**Auto Refresh**] attribute of the panel if it has not been activated. You can also turn it Off if you wish.

SCRN_ID$: Name of the current panel/shape.

SCRN_LIB$: Name and path of the current library.

There are many more variables! PxPlus offers complete control over the functionality and manageability of NOMADS.

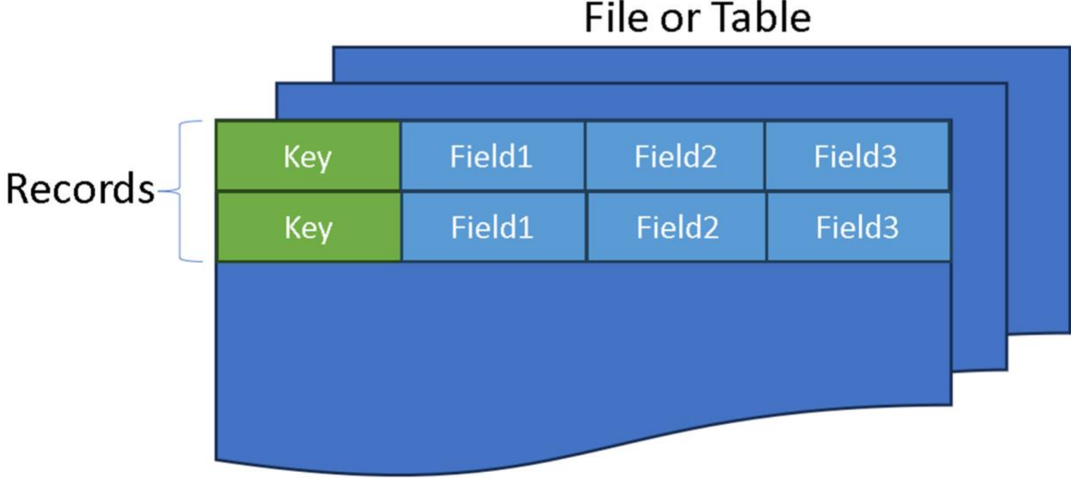
Refer to **NOMADS Variables** in the PxPlus Help documentation.

4. Introduction to Files and Tables

PxPlus is a language with more than 40 years of history oriented to the development of administrative and commercial applications where saving and retrieving information quickly and reliably is vital.

Although all those years have passed, a table is basically the same type of file, and the principle remains the same: handling information quickly and accurately.

The file or table is basically a storage unit with a physical name and some data organized as a group (normally called ***records*** or ***columns***) and divided into ***rows*** or ***fields***. Each record is identified with a unique value called a ***key*** or ***access key***.



At the program level, a file or table is defined, the length and/or name of the key is specified, as well as the name and length of each field or row. Visually, it can be seen like this:

Data Elements

Field	Dtl	Field Name	Data Class	Description	Type	Len
1		Code		Code	Str	6
2		Name		Name	Str	50
3		Age		Age	Str	2
4		Sex		Sex	Str	1
5		Course		Course	Str	2

PxPlus continues to use the basic syntax for file access: logical communication channels. This allows using the same routine for several files. The use of a table requires an assignment or channel opening:

open (channel_number)"file_name"

Example:

```
open (7)"STUDENT.DAT"
```

From this moment on, the logical channel (7) will be for communication with the STUDENT.DAT table until it is terminated using the command **close** (7).

Example: Let's see an example of using a file.

```
open (9)"student_data"
read (9)record$
…
close (9)
```

The use of channel 9 is exclusive (at this time) for the "student_data" file until it is released using the command close (9); that is, each table must have a unique access channel, up to 65,356 channels. This limit is normally higher than that imposed by the operating system.

The basic table/file operations are:

OPEN	Assigns a logical communication channel with a file or device.
CLOSE	Releases a logical communication channel.
READ	Reads the content of a table, according to a list of data or records.
WRITE	Writes information to a table.

When we save or read information, we can/should specify what each field contains:

```
input "Code: ",code$
input "Name: ",name$
open (25)"people_table"
write (25,key=code$)code$,name$
close (25)
```

In this case, we ask for two values and store them in the variables code$ and name$. We open the file through logical channel 25. We save the information using a clause **KEY=** that indicates that the internal sorting method will be the variable below, and that the record will be made up of two columns or fields: CODE and NAME.

To fill a file with many records, we modify the previous routine:

```
start:
        open(25)"people_table"

ask_for_fields:
        input "Code: ",code$
        input "Name: ",name$
        write(25,key=code$)code$,name$
        input "More: ",resp$
        if resp$="YES" then goto ask_for_fields

finish:
        close(25)
        end
```

This routine has three tags: **start**, **ask_for_fields** and **finish**. The **input** command is used to enter information from the keyboard. The **if…then…goto** condition is used to return to the ask_for_fields routine if the answer is "YES", and the finish routine does that.

Several considerations about the previous example:

a) It is possible to use any integer for the channel from 1 to 65535. It just has to be free.

b) The **end** command also closes the channels so we can say that the close (25) is unnecessary, but we wanted to show more commands.

c) The condition only accepts "YES" as an answer; any other will cause the program to end.

d) It is possible to set the list of input/output variables of the table in a separate statement:

```
start:
   a_list: iolist code$,name$
   open(25)"people_table"

ask_for_fields:
   input "Code: ",code$
   input "Name: ",name$
   write(25,key=code$)iol=a_list
   input "More: ",resp$
   if resp$="YES" then goto ask_for_fields

finish:
   close(25)
   end
```

Both routines are equivalent, and the second provides a little more flexibility since, if there are several input and output operations, the list of variables must be repeated each time one is performed. On the other hand, if it is defined in a list called an **IOLIST** (Input Output List), it is defined only once and will be used many times.

Note that the structure of the **write** command changes to reflect that the list of fields is in the "a_list" label.

Once we have stored information, we can read this information. To do this, we are going to incorporate some of our PxPlus words into our vocabulary. This routine reads the entire content of the table and displays it on the screen:

```
start:
   open(5)"people_table"
read_fields:
   read(5,end=end_cycle)code$,name$
   print code$,name$
goto read_fields
end_cycle:
end
```

Here, we see the **read** command, which allows us to read the contents of a table. It has an **,END=** clause that indicates that it should go to the next routine if the end of the table is found (the routine is end_routine). Then, we have the **print** command that displays the content of the variables code$ and name$ on the screen. Note that we have not placed the close statement. We simply end with the **end** command.

Important Note: Don’t worry about the aesthetic or functional aspects now. Let's continue reviewing the syntax and construction of table reading/writing routines, and then let's move on to something much simpler and faster. But for now, let's reinforce the theoretical part.

We will change the routine to incorporate a reading of the fields according to a user's input:

```
start:
   open(5)"people_table"
   a_list: iolist code$,name$
ask_for_data:
   input "Code to query (F4=leave): ",data$
   if ctl=4 then goto end_routine
read_fields:
   read(5,key=data$,dom=no_exist)iol=a_list
   print code$,name$
   goto ask_for_data
end_routine:
   end
```

This routine uses a condition in a way that we do not know, comparing a variable called **CTL** (**C**on**T**ro**L**), which uses some special values. The keys F1 to F4 return the CTL value 1 to 4; that line indicates that if the user presses the F4 key, they must go to the **end_routine** routine. The next highlighted line is the one for reading the file. It uses a **,key=** clause that specifies which variable will be used as the access key. The variable **data$** contains the value entered by the user (**,key =data$**). There is also a clause **,dom=** that indicates if the searched record does not exist (**DOM** means **D**uplicated **O**r **M**issing), it must go to the routine NO_EXIST (**,dom=no_exist**).

Note: The command **print code$, name$** is equivalent to using the command **print iol=a_list**.

Refer to these sections on file/table handling within this book: **A Complete Routine for Table/File Handling** and **How to Read a Table from Last to First Record (Reverse Order)**.

Although it is interesting to know this manual part of programming, we are going to explore in more detail the definition of a table for a simple application, but graphically, using NOMADS. We will take an example, and we will analyze the steps to solve this first stage of our problem.

Normally, a table is composed of a group of data or elements that have unique characteristics, such as Name and Length.

Note: The design of a table can have a lot more information, such as: Relationships, Data Type; Format; Validations, etc.

For now, we are going to keep the exercise as simple as possible.

Apart from the Name and Length of the elements, PxPlus requires only one additional piece of information - the internal ordering method of the table; that is, what ordering criteria the system will follow. As we will see later, all this could be variable and dynamic.

Along with this information, we must provide the name of the table or file.

Exercise: Defining a Table

Our client requires an application from us to register their student enrollment at their institution. The basic information of the student would be:

ID or Student Code (6) << Ordering Factor or Code

Student Name (50)

Student Age (2)

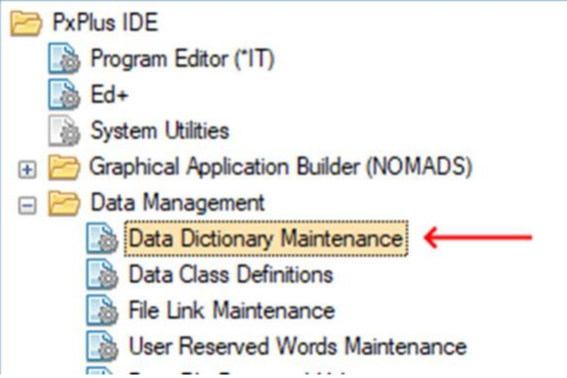
Sex (1)

Course (2)

This data must be added in PxPlus with the name of the table, which is actually two names: a Logical name and a Physical name (name of the file on disk). Later, we will see the reason for this.

Summing up, we have a Students table that will have Student Code, along with their Name, Age, Sex and the Course they are studying.

From the main menu of the PxPlus IDE, we open the **Data Management** category and double click on the **Data Dictionary Maintenance** task:

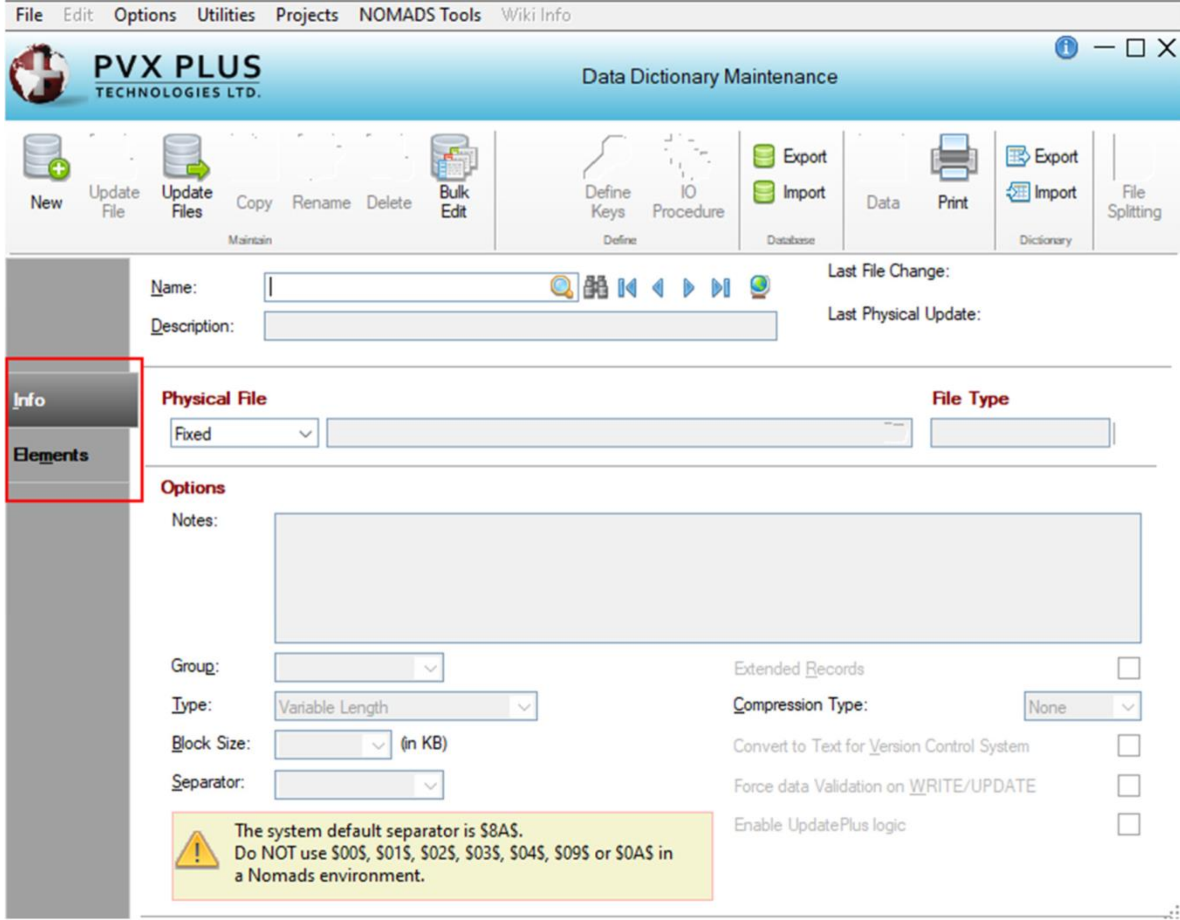


If PxPlus has been recently installed, it is possible that some control files do not exist, and it will ask for confirmation to create them. You must answer affirmatively in all cases.

Then, the **Data Dictionary Maintenance** window will display, similar to the one below.

Refer to **Data Dictionary Maintenance** in the PxPlus Help documentation.

On the left side of the screen, there are two tabs: **Info** and **Elements**.



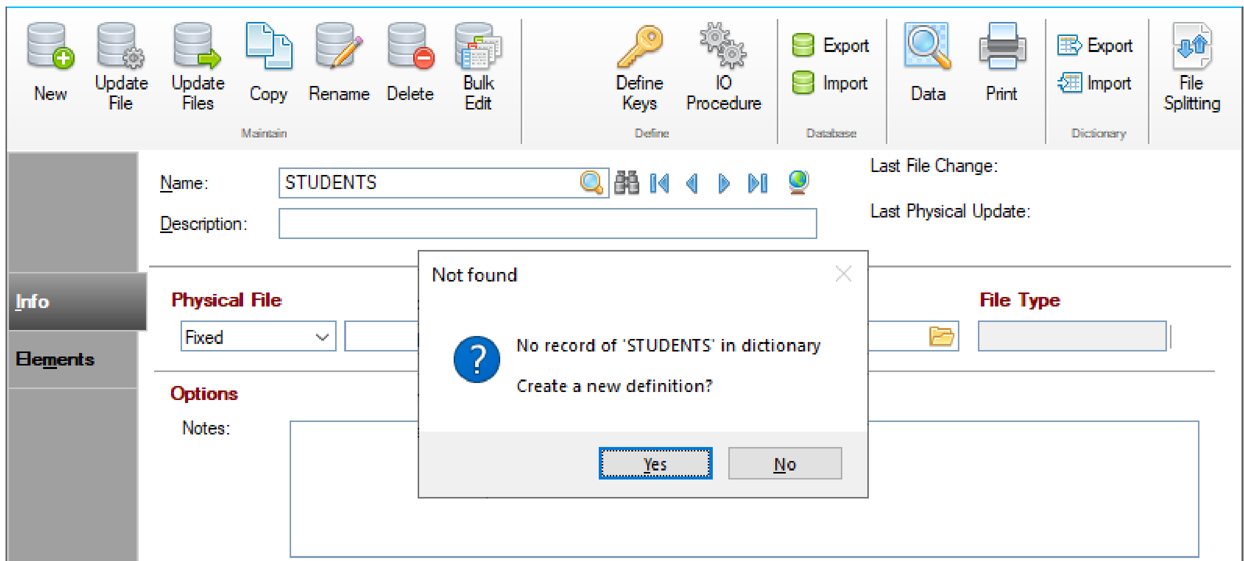
The **Info** tab allows us to define the basic data of the table: the **Name** and **Description** of the table and the name of the **Physical File** on the computer.

Note: You can see that there is much more information, but for now, we will only need those three pieces of information.

The second tab, called **Elements**, is used to define the elements or fields of the table.

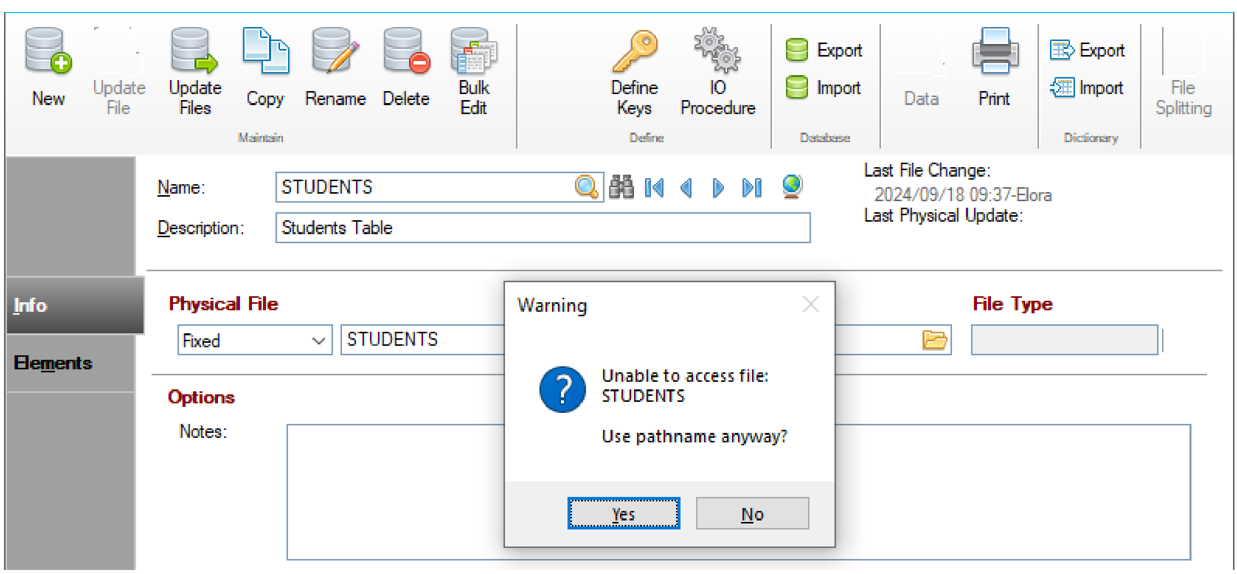
For the **Name** of the table, we will enter **STUDENTS**.

Since the table does not exist, this message will appear:



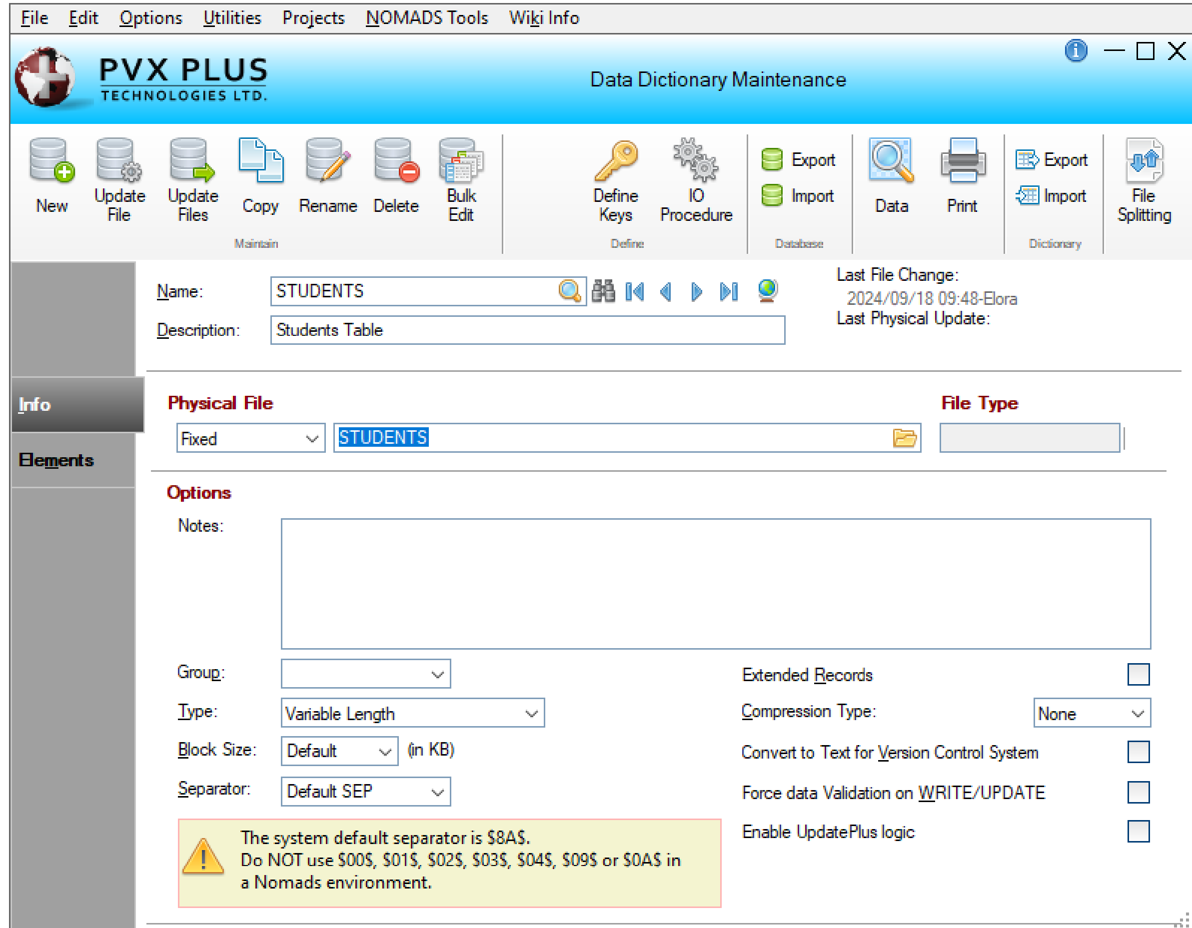
Answer **Yes** for that definition to be added in the system.

In the **Description** field, enter **Students Table**. For the name of the **Physical File**, enter **STUDENTS**.



Again, since that file does not exist, the system asks us for confirmation. Answer **Yes.**

The **Data Dictionary Maintenance** window looks like this:



This is the basic and minimum information for creating a table: a table (Logical) name, which is the identifier of the storage structure or file. This name may ***be different*** from the physical file name.

A table for PxPlus is an entry to the Data Dictionary and serves to define a data structure, detailing the content of the stored information, the characteristics of each element and its order (among other things).

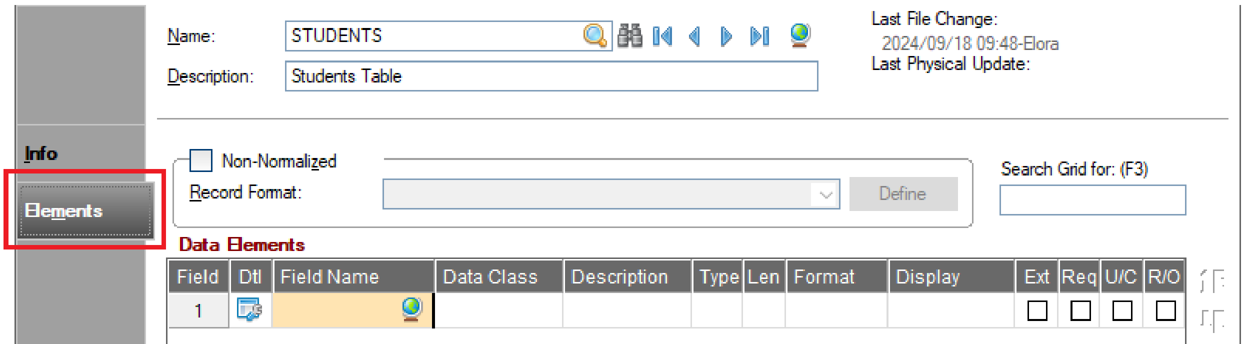
On the other hand, a physical file or file name is a unique entity on disk, which has data stored according to the specification (structure) of the previous table.

It could be the case that ***one table*** defines the structure of ***several physical files***.

For instance, in our example, we could use the same table **STUDENTS** to define two files or physical files: SCHOOL_1 and SCHOOL_2. Although it is different information, both use the same structure.

Later, we will continue with other examples to better analyze and understand the difference between the two.

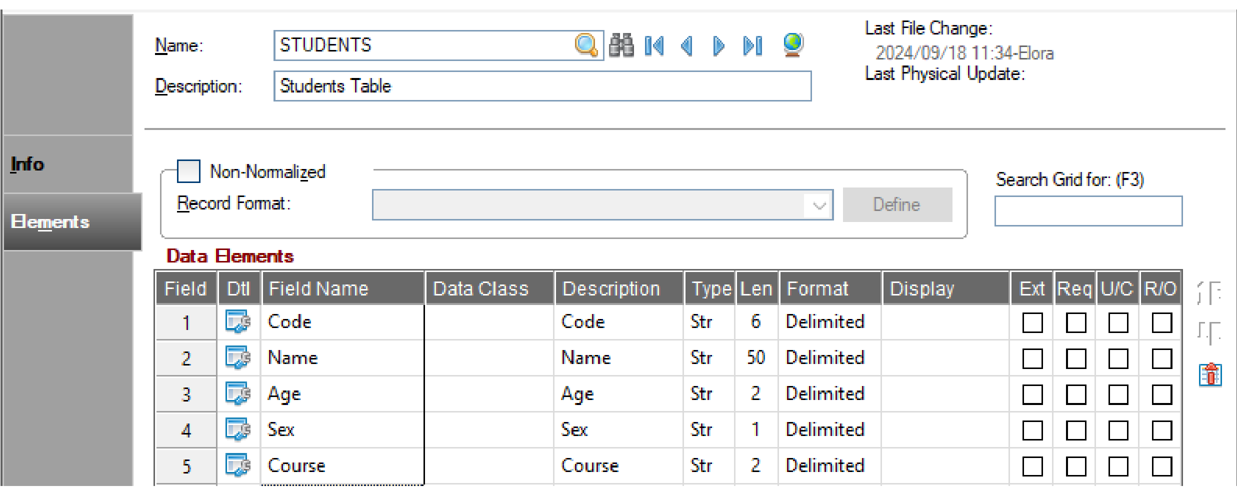
Once the basic data of the table has been defined, clicking the **Elements** tab (on the left side of the window) displays the **Data Elements** definition:



We are going to enter the minimum necessary data: **Field Name**, **Description**, **Type** and **Length.** The basic information would be:

Student Code (6)
Student Name (50)
Student Age (2)
Sex (1)
Course (2)

After entering this information, the **Data Elements** definition should look like this:



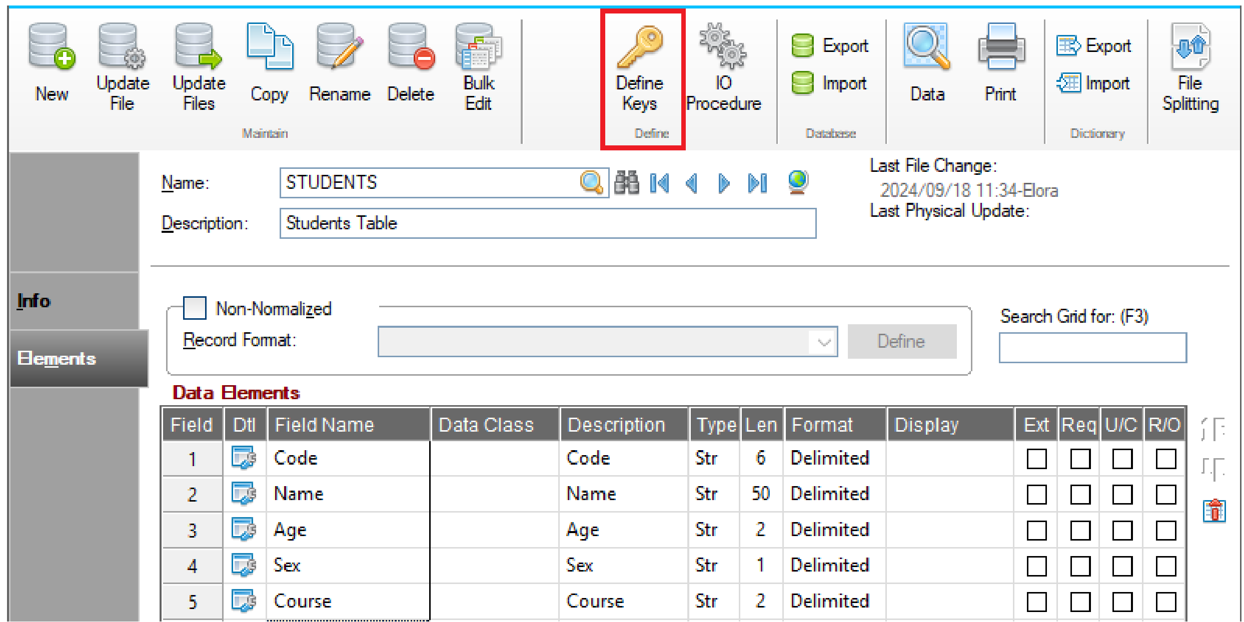
We will duplicate the **Field Name** and **Description**. We will need this for when we use automatic generation, as well as for other future cases. The **Type** is the internal storage structure. The **Age** could have been entered as numeric. For now, we will leave it like this, and then we will see the difference.

An additional advantage that this system has is that this information we are defining will be registered within the file so that any other PxPlus tool can use this information.

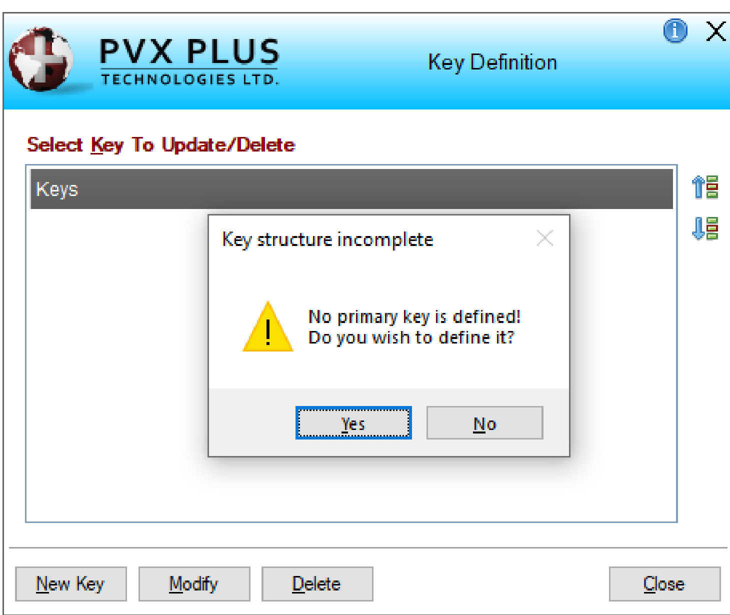
Once the elements that make up the table have been defined, we must define a key or sorting key, which will be the ***main*** criterion (but not the only one) for ordering the information contained in the

table. It is important to note that PxPlus requires at least one sort key or primary key.

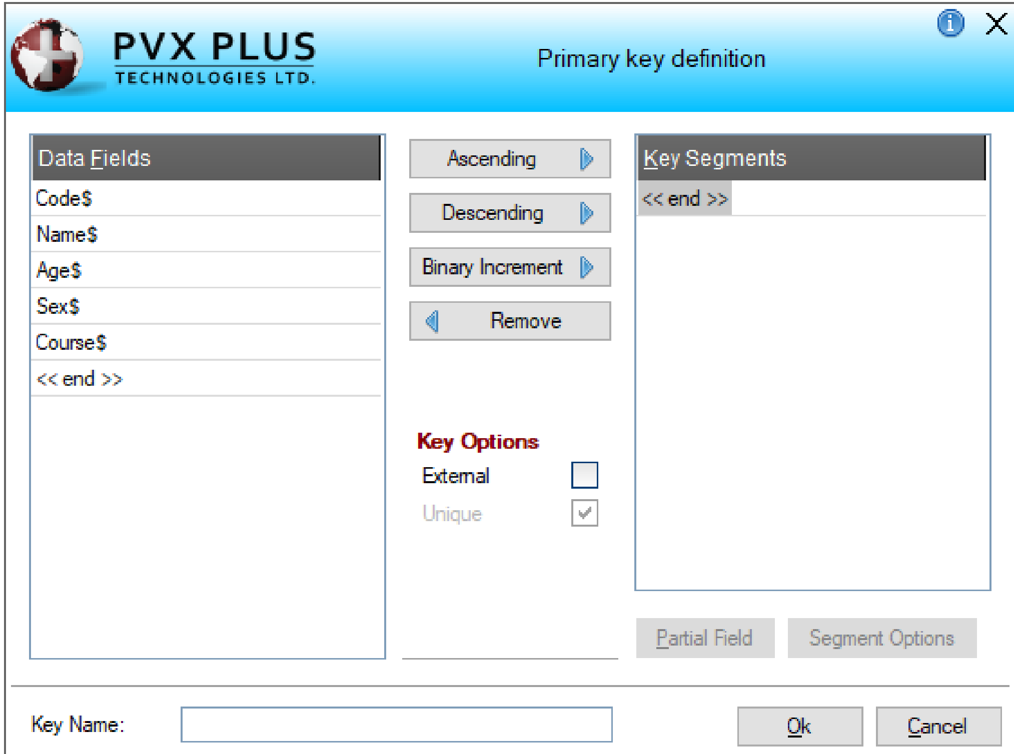
To define our sort key, we must click the [**Define Keys**] option at the top of the panel.



We see a message that indicates we do not have a defined key (remember that at least one key is mandatory). Answer **Yes**.



A **Primary key definition** window displays, similar to the one below:



Refer to **Defining Keys** in the PxPlus Help documentation.

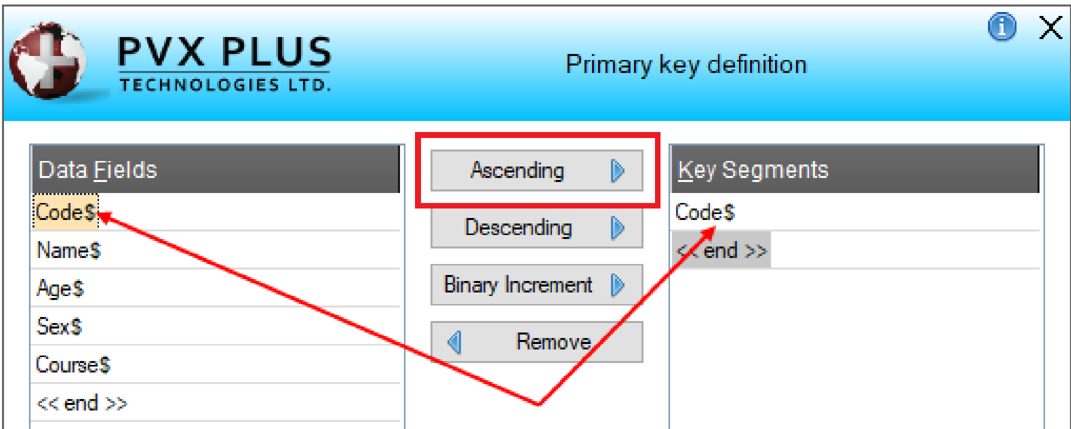
We have a list of the table elements, at least one of which will be defined as the sorting method, either ascending or descending.

Another unique characteristic is that we can define composite keys, which is made up of more than one element (**Example:** CODE$+NAME$).

The primary key ***must*** be unique. In this example, there cannot be two or more records stored under the same **CODE**. In case it repeats itself, we must redefine our table.

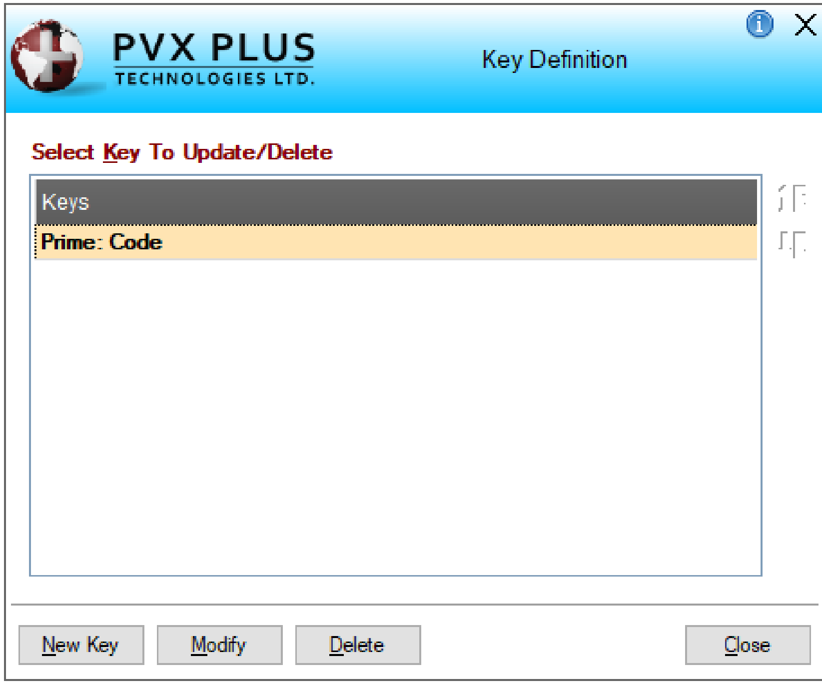
Note: There are structures that allow records with duplicate keys.

To define **CODE** as our primary key, double click on **Code$** or select it with the mouse and then click the [**Ascending**] button.



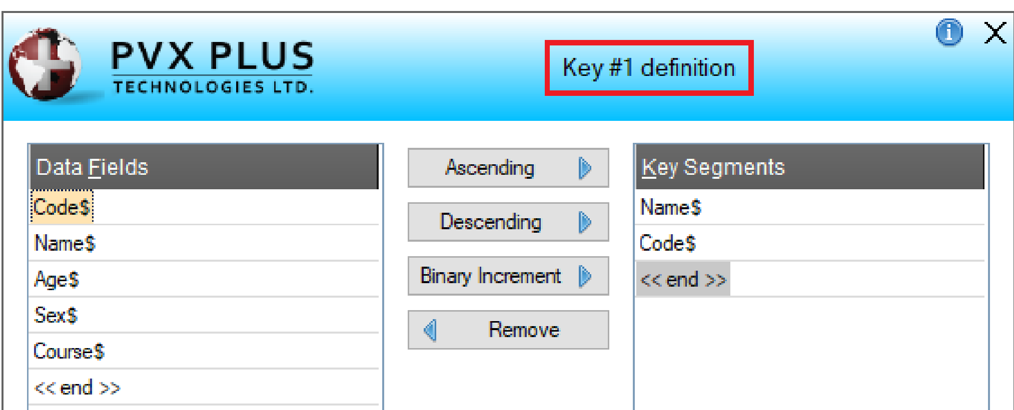
Normally, several keys or sorting criteria are defined. In our case, we could select NAME$+CODE$ as an alternate key.

To do this, we must first complete the definition of the primary key by selecting the [**OK**] button. With this step, we will have defined the primary key (or Key #0).

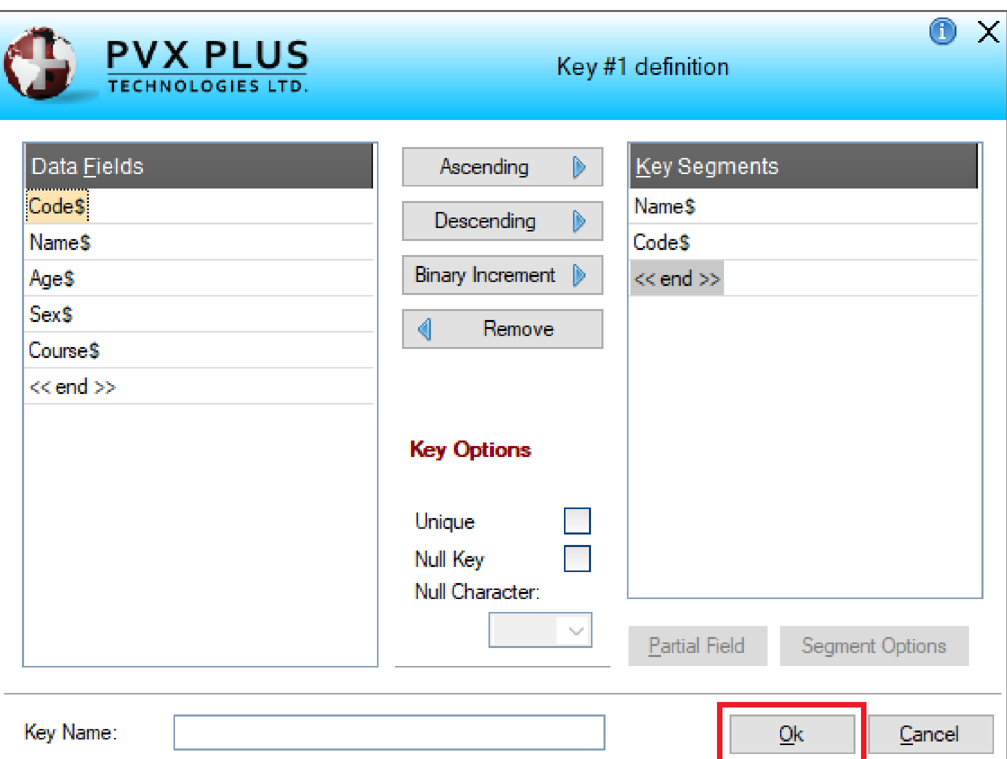


Run the **Key Definition** panel again by clicking the [**New Key**] button.

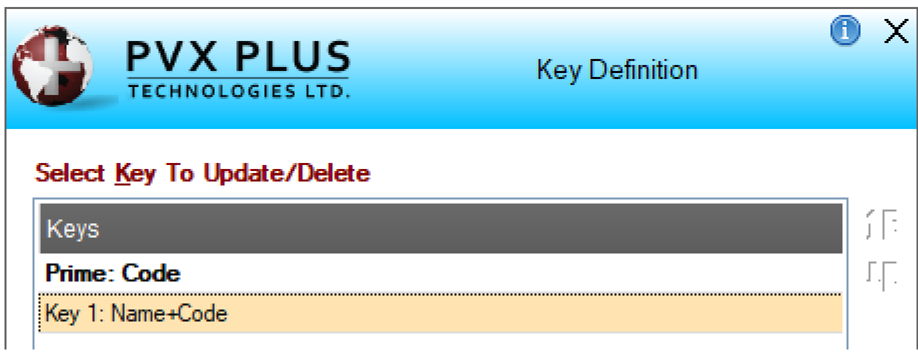
To define the second key (Key #1), we must first double click on **Name$** and then double click on **Code$**. Another way is to select **Name$** using the mouse, then click the [**Ascending**] button, and then select **Code$** and then click the [**Ascending**] button again.



To finish, click the [**OK**] button at the bottom.



The **Key Definition** panel now shows two keys:



It shows the Prime (or primary) key that is made up only of the **CODE** element and an alternate Key 1 (Key #1) that is made up of the **NAME+CODE**.

Why define the alternate password with **NAME+CODE** and not just define it with the **NAME**?

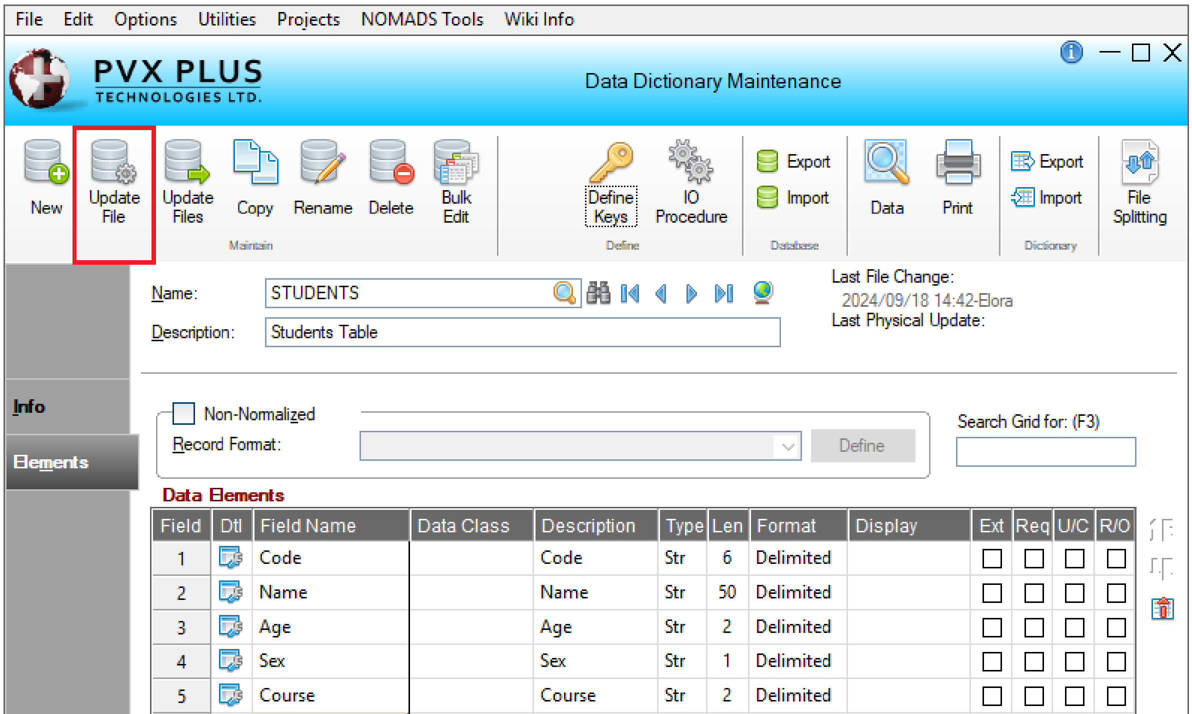
Repeated names are very common in some countries (**Example:** "John Smith" or "Peter Lewis"). It is possible that there are records with duplicate alternate keys (a situation fully permitted by PxPlus), but that, for practical purposes, could constitute a headache. To avoid this, it is suggested to add the code as an additional component of the key. Thus, if there are two people with the same name, PxPlus will try to order both names according to their student code; effectively avoiding this possible duality of records.

Once the name of the table, its elements and its keys have been defined, the last step is to tell PxPlus to define this structure (file) on the disk; that is, to proceed with the creation of a file with the previously specified characteristics.

Physically updating the file will also adjust the file, if it has content, so that it adapts to the new structure. That is, if there is information, it will convert that data to reflect the proposed changes.

If the file is new, it will simply create the same file on the disk with those specifications.

To proceed with the creation or physical update of the file, click the [**Update File**] button located in the upper left part of the **Data Dictionary Maintenance** window.



Once the file is created and the process is finished, a confirmation window will appear. With that, we have defined our first storage structure or table.

To summarize:

To define a table, we select **Data Dictionary Maintenance** (after opening the **Data Management** category from the PxPlus IDE main menu). We enter the name of the table and the file, and then the elements or fields.

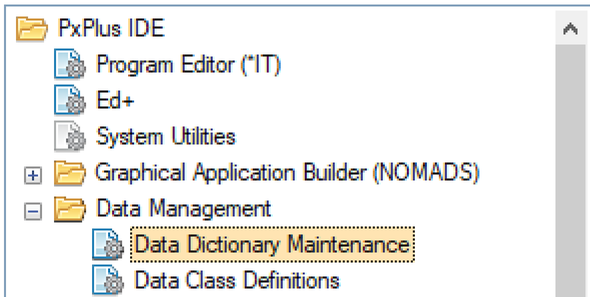
For the next step, we define the keys or sorting methods. At the end, we select the [**Update File**] button.

To learn about files and tables in more detail, refer to the section **Files and Tables: An In-Depth View** within this book.

5. Automatic Tools: File Maintenance and Query Manager

A little knowledge is enough to understand that there are many variables at play when defining a file and then controlling access to it or obtaining information back. This involves many commands to read, verify and display information. However, PxPlus offers, through NOMADS, tools that allow you to take advantage of the functionality of recording, reading and deleting information in a table from scratch. This is known as **File Maintenance**.

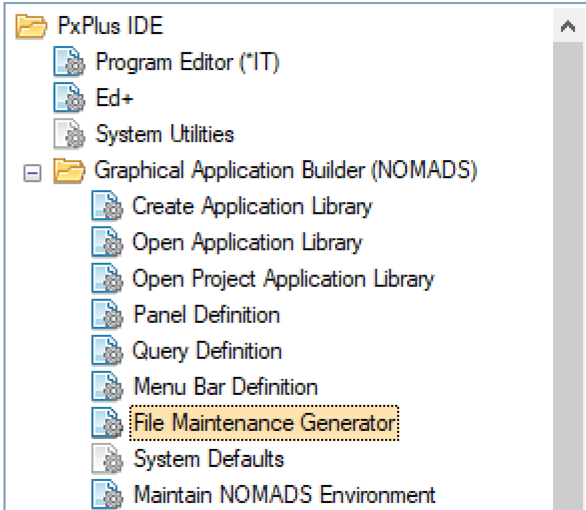
For these examples, it is necessary that we have a table defined through **Data Dictionary Maintenance** (open the **Data Management** category from the PxPlus IDE main menu):



If you completed the previous exercise, you will have a table called **STUDENTS** already created and defined.

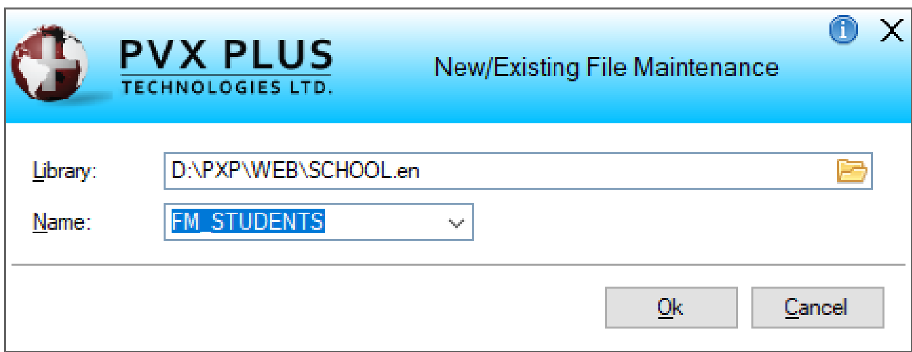
NOMADS File Maintenance

To run the **File Maintenance** tool, double click on the **File Maintenance Generator** task (open the **Graphical Application Builder (NOMADS)** category from the PxPlus IDE main menu):



Refer to **Accessing File Maintenance Generator from IDE** in the PxPlus Help documentation.

A window will display, asking for the name of the library and the panel:

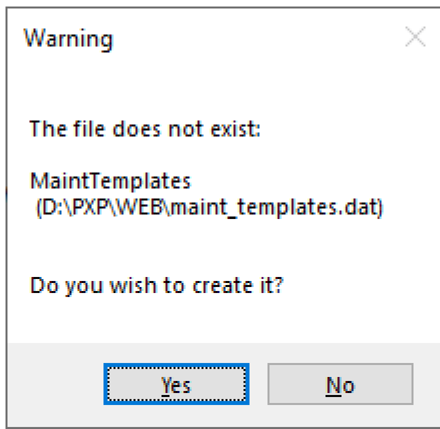


Note: Remember that a panel is a form or window that is defined within a NOMADS library.

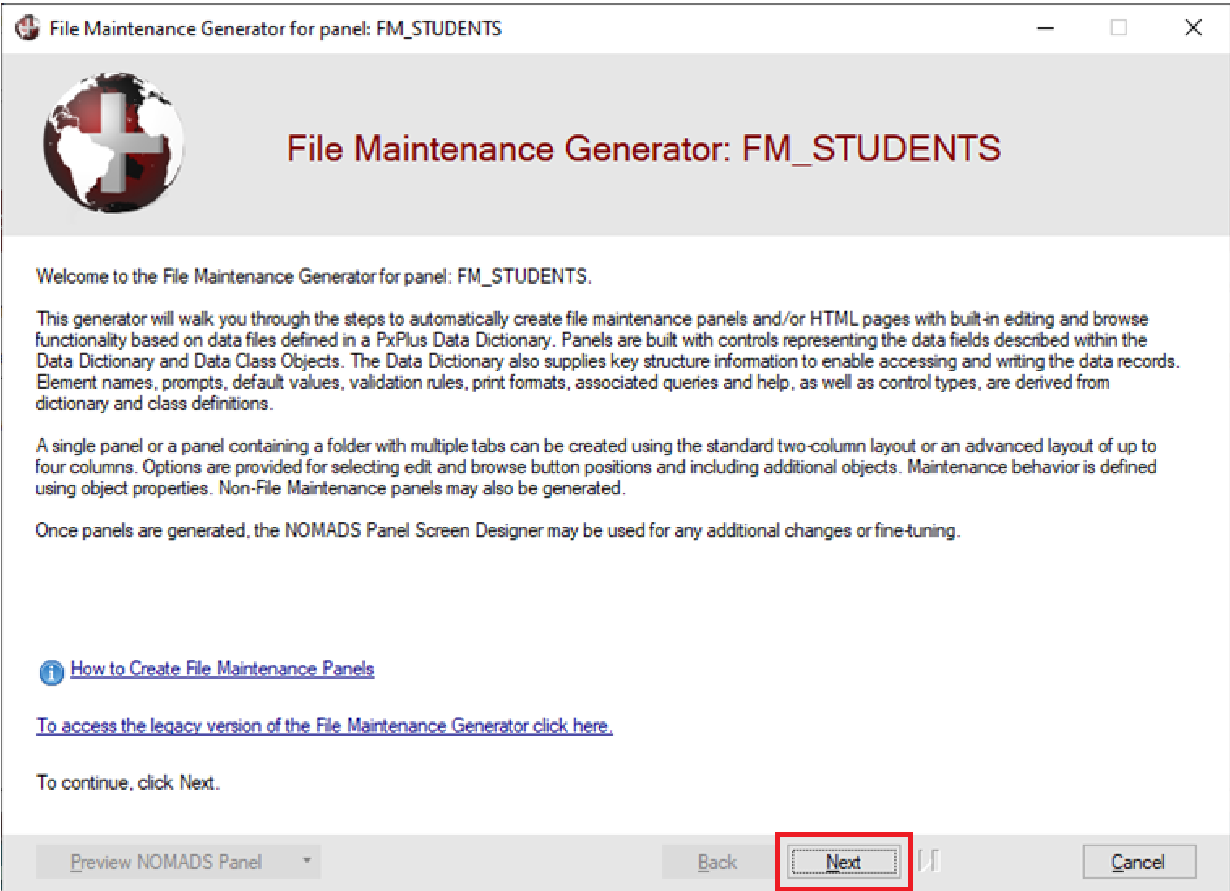
Enter the name of the Library that we have created (**SCHOOL.EN**) or select it by clicking the Query button on the right (yellow folder).

Enter the name of the panel as **FM_STUDENTS** (FM is for File Maintenance), but you can enter any name. Click the [**OK**] button.

The system may display a message to ask for confirmation to create the Templates data file if it does not exist. If this happens, answer **Yes**.



The **File Maintenance Generator Welcome** panel will display. Click the [**Next**] button:



A wizard will appear that will guide us through seven steps to create the file maintenance panel:

1. **Definition:** Define the type of form(s) to be created, select the data dictionary table, select the object to control the program.
2. **Properties:** Define the options for updating records, screen and messages.
3. **Screen:** Define the options for screen position, action buttons and header.
4. **Controls:** Define the field options, type of sources, etc.
5. **Keys:** Define the registry query options, keys, etc.
6. **Fields:** Define the presentation format of the fields, options for tabs (folders) and additional functionality.
7. **Finish:** Complete the information and generate the panels. Before panels are generated, review the information provided. Save these settings as a template for later use.

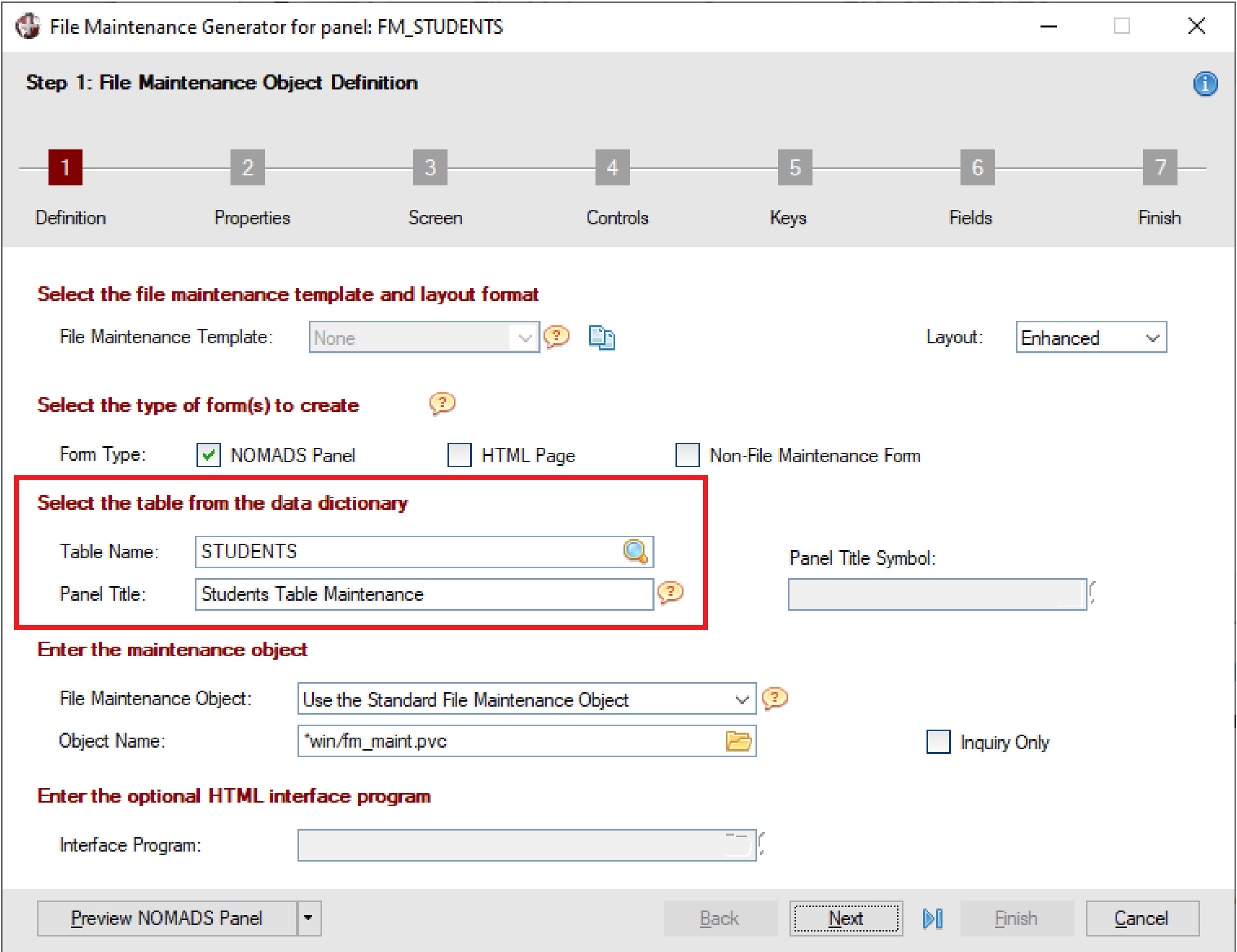
Refer to **File Maintenance Generator** in the PxPlus Help documentation.

In **Step 1 (Definition)**, select the type of form(s) to create: a **NOMADS Panel** (default), an **HTML Page** (to use with Webster+) or both types. Refer to **Webster+** in the PxPlus Help documentation.

Enter the name of the data dictionary table to use, which is **STUDENTS**.

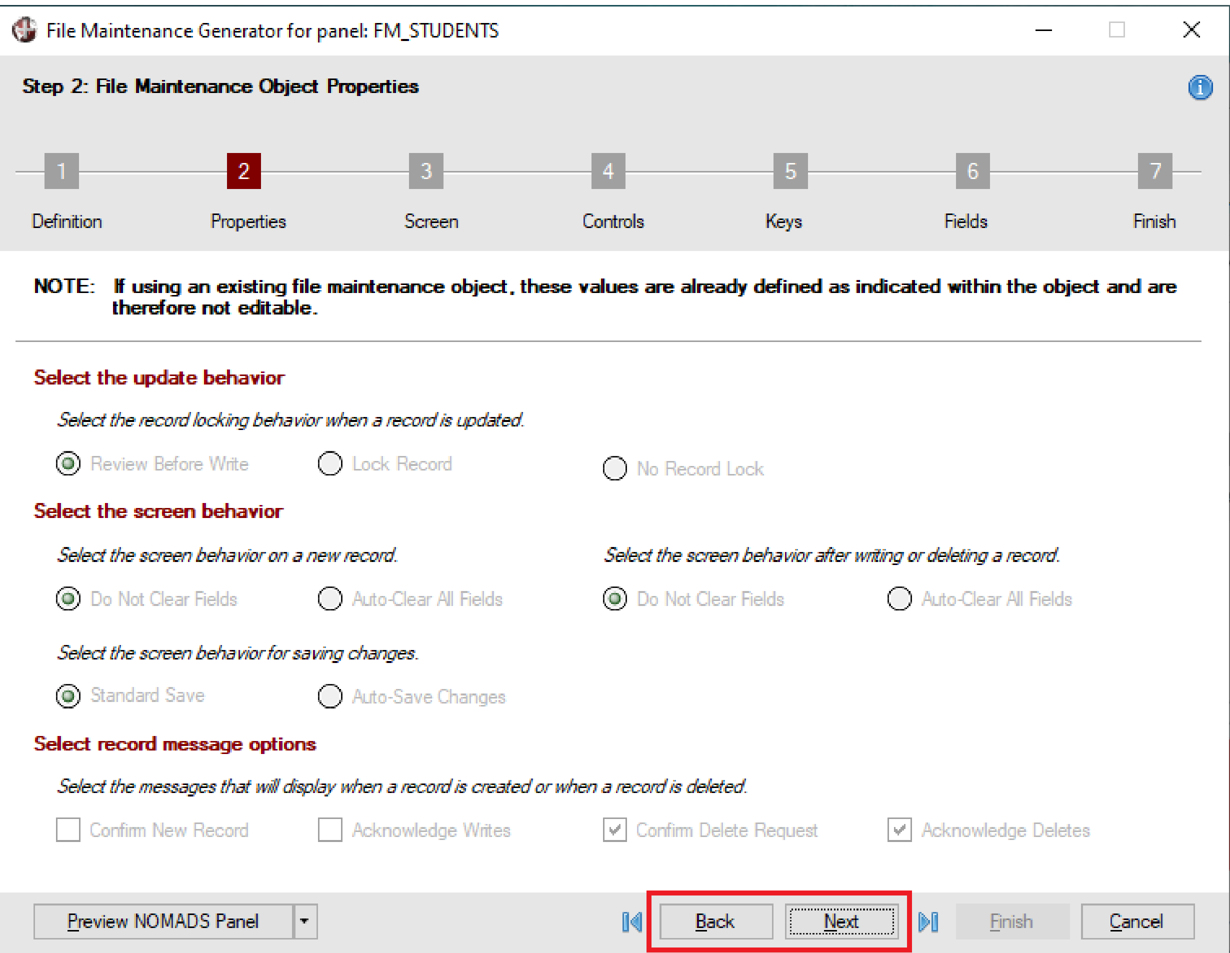
If we want, we can modify the title of the panel or form, which defaults as **Students Table Maintenance**.

Click the [**Next**] button to continue.

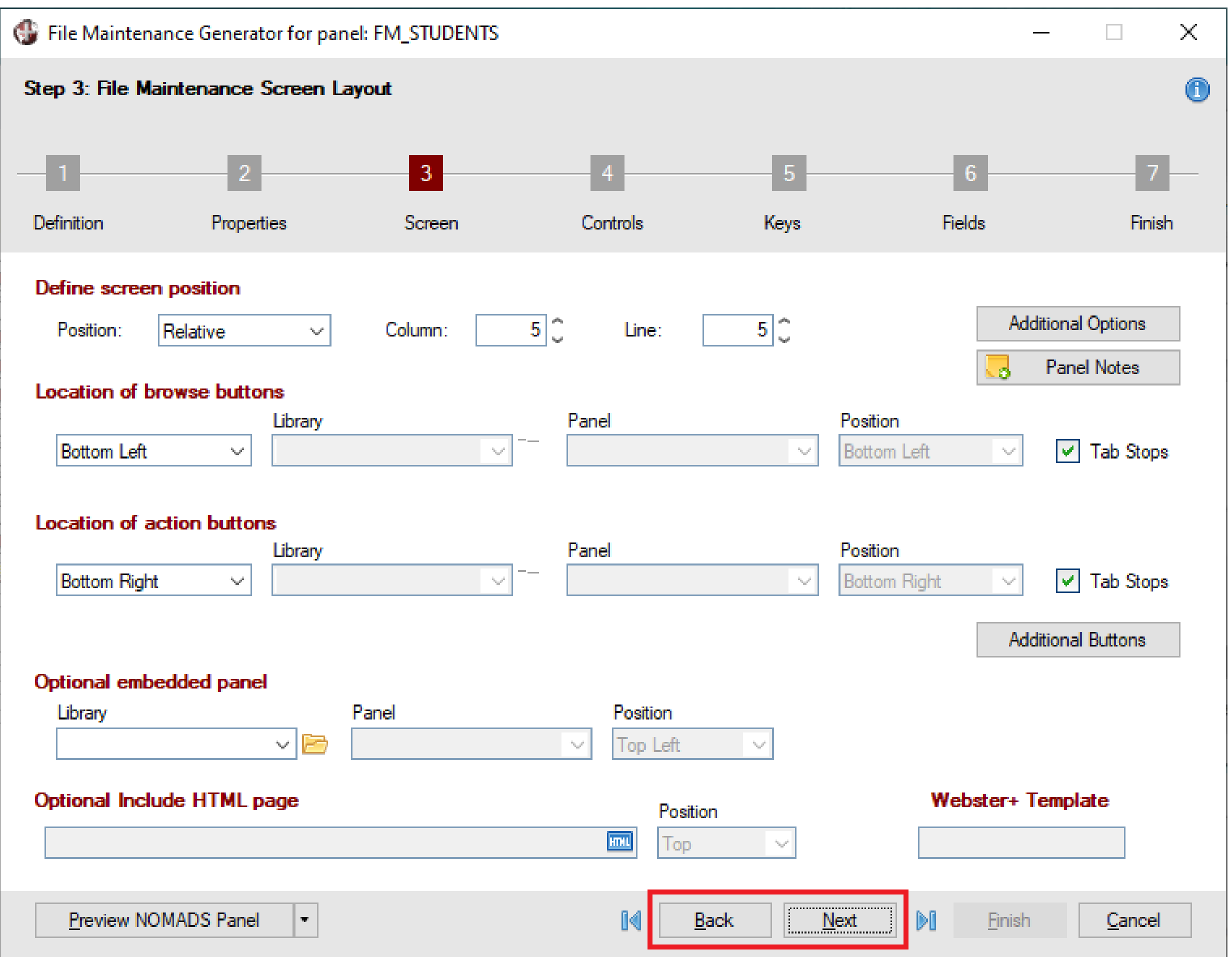


Step 2 (Properties) lets you change the record blocking options; for example, if you want the fields to be cleaned after recording the information or if they should keep the previous information, etc. Depending on the mode selected, these (and other options) may or may not be available to be changed.

If you make a mistake, click the [**Back**] button to return to the previous screen or click the [**Next**] button to continue.

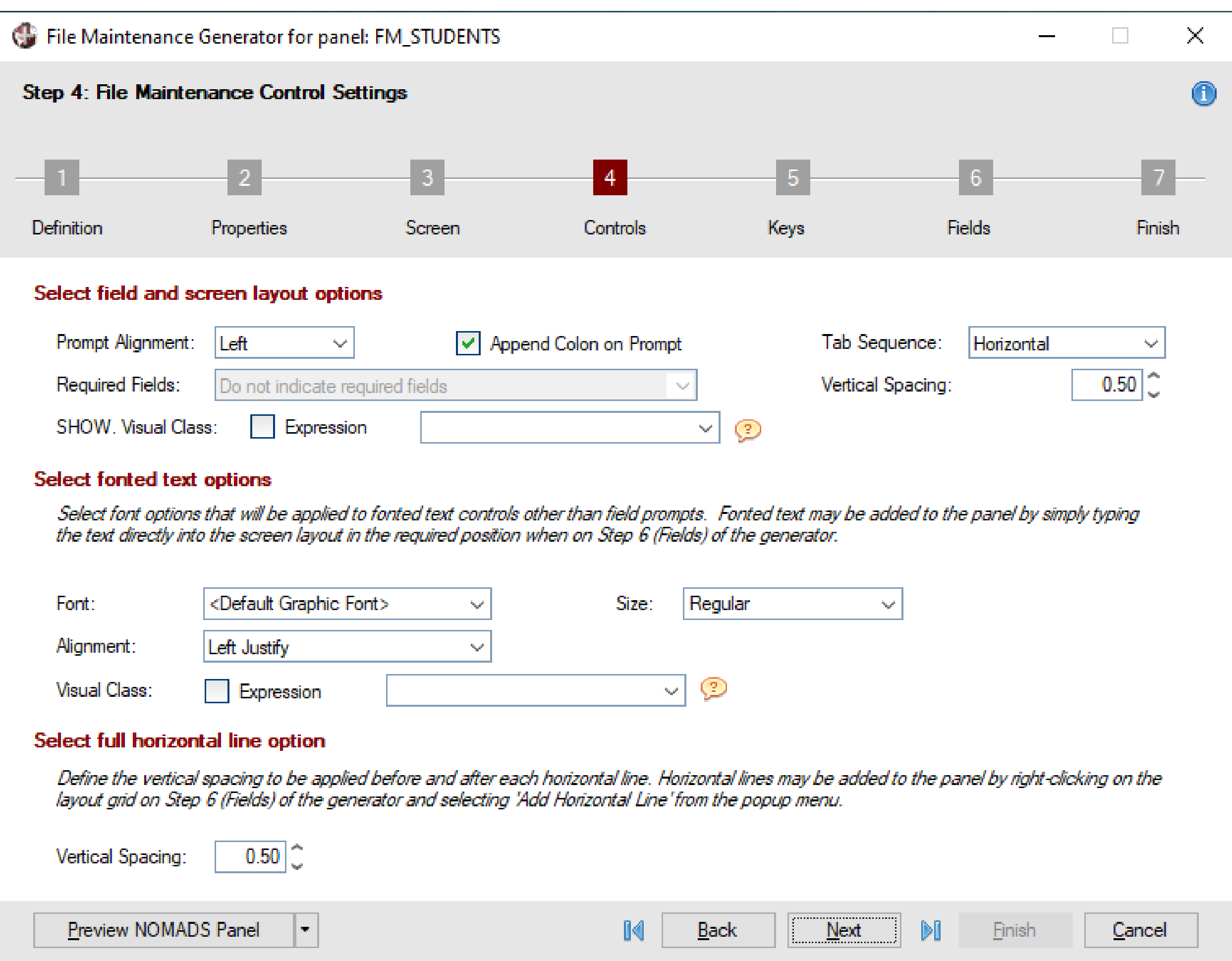


Step 3 (Screen) allows you to specify the location of the panel, as well as the location of the navigation buttons (browse buttons) and action buttons (such as Write, Delete, Clear, Exit, etc.).



You can use the [**Back**] button to return to the previous screen or click the [**Next**] button to continue.

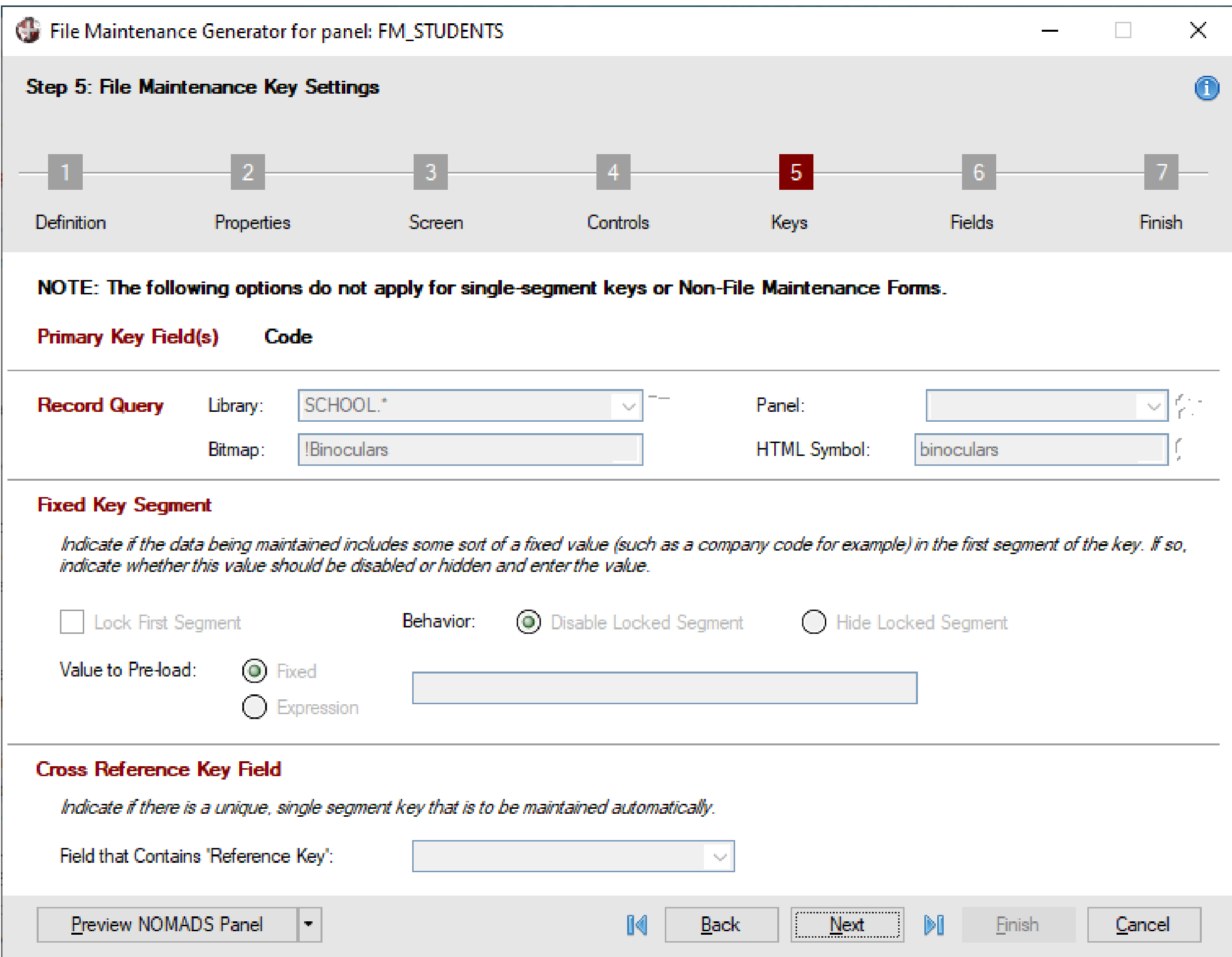
Step 4 (Controls) allows us to modify the location of the titles of the headers of each field (prompt alignment), the type of navigation when the Tab key is pressed, the type and size of the font, etc.



If you want to return to the previous screen, click the [**Back**] button. Click the [**Next**] button to continue.

Step 5 (Keys) allows us to make finer adjustments to the way records are selected (if established when defining the key(s) on the Data Dictionary definition screen).

Example: If we define the primary key as a composite key (that is, it has more than one segment), we could prefix the first segment and make queries only with the remaining ones. (Imagine that you want to filter only the products of a certain manufacturer or those that are only in a certain inventory line.)

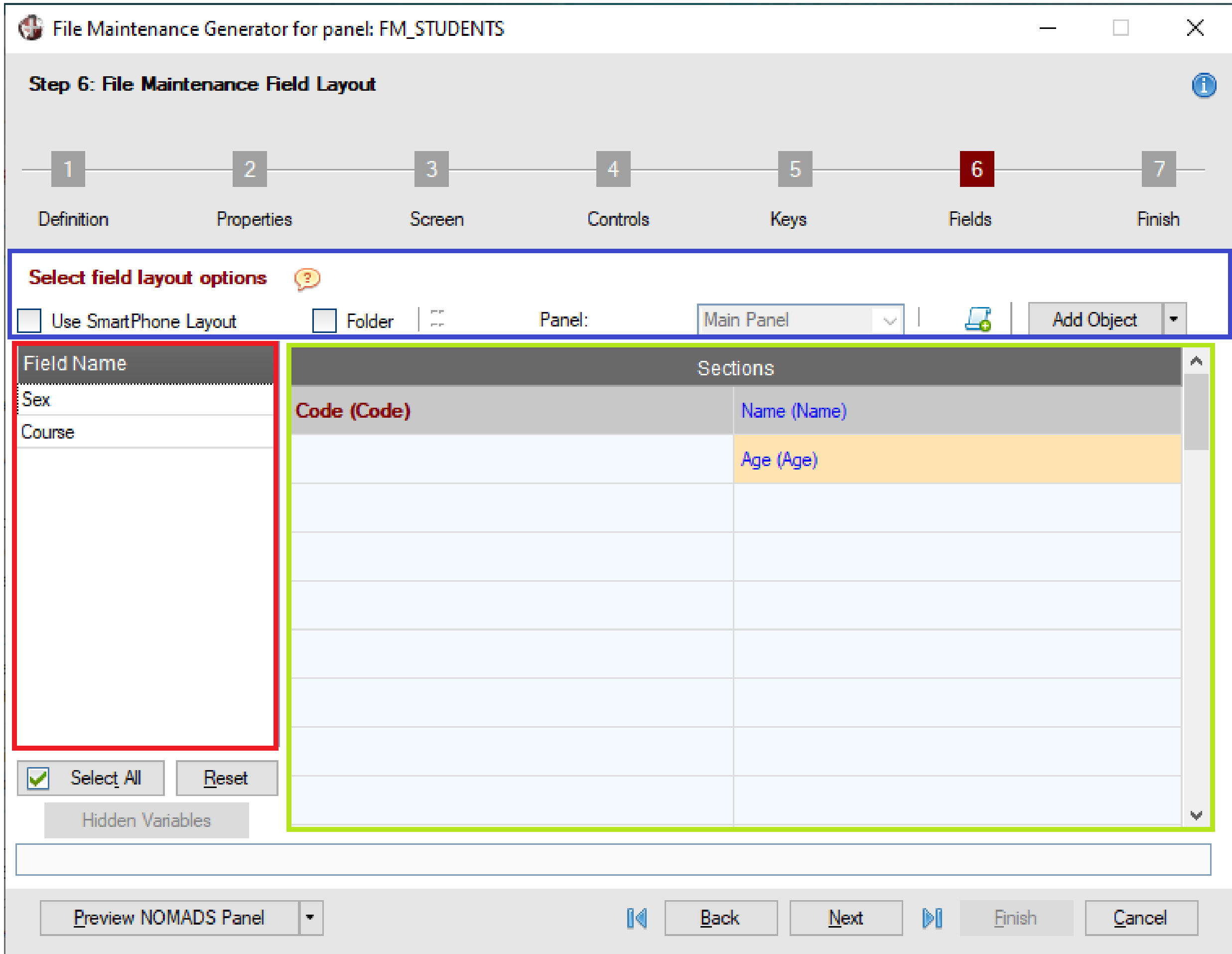


If you want to return to the previous screen, click the [**Back**] button or click the [**Next**] button to continue.

Step 6 (Fields) allows us to select which elements or fields of our table will be shown in the file maintenance panel and, if there are many elements or fields, to place a group of fields in different tabs or folders and to logically group the information.

If the maintenance panel looks very busy, we could also have hidden or blocked fields, which may not be able to be modified (because they are filled after performing a calculation, for example).

The **Step 6 (Fields)** panel is divided into three large parts and looks like this:



The upper blue part provides options that allow us to define additional folders or tabs and to generate a format for mobile phones (see **Note** below).

The red part on the left shows a list of the fields or elements in the selected table. We must select from the list (in red) and drag/drop to the desired area on the maintenance panel to be generated (in green on the right).

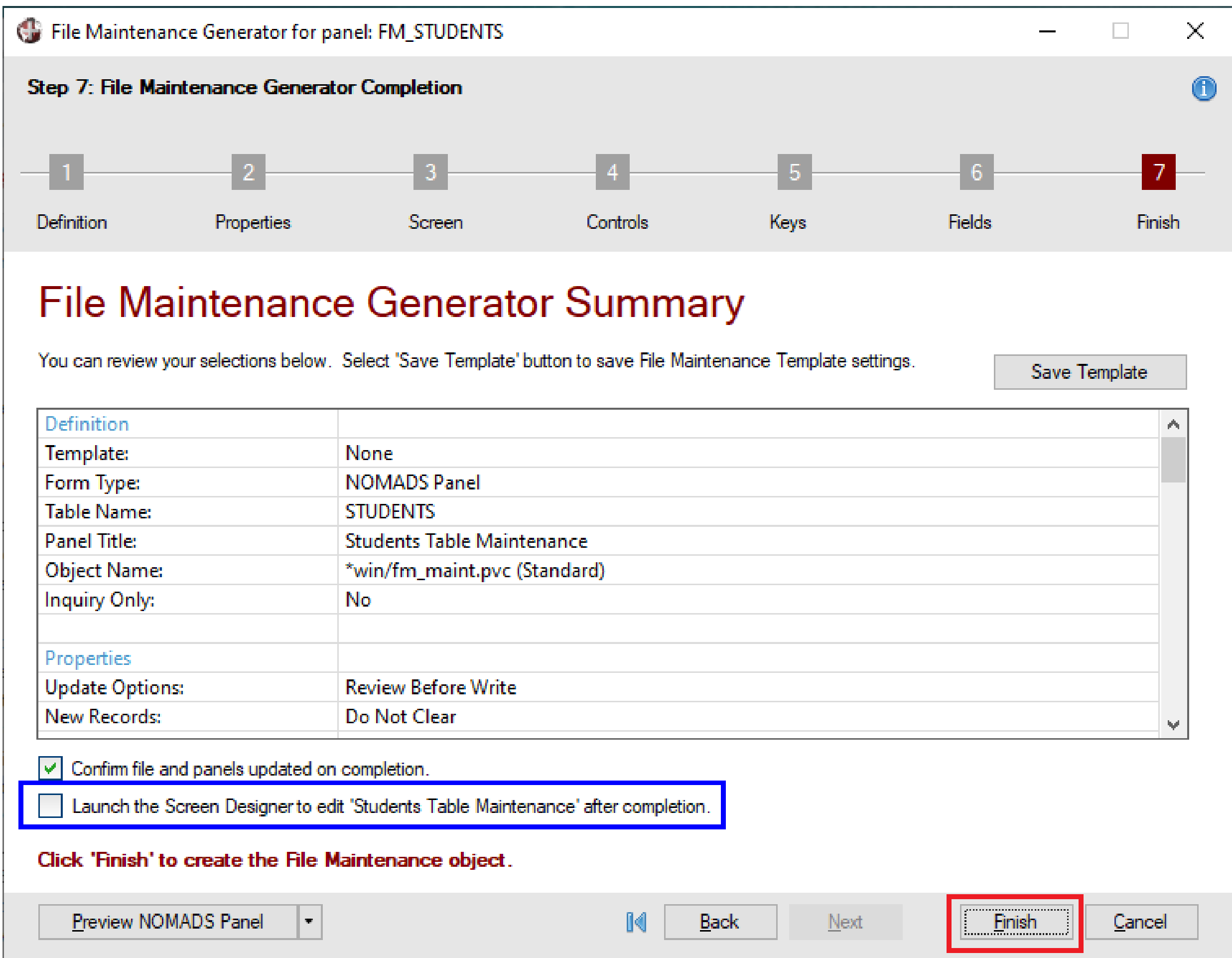
For this example, place all the fields on the right side, with the exception of the Key, which is the **Code** field.

The placement of the fields looks like this:

Important Note: These tools (**File Maintenance** and **Query Manager**, among others) allow us to generate pages/windows compatible with HTML (to be used in a browser via the Internet), so we are "ignoring" several options. Later, we will be returning to them to see how these panels can be generated and viewed in a Web browser.

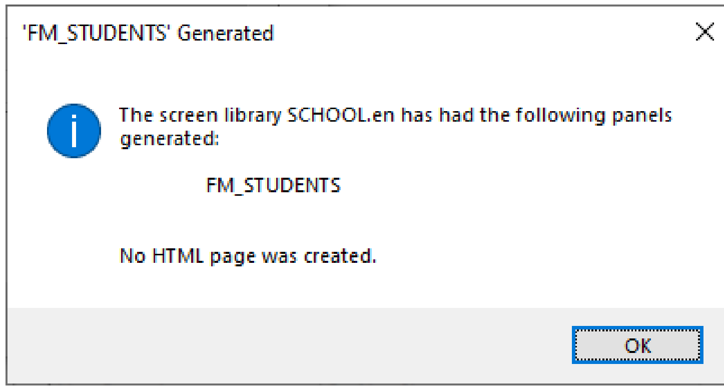
Once the field selection is complete, click the [**Next**] button to continue. If you want to return to the previous screen, click the [**Back**] button.

In **Step 7 (Finish)**, the **File Maintenance Generator Summary** panel displays:



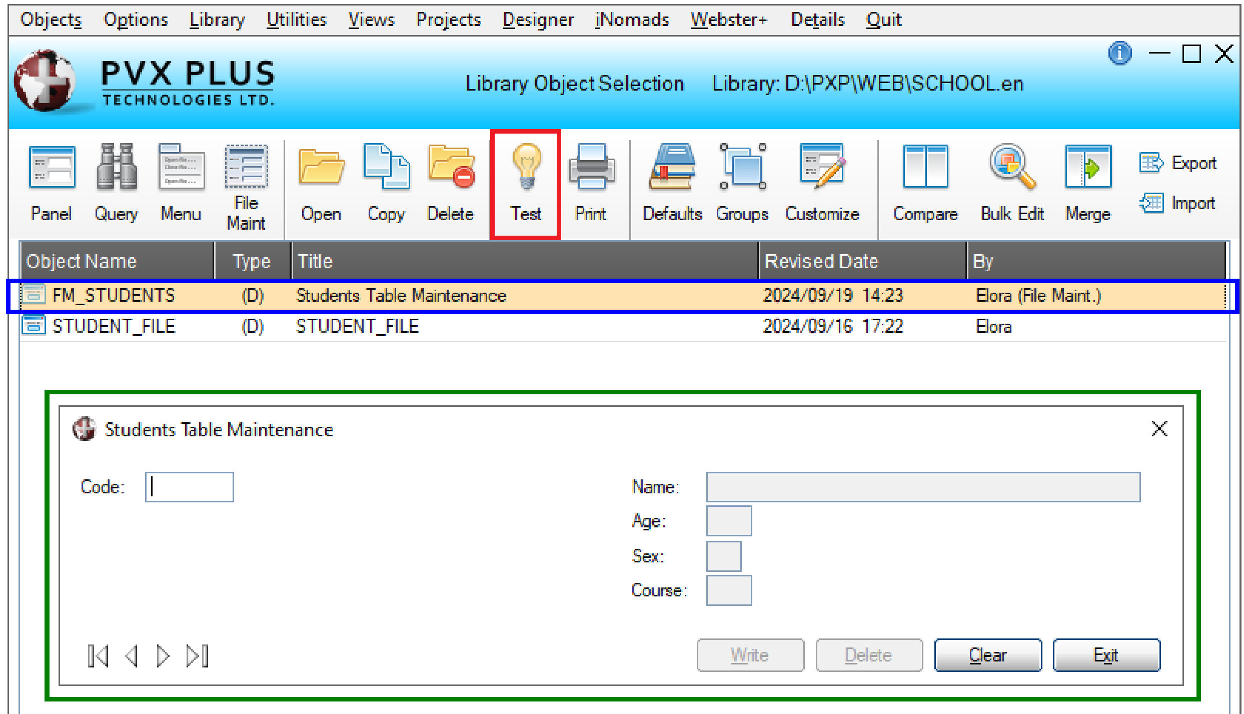
If you select the option [**Launch the Screen Designer to edit 'Students Table Maintenance' after completion**], the NOMADS panel designer will open. You can make other adjustments and adjust the panel to your needs and preferences.

For now, click the [**Finish**] button. A confirmation message will display, similar to the one shown below:



Although it may seem that we have spent a long time making adjustments and options, in reality, we only enter the name of the Library (**SCHOOL.EN**) and the panel (**FM_STUDENTS**). Then, we enter the name of the table (**STUDENTS**), and finally, we select the fields to place on the file maintenance panel.

To test our panel, we must load the library in NOMADS (**SCHOOL.EN**). A window similar to the one below will display:



Select the **FM_STUDENTS** panel (marked in blue) and click the [**Test**] button at the top (marked in red). The newly created panel will display (marked in green).

This file maintenance panel was generated with four simple steps. If we need to program it, it would be many lines of program and at least a couple of hours of work. On the other hand, the NOMADS **File Maintenance Generator** allowed us to do it in less than a minute, with many options available to customize it. In addition, we can open it in the NOMADS panel designer to make even more adjustments.

To run our panel from NOMADS, we only need to make a "link" or "jumpto" to this panel (Name: FM_STUDENTS, Library: SCHOOL.EN) or in a PxPlus program through the **PROCESS** command: "FM_STUDENTS", "SCHOOL.EN".

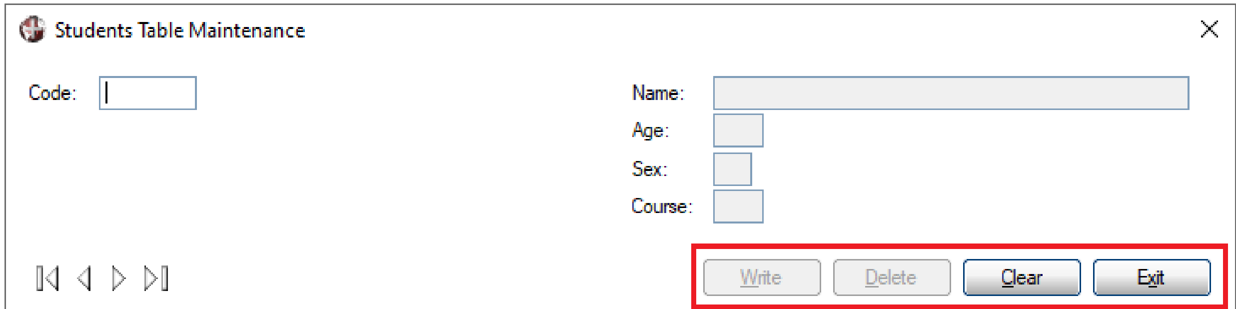
This panel does not have a single line of PxPlus code. Its integration with any panel is transparent, and it can be used in different projects and combinations.

For now, we are going to use our test panel to enter the sample data below. After each record, click the **Write** button to save the record to our **STUDENTS** table and then click the **Clear** button to clear the fields and enter the next record.

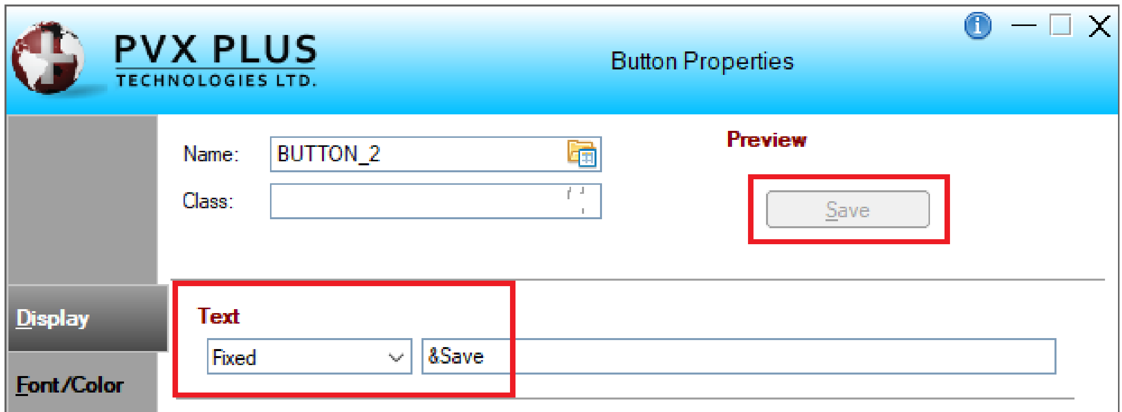
Code	Name	Age	Sex	Course
000001	John Hendersen	40	M	A1
000002	Ana Victoria	19	F	A1
000003	Irma Rose	45	F	A1
000010	Amy Lucia	30	F	A3
000011	Michael Ross	34	M	A3
000012	Diana Martin	37	F	A1
000013	Louis Britton	25	M	B1
000020	Janet Perry	40	F	B1
000021	Veronica Diaz	37	F	B3
000022	Joseph Diego	52	M	B1
000030	Michael Martin	60	M	B3
000100	Brian Peters	41	M	A3
000101	Simon Soler	27	M	B1
000102	Diana Ruiz	36	F	B3
000201	Fiona Lopez	47	F	C1
000202	Jordan James	28	M	C1

You will have noticed that on the action buttons (marked in red below), there are some underlined letters. These keys are called "accelerators", "quick keys" or "hot keys". The idea is that you can quickly execute the associated action by pressing the [**Alt** + *key*] combination.

Example: [**Alt X**] is equivalent to selecting the [**Exit**] button while [**Alt W**] can be used as a shortcut for the [**Write**] button.



When you are defining a control (**Example**: a Button control), in the properties window on the **Display** tab, you can prefix a letter in the **Text** field with an **&** (*ampersand*) sign to specify the following letter as a quick key, as shown below:



Note: If there are several controls with the same quick key specified, they will be executed progressively. Numbers can also be used as quick keys or accelerators.

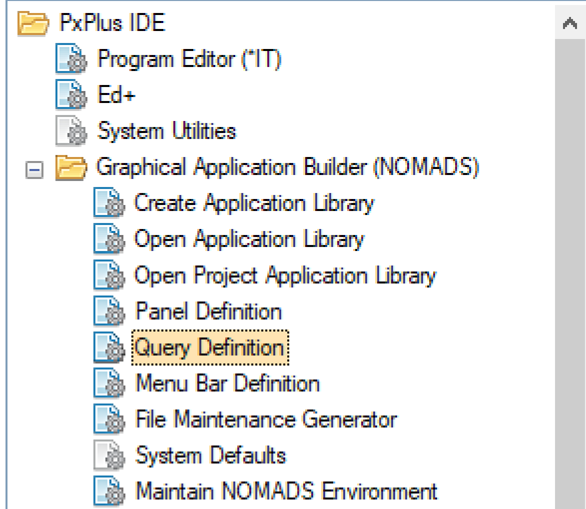
Important Note: These tools, and many others from PxPlus and NOMADS, have messages in English. That is where the suffix of the NOMADS panels comes from (.EN = English) so all messages from the automated tools and the environment itself will be in English. However, it is possible to change it by directly editing the panel, using a translated message file or by subscribing to a paid service (Microsoft or Google). The simplest and most expeditious method is to manually change the message on each panel. The most efficient method is by translating the message file.

To summarize:

To define a file maintenance panel, we must have a table definition in the data dictionary. Then, we select the NOMADS **File Maintenance** option. We specify a library, a panel name and the table name. We select the presentation options, format, elements to display and that's it!

NOMADS Query Definition

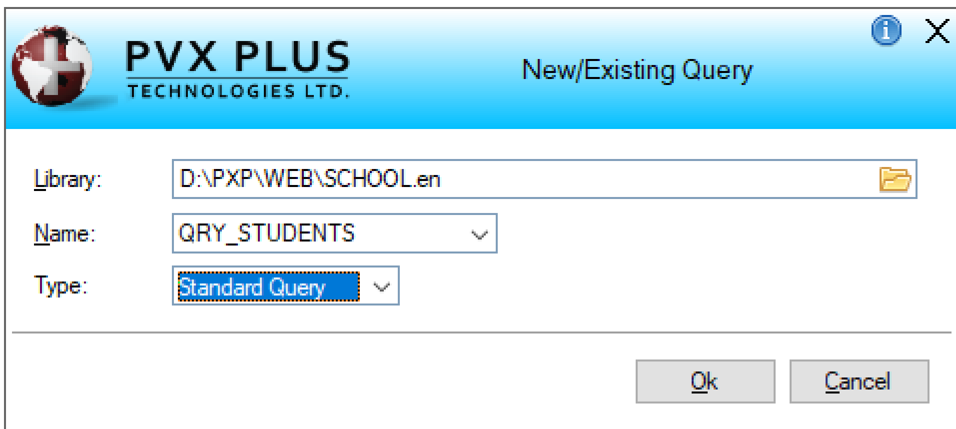
To run the **Query Definition** tool, select the **Query Definition** task (after opening the **Graphical Application Builder (NOMADS)** category from the PxPlus IDE main menu).



Refer to **Accessing Query Definition from IDE** in the PxPlus Help documentation.

A window for **Query** maintenance and creation will display, asking for the name of the Library, the name of the query panel and type of query (either **Standard Query** or **Query List**).

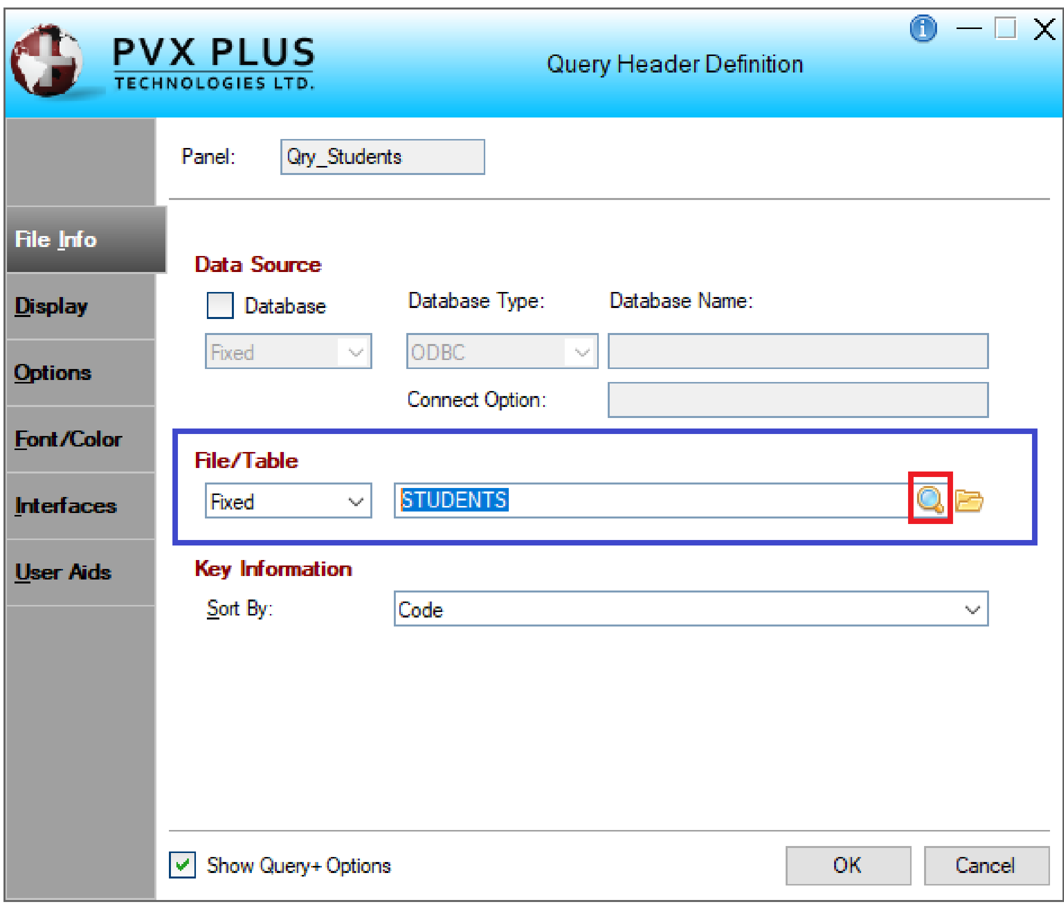
We will take a look at the difference between these two types. For now, let's select **Standard Query** and click the [**OK**] button:



Once this information has been entered, the **Query Header Definition** window displays where, on the **File Info** tab (on the left side), we can define a **Query** for a database or for a table.

Refer to **Query Header** in the PxPlus Help documentation.

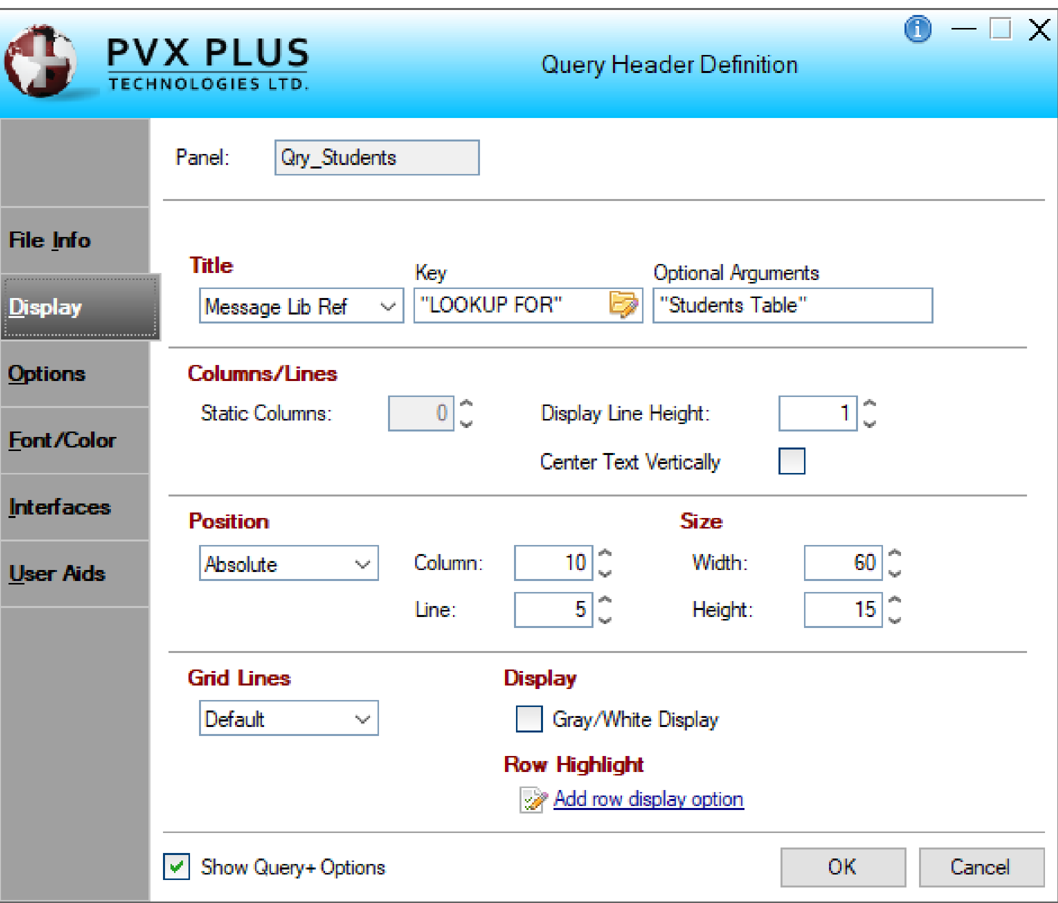
At this time, we will define a Query for our **STUDENTS** table (entry marked in blue). If we do not know the name, we can search for the file/table by clicking the magnifying glass icon (marked in red).



The second entry in this window is on the **Display** tab (on the left side) where you can change the title of the query panel.

We can also leave some columns fixed. If the table has many fields/elements, it is possible to leave the first two or three columns fixed and allow other columns to be moved. **Example:** We could leave the CODE and DESCRIPTION columns fixed and make the other columns moveable.

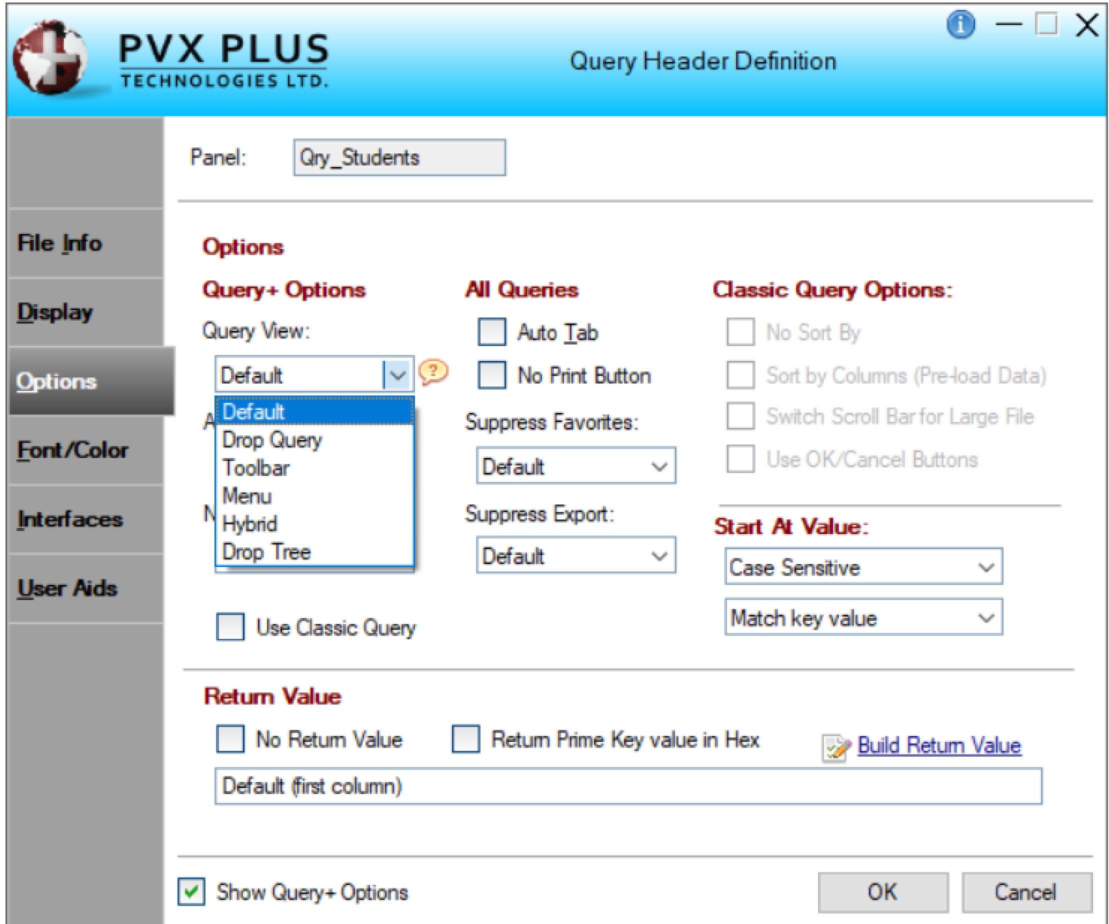
It is also possible to place alternating lines and other visual adjustments.



By selecting the third tab on the left, **Options**, we have a [**Query View**] option for choosing the type of presentation for how the query will display (drop-down, toolbar, menu, hybrid and drop-down tree):

Default:	By default, it will use the Toolbar view.
Drop Query:	Displays data in Report View style.
Toolbar:	Displays a toolbar with other available functions/tools.
Menu:	Functions are displayed in a menu, not in a bar.
Hybrid:	It is a combination of menu and toolbar.
Drop Tree:	Displays the information in a tree-like list, directly on the panel.

We can also specify other options, such as the value returned when selecting a record. By default, it will be the first column of data.



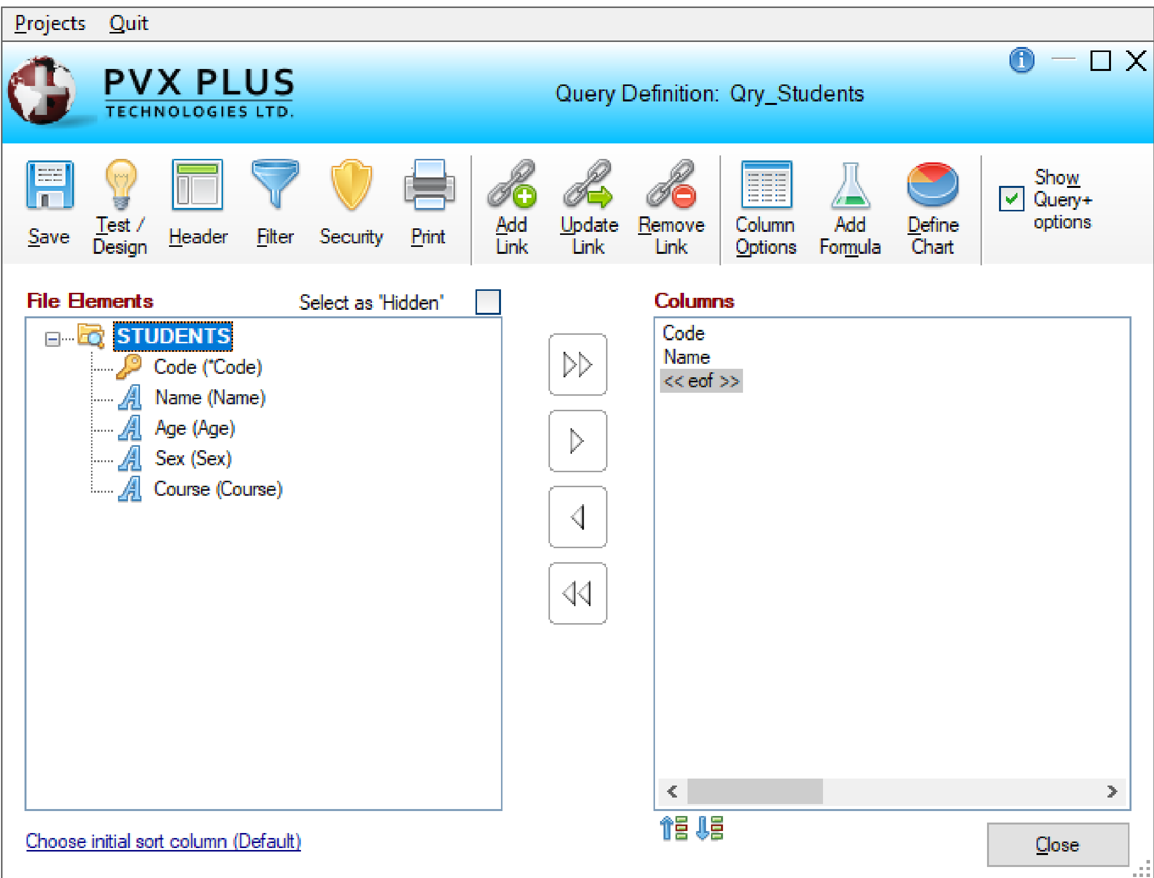
Important Note: Some of these types of queries read the entire content of the table before displaying the information on the screen while others read progressively. It is important to know if the number of records is too large to properly select the type. **Drop Query** and **Drop Tree** read the entire contents of the table before displaying the content on the screen.

The other side tabs (on the left side) provide options that allow you to select the **Font/Color**, the programming **Interfaces** (to prepare and/or filter the information), and the **User Aids**. For now, click the [**OK**] button.

The **Query Definition** window displays.

Once the basic parameters of the query header have been defined, we have the query definition panel itself where we have a command toolbar at the top. Below that, the left side lists the elements or fields in the table, as well as any links and formulas. The right side lists the columns that are to be displayed in the query. In the middle, the four selection buttons allow you to move one or more selected elements from one side to the other.

Refer to **Query Definition** in the PxPlus Help documentation.



The command toolbar is made up of the following buttons:

Save: Saves the definition of the query.

Test/Design: Calls the column/test design panel.

Header: Returns to the query header layout panel.

Filter: Allows you to define information selection criteria, range, selection criteria, filtering program, etc.

Security: Calls the user and access control definition panel.

Print: Print a copy of the query definition.

Add/Update/Remove Link: Defines and modifies the link fields (join) between the columns of the table of this query with other external tables.

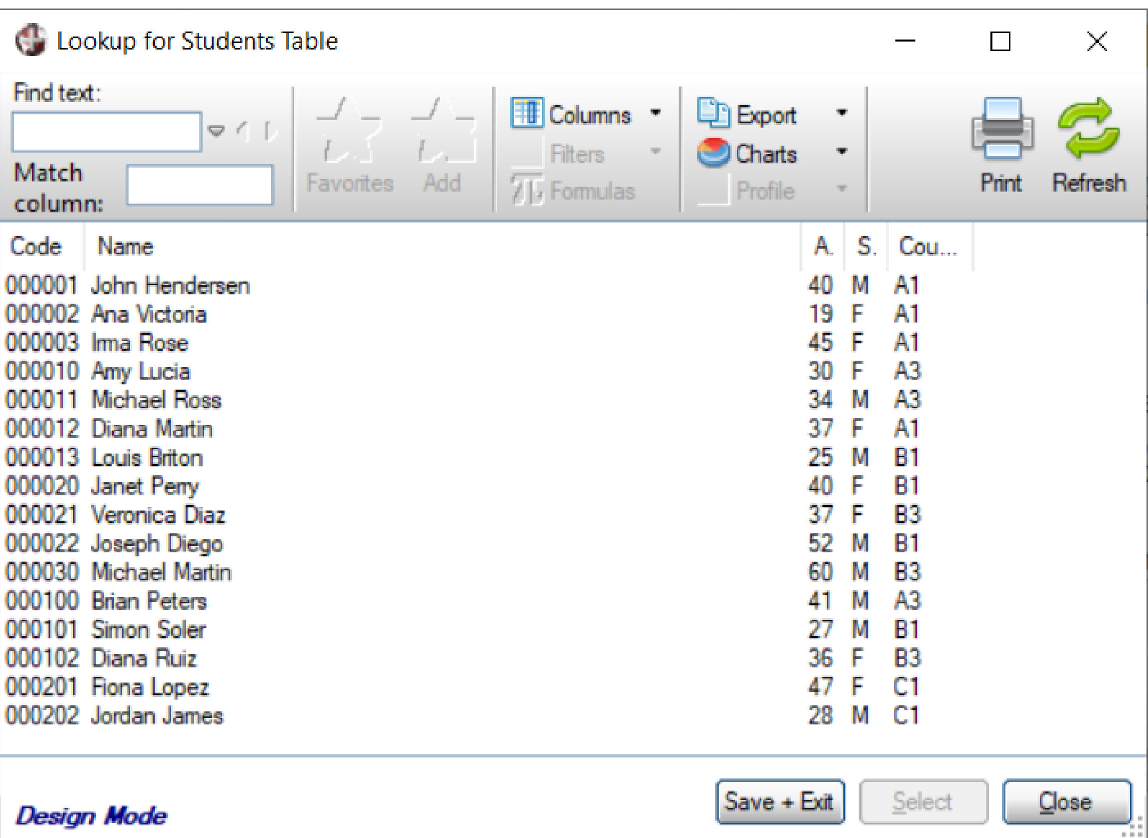
Column Options: Allows you to modify the presentation and format of the displayed data, allowing you to apply sorting methods, width, title, etc.

Add Formula: Allows you to define formulas and calculations for the data, including formulas from other queries (and previously defined formulas), specifying width, color, format, etc.

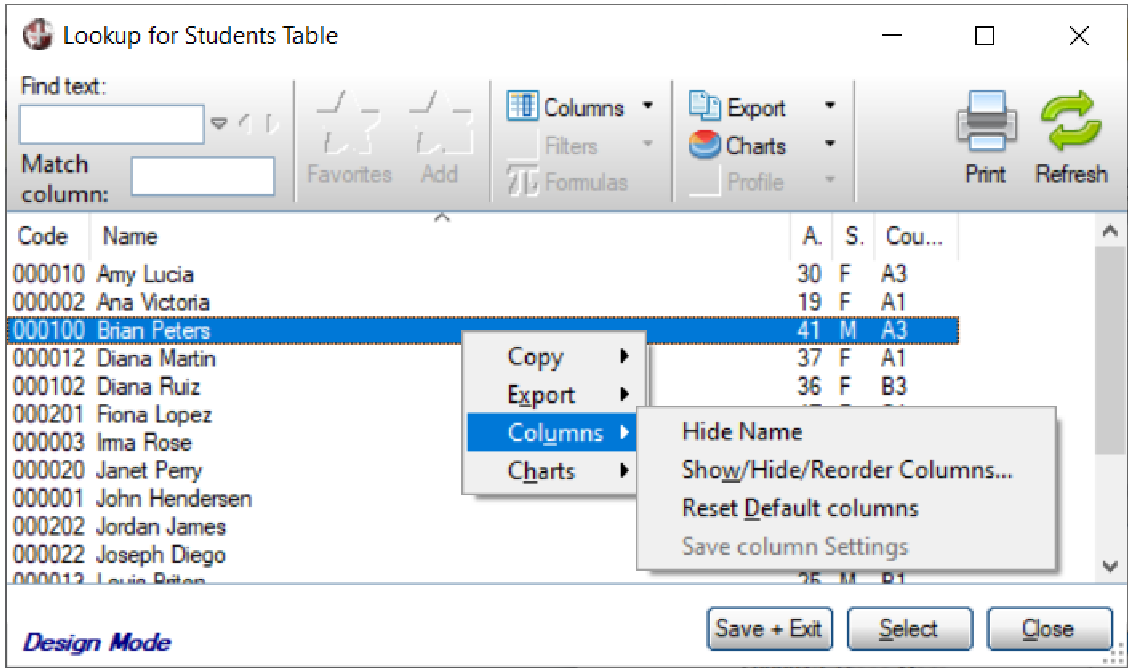
Define Chart: Allows you to define a chart based on the values of a column; this chart can then be embedded in another panel or widget.

Show Query+ options: Option that allows you to disable/enable Query+ options (Query+ is the default mode of display).

If we click the first button in the middle (with the double right arrow) to select all the file elements (fields) and then click the [**Test/Design**] button in the top toolbar, a query panel similar to this will display:

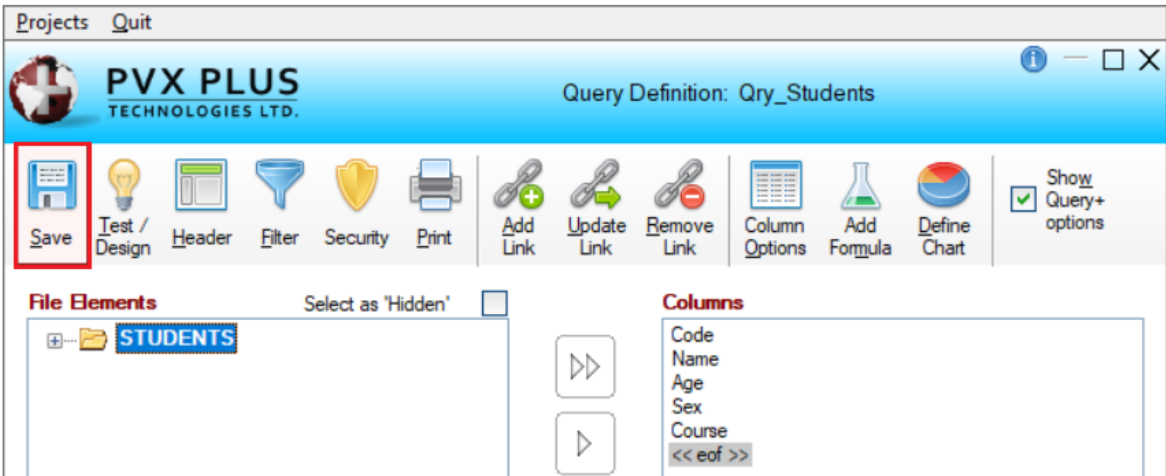


In this panel, additional options are available to carry out tests and adjustments for data selection and filtering:

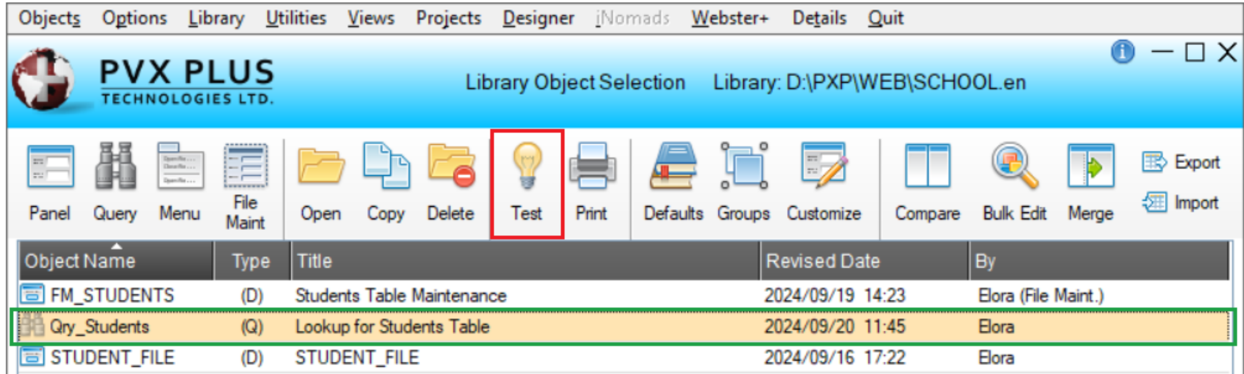


To return to the **Query Definition** panel, click the [**Close**] button.

Once we are finished with the query definition and adjustments, save the query. On the **Query Definition** window, click the [**Save**] button (diskette icon) and then click the [**Close**] button at the bottom.

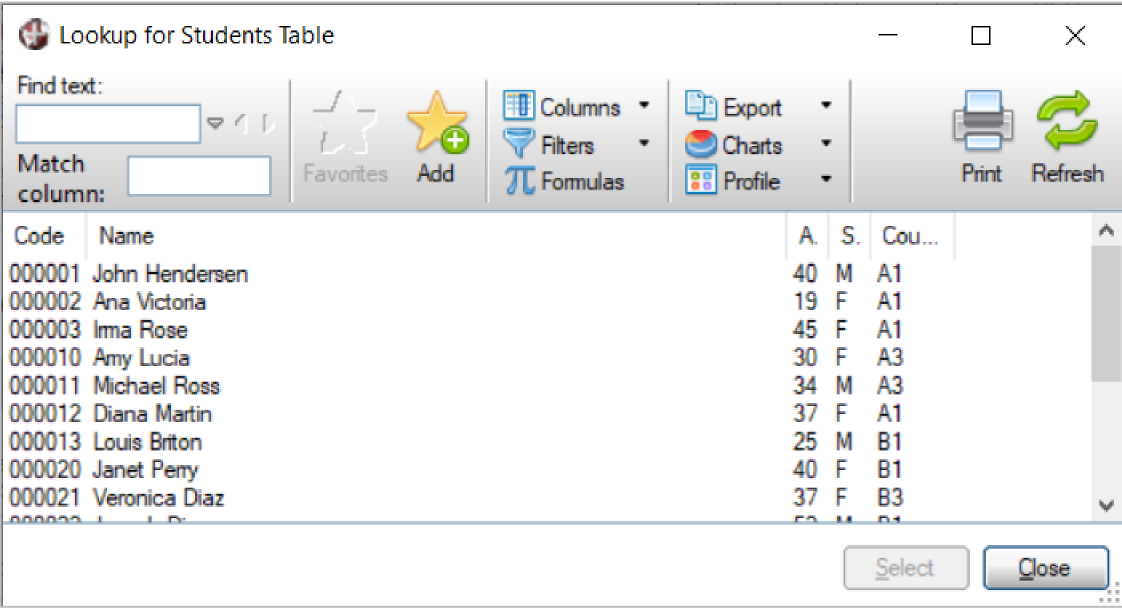


Let's return to the **NOMADS Library Object Selection** window for the library **SCHOOL.EN**.



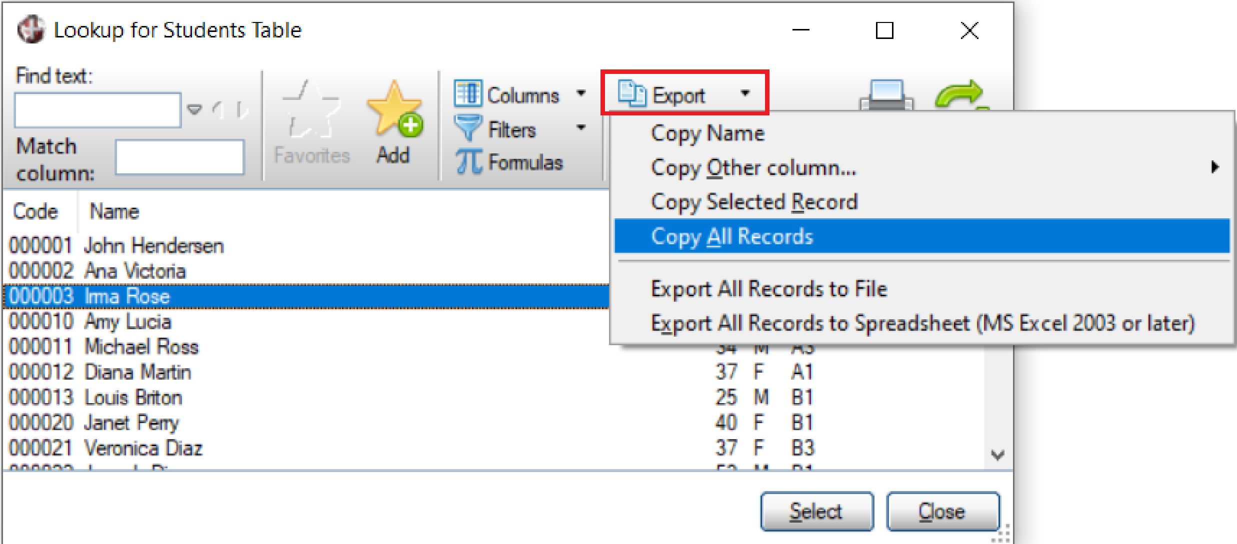
In the **Library Object Selection** window, our query (marked in green) is in the list of available panels or objects in the **SCHOOL.EN** library. Click the [**Test**] button (marked in red).

The query panel that we just defined will display:



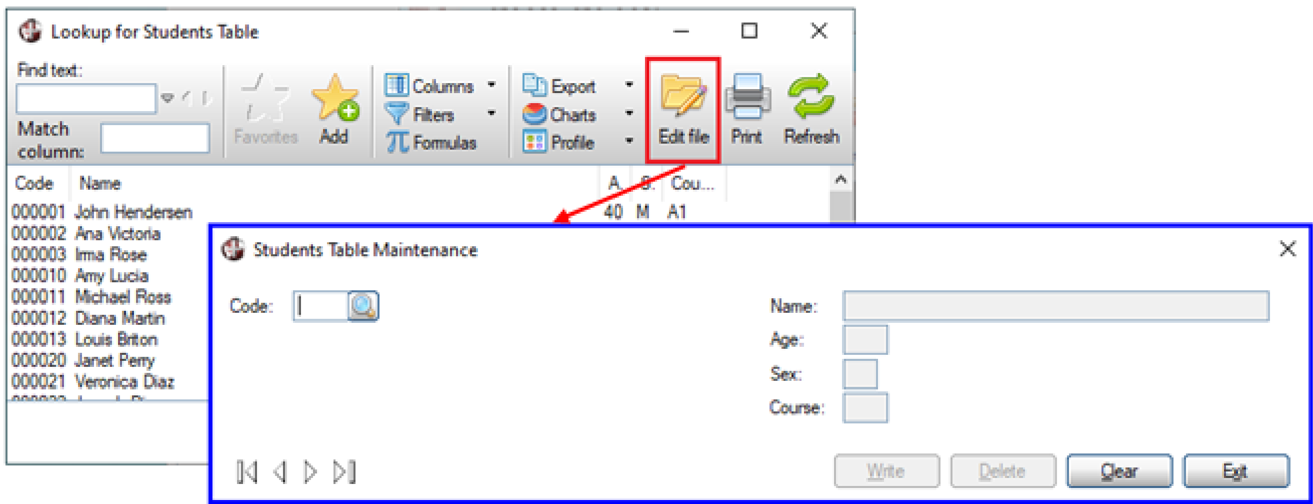
We can see that there is similar functionality to what we had in the panel designer (these can be turned Off, if desired).

We can also create selection criteria, filtering, etc. It is easy to copy or export the information to another file or Excel spreadsheet.



When we defined the **Query**, we could have specified a **Maintenance Library** and **Panel** on the **Interfaces** tab when entering the **Query Header Definition**.

Assuming we had entered the information for the **FM_STUDENTS** panel previously created with the **File Maintenance Generator**, an [**Edit file**] button would also appear on the query toolbar beside the [**Print**] button. Clicking this [**Edit file**] button would launch the maintenance panel (that we previously created) from within the query panel.



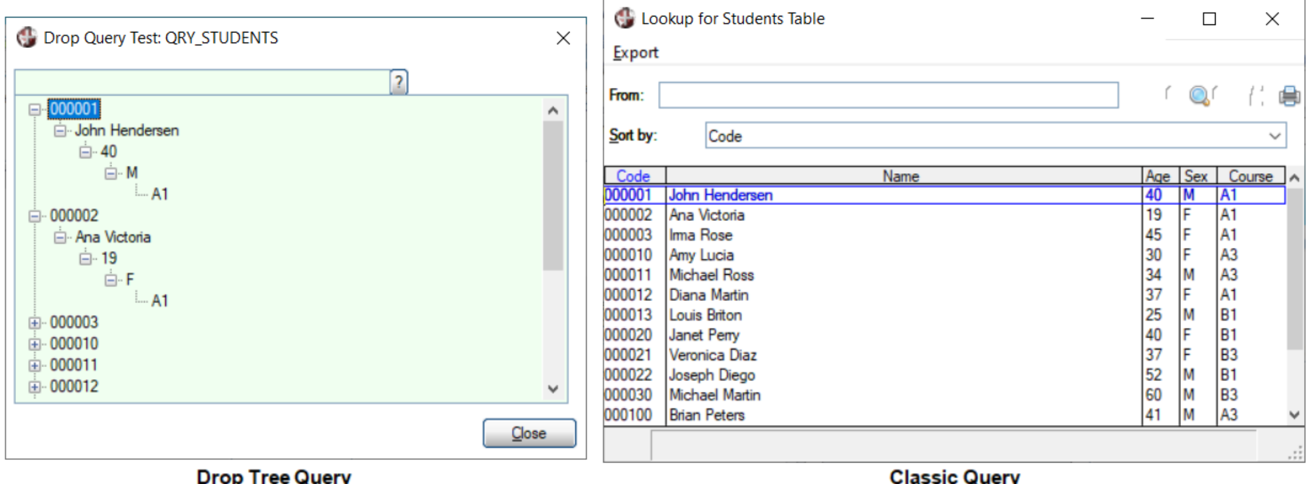
We can see that by simply selecting a series of options and arguments, PxPlus (through NOMADS) can generate complete and attractive panels, fully functional without writing a single line of code.

Both **File Maintenance** and **Query** have object interfaces that allow direct manipulation from our programs without having to define and/or use the menu bar options.

Remember that, although it seems like a tool with a single purpose, it is possible to define different types of queries.

Example:

This shows a **Drop Tree Query** on the left and next to it, a **Classic Query**:



Drop Tree Query **Classic Query**

For now, we will leave this part about the initial creation of queries. We will return to this topic later.

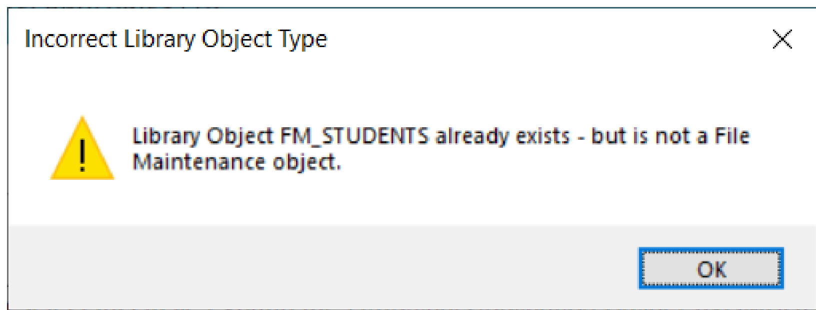
Connecting Panels

Many times, even if we only see one panel on the screen, it could be made up of several, as is the case with embedded panels or folders/tabs. It is also possible to create alternate panels (depending on the situation, one or the other opens).

However, the preferred method of "connecting" between panels is **Link/Jumpto**. As is the case with the Query, it is possible to do it in a pre-established way. **Example:** In File Maintenance, a panel can be modified to add a **Query** button so that we can look up the existing information and by clicking on a record, select it and access it for modification.

To do this, select **File Maintenance Generator** (after opening the **Graphical Application Builder (NOMADS)** category from the PxPlus IDE main menu). Enter the name of the Library (**SCHOOL.EN**), and select the name of the panel (**FM_STUDENTS**).

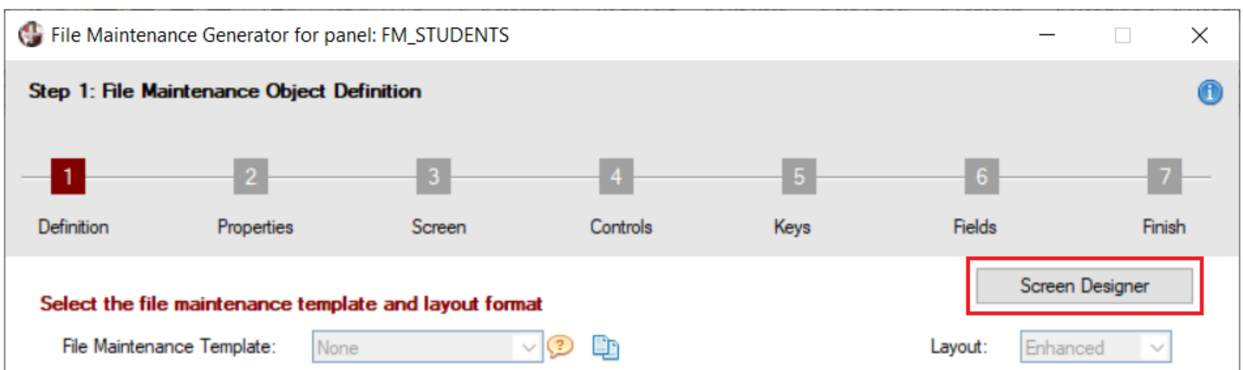
Note: If adjustments were made to this panel using the NOMADS designer after it was generated, it is possible that, at this moment, the message below will display:



This message displays because the **File Maintenance Generator** generates a different object type than the one used by the **NOMADS** panel designer. Once this object is converted to the NOMADS format, this panel can no longer be used with the **File Maintenance Generator**. In reality, it is not complicated at all to regenerate or create another object of this type again.

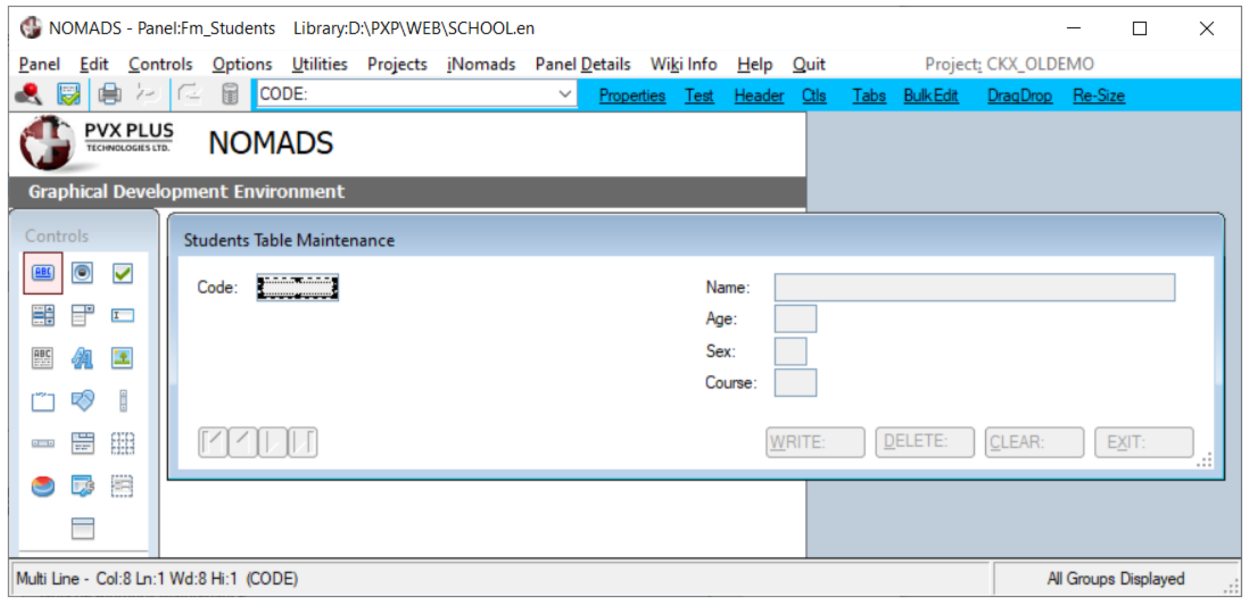
In our case, no message was displayed. The **Step 1 (Definition)** screen is displayed.

Click the [**Screen Designer**] button (marked in red):

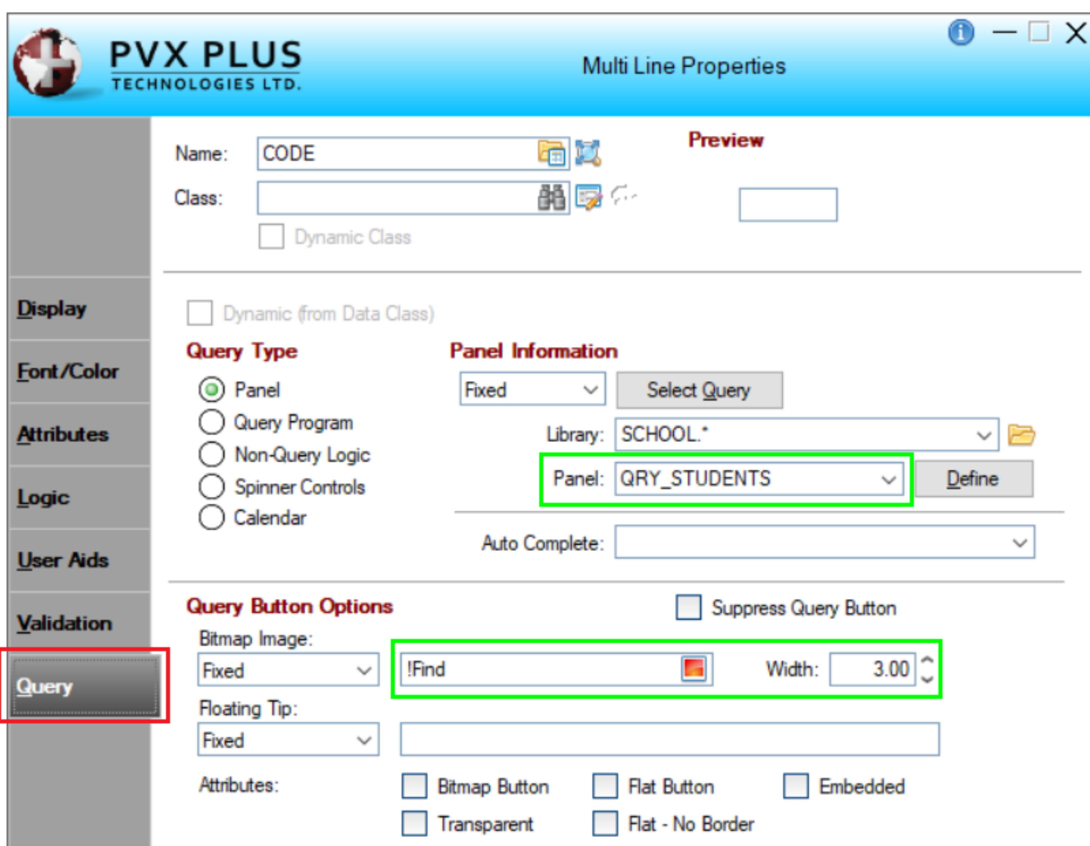


The NOMADS panel designer will open, and it is possible to modify any control in it.

We are going to modify the Multi-Line control for Student **CODE** by double clicking on it.



The **Multi-Line Properties** window for that control displays.



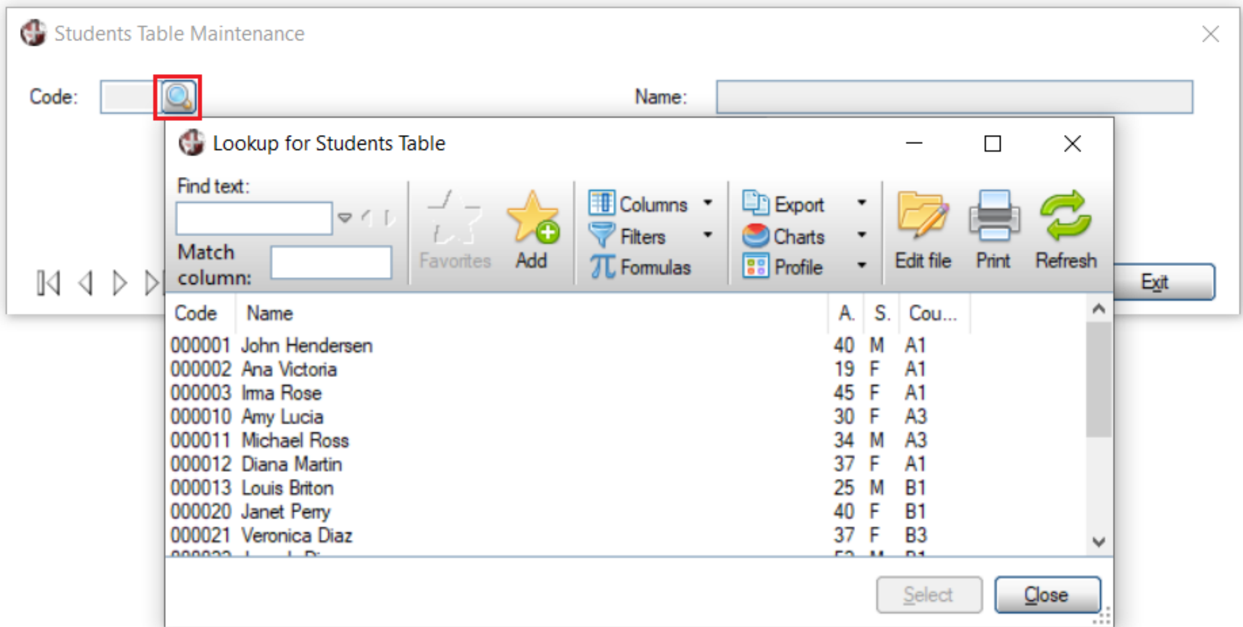
Click the **Query** tab (marked in red) on the left. This is where we specify the query panel to use. Click the **Panel** drop down (marked in green) and select the **Query** panel that we defined earlier

(**QRY_STUDENTS**).

In the [**Query Button Options**] section, we can also change the bitmap image that will display in this field and specify its width, as we have done above (marked in green).

After we are done making these changes, click the [**OK**] button to close the properties window. Save the panel by clicking the [**Save**] button (diskette) or by selecting [**Panel**] -> [**Save**] in the top menu bar or by pressing [**Ctrl S**].

Once saved, select the [**Test**] option at the top. When the test panel displays, click the query button associated with the **Code** field:



You can check that when any record is selected, it will be brought into our maintenance panel for modification.

Tip: Remember to familiarize yourself with common functions, such as saving panels, testing, etc., since we will progressively eliminate the explanations for these actions.

6. Panels: Look and Feel

We can say without a doubt that, in modern applications, the look and feel (which we could define as "style and appearance") is one of the most important factors for the user. To clarify - an application can be very attractive, but if it does not fulfill its purpose, it will be discarded and replaced very soon by another, which may not be as attractive, but that works!

PxPlus offers a whole series of tools to make our panels look modern and functional, as well as attractive to users.

We see an "attractive" design that would serve as an access menu to other panels. Remember that in terms of interface design, there is nothing definitive. **Example:** When we got used to an interface, such as Windows 8, then Windows 10 came out, and other "presentations" with different fonts and color schemes followed. The same thing happened with Android, IOS and almost any variant of Linux, so the limit is your imagination. In any case, being restrained and conservative (in the case of business applications) does not sound like a bad plan.

Exercise: Defining a Panel with Look and Feel

In this exercise, we have an example of what a modern and fresh entrance panel would be. We will see how to define a similar panel, but only the aesthetic part. We will not assign logic or actions to any control.

To begin, run NOMADS and create a panel called **PRETTY_PANEL** (or any other name).

1. Define the panel **Size** (Width 64, Height 20). Go to the **Font/Color** tab and select White for the **Background Color**. Enter the **Title** as "Doggie's Hair Salon". Click the [**OK**] button.

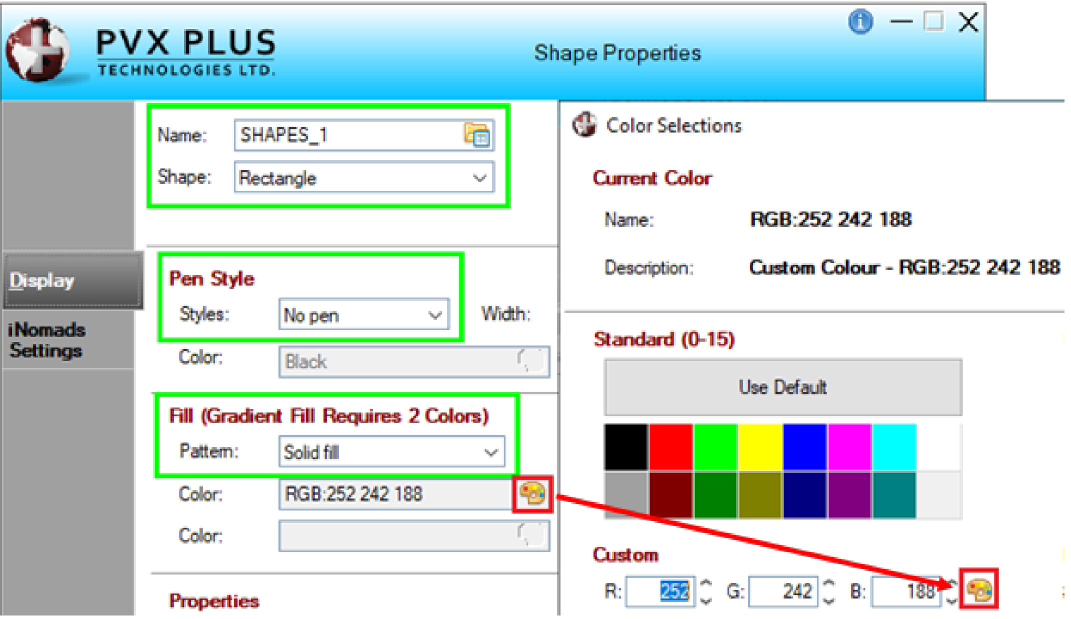
   Tip: Remember that once the panel is created, click the [**Header**] option in the top menu to make any adjustments.

2. On your computer (or on the Internet), look for an image or logo of a similar size to the one we are using (about 300x300 pixels). In the NOMADS designer, select the [**Picture**] control in the **Controls** box on the left. In the **Image/Picture Properties** window, set the position (Column 2, Line 0) and the size (Width 20, Height 10). Click the icon (yellow folder with an image) to the right of the **Picture Path** input field. When the **Bitmaps** window displays, click the [**External**] button, which opens a file selection box. Browse and select the image to insert. For the [**Image Format**] option, select Scaled. Click [**OK**].

3. In the NOMADS panel designer, select the [**Shapes**] control in the **Controls** box. Then, verify that the [**Shape**] option shows Rectangle.

   In the **Pen Style** section, for the [**Styles**] option, select No pen.

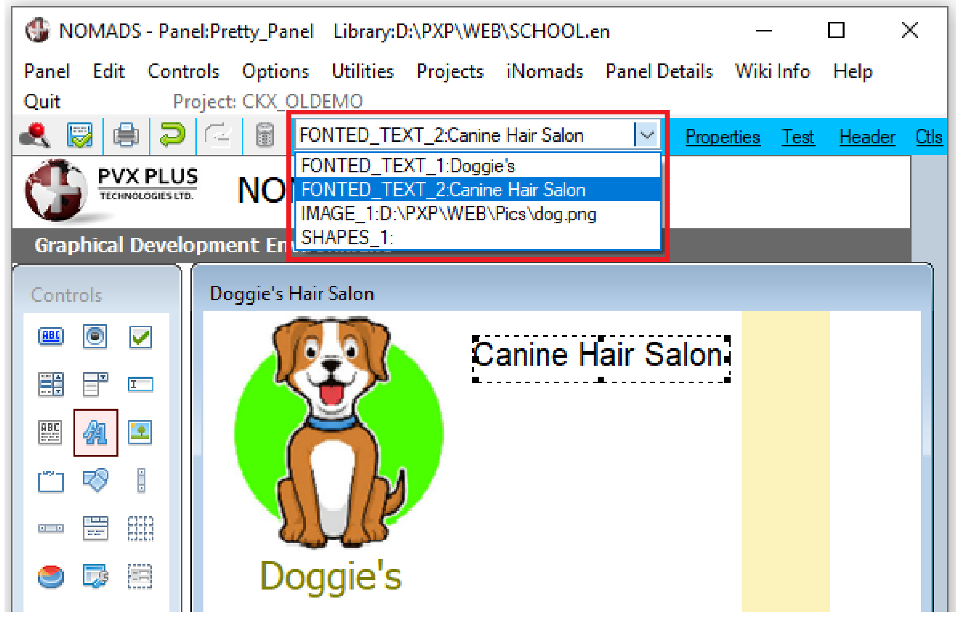
   In the **Fill** section, for the [**Pattern**] option, select Solid fill. Click the paint palette icon to the right of the [**Color**] option. When the **Color Selections** window displays, in the **Custom** section, enter the **RGB** values (R:252 G:242 B:188) or select the paint palette icon again. Return to the previous **Shape Properties** window. Set the position (Column 48, Line 0) and the size (Width 8, Height 20). Click the [**OK**] button.



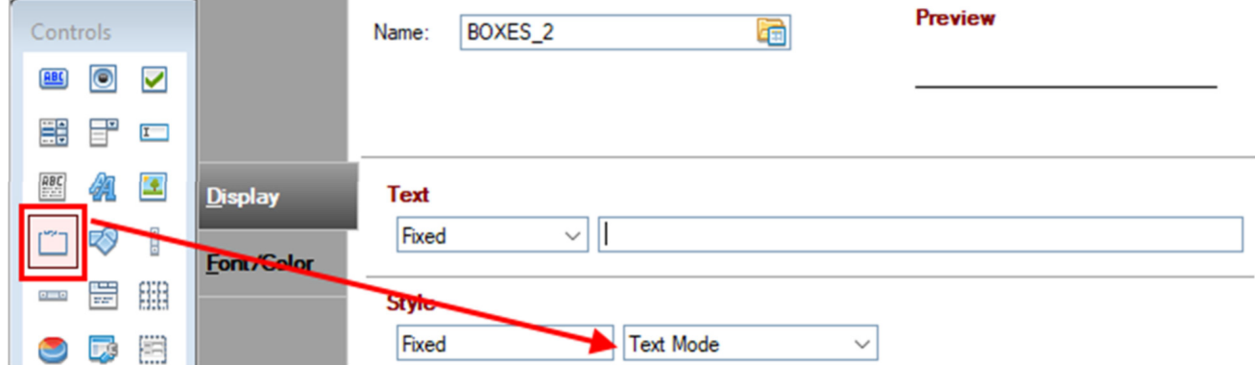
4. Select the **Fonted Text** control in the **Controls** box, and for [**Text**], enter "Doggie's". Set the **Position** (Column 5, Line 10) and the **Size** (Width 15, Height 2). In the same properties window, go to the **Font/Color** tab. For [**Font**], select Tahoma, size 24 points. For the [**Foreground Color**] option, select Dark Yellow as the text color. Click the [**OK**] button.

5. Select the **Fonted Text** control again, and for [**Text**], enter "Canine Hair Salon". Set the **Position** (Column 24, Line 1) and the **Size** (Width 23, Height 2). In the same properties window, go to the **Font/Color** tab. For [**Font**], select Arial and with a [**Size**] of 20 points. Click the [**OK**] button.

Tip: If you are using the ***Folder Style*** version of the NOMADS panel designer, another way to select the controls is to use the collapsible drop box in the top menu. You can use the green arrow button on the left to undo your actions. Remember to save your work.



6. Select the **Frame** control (also called **Boxes**). Set the **Position** (Column 24, Line 2) and the **Size** (Width 23, Height 1). Change the [**Style**] option to Text Mode.



7. Draw another **Frame** control, and this time, leave the [**Style**] option set to the default (it should be 3D Frame). Set the **Position** (Column 1, Line 14) and the **Size** (Width 22, Height 5).

8. Select the **Button** control. For [**Text**], enter "&Other Processes". Set the **Position** (Column 4, Line 16) and the **Size** (Width 16, Height 1.60). Go to the **Font/Color** tab and change the [**Background Color**] to Dark Yellow.

9. Define another **Button**, and for [**Text**], enter "Pet &Reception". Set the **Position** (Column 28, Line 7) and the **Size** (Width 32, Height 1.60). For the [**Background Color**], select Dark Yellow.

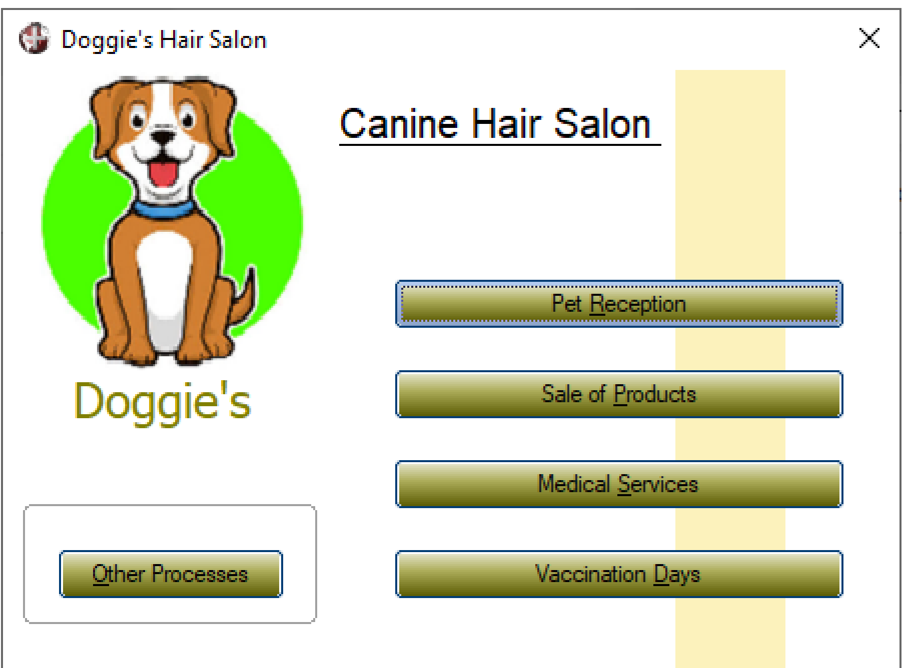
10. We are going to duplicate this **Button** control and add three more buttons. Select this button with the mouse and then right click to bring up the context menu. Select the [**Copy**] option, and then go to where we want to place it, right click and select [**Paste Object**].

    The values to change in the other buttons are:

    Text: "Sale of &Products" (Column 28, Line 10)
    Text: "Medical &Services" (Column 28, Line13)
    Text: "Vaccination &Days" (Column 28, Line 16)

Save and test the panel.

The final panel should look something like the one shown below:



Note: Do you remember what the **&** (*ampersand*) does when it is placed in the text description of a control (in this case, in the buttons)? It is called the accelerator or hot key and allows the execution of that control when the key combination [**Alt** + hot key] is pressed. **Example:** If the button has the text "&Record", the hot key will be "R", so pressing [**Alt R**] will be equivalent to selecting this button.

Note: Through the use of **Themes**, you can define color combinations for the different controls and then apply that theme to a particular panel or to an entire library. **Visual Classes** do the same at the control level.

Custom Title Bars

To make our panels more attractive and with a consistent design throughout the development phase, a **Title Bar** can be added, which is nothing more than a panel with certain characteristics that will be automatically "embedded" at the top of all panels that are denoted to use a title bar.

This "embedding" process means that the title only has to be defined once, and then it will be automatically added to all the panels. This means that if you want to change any element of it, it will be done in only one place, and the system will automatically replicate the change in all other panels. You can define the title for just a particular panel or for an entire library.

NOMADS incorporates a series of variables and controls to facilitate use. To define a panel as a **Custom Title Bar**, it must have the [**Sizing**] attribute set to **Resizable/Custom**, and its title must be a **Fonted Text** control with the name **Px_TitleBar_Caption**.

You can use many types of controls, but best practice is to be careful about which controls you use and give them unique names.

Refer to **Custom Title Bars** in the PxPlus Help documentation.

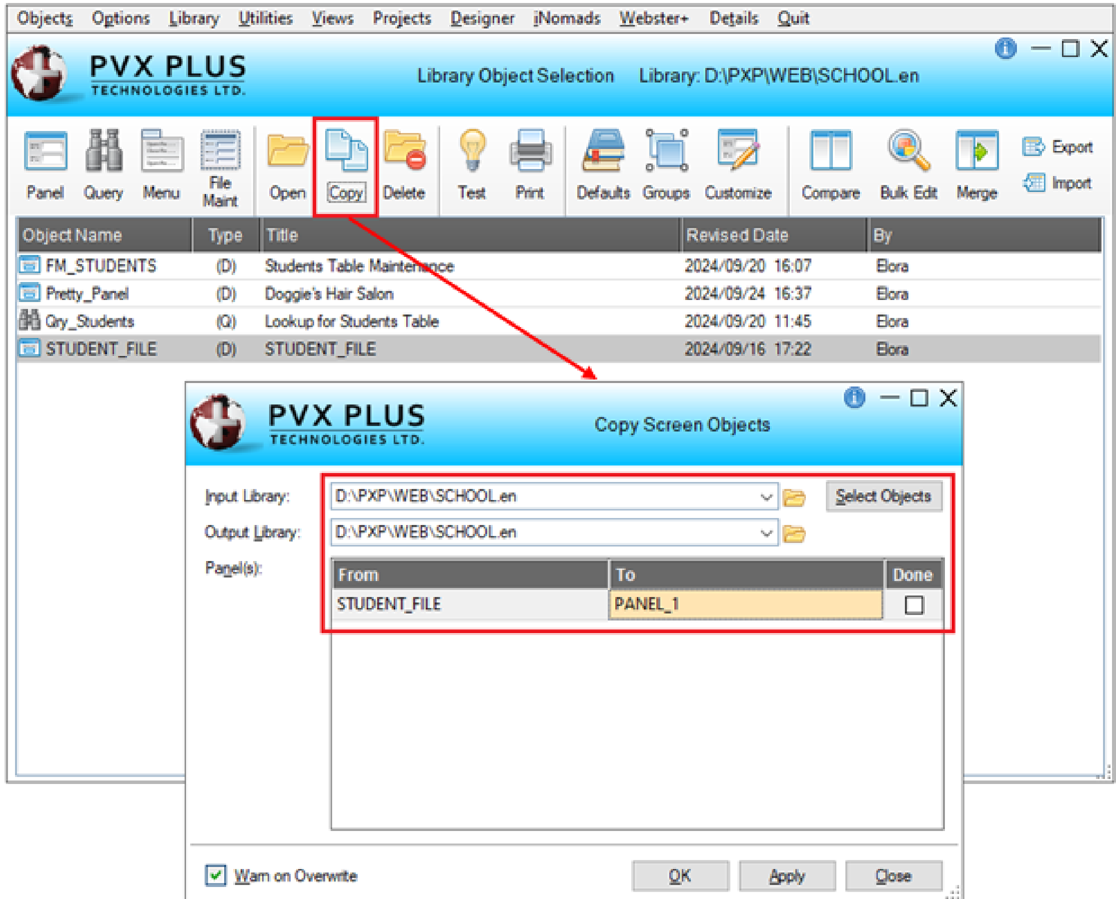
Exercise: Adding a Custom Title Bar

We will add a title bar to a copy of an existing panel, **STUDENT_FILE**.

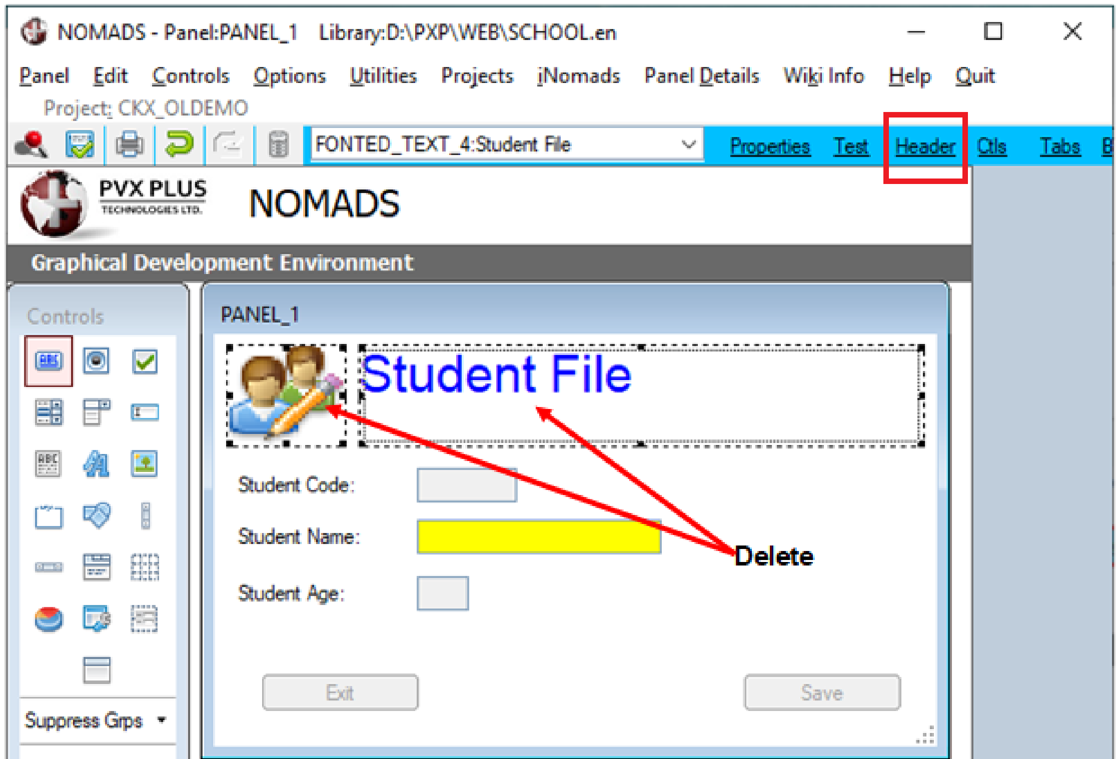
First, we will make a copy of this panel. From the PxPlus IDE main menu, select **Open Application Library** under the **Graphical Application Builder (NOMADS)** category.

Select the library, **SCHOOL.EN**. This opens the NOMADS **Library Object Selection** window, which lists the objects that make up this library. Select the **STUDENT_FILE** panel.

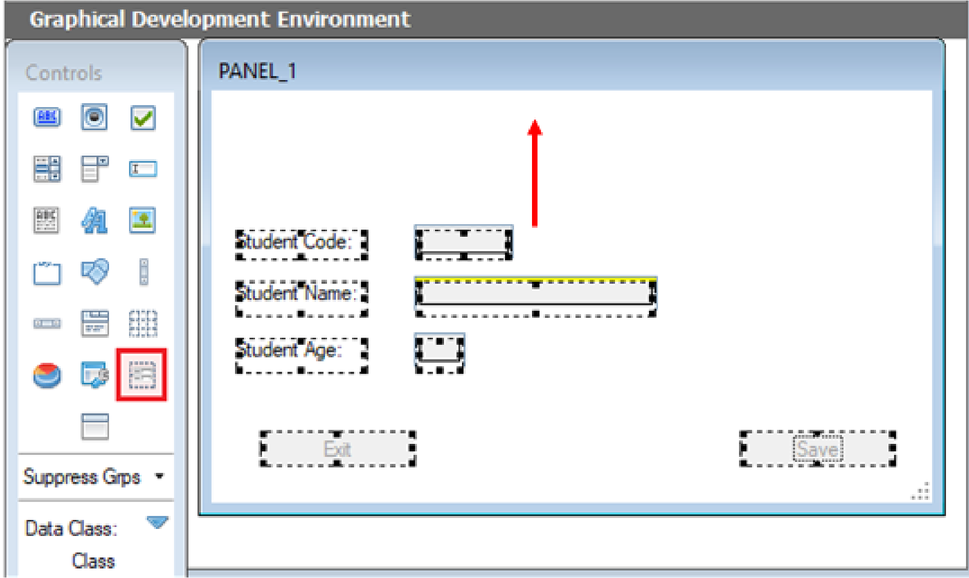
Next, click the **Copy** button at the top. When the **Copy Screen Objects** panel displays, enter the name of the new panel to copy to as **PANEL_1**. Click [**OK**]. A message confirms that this file was copied.



Open the newly copied panel. Select the [**Header**] option in the top menu and change the title to **PANEL_1**. Then, delete the image and the title from the top. To delete a control, select the control with the mouse and press the [**Delete**] key.

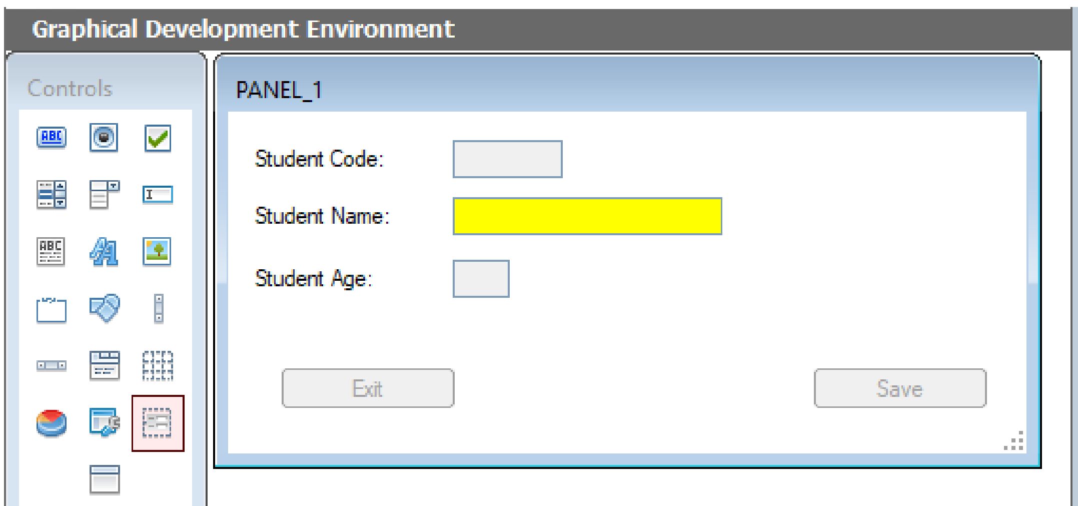


The rest of the controls need to be moved up to eliminate the empty space at the top of the panel. Select the [**Group Items**] tool in the **Controls** box on the left side. With the mouse, draw a selection box that covers all the controls so that all of them are selected. To move the selected controls, drag the mouse or use the cursor keys.

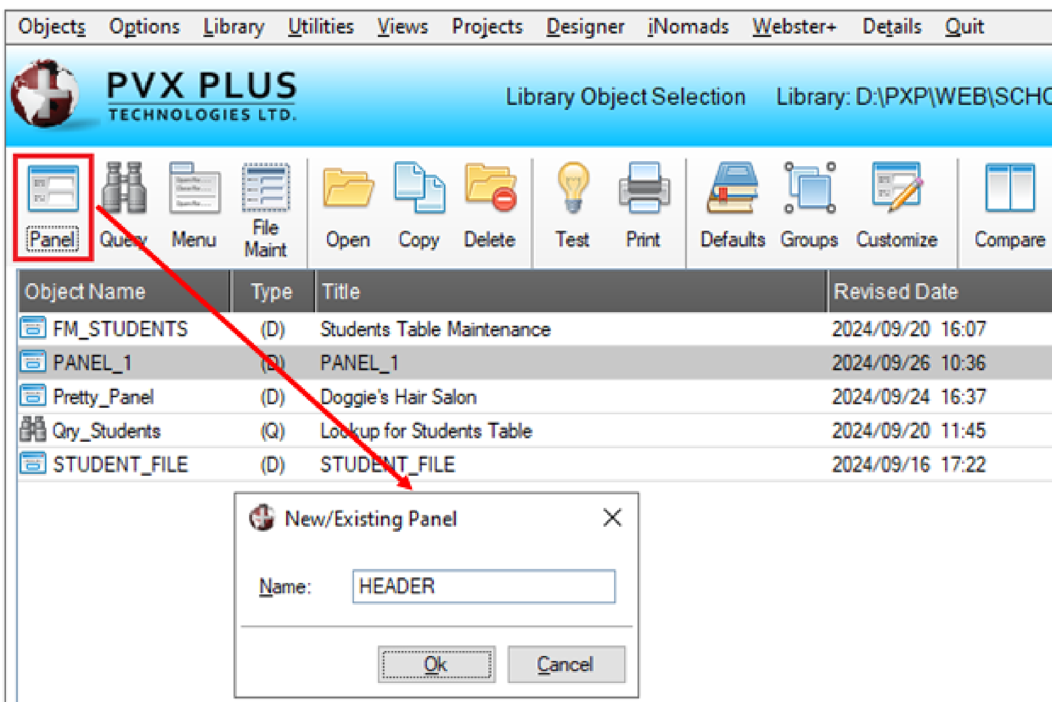


Another way to select multiple controls is to click on the first control and then continue selecting the other controls while holding down the [**Shift**] key.

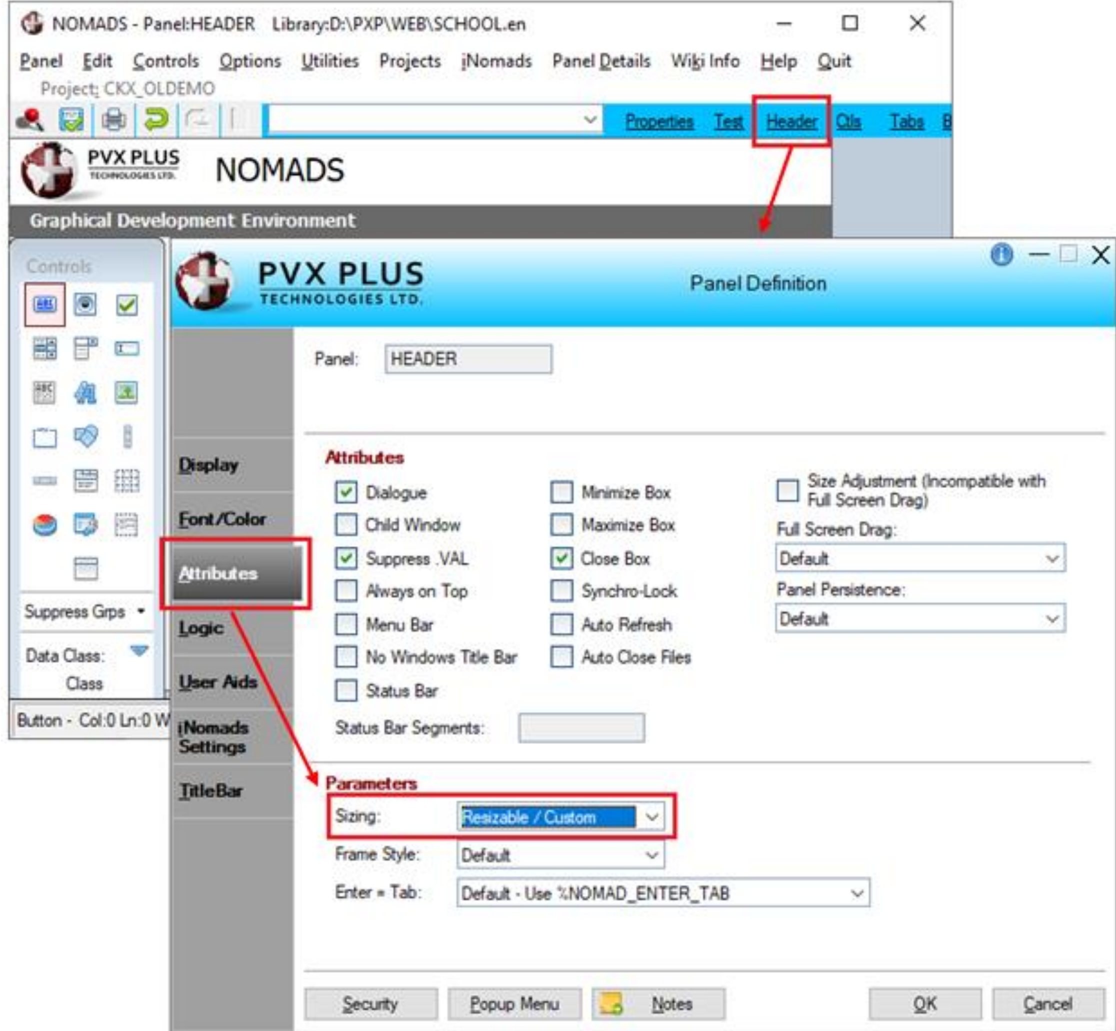
After the controls have been moved, go to the Panel properties (the [**Header**] option in the top menu) and make the height a little smaller (four lines were removed). It should look similar to this:



Now, save the panel and return to the **Library Object Selection** window where we are going to define a new panel. We will call it **HEADER**.



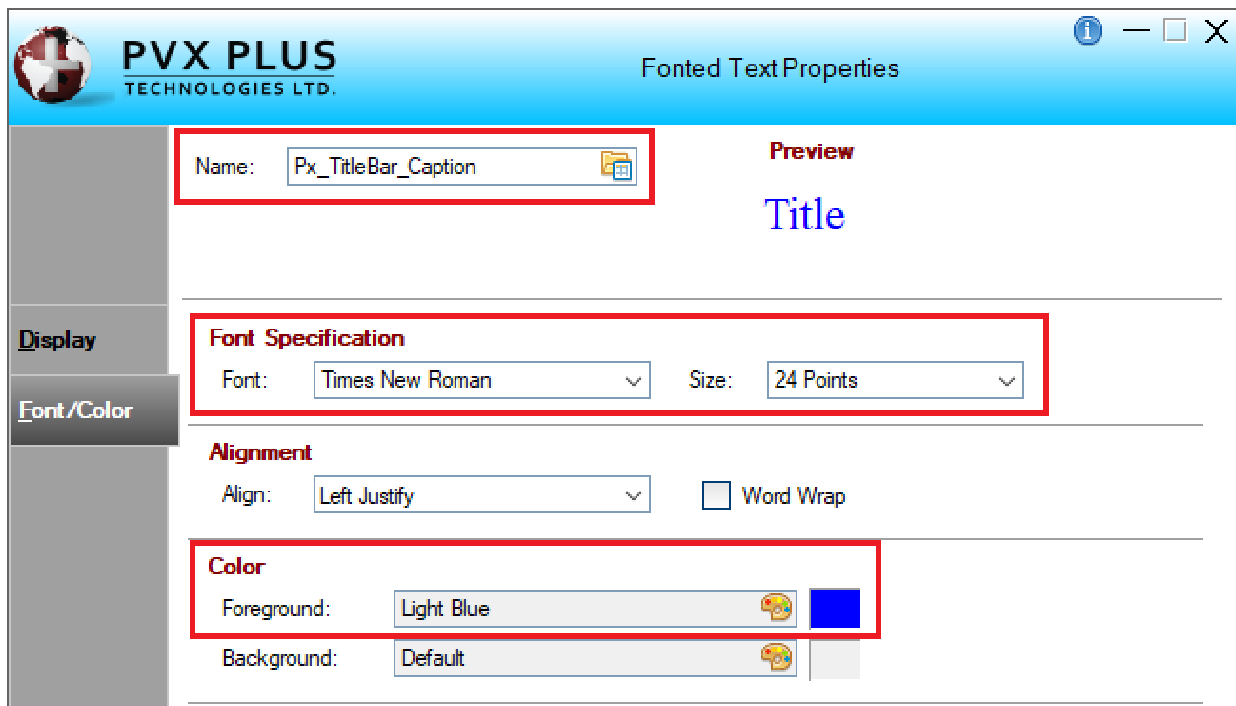
This will be the panel to embed as a **Custom Title Bar** in the **PANEL_1** panel. Once the panel is created, select the [**Header**] option in the top menu to go to the Panel properties. On the **Attributes** tab, in the **Parameters** section, change the [**Sizing**] attribute to **Resizable / Custom.** This will allow the panel to fit the size of the receiving panel.



Note: Is the [**Auto Refresh**] attribute turned Off? Please check that it is turned On. Later, we will see a way to make this "automatic".

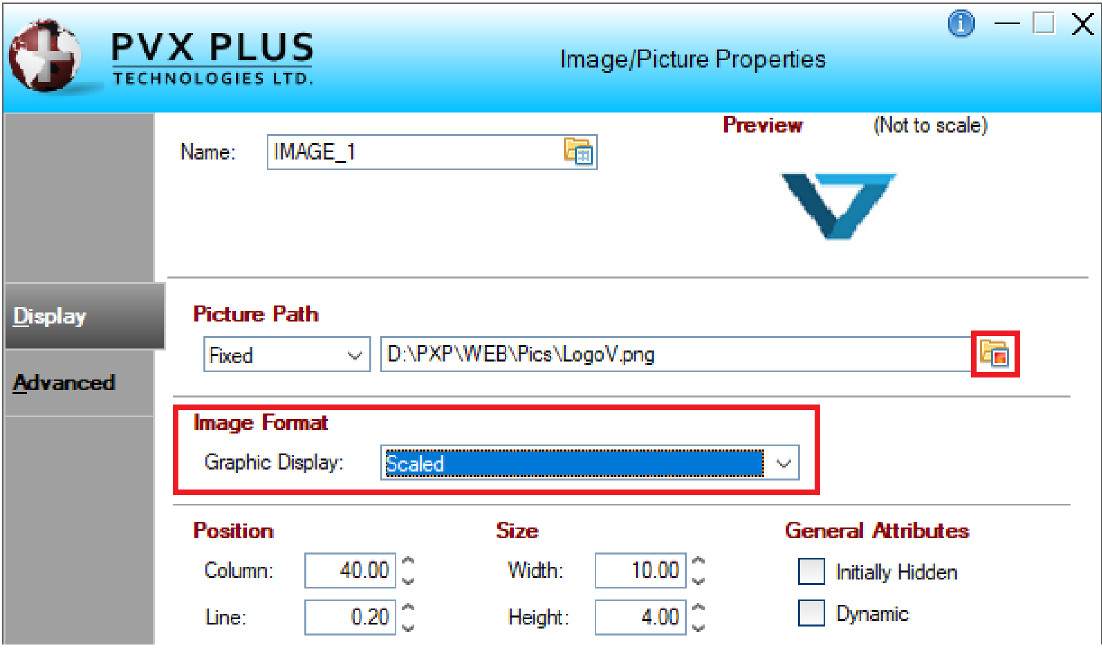
We will add an image as a logo and add a **Fonted Text** control, but this control must be named **Px_TitleBar_Caption** so that NOMADS interprets it as the title.

Let's do the title part first. Create the **Fonted Text** control, and on the **Font/Color** tab, set [**Font**] to Times New Roman and set [**Size**] to 24 points. Set [**Foreground Color**] to Light Blue. Remember to change the [**Name**] of the control to **Px_TitleBar_Caption**. For [**Text**], enter Title so that we have an idea of its appearance and location.

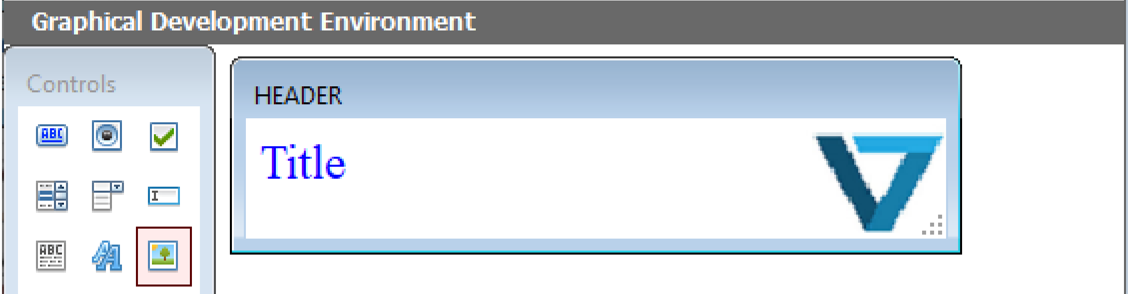


Next, create the logo of the title panel. Select the [**Picture**] control in the **Controls** box. In the **Image/Picture Properties** window, click the yellow folder icon with an image, which is located to the right of the [**Picture Path**] input field. When the **Bitmaps** window displays, click the [**External**] button and select the image.

Change the [**Image Format**] option to **Scaled**. This option will make the drawing fit the size of the control. Place the control on the right side of the panel.



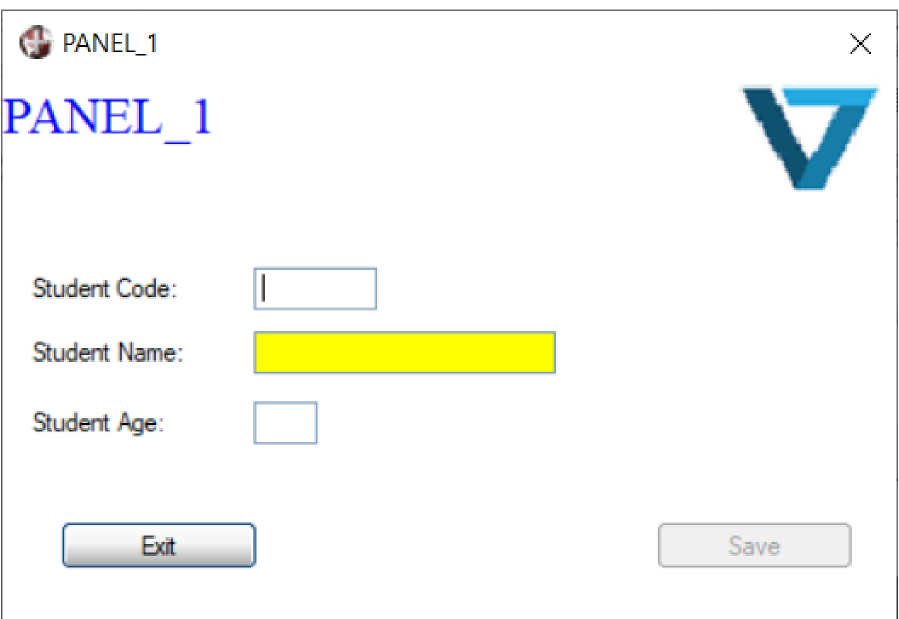
Once the logo is defined, let's proceed to make the panel 4 lines high. Select the [**Header**] option in the top menu and change the [**Height**] field. Our panel should look similar to the one shown below:



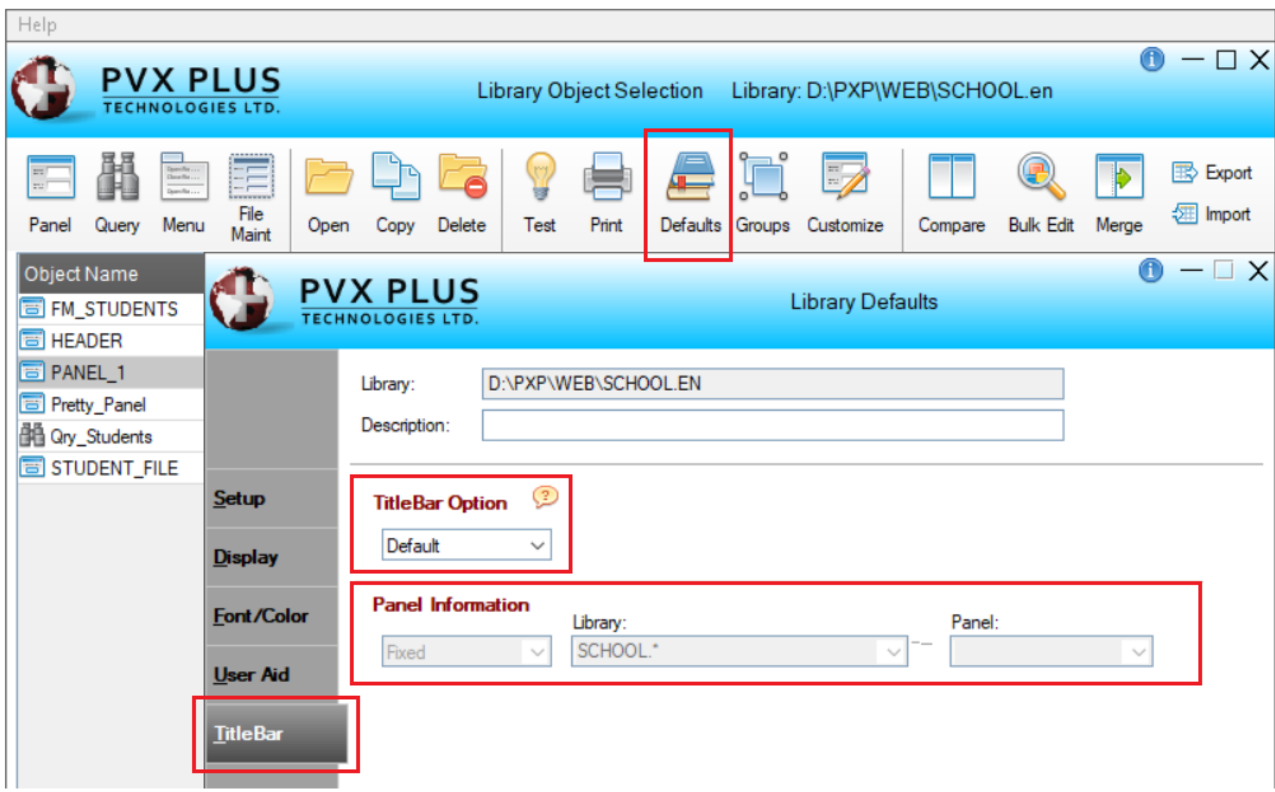
Remember to save the panel and return to the NOMADS **Library Object Selection** window.

With these simple steps, we have defined the panel that will be automatically embedded in the other panels that we define in this way. We can use the panel **PANEL_1** we defined previously, open it and modify its properties by using the [**Header**] option, go to the **TitleBar** tab, select the [**TitleBar Option**] and fill the [**Panel Information**] with the name HEADER:

Save and test the panel. It should look similar to the one shown below:



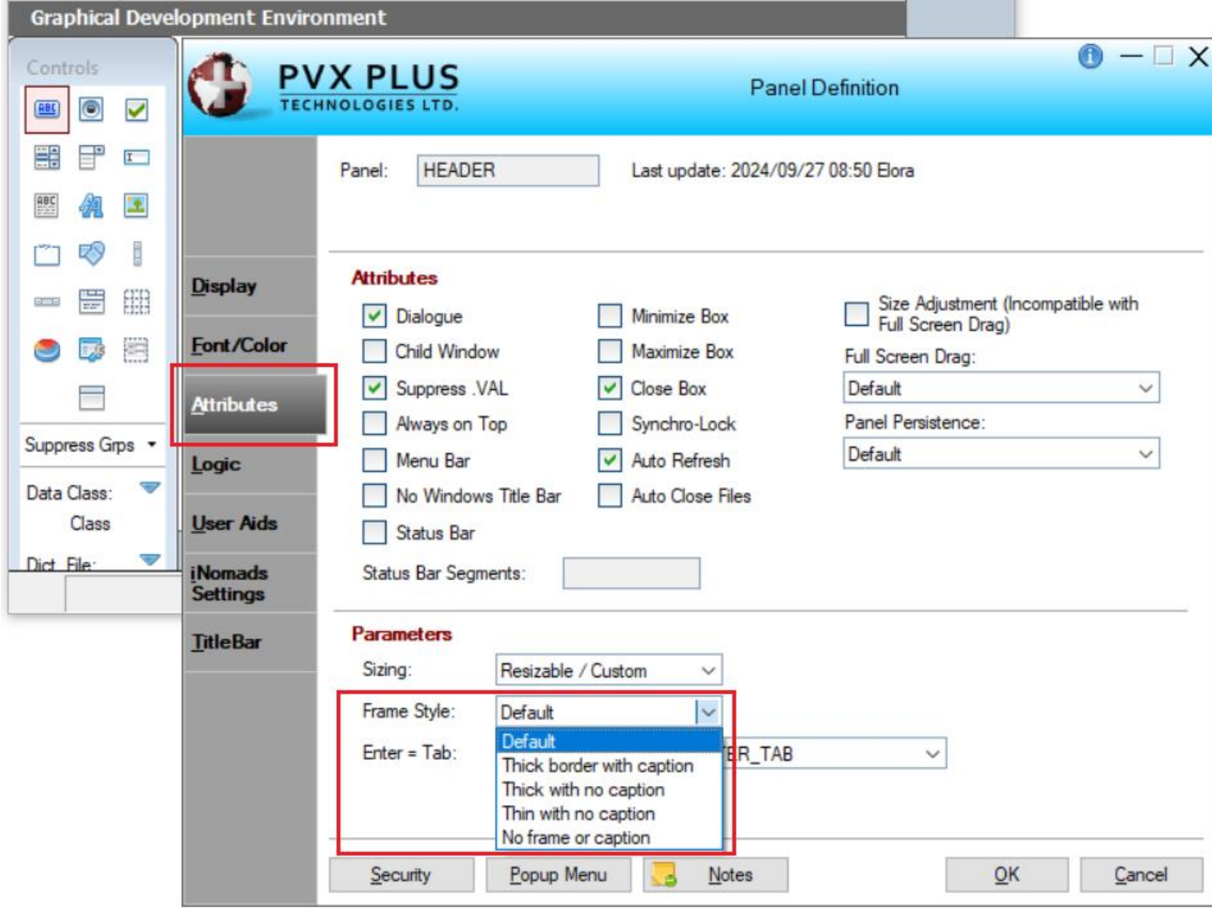
In this way, all the panels designated to have a title bar will automatically have this title (at the time of execution). In the **NOMADS Library Defaults**, select the **TitleBar** tab, and select a panel as the title bar or header for an entire library.



Important Note: For those who prefer the programmatic option, you can use the global variable **%NOMADS'TitleBar$** to define a header-type panel. This method allows you to do it even for several libraries until the content of this variable is deleted or changed.

You may have to return to the Header panel to make minor adjustments. Soon, you will have more experience in figuring out the best way to place elements on the Header panel.

You can play with some additional settings for the presentation of the window title and title bar by selecting the [**Frame Style**] setting in the **Attributes** side tab of the panel properties window (select the [**Header**] option in the top menu).



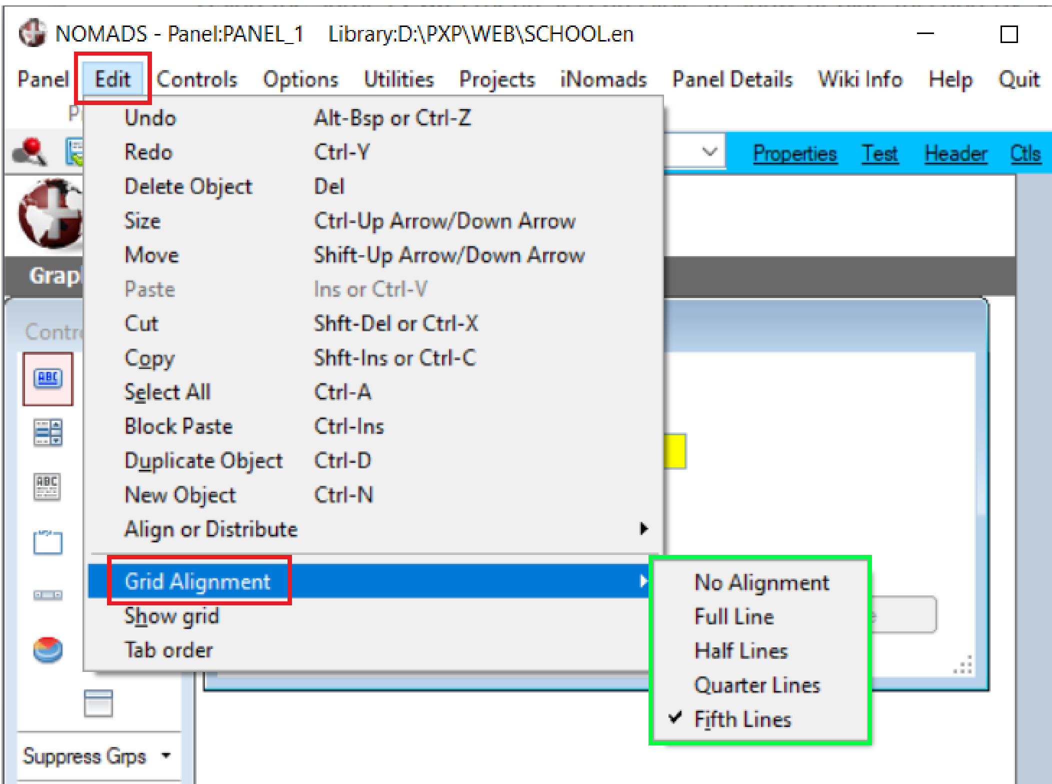
Aids to Panel Editing and Design

PxPlus, through NOMADS and its other tools, offers shortcuts to speed up our work and help us be more productive, from automatic generation of panels to the possibility of performing multiple changes simultaneously. Let's take a look at some of these features.

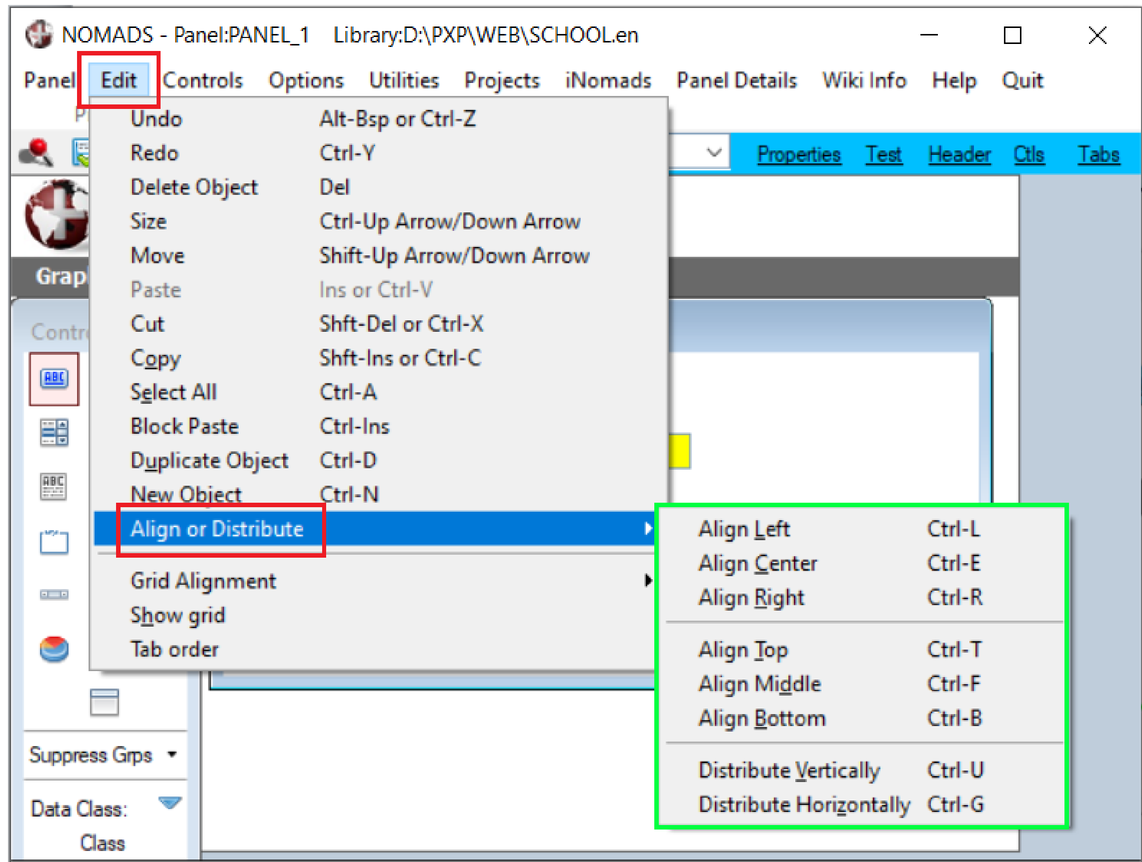
One of the basic aspects of editing in the **NOMADS Graphical Application** panel designer is the possibility of presetting a grid (invisible), which allows precise positioning of the controls. Some people use it; others prefer to turn it off. This positioning can be in lines – Full Line, Half Lines, Quarter Lines and Fifth Lines.

You can activate (or deactivate) this functionality through the [**Edit**] menu by selecting [**Grid Alignment**] and selecting the appropriate option. To turn this Off, select the **No Alignment** option.

Using the same [**Edit**] menu, it is possible to show or hide this grid by selecting the [**Show Grid**] option, which is located below the [**Grid Alignment**] option.



NOMADS allows the precise placement of controls through the [**Align or Distribute**] option, which is located on the same [**Edit**] menu at the top of the designer:



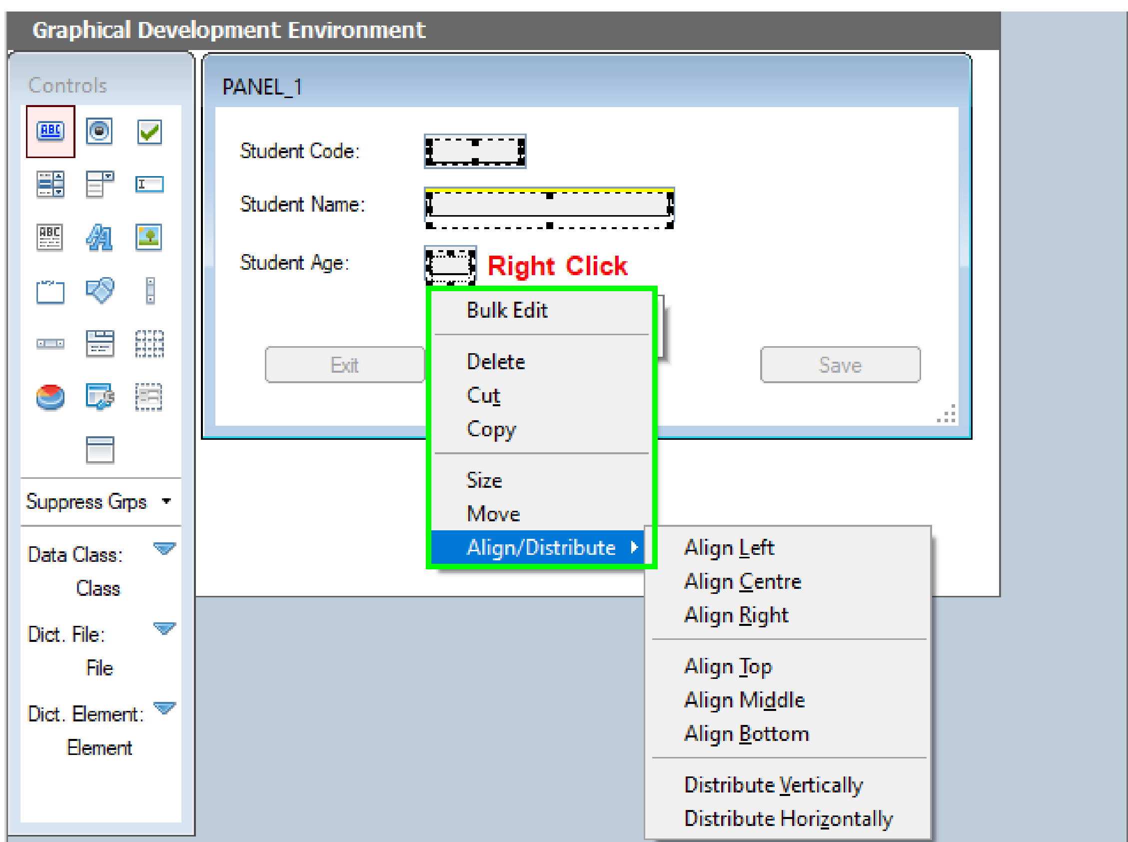
After selecting the controls, either by using the [**Group Items**] tool or by pressing the [**Shift Click**] combination, we can select any of the alternatives or press the sequence of quick keys:

Align Left [**Ctrl L**]
Align Center [**Ctrl E**]
Align Right [**Ctrl R**]

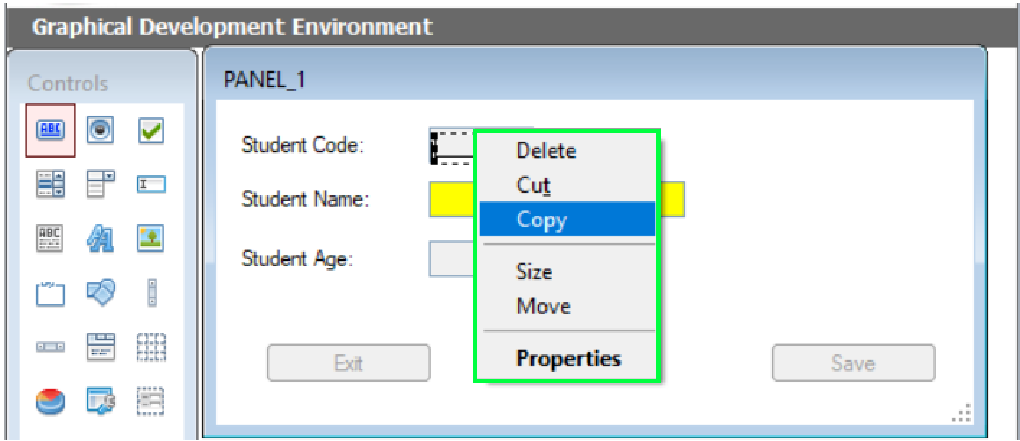
Align Top [**Ctrl T**]
Align Middle [**Ctrl F**]
Align Bottom [**Ctrl B**]

Distribute Vertically [**Ctrl U**]
Distribute Horizontally [**Ctrl G**]

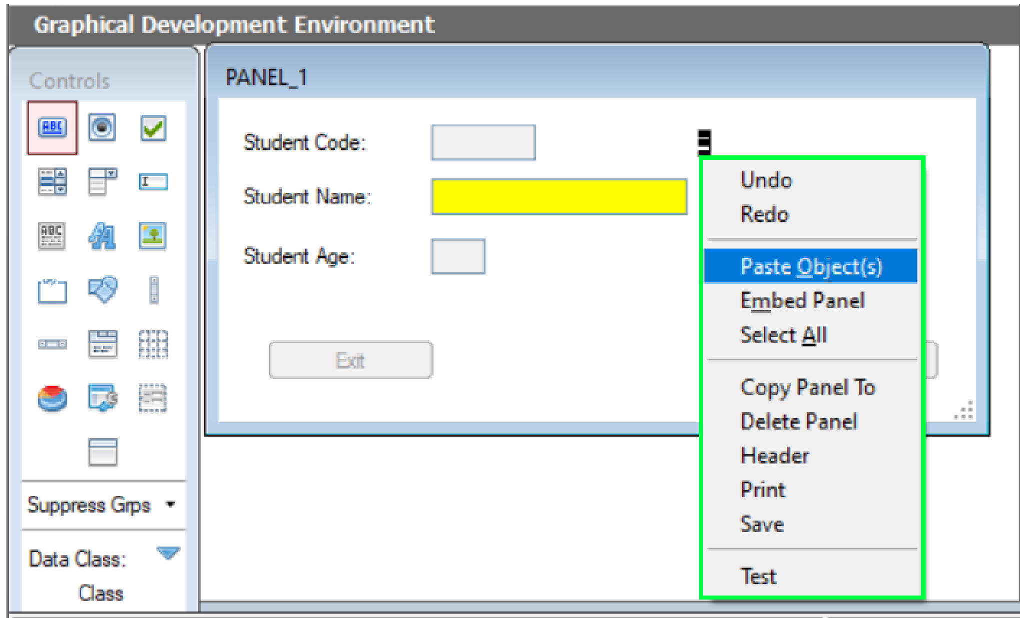
You can get that same menu by selecting several controls and right clicking to display a context menu. Select the [**Align/Distribute**] option:



In the [**Edit**] menu, you have options to delete, cut, copy and paste controls. You have to select the control and initially select [**Copy**] or [**Cut**]. With this, the control will be copied in memory.



Then, select the [**Paste Object(s)**] option to place a duplicate of the object on the panel.



Important Note: You should try to ensure that the dimensions of the box where you are going to place the control are appropriate; that is, similar to its actual size. If the dimensions are very small, the control may be partially displayed. If the dimensions are very large, the selection and manipulation of the different controls will become more laborious.

It is possible to create panels and controls that dynamically modify their sizes according to what is selected by the user (using the different buttons or by dragging the edge of the panel). As programmer, you can choose what behavior each of the controls will have. If you have a panel with buttons, by maximizing the size, you can leave the size of the buttons fixed and maximize the information display area, such as lists or drop-down boxes. You can also select the anchor or default position of each control in case the panel dimensions change.

It makes no sense to re-scale or modify the size of some panels; however, others greatly benefit from the larger workspace, especially those that display a large amount of information or have list boxes or grids.

The same panel may or may not have the Minimize, Close and/or Maximize buttons. You can look at the [**Sizing**] option in the **Attributes** side tab in the panel properties. Once the type of panel is defined, you can go to the top menu and use the [**Re-Size**] option, which we will see below.

In the case of the **PANEL_1** panel (which has just a few controls, multi-lines and buttons), maximizing or resizing doesn't make much sense. We are going to make a modification to it to include a list or **LIST_BOX** control and see how its size can be altered.

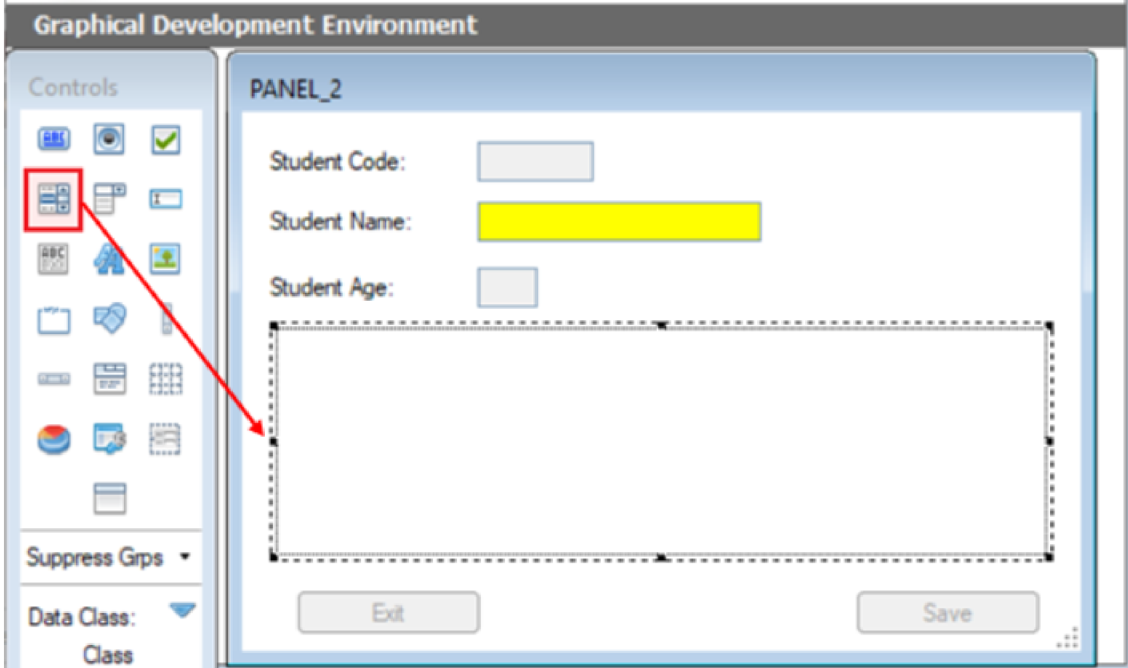
First, make a copy of this panel. In the NOMADS **Library Object Selection** window, select the **PANEL_1** panel. Click the [**Copy**] button at the top. When the **Copy Screen Objects** window displays, enter the name of the new panel to copy to: **PANEL_2**. Click [**OK**]. A message confirms that this file was copied.

Open the newly copied panel. Select the [**Header**] option in the top menu and change the title to **PANEL_2**.

Since we will be adding a **LIST_BOX** control, we need to increase the panel [**Height**] to 18. Next, select and move the **Save** and **Exit** buttons down to the bottom of the panel.

On the **TitleBar** tab, change [**TitleBar Option**] to Default.

Select the [**List Box**] control in the **Controls** box and draw the control on the panel:



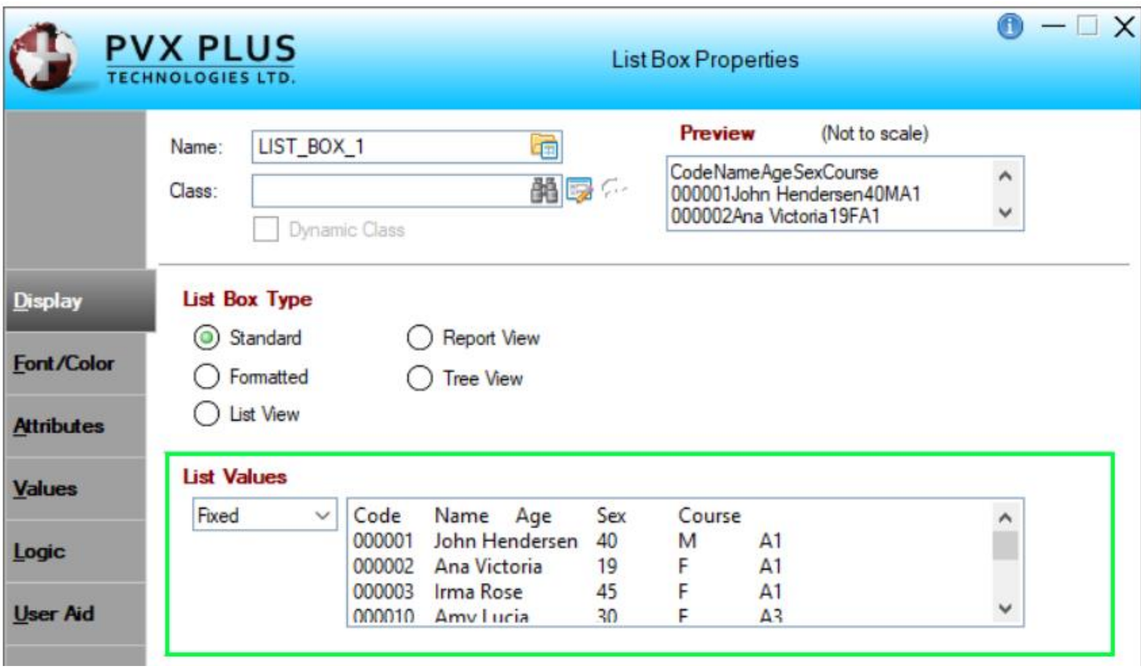
When we finish drawing the control, the **List Box Properties** window will display. This will be discussed in more detail later.

We need to enter some values to display in the List Box. To save some time, use the mouse to select the list provided below. Right click inside the highlighted list and select [**Copy**].

Then, in the **List Box Properties** window, right click inside the **List Values** box (on the **Display** tab) and select [**Paste**].

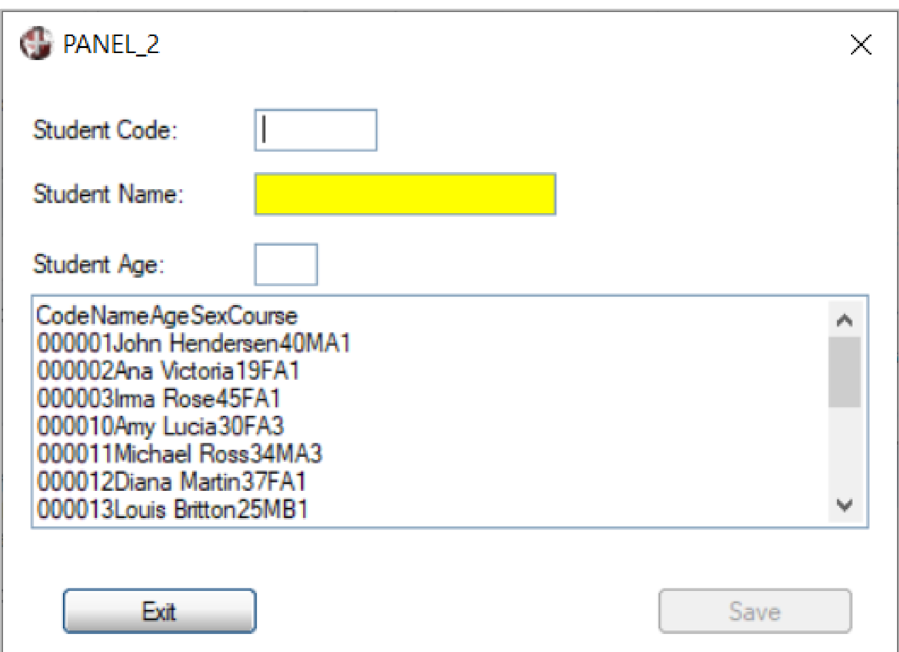
Code	Name	Age	Sex	Course
000001	John Hendersen	40	M	A1
000002	Ana Victoria	19	F	A1
000003	Irma Rose	45	F	A1
000010	Amy Lucia	30	F	A3
000011	Michael Ross	34	M	A3
000012	Diana Martin	37	F	A1
000013	Louis Britton	25	M	B1
000020	Janet Perry	40	F	B1
000021	Veronica Diaz	37	F	B3
000022	Joseph Diego	52	M	B1
000030	Michael Martin	60	M	B3
000100	Brian Peters	41	M	A3
000101	Simon Soler	27	M	B1
000102	Diana Ruiz	36	F	B3
000201	Fiona Lopez	47	F	C1
000202	Jordan James	28	M	C1

The **List Values** box will look similar to the one shown below:



Click the [**OK**] button. Save and test the panel.

The List Box will look similar to the one shown below:

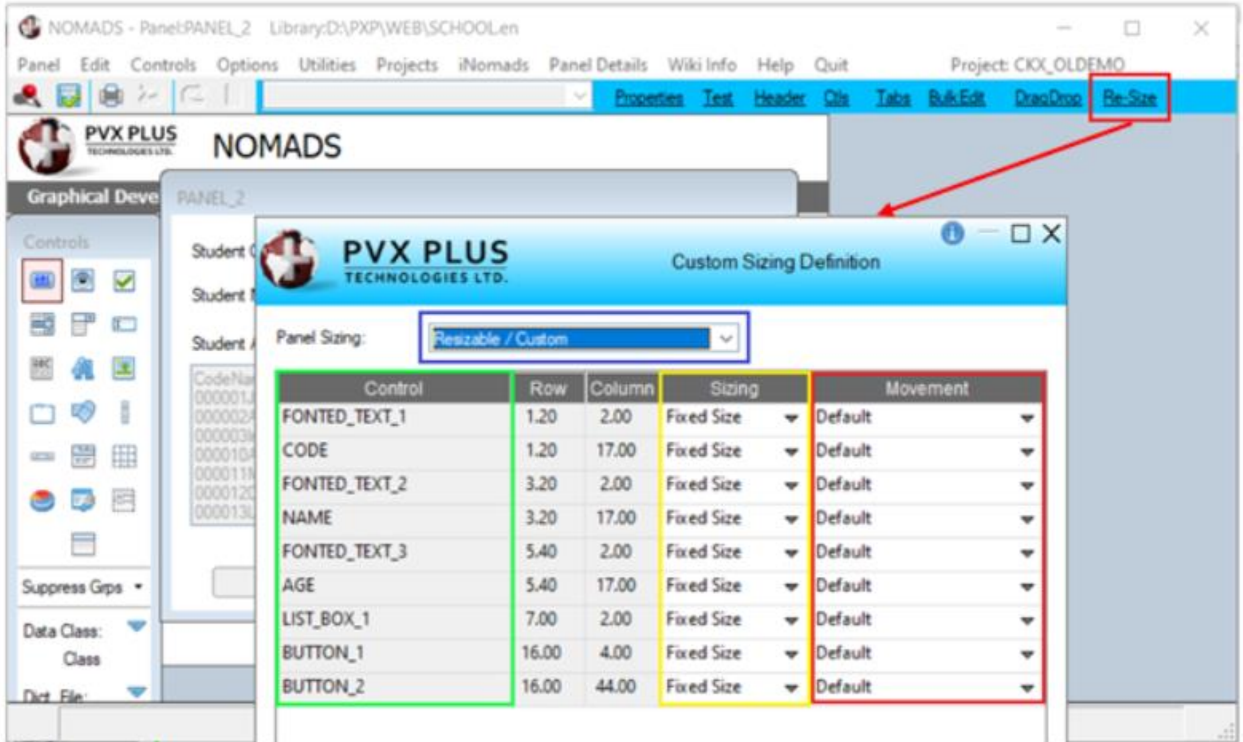


Looking at the panel, you will notice that the only thing that really benefits from the **Resizing** option is the List Box control at the bottom, which is called LIST.

Let's run the [**Re-Size**] option. Invoke it by clicking the [**Re-Size**] button at the top of the NOMADS panel designer (marked in red). Select the [**Panel Sizing**] option (marked in blue) and select **Resizable / Custom**.

Analyze the different controls/elements of the panel (marked in green). Once determined, select the type of sizing change (marked in yellow) and finally, the action or behavior (marked in red), depending on the style selection size change.

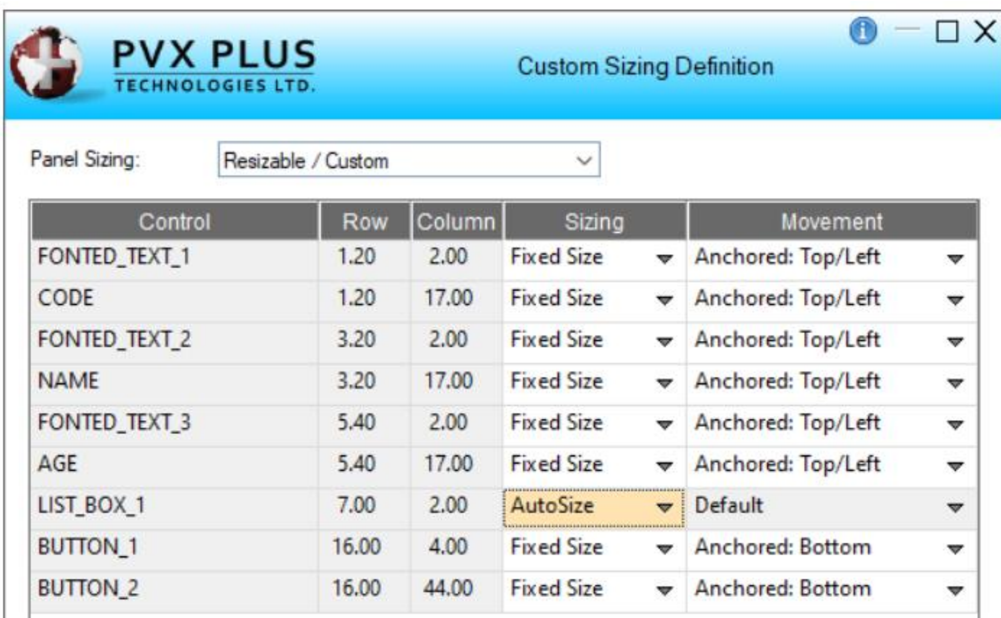
Refer to [Panel Resizing](#) in the PxPlus Help documentation.



In our panel, you can see that both the **FONTED_TEXT** and the input box controls called **CODE**, **NAME** and **AGE** should be the same size (**Fixed Size**) and "anchored" to the upper left corner; that is, if the panel grows in size, they will retain their position with respect to that corner.

The buttons should be the same size and anchored to the bottom of the panel; therefore, only the **LIST_BOX** control will be left to change the size, which we can let float in the center.

The changes would look like this:

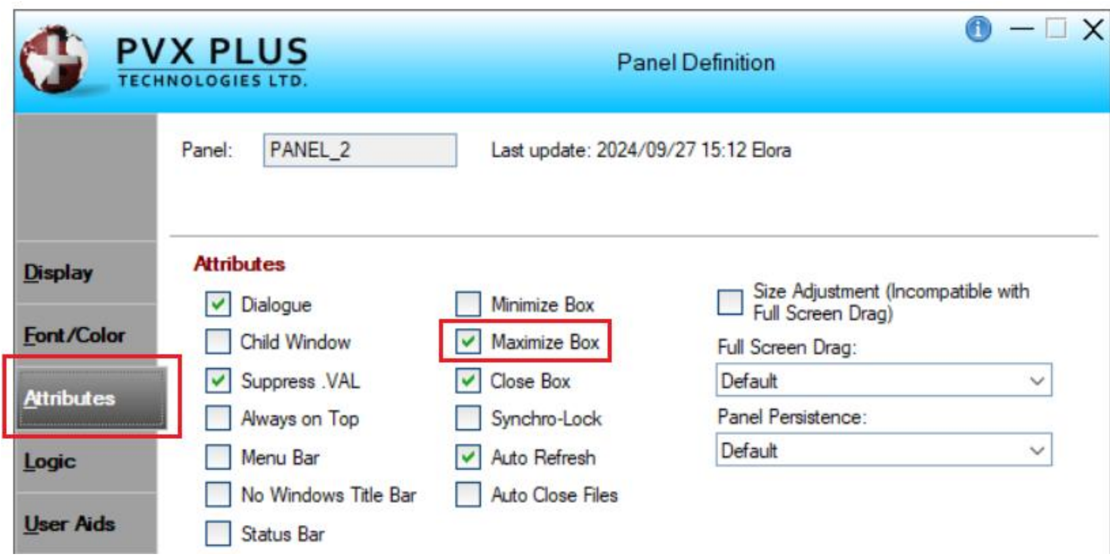


Control	Row	Column	Sizing	Movement
FONTED_TEXT_1	1.20	2.00	Fixed Size	Anchored: Top/Left
CODE	1.20	17.00	Fixed Size	Anchored: Top/Left
FONTED_TEXT_2	3.20	2.00	Fixed Size	Anchored: Top/Left
NAME	3.20	17.00	Fixed Size	Anchored: Top/Left
FONTED_TEXT_3	5.40	2.00	Fixed Size	Anchored: Top/Left
AGE	5.40	17.00	Fixed Size	Anchored: Top/Left
LIST_BOX_1	7.00	2.00	AutoSize	Default
BUTTON_1	16.00	4.00	Fixed Size	Anchored: Bottom
BUTTON_2	16.00	44.00	Fixed Size	Anchored: Bottom

Notice that the only one that has **AutoSize** is the **LIST_BOX** control, which has Default movement; the others have a Fixed Size and are Anchored.

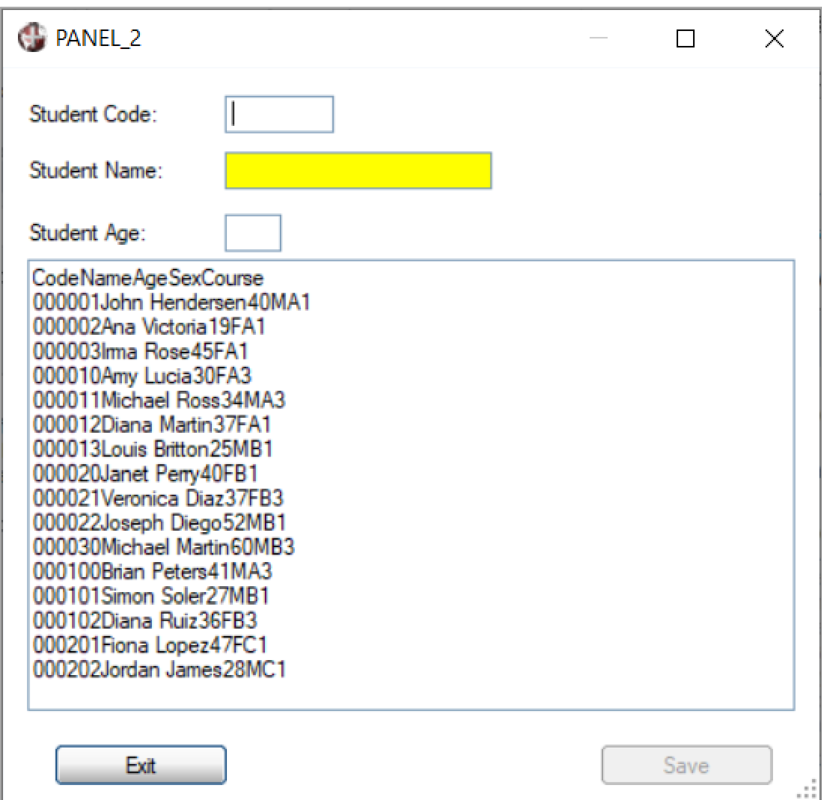
To make the panel resizable, it must also have a Maximize Box. Go to the panel properties by selecting the [**Header**] option in the top menu of the designer.

On the **Attributes** tab, select the [**Maximize Box**] option.



Save and test the panel.

You can also manually resize the panel. Hover the mouse pointer over the bottom right corner of this window until it changes to a double-headed arrow. Then, click and drag the mouse until the window is the desired size.



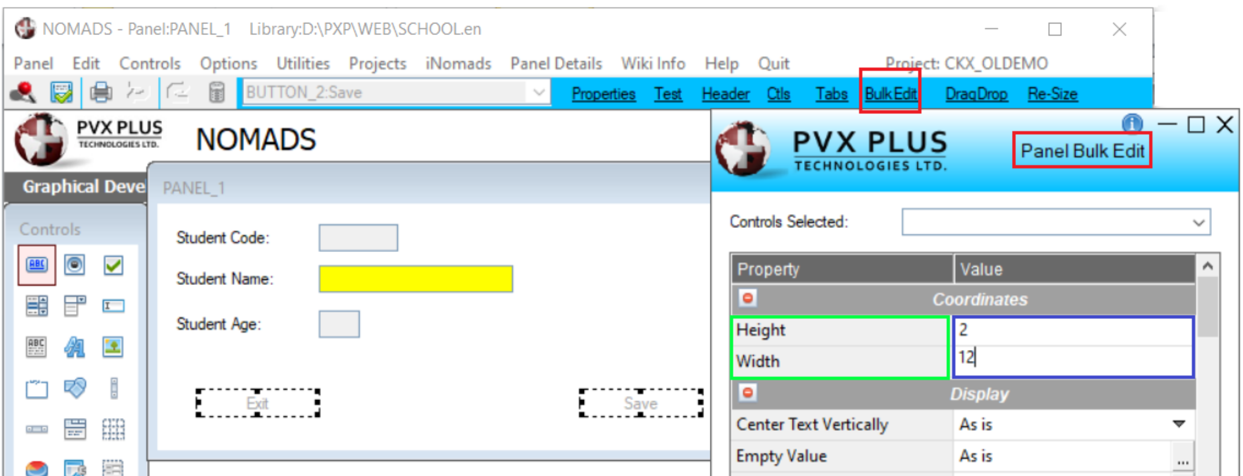
All controls maintain their size and location, and only the **LIST BOX** control grows.

Using the Bulk Edit Utility

The **Bulk Edit** utility is a powerful and very easy tool to use! We can select multiple controls and simultaneously modify several properties for all the selected controls.

Example: We have a panel with several buttons, which do not have the same dimensions, and we want to make them the same size. We select these buttons and then click on the [**Bulk Edit**] option in the top menu of the NOMADS panel designer.

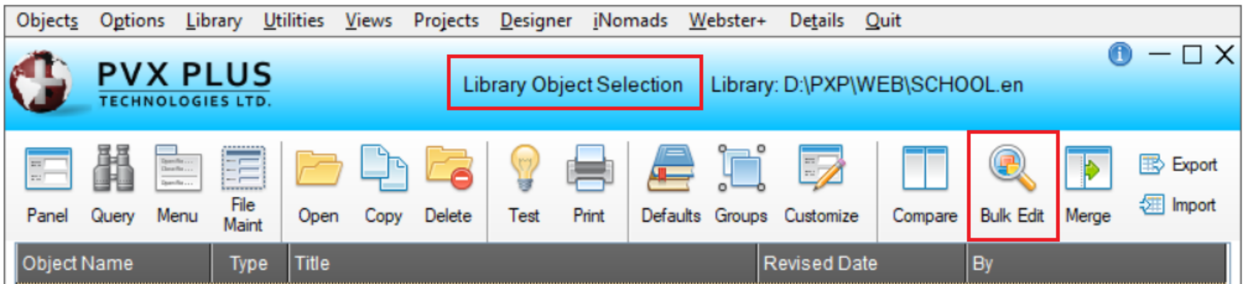
The **Panel Bulk Edit** window displays, showing two columns and several categories. We locate the property that we are interested in changing and, in the right column (**Value**), we enter the new value of that property.



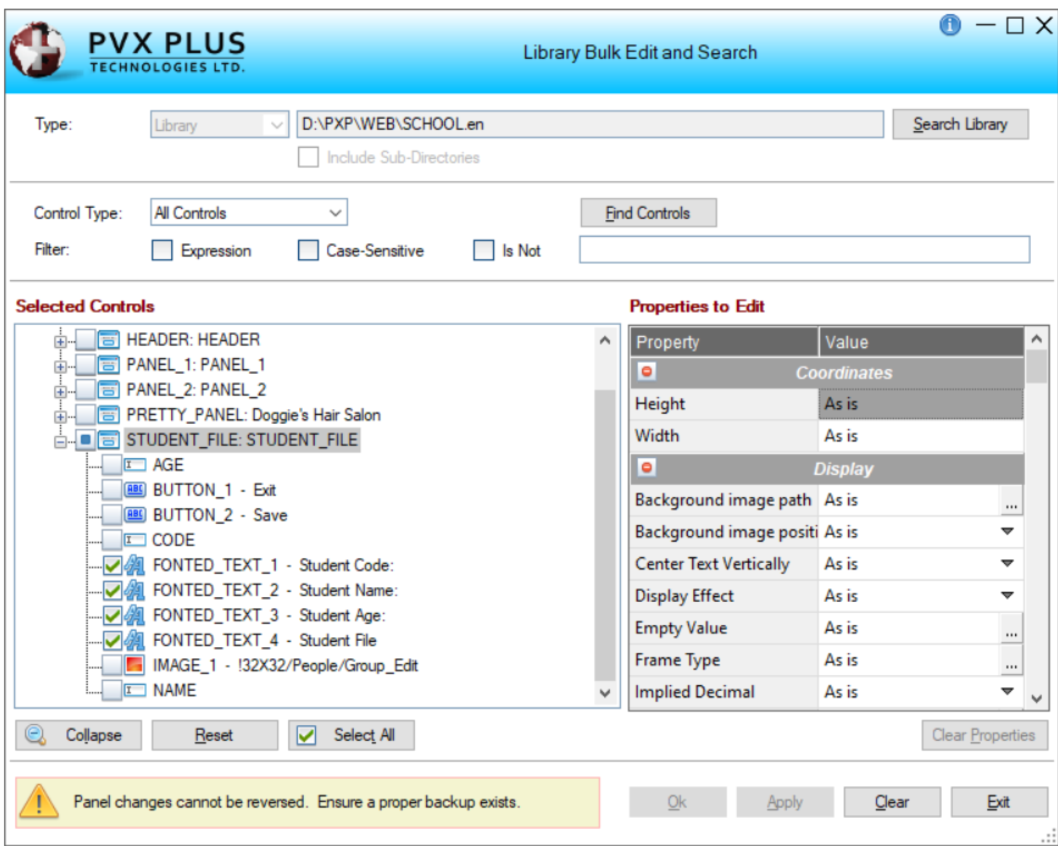
Using this method, you can make all the buttons look the same or make all the titles the same size. This utility works for the controls selected in the same panel.

Refer to **Panel Bulk Edit Utility** in the PxPlus Help documentation.

In more complicated cases where we want to do this type of property editing in bulk (at the level of an entire library), we can use the **Library Bulk Edit and Search** utility by clicking the [**Bulk Edit**] option in the top menu of the NOMADS **Library Object Selection** window:



The operation of this utility is very similar to the previous one, but instead of selecting the controls within the panel, a collapsible tree-type hierarchical structure will open where we can select the controls that we want to change in multiple panels:



Note: If the **Selected Controls** list is empty, click the [**Find Controls**] button, which will cause NOMADS to scan all panels in the library and display them on the left.

Important Note: The changes made by this utility ***cannot be undone***. It is strongly recommended to ***make a backup*** copy before running it.

Refer to **Library Bulk Edit and Search Utility** in the PxPlus Help documentation.

7. NOMADS: Using Panel Controls

Now, we can venture into designing a new panel where we can experiment with the controls and know them more in depth.

From the PxPlus IDE main menu, open the **Graphical Application Builder (NOMADS)** category, select **Open Application Library** and open our library **SCHOOL.EN**.

Create a new panel called **PNL_COURSE**, which will be used to experiment with the different controls. Once the panel is created, click the [**Header**] option in the top menu of the panel designer. On the **Attributes** tab, select the [**Auto Refresh**] option. Remember to save before testing the panel.

Define an **Exit** button in the lower right part of the screen. In the **Button Properties** window, enter the [**Name**] and [**Text**] for the button. Move on to the **Logic** tab, and for the **When Button Pressed** event, select the **End** action.

Radio Button

Once the **Exit** button is drawn, we are going to start creating a **Radio Button**. Basically, it is a control that can have several instances (minimum two), but one (and only one) must be selected at a time.

This control must be "drawn" as many times as it has options, and each time we create it, we must specify the name, text and value.

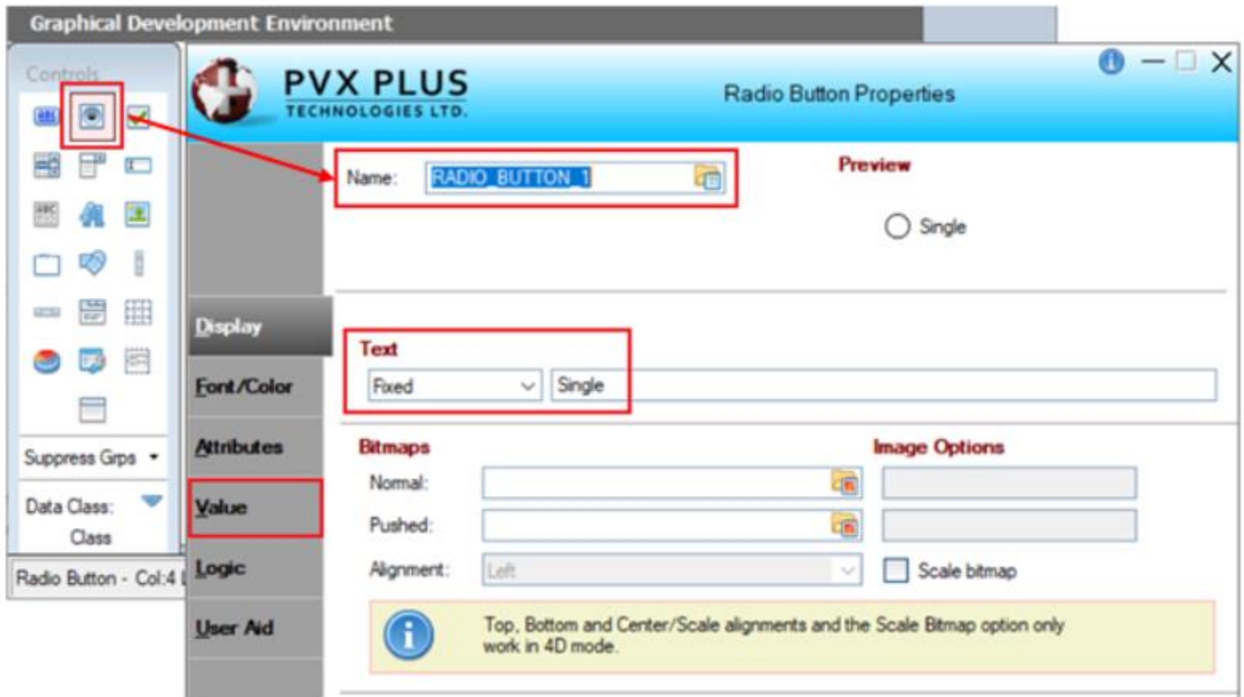
Example: Let's define a **Radio Button** to ask about the marital status of a person, which can be Single, Married, Widowed and/or Divorced. A person cannot have two marital statuses at the same time, but neither can a person be without having one defined.

Select the **Radio Button** in the **Controls** side box and draw its outline. When the **Radio Button Properties** window displays, we must specify the name. For this example, we will use the name provided, which is **RADIO_BUTTON_1**.

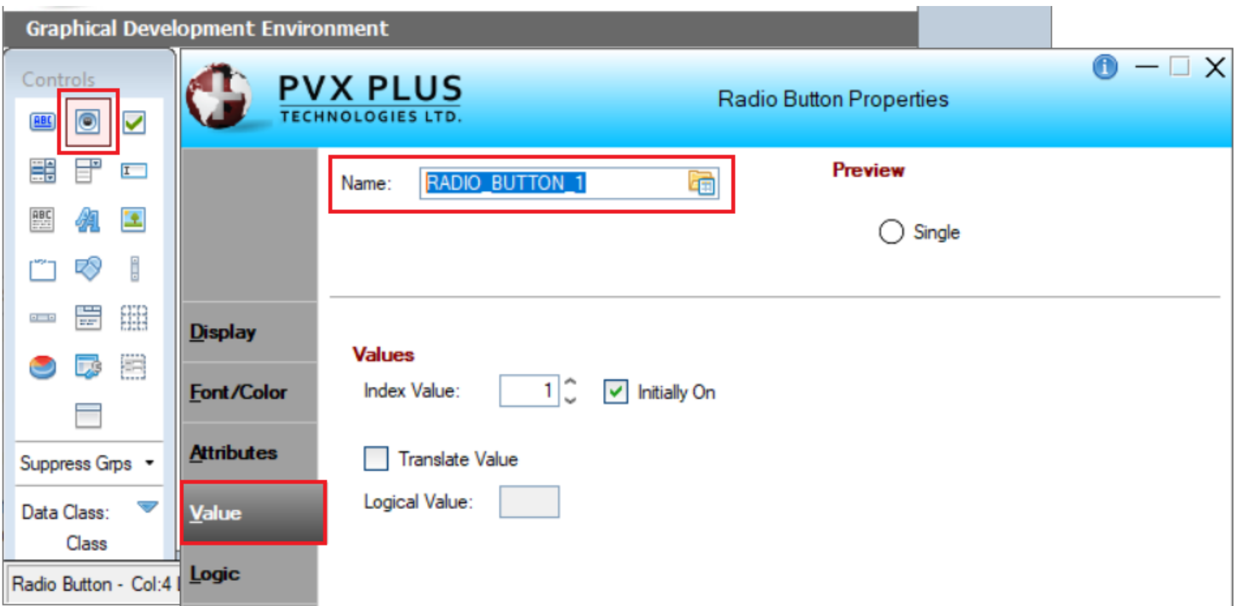
Note: This is the control indicator. ***All the options of the same Radio Button must have the same name***.

Enter the text for the first option, **Single**. Then, we must go to the **Value** tab to specify the index and whether this option is On or Off.

Note: When defining a **Radio Button** control, you must specify only one of the options (number) as initially On.



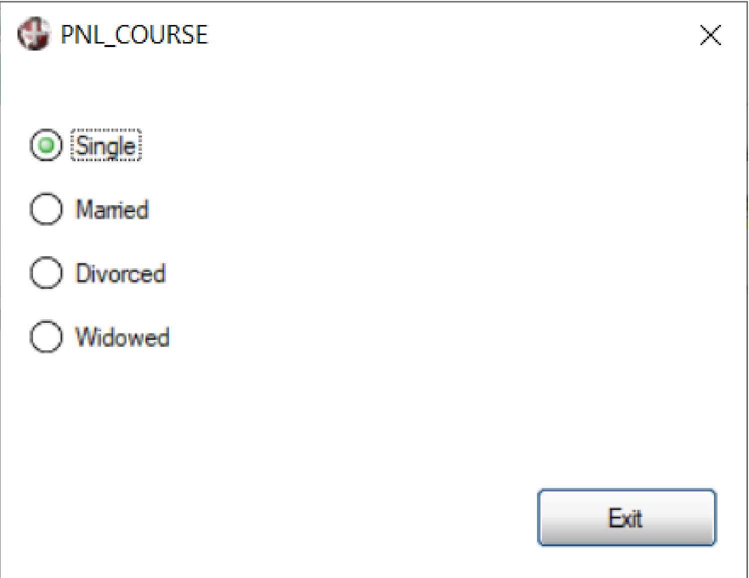
In addition to the index and its value, it could have an alternate **Translate Value**; for example, S for Single, M for Married, D for Divorced and W for Widowed.



Note: Remember that the four options belong to the same Radio Button control, so ***all four options must have the same name***.

Once we have drawn the four options (using the same control name but changing the [**Text**] and the [**Index Value**] for each option), only one option should be On at the beginning.

The panel should look like the one shown below:



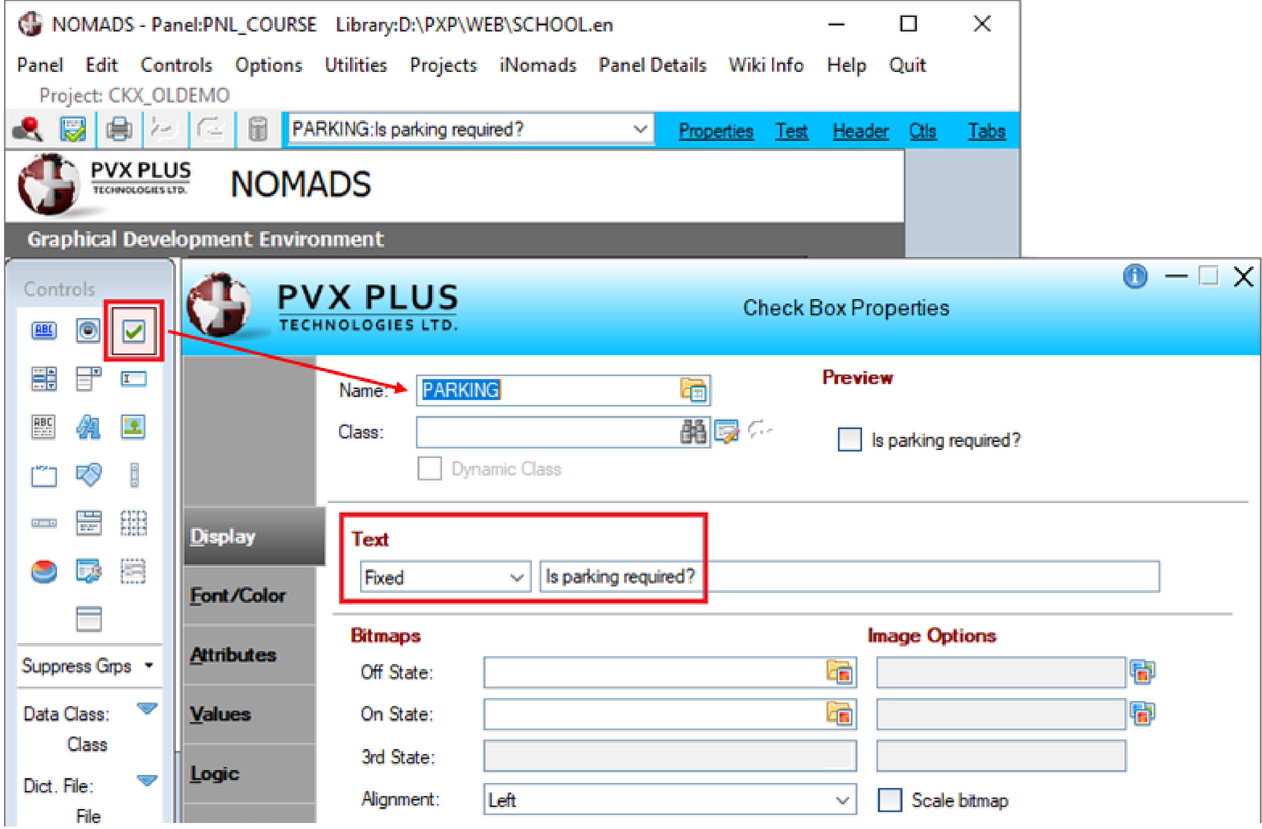
By testing the panel, we see that the options are mutually exclusive; that is, we cannot select two of them at the same time. It is recommended that you study the different options that Radio Buttons have, such as location variations, icons, letters and colors, etc.

Note: It is possible to have two or more Radio Button controls on the same panel. In our previous panel, we have one control with four options. What indicates the number of controls is the name, not the location on the screen; that is, we can have the options grouped (the common thing) but what makes them work as a single control is that they all have the same name. If we define a second Radio Button control and assign it a different name, we will effectively have another control, not another option from the previous control.

Refer to **Radio Button Control** in the PxPlus Help documentation.

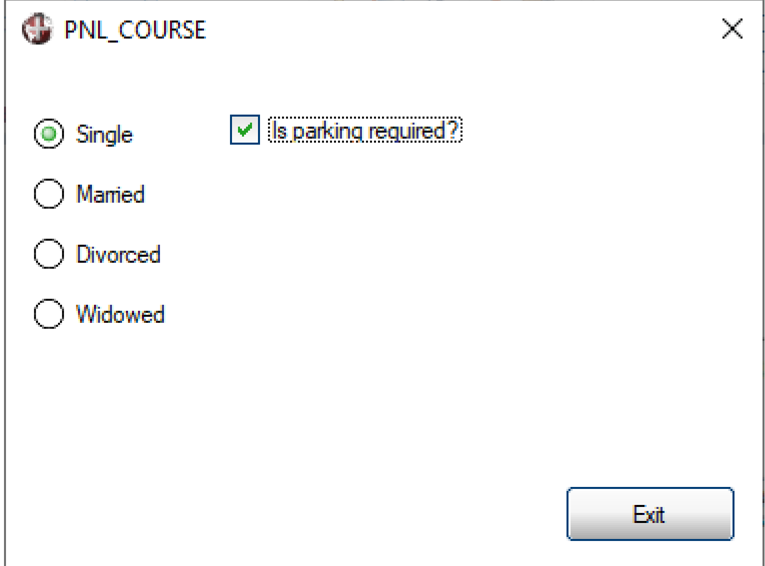
Check Box

The **Check Box** is a control that can be in two states: On/Off, Checked/Unchecked, etc. We will use the control to ask a question, such as whether parking is required. If the control is On (checked), the answer will be Yes.



This control functions similar to a flag, which can be optionally turned On to indicate a state. This control allows multiple variations, such as placing the box on the right side and changing the states (On/Off) with two icons so that when it is Off, an icon appears, and when it is checked, the icon changes. It also has a feature called [**Sticky Button**] that simulates when a button is pressed and becomes depressed, and if it is pressed again, it will appear highlighted.

Another interesting functionality of the Check Box is the **Tri-State** state where the control, instead of having two possible values, will have three.



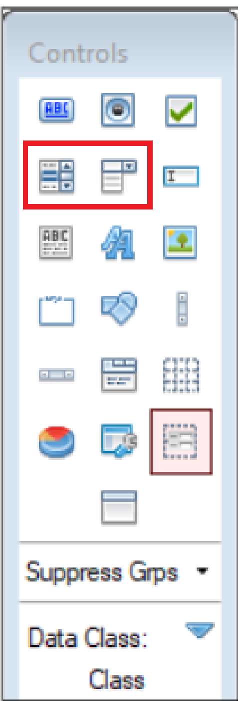
Tip: It is a good idea to experiment and try the different alternatives of these controls. Practice will determine your skill and thus help you know the scope of each of the controls.

Refer to **Check Box and Tri-State Control** in the PxPlus Help documentation.

List Boxes

When a lot of information must be displayed on the screen, one of the most popular ways is through List Boxes (or simply, Lists), an information container that allows you to display a quantity of information at a time and also select any of these values. PxPlus offers several types of lists: unformatted, formatted, report type, expandable, tree type, etc. The basic operation of all of them is similar. Some are more complete, others more basic, but in essence, they all allow information to be displayed.

At the NOMADS level, two controls are offered in the **Controls** side box to define lists: the control called **LIST_BOX** and another control called **DROP_BOX**. The fundamental difference is that the **DROP_BOX** takes up much less space (basically, the same space as a button, for example), and it is necessary to select it to show more data, while the **LIST_BOX** offers a larger box or control that can show more information at all times.



The **LIST_BOX** control has five sub-types or presentation formats: **Standard**, **Report**, **Formatted**, **TreeView** and **ListView**.

Refer to **List Box Type** in the PxPlus Help documentation.

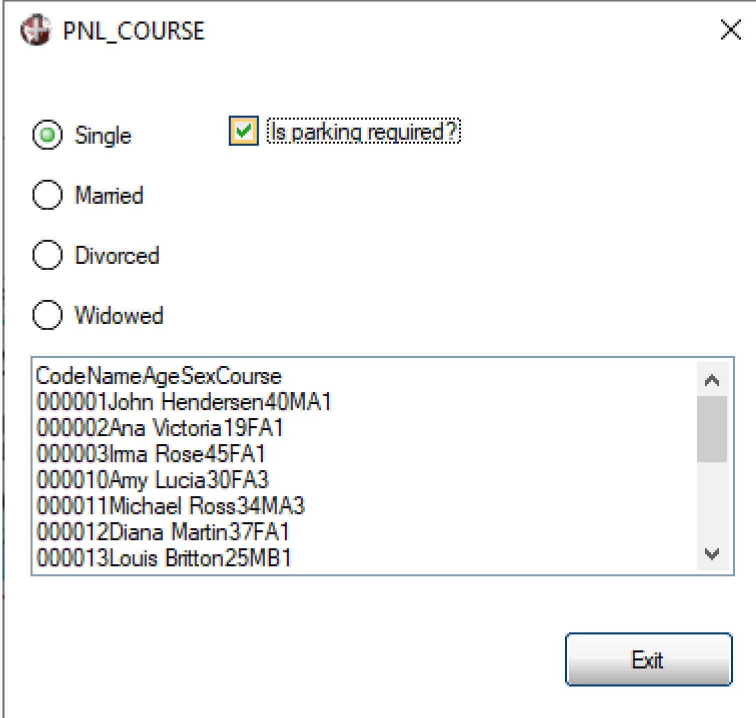
We are going to start by studying the **Standard LIST_BOX** control, and then we will see some variants.

We are going to modify the **PNL_COURSE** panel to make room to define a **LIST_BOX** control with enough space (in this example, the **LIST_BOX** dimensions are 50 columns x 8 lines). The control will be called **LIST** and be of the **Standard** type.

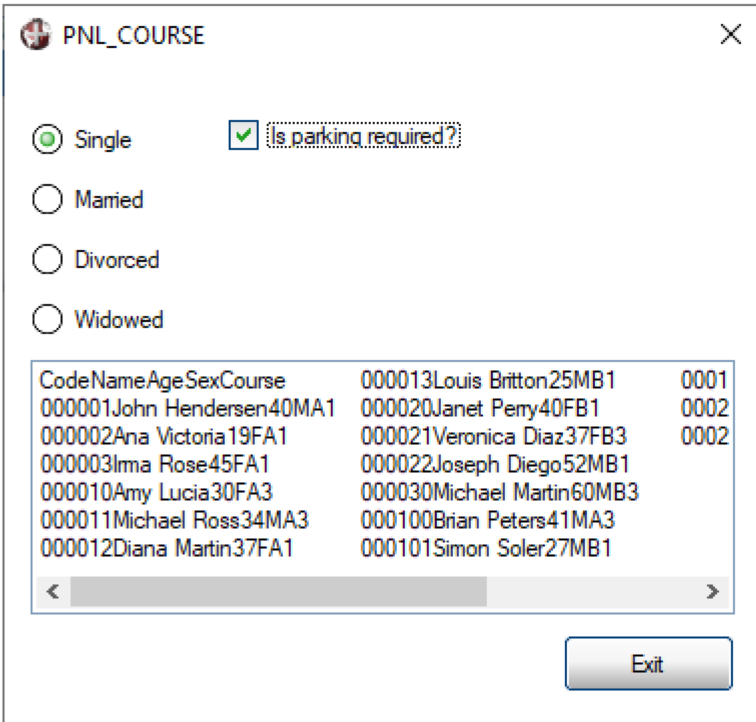
In the **List Values** box, enter some sample information so that we are able to perform the exercise. To save some time, use the mouse to select the list provided below. Right click inside the highlighted list and select Copy. Then, in the **List Box Properties** window, right click inside the **List Values** box (on the **Display** tab) and select Paste.

Code	Name	Age	Sex	Course
000001	John Hendersen	40	M	A1
000002	Ana Victoria	19	F	A1
000003	Irma Rose	45	F	A1
000010	Amy Lucia	30	F	A3
000011	Michael Ross	34	M	A3
000012	Diana Martin	37	F	A1
000013	Louis Britton	25	M	B1
000020	Janet Perry	40	F	B1
000021	Veronica Diaz	37	F	B3
000022	Joseph Diego	52	M	B1
000030	Michael Martin	60	M	B3
000100	Brian Peters	41	M	A3
000101	Simon Soler	27	M	B1
000102	Diana Ruiz	36	F	B3
000201	Fiona Lopez	47	F	C1
000202	Jordan James	28	M	C1

The modified panel looks like this (remember to save before testing):



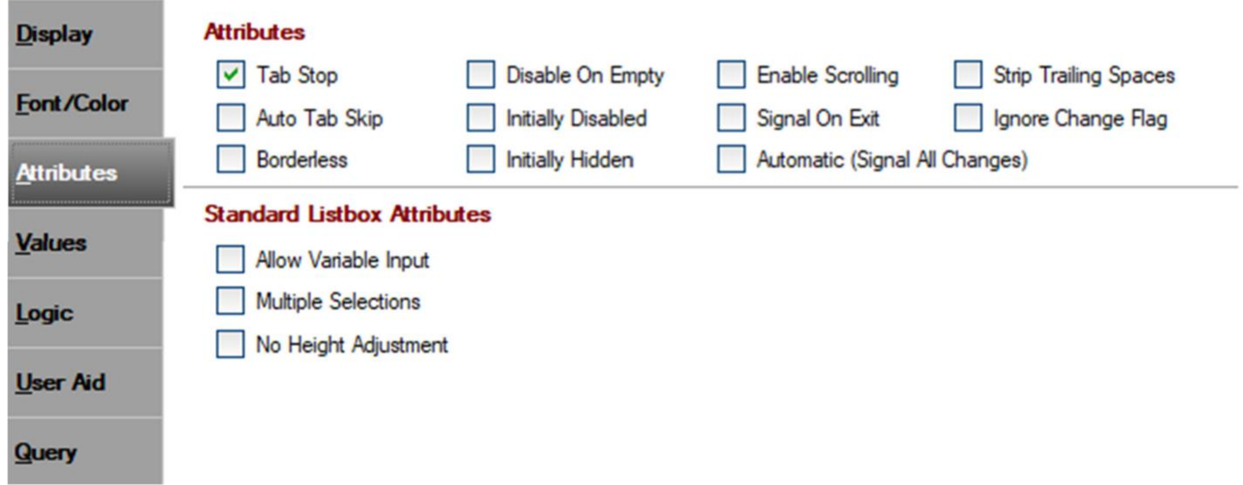
In the **List Box Properties** window, if you change [**List Box Type**] to **List View**, it looks like this:



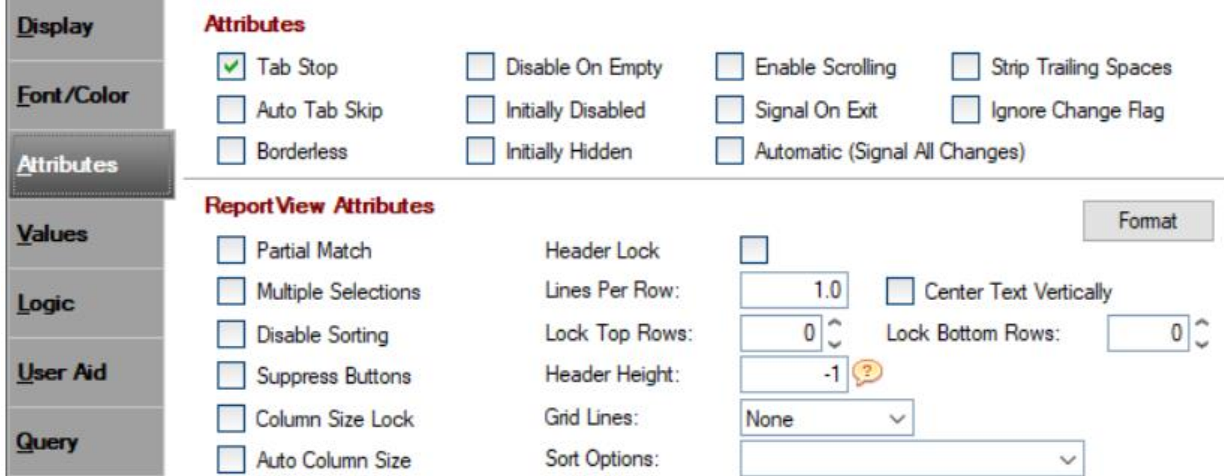
Let's now look at the properties of the **LIST_BOX** control in more detail. Note that depending on the selected control sub-type, some functions or attributes could change or simply not be displayed. Research, practice and experimentation will be your best allies in this learning process.

One of the functionalities of List Boxes is that some allow the entry of information. To do so, they must look to see if it has the [**Variable Input**] attribute.

Let's look at the attributes of a **Standard** List Box:



These are the attributes of a **Report View** List Box:



Some of these lists allow you to define a format or division of the data, to have a columnar presentation, and even simulate a grid. In other cases, it is possible to define titles or headings, as well as sort by any column in the list. It is also possible to assign conversion or equivalence values.

In a **List Box** control, there are three events: **Post Create**, **When Receiving Focus** and **When Entry is Selected from List Box**.

You don't have to be very inquisitive to realize that preloading the data in the control's properties window is not very efficient. To do this, it is suggested to create a routine (program) that is executed after creating the control and that copies the content from a table to the list.

The routine could be similar to:

```
01  begin:
02  channel=unt
03  open(channel)"table"
04  data_read_cycle:
05  read(channel,end=finish_read_cycle)data1$,data2$
06  list_box load list.ctl,0,data1$+data2$
07  goto data_read_cycle
08  finish_read_cycle:
09  close(channel)
10  exit
```

Note: We have added the line numbers to make it easier to study the routine. There is no need to add them when you are doing your programs.

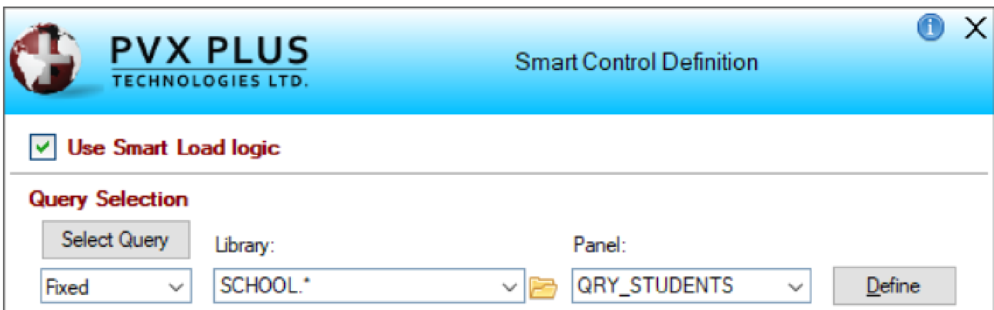
Let's analyze each line separately:

01	Start label, not mandatory.
02	The next free channel (variable unt) is located and assigned to the variable channel.
03	The table "table" is opened and the logical channel (channel) is assigned to it.
04	Label to mark the table reading cycle.
05	The table is read, and in case of end (clause, end=), it branches towards the label finish_read_cycle. The read values are assigned to the variables data1$ and data2$.
06	The list_box load command allows you to load the variables data1$ and data2$ to a list that has the name list, so its object identifier is list.ctl.
07	Return the program flow to the data_read_cycle label.
08	Label to mark the end of the file read cycle. When the command on line 05 finds that the table is over (no more information), it will send the program control here.
09	Close the open channel (channel).
10	End the program and return to NOMADS.

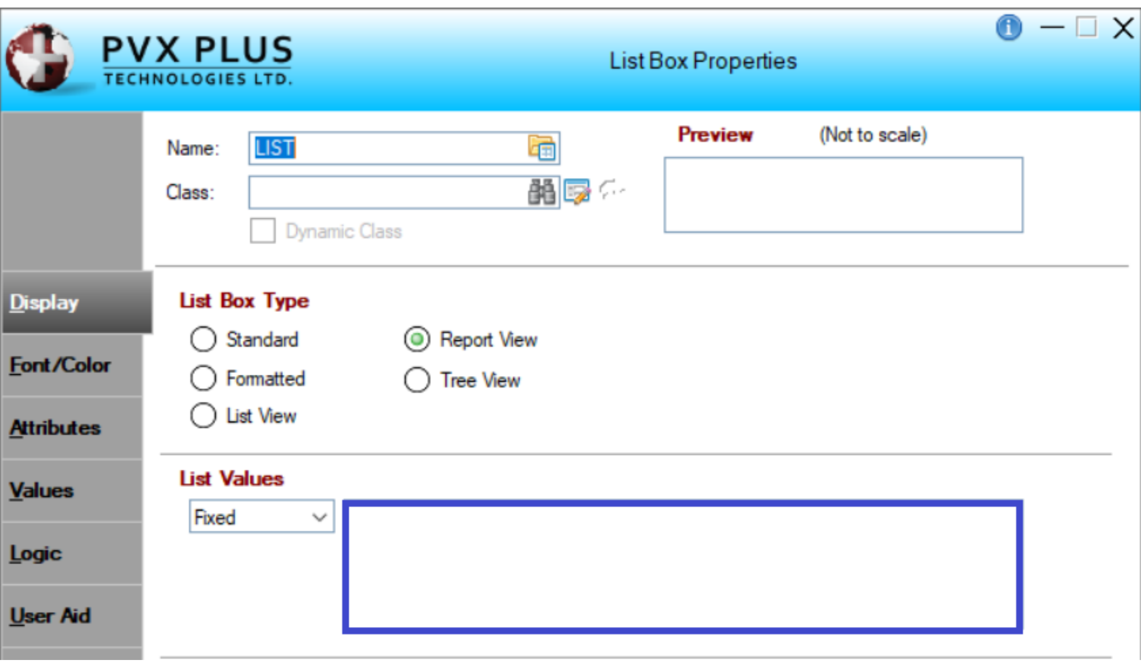
If you are wondering whether it is possible to leave this task of uploading the list to NOMADS, the answer is Yes - this is because of the **Smart Load** feature:

How does Smart Load work?

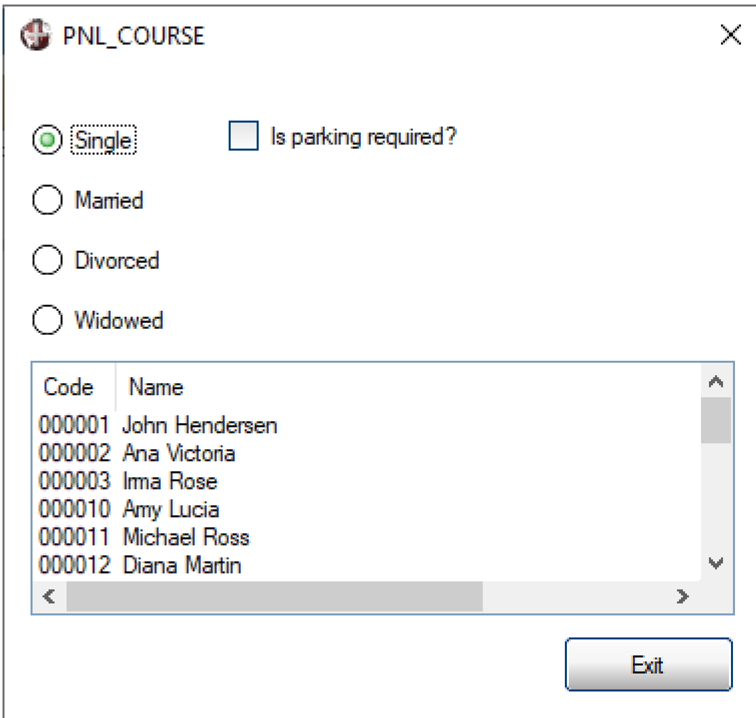
A query must be defined, and then the query is associated with the control (**LIST_BOX** in this case). It is as simple as doing this:



We are using the query defined previously and also eliminated the values we entered manually in the definition of the List Box control:



We save and test the panel, which looks similar to the one shown below:



We see the data from the **QRY_STUDENTS** query defined using the data from the **STUDENTS** table.

Refer to **Smart Controls** in the PxPlus Help documentation.

Understanding the Use of .CTL Variables in NOMADS

This is now a good time to deepen the use of **C**on**T**ro**L** variables in NOMADS. These variables allow you to identify all the controls that are defined in a panel. Basically, the **CTL** variable is composed of the name of the control plus the .CTL prefix. For example, if the control is called CODE, the CTL variable for that control will be CODE.CTL. (The content of this variable is numeric, although we do not need to know its value, just the name itself.)

What are CTL variables actually used for? There are basically two uses for these variables:

1. To reference a control in NOMADS. This allows us to control certain aspects of it.

   Example:

   ```
   disable CODE.CTL control
   ```

   It will be used to disable the CODE control. It also serves to force the processing flow:

   ```
   next_id=CODE.CTL
   ```

   Some commands require the identification of the control to perform operations, such as:

   ```
   list_box load LIST.CTL,0,data$
   ```

2. To reference a control ***in the form of an object***. That is, the CTL variable will be the object identifier, and therefore, you will be able to access its methods and properties.

   Example:

   To change the background color of a control called CODE, we could do:

   ```
   code.ctl'backcolour$="Light blue"
   ```

   We can also change properties, such as the text of a button called BUT_EXIT:

   ```
   but_exit.ctl'text$="Exit"
   ```

Tip: Confused? Refer to the next point about dynamic properties of objects and this will clear up your doubts.

Dynamic Object Properties

For PxPlus and for many other environments compatible with OOP (Object Oriented Programming), an object is a unique entity with certain functions and characteristics, which can be manipulated in some way.

Tip: You were expecting a class on abstraction, inheritance, polymorphism and encapsulation, with inherited instances and methods, with replication or inheritance of methods and properties? No, this is one step at a time!

For PxPlus and its tools (including NOMADS), many of the controls that are defined in the panels are objects; that is, they have a series of attributes and functions, and it is possible to modify and interact with them. In addition, if you copy an object, the destination object will have (inherit) all the properties and attributes of the previous one.

Summarizing, we can say that each of the elements of a panel, for NOMADS, is an object. Objects need a unique number or identifier (known as Object ID). In NOMADS, that ID is represented by the CTL variable of an object.

To refer to a property of an object, we have a general syntax:

objectID'property

The properties of the objects are of two types: alphanumeric and numeric. To differentiate them, we use the same syntax as for the variables in PxPlus: if they end in **$** (*dollar sign*), they are alphanumeric; if not, they are numeric.

Therefore, a generalized way of referring to a property would be:

ObjectID'property: Numeric property, such as the width of a button

ObjectID'property$: Alphanumeric property, such as button text

We also know that a control's ObjectID is its name plus the .CTL suffix or extension, so we'll change the references above:

objectname.CTL'property: Numeric property (width of a button)

objectname.CTL'property$: Alphanumeric property (button text)

So, using the programmatic way of referring to a control, we can say that to change the value of some property of an object, we only need to know the name of the property and the name of the control.

Note: Not all controls share all properties. By their nature, some will have fewer properties than others.

Example:

Let's look at the properties of a Button:

ActiveBackColor, ActiveBorderColor, ActiveTextColor, BackColor, BitmapPosition, Border, BorderColor, BorderWidth, BringToTop, Col, Cols, CtlName, Cursor, DisableBackColor, DisableBorderColor, DisableTextColor, Enabled, Eom, Flat, Focus, FocusBackColor, FocusBorderColor, FocusTextColor, Font, Height, HoverBackColor, HoverBorderColor, HoverTextColor, hWnd, ImageCount, Key, Left, Line, Lines, LookAndFeel, MenuCtl, Moveable, Msg, ObjectID, OnFocusCtl, OnTipCtl, Parent, SignalOnly, Text, TextColor, Tip, Top, Underline, Visible, Width

As you can see, there are many!

Refer to **Control Object Properties** in the PxPlus Help documentation.

Let's ignore our confusion, knowing that there are so many properties, and let's focus on knowing their use.

Suppose we have a Button control called BUTTON_3. We can change the Background Color to Red:

```
Button_3.ctl'backcolor$="RED"
```

The change is immediate. We do not have to go to NOMADS, edit the panel, select the control, etc.

This opens up endless opportunities and provides a lot of flexibility to our system.

Tip: We should let PxPlus take care of as many details as possible, and only modify and try to control what is really important.

We can do several operations to change the property of an object in a single routine:

```
If condition=1 then button_3.ctl'backcolor$="red";button_3.ctl'textcolor$="white"
```

Some properties cannot be modified by the user. We must familiarize ourselves with how the control works to see the best way to use it. Furthermore, we will see that the correct use of dynamic properties is the only way to efficiently control some controls, mainly the grid.

It will be experience that helps us determine if it is better or more efficient to use a particular property. **Example:** To copy the elements of a list to a table, there are several methods. We will soon see that there are several alternative commands to do the same, some using dynamic properties and others using commands for direct control manipulation.

```
LIST_BOX LOAD LIST.CTL,""
List.ctl'value$=""
```

The first is a command; the second is the change or assignment to a property of an object.

A widely used property regarding the management of List-type controls is **ItemCount**. The correct syntax would be:

list.ctl'itemcount

We can make a routine to explore the elements of a List like this:

```
for i=1 to list.ctl'itemcount
    list_box find list.ctl,i,value$
    …
next i
```

We have several new instructions. Let's briefly review:

The **FOR** command is part of what is known as a loop and is actually several "commands" grouped as follows:

```
for variable=start to limit
   CYCLE
next variable
```

The idea of this is to take a variable from a start (often 1, but it can be another value) to a limit in the example, and the instructions that are between **FOR** and **NEXT** will be executed as many times as necessary. We have denoted those instructions as CYCLE.

Example: To count the numbers from 1 to 100, we can do:

```
for i=1 to 100
    print i," ",
next i
```

This set of instructions is known as a **FOR..NEXT** cycle. PxPlus will increase the value of the variable "i" from 1 to 100 (1 at a time), so the instruction:

```
for i=1 to list.ctl'itemcount
next i
```

… will execute the command:

```
list_box find list.ctl,i,value$
```

… substituting the value of "i" as many times as the value of the **ItemCount** property, from 1 to the maximum value, and that particular command will read the item number "i" from the List Box and assign it to the variable *value$*. The command:

```
list_box find list.ctl,44,value$
```

It will read element number 44 (1 being the first element in the list) and assign it to the variable *value$*.

Tip: Remember that, in PxPlus, we (often) have several ways to do the same job.

Example:

```
        c=0
cycle:
        c++
        list_box find list.ctl,c,value$
        if c<list.ctl'itemcount then goto cycle
```

Note: You are not sure what the c++ instruction does? It adds one (1) to the value of the variable "c"; that is, if "c" is "1", when "c++" is executed, the value will be "2", and so on.

Example:

An example of an equivalent command and dynamic property would be:

```
button disable button.ctl
button.ctl'enabled=0
```

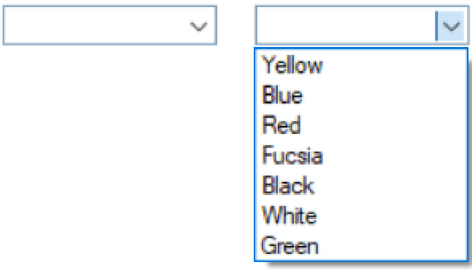
Both disable the BUTTON control; that is, it will be present in the panel, but it will not be active, and it cannot be pressed.

DROP_BOX Control: Drop-Down List Boxes

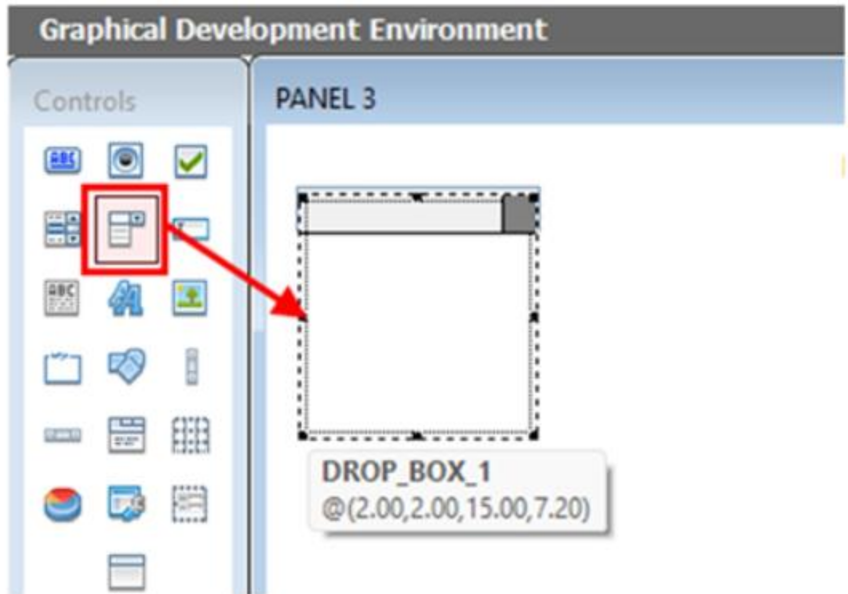
Knowing how the List Boxes work, the **DROP_BOX** control will be very easy to understand, since it is basically the same control but takes up less space on the screen and offers the possibility of being displayed or opened to show more information.

Example:

Let's see an example that shows the same control, closed (as it normally is) and open or displayed:



When we draw the control (selecting the corresponding icon in the **Controls** box on the left), we must leave enough vertical space so that when the **DROP_BOX** control is clicked, it has space to display; if not, it will seem that it does not work. Making it at least 4 or 5 lines high seems enough, but it will depend on your needs and the size of the panel:



Almost all of the considerations and properties of the **LIST_BOX** control are applicable to this control, although the command to control it is the **DROP_BOX** command, which coincidentally has the same syntax and accepts the same arguments as the **LIST_BOX** control.

Example:

```
! Read the item no 7 from DROPBOX and store it in element$
drop_box find list_drop.ctl,7,element$
!
!
num_elem=drop_box.ctl'itemcount
num_elem$=str(num_elem)
msgbox "This control has "+num_elem$+" items"
```

You can see that some lines start with **!** (*exclamation point*). These lines are ignored so you can use them to insert some comments in your program. These "comments" can be at the end of most commands: **command ! comment**.

We have done this on purpose to illustrate some things to you. As we know, some properties are numeric and others are alphanumeric. In PxPlus (as in many other languages), it is not allowed to mix numeric content with alphanumeric content. That is why we are initially using a variable NUMERIC (note that it does not end with "$") to store the number of elements (**NUM_ELEM**) of the **DROP_BOX** control. Then, we must convert from number to alphanumeric (also called "literal" or "string") using the **STR()** function. The content of the numeric **NUM_ELEM** variable is stored in **NUM_ELEM$** as a literal or text.

Note: **NUM_ELEM** and **NUM_ELEM$** are ***two different variables***.

The last statement simply displays the message with the number of elements contained in the **DROP_BOX** control. We can summarize these lines:

```
!
num_elem=drop_box.ctl'itemcount
num_elem$=str(num_elem)
msgbox "This control has "+num_elem$+" items"
```

We can use a more condensed form, concatenating the **STR()** function with the **ItemCount** property, all in one statement:

```
!
num_elem$=str(drop_box.ctl'itemcount)
msgbox "This control has "+num_elem$+" items"
```

Since many functions and commands allow concatenation, we could reduce the above even further:

```
!
msgbox "This control has "+str(drop_box.ctl'itemcount)+" items"
```

But, it is not recommended to try to "shorten" the programming by taking so many shortcuts that later make it difficult to understand and debug the programs.

Numeric and Text (Alphanumeric) Variables

A ***variable*** is a piece of info (data) with a unique name, able to store a single data. In PxPlus, the variable names start with a letter (not with a number), such as "A". The variable name can be longer than 32 characters. In practice, the programmers keep the variable name shorter. The variables are a temporary storage within a program.

Example:

Below are some examples of numeric variables.

```
A
AMOUNT
MY_LAST_BONUS
B1
Counter3
```

The content of the variable can be integer or with decimals, and it is possible to perform calculations and mathematical operations with them:

```
ADD=SALARY+BONUS
A=B+C
NAME1=BONUS*20.5
M=7500
D=T
RATE=VALUE^2+(SALARY-3)
```

In these examples, PxPlus evaluates the value of the expression to the right of the "**=**", and the result will be stored in the variable located to the left of the "**=**" sign. Let's study what each example does:

```
! Assign 2 to A variable, value 7 to B and 3 to C
A=2
B=7
C=3
! Add the value of A (2) to B (7) and store it on D (D = 9)
D=A+B
! The name of the variables is the same in upper and lowercase
SUM=C+A
! Multiplies (* asterisk sign) the value of C (3) by 20, that is, 60
! and assigns it to the variable BONUS
Bonus=C*20
! As there are operations inside parentheses, these must be performed first; therefore, C (3) is
subtracted from the value of B (7), resulting in 4, which will be multiplied (* asterisk sign) by 2, and the
resulting value, 8, is stored in the OPE variable
OPE=A*(B-C)
! Assigns the content of the variable "a.b" (which does not exist) to the variable D
D=a.b
! Square the number 200 (value of A, 2)
Rate=200^a
! Assigns the value 9.27342 (nine to several decimal places) to the variable TASA
TASA=9.27432
```

Basically, in PxPlus, any variable whose name ***does not end*** with the **$** (*dollar sign*) will be considered numeric.

There is a second type of variable that allows letters to be stored. These variables are called text, alphanumeric or "string" type variables (string or literal). To denote these variables, the **$** (*dollar sign*) must be added to the end of the name; for example, A$. The content of these variables must be enclosed in double quotes at assignment time:

```
A$="THIS IS "
B$="A TEST"
Name_student$="Ana Victoria"
Sex$="F"
Variable$=ANOTHER_VARIABLE$
```

You can concatenate (attach) the content of two or more variables with the "**+**" sign:

```
! Add the content of A$ (THIS IS) with B$ (A TEST) and store in C$ (the new content will be:
THIS IS A TEST)
!
C$=A$+B$
! D1$="A"
! D2$="B"
D$=D1$+D2$
! D$ will be "AB"
```

Note: ***It is not possible*** to "mix" numeric type variables with literal or alphanumeric type variables in an addition/concatenation operation:

```
A=3
C$="HOLA"
D=A+C$
```

This operation will result in an error (Error #26 - Type mismatch), although there are appropriate functions to convert numeric type variables to literal and vice versa:

```
a=num(var_literal$)
literal$=str(value_numeric)
```

The **NUM()** function allows you to convert the content of a variable to numeric so that it can be processed mathematically.

Note: The content must be numbers. It cannot have alphanumeric characters as part of its value.

```
! We'll get an error because PxPlus cannot convert NADA to a numeric value
A$="NADA"
a=num(a$)
! BALANCE is 8.3
VALUE$="8.3"
Balance=num(value$)
```

It is also possible to convert in reverse; that is, a variable or numeric value to a literal (to concatenate or paste it with other text):

! The literal variable tmp$ will now have the value 32 (but no longer has a numeric value; it is simply two characters or symbols, the "3" and the "2"):
value=32
tmp$=str(value)
\
! The numeric variable A is set to the same as the unknown numeric variable B and then stored in the alphanumeric variable NUMBER$. Note: The name does not indicate anything. What determines whether it is numeric or alphabetic/literal is the **$** (*dollar sign*) at the end. Because Number$ has a dollar sign at the end, it is an alphanumeric/ literal type variable. For example, if B is zero (0), the contents of Number$ will be "0".
A=B
Number$=str(A)

Some Literal Functions and Text Expressions or Substrings

An alphanumeric or literal type variable can be very large (long); that is, contain too many characters, say, more than 100 characters. PxPlus allows access to certain parts of an alphanumeric or literal variable (that part will be called subliteral or substring). We will do so by referring to the initial position and the length of what we want to process or display.

Example:

If we have a variable A$ with the following content:

```
a$="VENEZUELA"
```

Let's put a number in every letter:

```
123456789
VENEZUELA
```

We can find the length (number of characters) of that variable by using the **LEN()** function:

```
! The LITERAL variable A$:
a$="VENEZUELA"
! The NUMERIC variable A will value 9; that's the number of characters or symbols of the
variable A$
a=LEN(A$)
```

It is possible to have a literal variable without content. This content is called NULL and is represented with the "" (that is, two double quotes with nothing in between). Note that this is different from ZERO and is also different from " " (there is a space between the double quotes):

```
! The variable nothing$ does not have any characters, its content is NULL
Nothing$=""
! The L variable will have ZERO value
L=len(nothing$)
```

We can refer to a part of a literal or alphanumeric variable.

Example:

```
123456789
VENEZUELA
```

Suppose a$="VENEZUELA", and we want to extract the third character from it:

```
A$="VENEZUELA"
B$=A$(3,1)
! B$ Will have the char "N"
! We extract several characters: C$=A$(1,4)
! C$ Will have: VENE
! CANNOT overpass the max length: C$=A$(50,2)
```

Note: In the expression A$=B$(C,D), the value of C corresponds to the *nth* character (according to the value of C), and the value of D corresponds to how many characters starting from character C will be extracted:

```
Aa$="ABCDEFGHIJ"
! 1 char from position 1 = A
DATA$=AA$(1,1)
! 1 char from position 4 = D
DATA$=AA$(4,1)
! 2 chars from position 1 = AB
DATA$=AA$(1,2)
! 1 char from position 2 = B
DATA$=AA$(2,1)
! 3 chars from position 4 = DEF
DATA$=AA$(4,3)
```

Note: We can only specify one argument in the subliterals:

```
Aa$="ABCDEFGHIJ"
! The chars from position 5 = EFGHIJ
DATA$=AA$(5)
! The chars from position 9 = IJ
DATA$=AA$(9)
```

There are many other features that you will need to explore to get the most out of the language. New features and literal handling explanations will be added as we progress.

Managing Keyboard Input

The preferred method of entering information in a commercial, administrative or management system is undoubtedly the keyboard. Simple questions such as Yes/No or data entry or search parameters, names, quantities, etc. will be entered into the system with the keyboard.

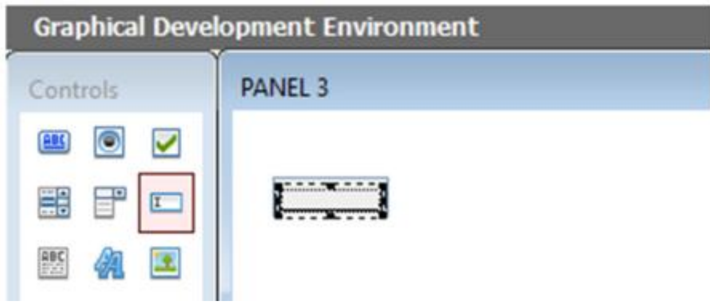
Some simple operations could become complicated if we do not make dimensions that allow us to "filter" the user's possible response. Let us remember that for a system, any data entered is just a sequence of symbols, without any type of content. Validation or acceptance of it will depend on us. The following answers are different for the system: "YES", "yes", "Yes", "Y", "y".

The same happens with dates. For us colloquially, 1/1/24 is January first of the year 2024. But for the system, these are all different dates or entries: "1/1/24", "01/01/24", "01/1/24", "01/01/2024", "1/1/ 2024" … you get the idea.

Many of the problems are avoided by using the appropriate type of control; for example, a **RADIO_BUTTON** or **DROP_BOX** to select the gender (you can only choose one of the preset options). We can also use a calendar to select the date or a drop-down box to select the city of birth.

But, in the end, we will be left with the need to enter information. For this, we have the most used control (that is why we preferred to leave a complete section for it) - a text entry box called a **MULTI_LINE**. There is also a command called **INPUT** that has a similar functionality but is used in non-graphical environments (text terminals or remote connections). We will look at this second command in other instances.

For now, we are going to look at the **MULTI_LINE** control.



Multi-Lines

The first peculiarity that this control has is that its name is very important. PxPlus will use a variable with that name to store what has been entered by the user.

Important Note: In NOMADS, initially there was a special variable for the VALUE or content of the control precisely called "**control_name.VAL**". For example, if the control was called **CODE**, the variable with the content (what was entered by the user) for the control would be **CODE.VAL$** (if the content is literal or alphanumeric) or **CODE.VAL** (if numeric content). This approach was soon changed to make use of only the control name, and a parameter was added to see whether or not to suppress the "**.VAL**".

Note: In this document, we assume that the "**Suppres .VAL**" attribute is turned On; that is, the "**.VAL**" suffix is not added to the control name.

Important Note: If the control is marked as numeric (**Numeric** attribute), the variable will be exactly the name of the control; otherwise, we must add a **$** (*dollar sign*) at the end of the control name to refer to its content.

The **MULTI_LINE** control has many properties. We will see the main ones, such as having a possible initial value. For example, if your client is located in Venezuela, you could save work by placing this value as the [**Initial Value**], or you could do it with the date process by reading the current date and setting it as the initial value.

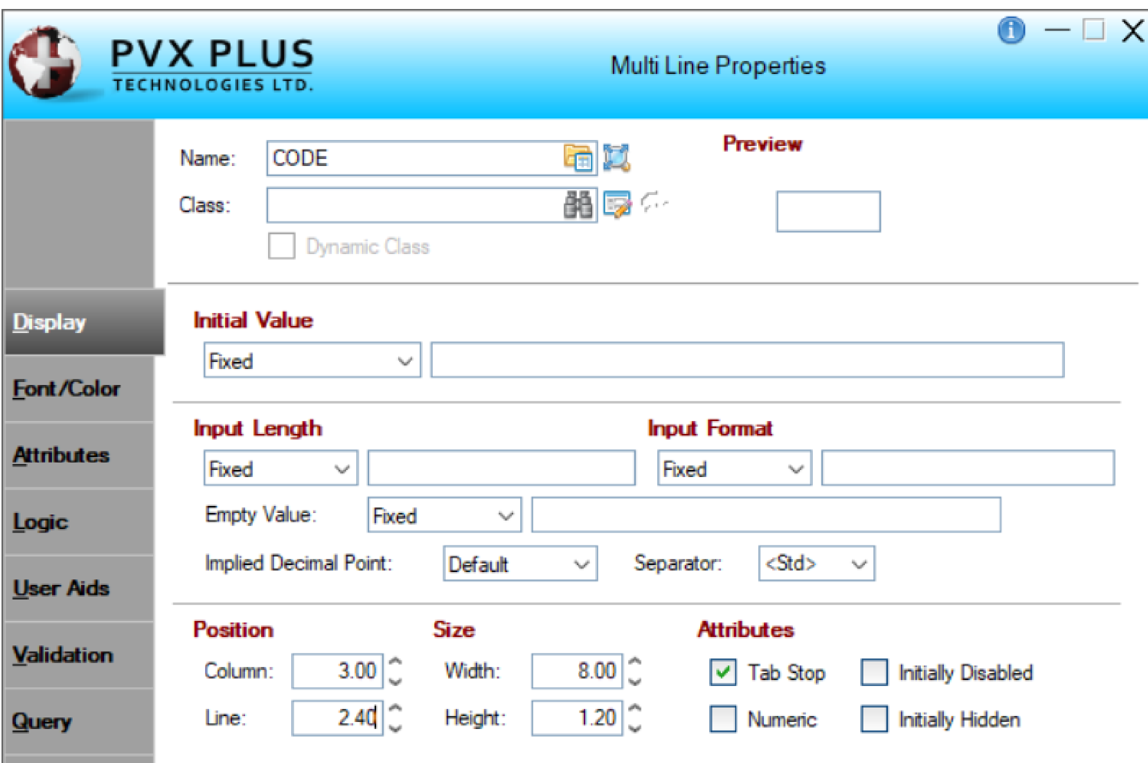
Another important property is the [**Input Length**] where you can establish how many maximum characters the user can enter for that field. For example, for a date, it could be 8 characters (2 for the day, 2 for the month, 4 for the year).

You can also set the [**Input Format**] according to the following table:

0	Only numeric digits, fill with 0s (zeros)
X	Any character
A	Any letter, uppercase
a	Any letter
#	Only numeric digits

Examples:

0000	Up to 4 digits, 29 looks like: "0029", looks "0000"
####0	Up to 5 digits, 29 looks like: " 29", 0 looks like: " 0"
XXXX	Up to 4 any characters
AAA	Only can input 3 letters, will be converted to uppercase
aaaaaaaaaa	Up to 10 letters



This filtering method is not widely used, but it will be up to you to determine its possibility of use and its applicability.

The other property we want to highlight is the [**Empty Value**], the value that will take control when it does not contain anything. For example, some audit guidelines require that if the user does not enter anything, they do not leave the field with a null value ("", which is two double quotes, indicating that there is nothing in the middle), but instead, they put some special word: "Null", "Blank", "Empty", etc. It can also be used to show if it is empty to indicate that something must be entered: "Enter code".

Other ***properties*** to highlight:

Locked: Allows text to be displayed without the user being able to modify it (effectively only functioning as an output control, not an input control).

Numeric: Only accepts input of numbers, not text/letters.

Password: Does not show what the user enters and replaces it with some character so that it is not seen; it is used to enter secret codes or keywords.

Borderless: Does not show the text box around it; that is, it seems that the text is entered directly into the panel, not in a box.

Reverse Input: Alters the order of entering characters, placing them backwards (the first ones are placed last).

Validation: Allows you to specify a program that will verify or change the entered content for further processing.

Formatter: Specifies a program to alter the appearance of what is entered; for example, if we enter 2129912312, it could change it to: (212) 991-2312.

Another important part of the **MULTI_LINE** control is related to ***events***. There are three possible events:

Post Create: (After creating the control) Allows us to do some checks and adjustments before allowing the user to enter something; for example, if the user level does not allow editing or modifying that field, it could be blocked and left alone reading.

When Receiving Focus: When the user is preparing to enter some data and the "cursor" is placed over the control, some action can be activated.

On Change: If the content (what was entered) is no longer null or changes (that is, when the user actually enters something), some action can be activated; for example, to see if what was entered is valid or has relation to what we are processing.

Exercise: Manual Creation of a CRUD Panel

Let's assume we have a **PRODUCTS** table composed of the following fields or elements:

Product Code (Length 5, would be our key)

Product Description (Length 50)

Origin of Product (I=Imported, N=National)

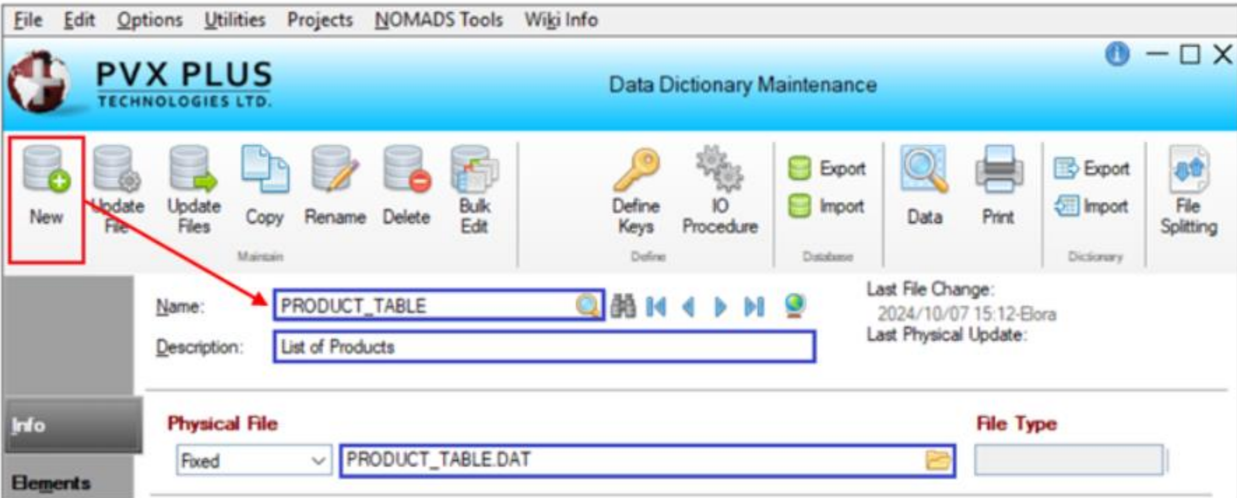
Although we know that PxPlus offers excellent tools for the **CRUD** process of a table (**C**reation, **R**ead, **U**pdate, **D**elete), we are going to perform this exercise completely manually.

We will define a table with its data dictionary, but the maintenance or data management part will be done manually.

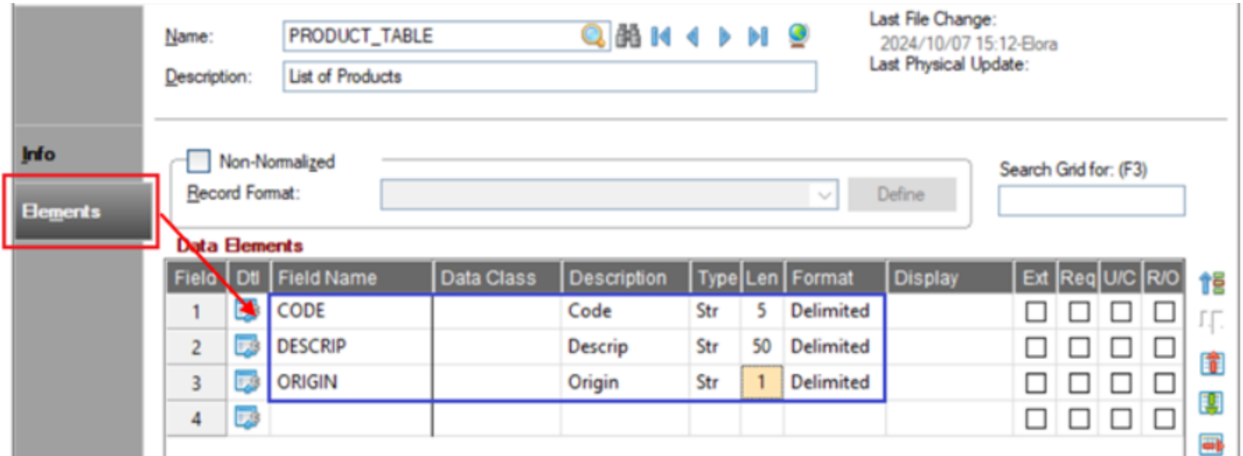
Note: We will still use the same library to keep everything in one place. In our case, the library is called **SCHOOL.EN**.

We will proceed to define our table. We will call it **PRODUCT_TABLE** and define the same name as the physical name plus the file extension ".DAT": **PRODUCT_TABLE.DAT**.

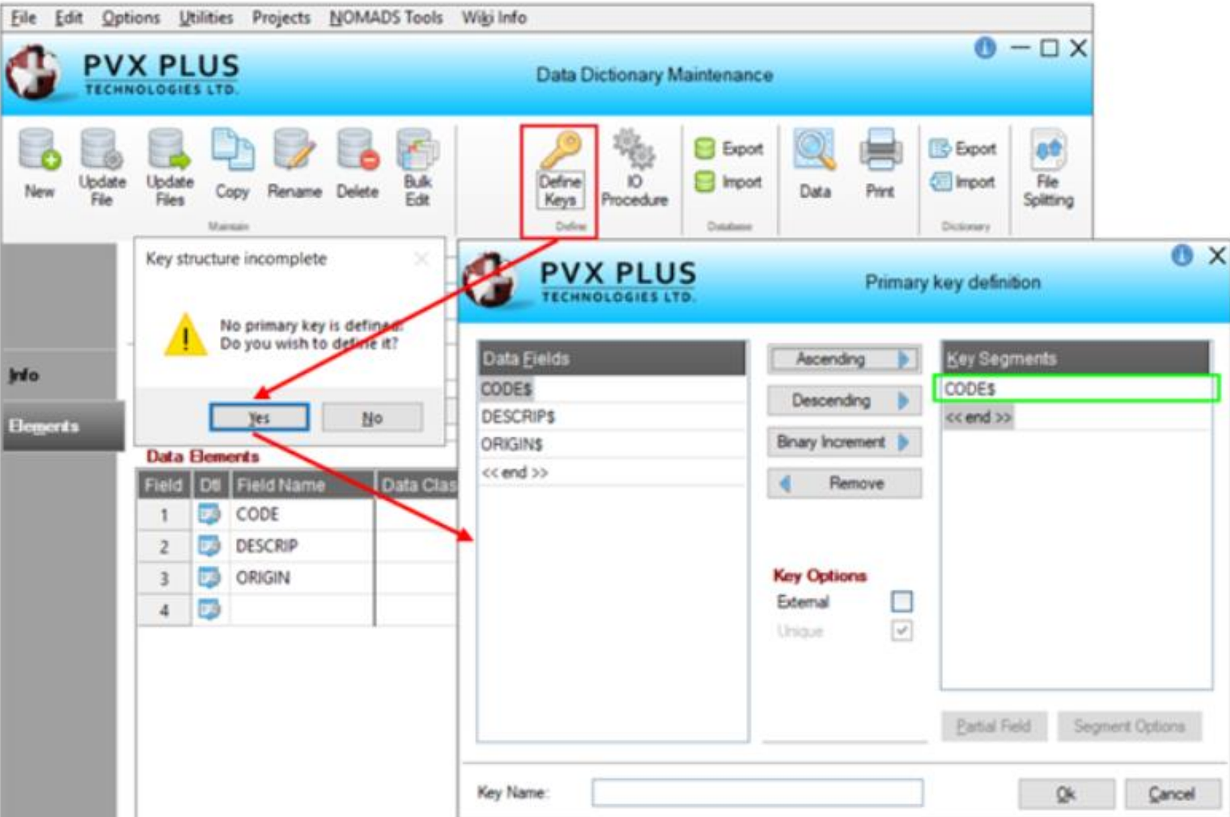
From the PxPlus IDE main menu, open the **Data Management** category and select **Data Dictionary Maintenance**:



Let's create fields or elements of the table:

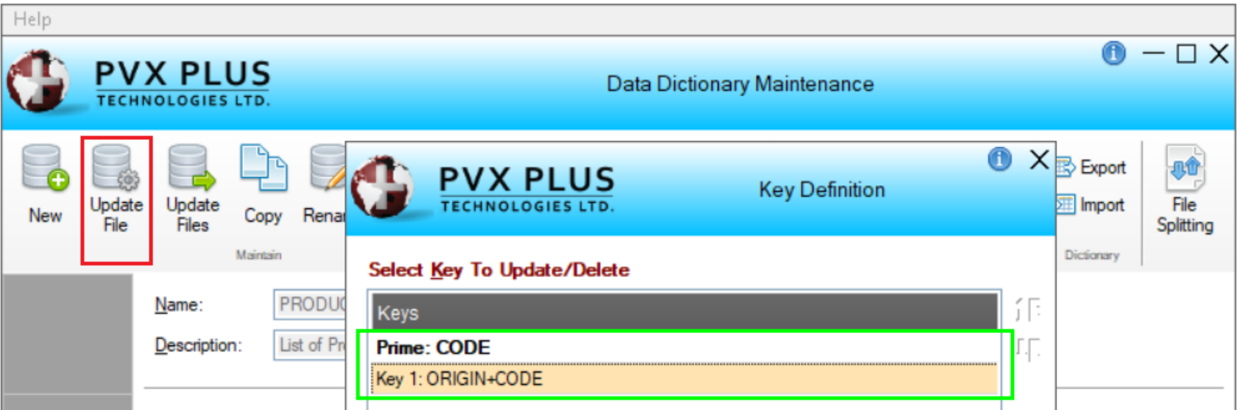


After the creation of the elements/fields, let's define the table's key (for sorting and access):



We will proceed to create a secondary (alternate) key to facilitate sorting by product origin.

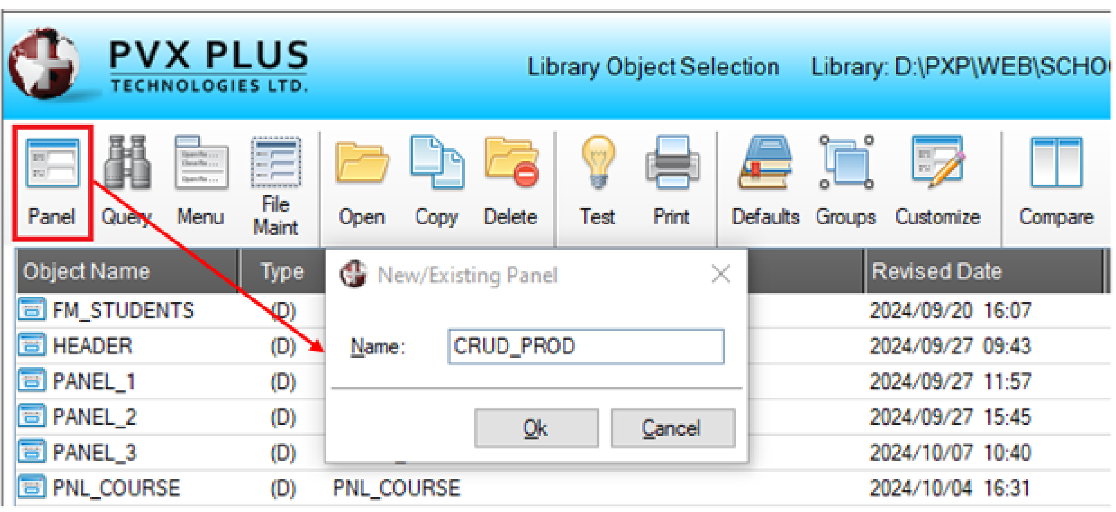
Note: The process of adding a new key is very simple and can be done at any time even if the file already has information and is in use. We will define the secondary key as ORIGIN+CODE so that, once sorted by origin, the second sorting criterion is the product code. Then, we proceed to physically create the table using the [**Update File**] option.



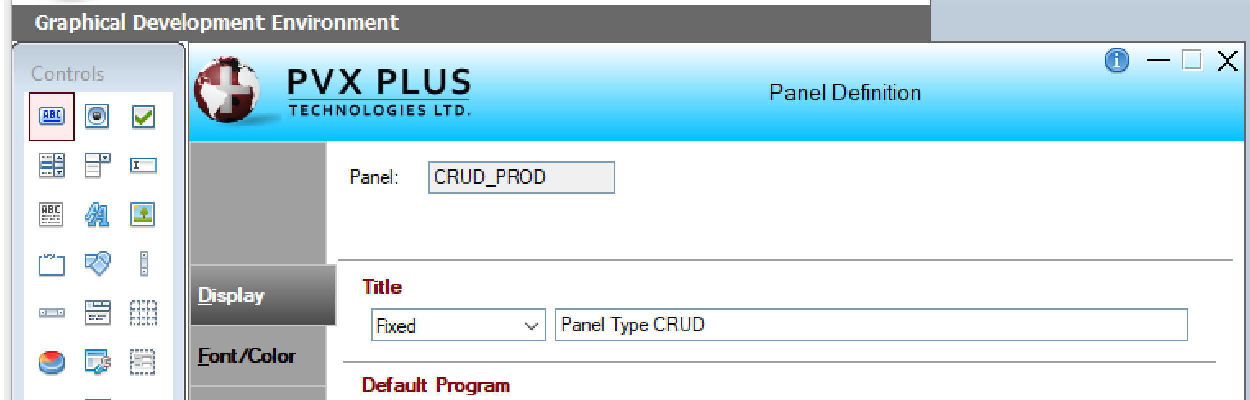
Note: Remember that we will be synthesizing processes already seen in order to advance faster. We will do it one step at a time without so many pauses (avoiding the use of detailed explanations in the already studied cases) to reach our destination in good shape!

Once the message displays to inform us that the table has been created, we will proceed to run the graphical designer to define our panel.

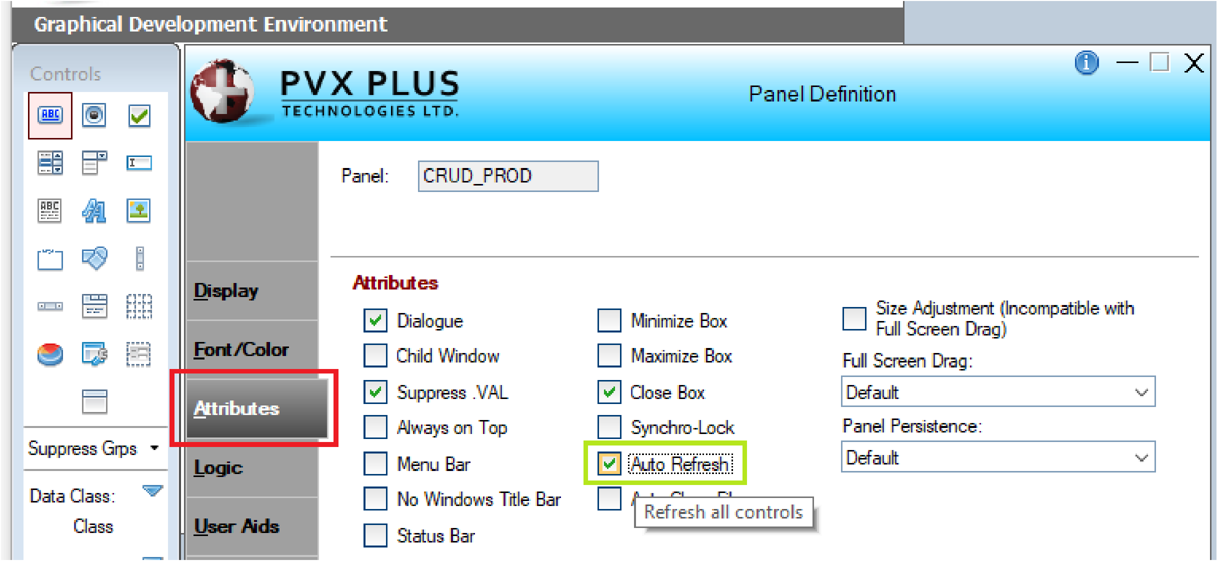
From the PxPlus IDE main menu, open the **Graphical Application Builder (NOMADS)** category and select **Open Application Library**. Select **SCHOOL.EN**.



Once the panel name has been entered, the NOMADS panel designer will open. We will go to the [**Header**] option to modify the panel itself to change the title.



We'll also select the [**Auto Refresh**] attribute:

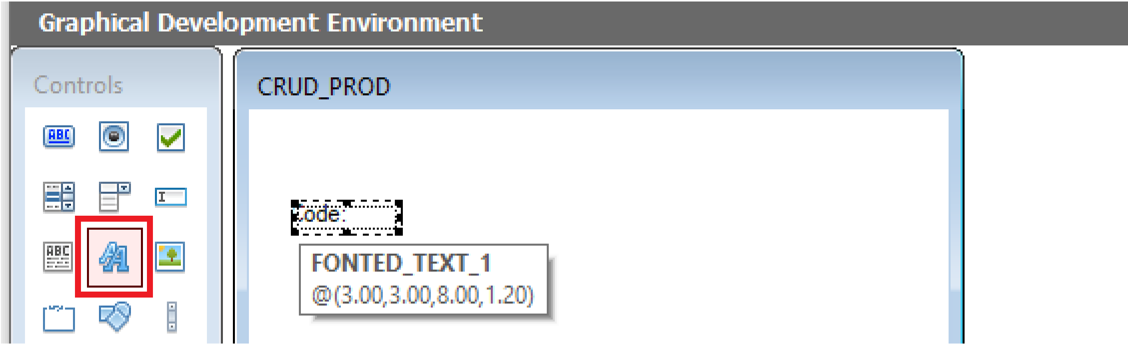


Important Note: We had said that there was a way to activate this attribute in case you want to turn it Off or On at your convenience. If the variable **REFRESH_FLG** is different from zero (for example, 1), the panel will be automatically refreshed.

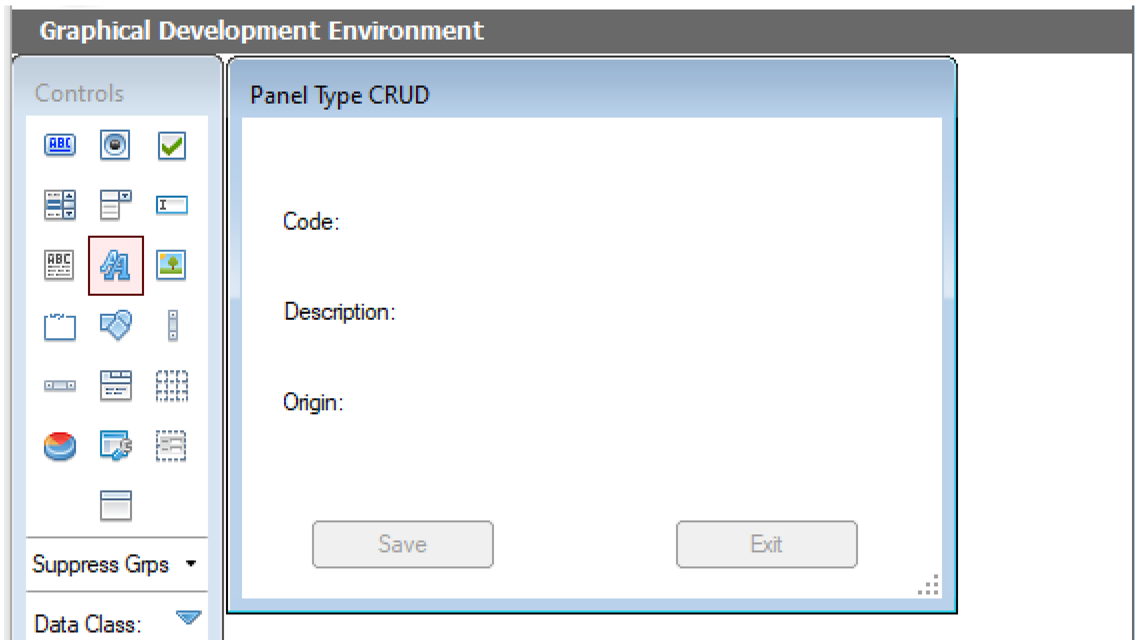
Let's save our panel (by clicking the diskette icon, by selecting the [**Panel**] -> [**Save**] menu options or by pressing the [**Ctrl S**] keys).

We will proceed to create several text labels (**Code:**, **Description:** and **Origin:**) to help document the loading of our information.

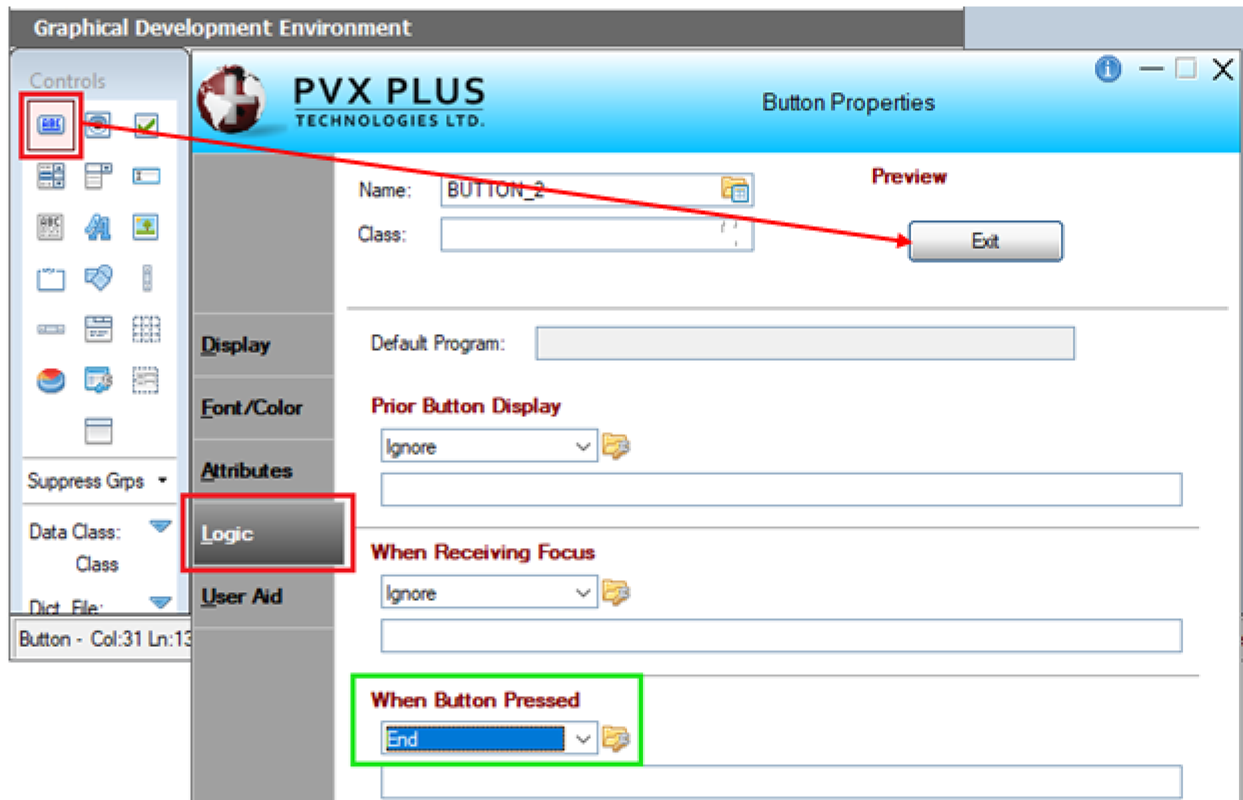
We will do them all with the **Fonted Text** control, using the type of default font and placing all in column 3 on lines 3, 6 and 9 respectively. You must take care of the size of the control so that it is not too large and overlaps other controls or too small and does not look complete. We use a width of 8, 10 and 8 respectively:



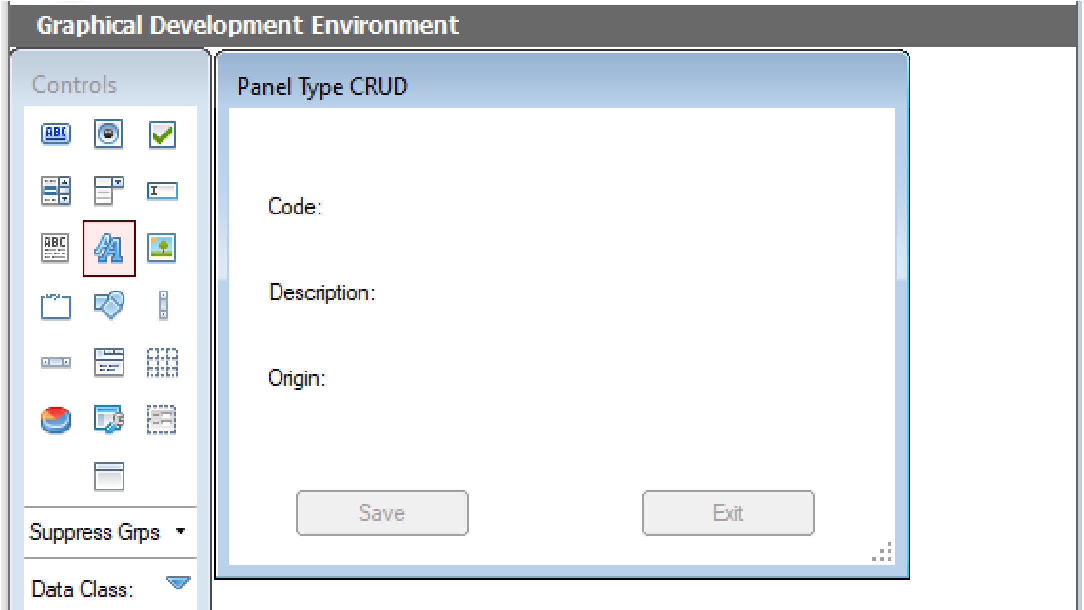
Our panel, with the text labels, will look similar to the one shown below:



Now, we are going to define two buttons, one for **Save** and another for **Exit**. We will select the logical action **End** for the **When Button Pressed** event.



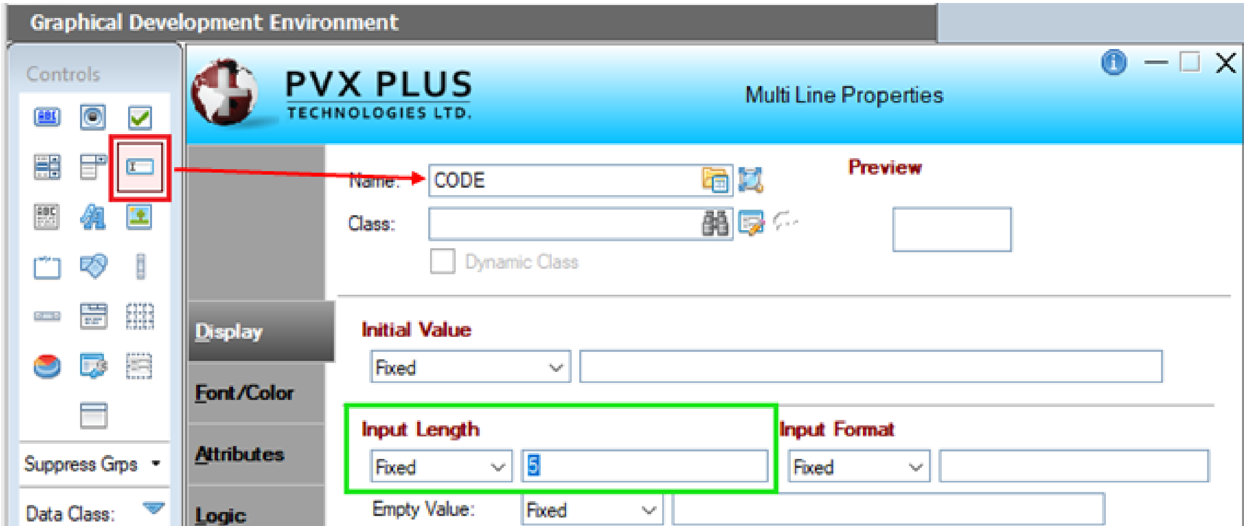
Let's save our work and check how it will look:



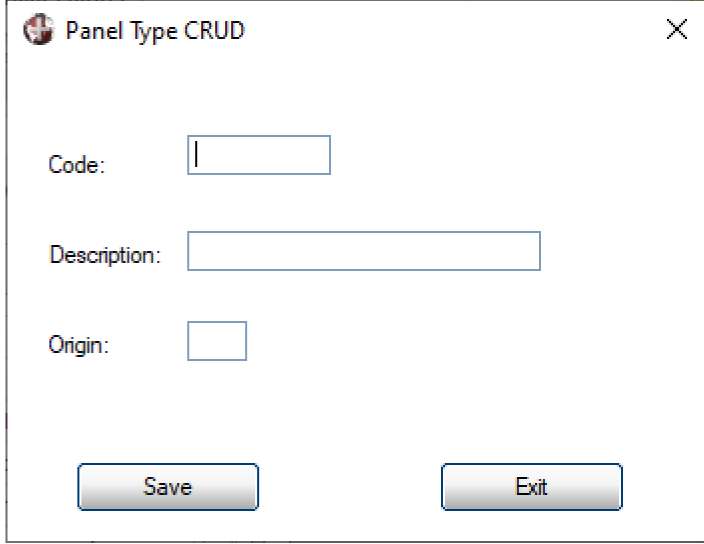
Note: Are you using advanced editing features (such as copying and pasting objects)? Do you have grid alignment enabled? Are you using the [**Align or Distribute**] function on the **Edit** menu? Remember that these functions are also available in the context menu by using the right mouse button. Remember that you can select multiple controls by using the [**Group Items**] tool located in the

Controls box or by clicking on the first control to select it and then selecting the others with the key and mouse combination [**Shift Click**].

Now, we are going to select the [**MULTI_LINE**] control in the **Controls** box and create three controls next to the respective text labels. Let's make the names of each control match the names of the fields (elements) of the PRODUCT_TABLE. We are going to enter the maximum [**Input Length**] of each one, namely: CODE (5), DESCRIPTION (50) and ORIGIN (1).



When finished, our panel should look similar to the one shown below:

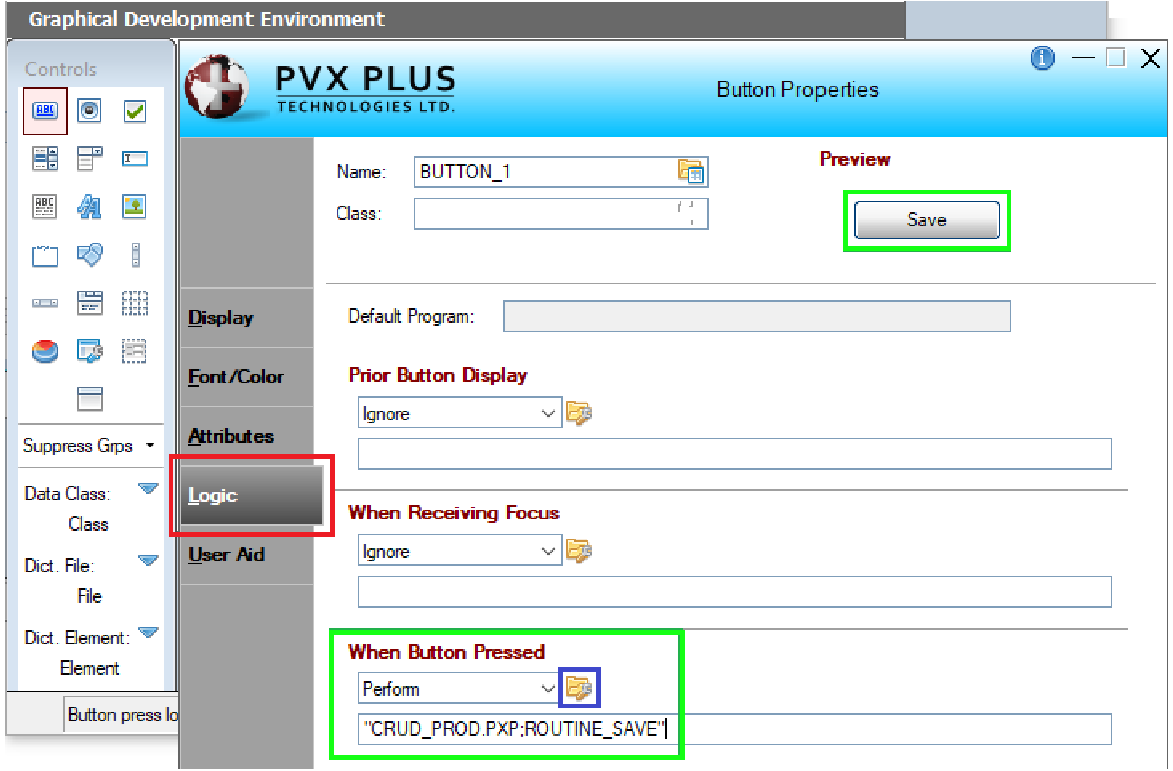


Note: Remember to save your work (by clicking the diskette icon, by selecting [**Panel**] -> [**Save**] options or by pressing the [**Ctrl S**] keys).

We have logic associated with the **Exit** button specifically for the **When Button Pressed** event. This is the **End** function.

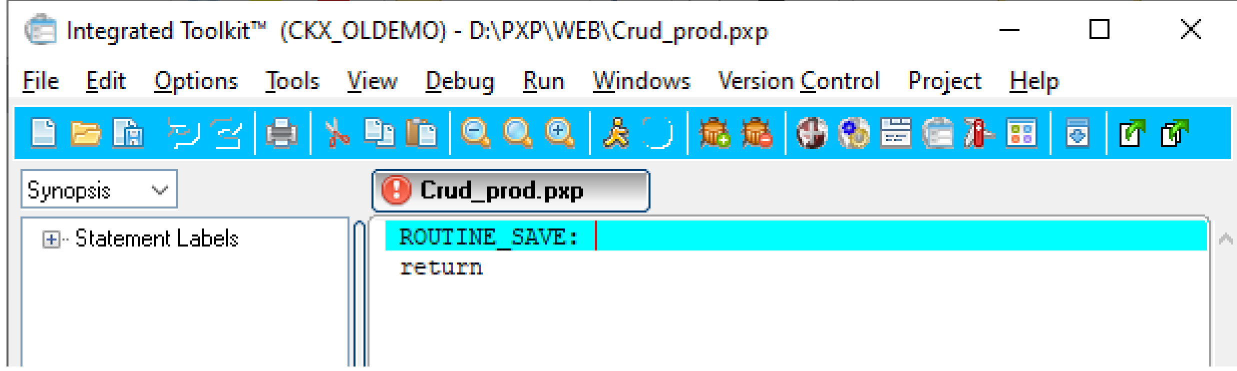
Now, we will associate a similar logic with the **Save** button, but we will associate a routine. To do this,

we open the properties of the **Save** button and select the **Logic** side tab. For the **When Button Pressed** event, select the **Perform** function, and in the corresponding input box, we enter **"CRUD_PROD.PXP;ROUTINE_SAVE"**. ***Let's not press anything else at the moment!***



Now, instead of selecting the [**OK**] button, we are going to select the yellow folder icon located to the right of the **Perform** action (marked in blue on the previous screen).

This opens an additional window with the program editor, placing the routine label in this example (ROUTINE_SAVE) and the **RETURN** command:



We will press [**Enter**] to insert a blank line and paste the following routine:

```
If code$="" then msgbox "The code is empty","Warning";next_id=code.ctl;exit
```

That instruction will apply if the CODE Multi-Line is blank. If so, it displays the message, places the focus on the control again and exits the program.

We will do the same with the other two Multi-Lines:

```
If description$="" then msgbox "The description is empty","Warning";next_id=description.ctl;exit
If origin$="" then msgbox "The origin is empty","Warning";next_id=origin.ctl;exit
```

Then, we open the table **PRODUCT_TABLE.DAT** and save the information:

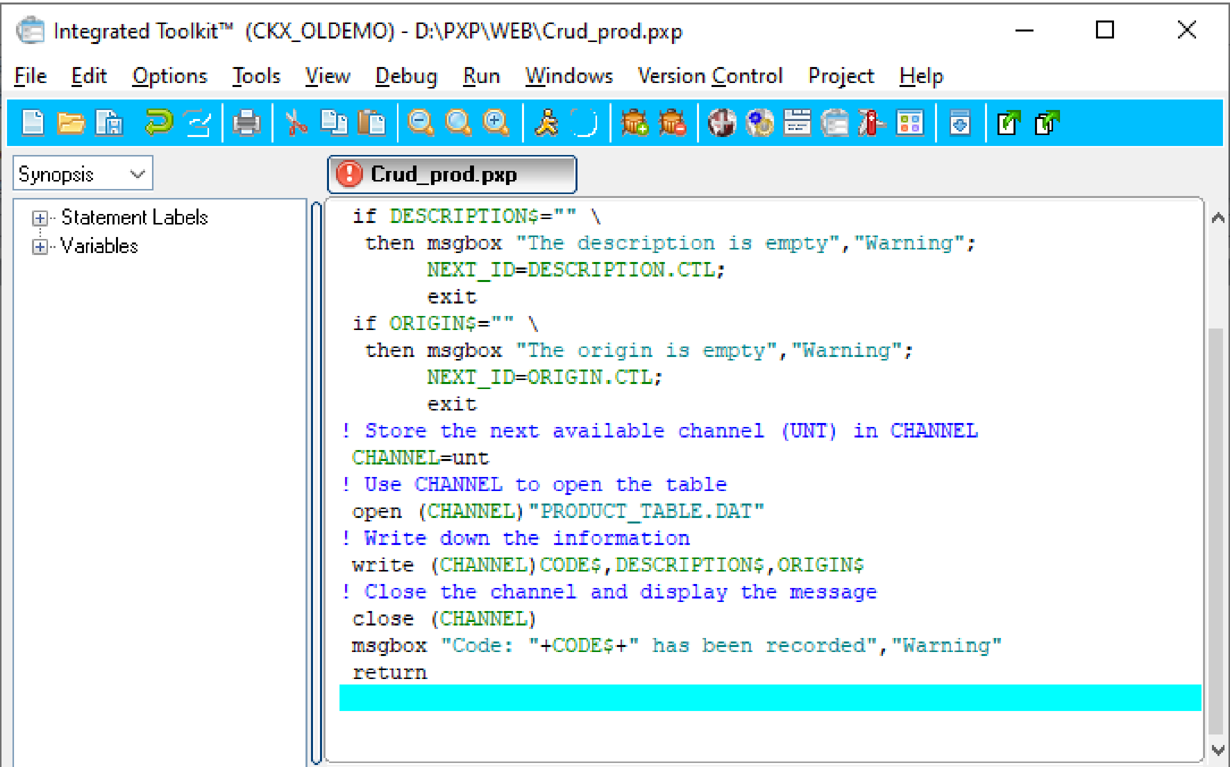
```
! Store the next available channel (UNT) in CHANNEL
channel=unt
! Use channel to open the table
open(channel)"PRODUCT_TABLE.DAT"
! Write the information
write(channel)code$,description$,origin$
! Close the channel and display the message
close (channel)
msgbox "Code: "+code$+" has been recorded", "Warning"
```

Note: Remember that an expression like "Code: "+CODE$ basically uses the text "Code: " and pastes it with the value of the variable CODE$ (in this case, associated with a MULTI_LINE control).

Let's look at the complete routine. (***Please do not paste it to the editor yet.*** We will do it only for teaching purposes. At the end of the example, we will post the complete routine.)

```
If code$="" then msgbox "The code is empty","Warning";next_id=code.ctl;exit
If description$="" then msgbox "The description is empty"," Warning";next_id=description.ctl;exit
If origin$="" then msgbox "The origin is empty"," Warning";next_id=origin.ctl;exit
! Store the next available channel (UNT) in CHANNEL
channel=unt
! open the table using channel
open(channel)"PRODUCT_TABLE.DAT"
! Write down the information
write(channel)code$,description$,origin$
! Close the file and report the recording
close (channel)
msgbox "Code: "+code$+" recorded", "Notice"
```

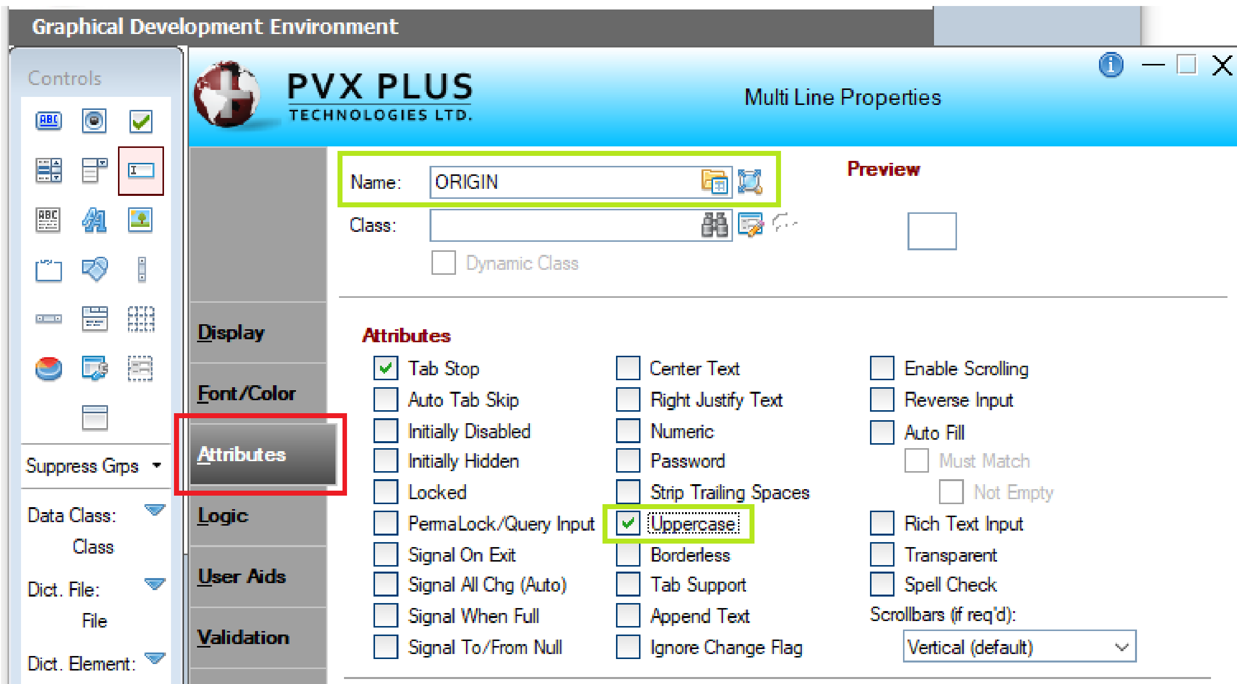
With this, the screen of our program editor will look like this (***do not paste it into your editor for now***):



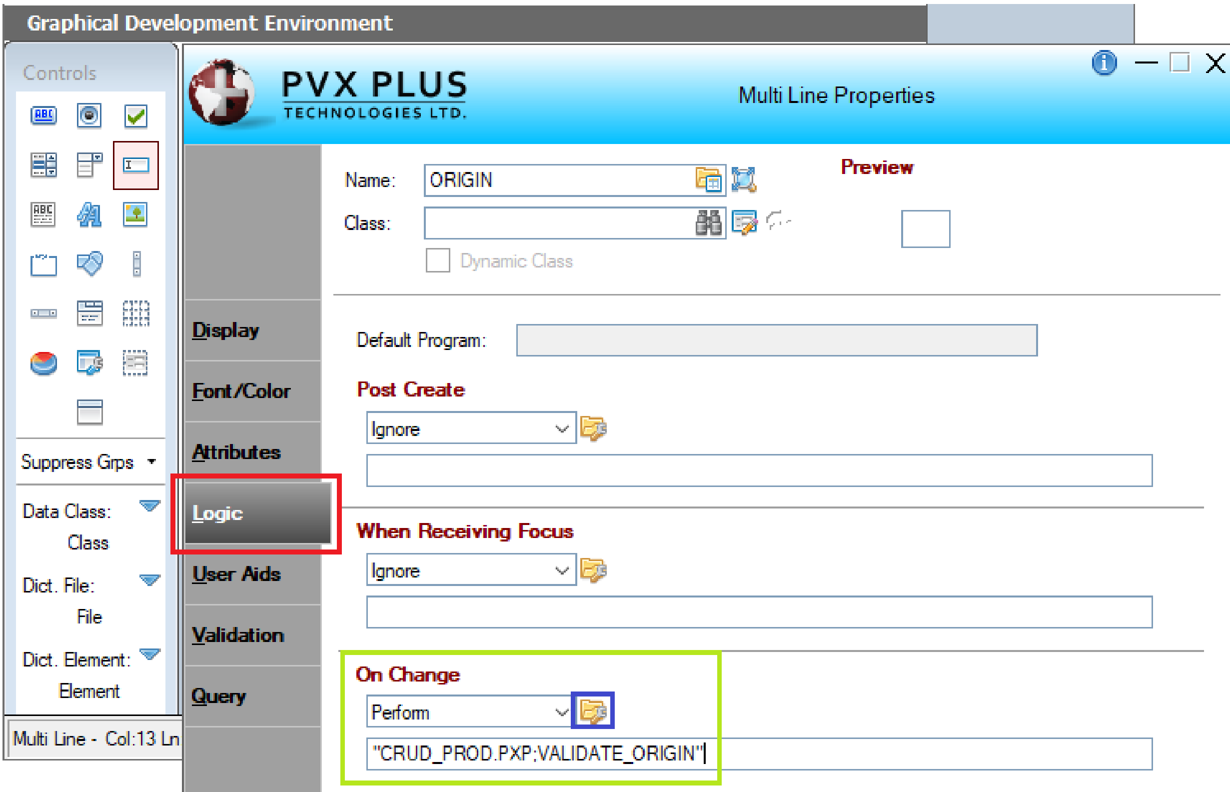
This program should be functional, but we quickly see that we can improve it with some things. For example, it is necessary to validate that the content of the ORIGIN Multi_Line will be only **I** or **N** (<I>imported or <N>ational) or something similar. We could also have placed a **Check Box** to show that if selected, it is Imported, and if not, it is National. For now, let's do that validation.

We will activate the properties of the ORIGIN control so that it converts to uppercase, and thus, we will avoid an additional verification.

Note: One of the biggest advantages of the PxPlus IDE is the possibility of having multiple windows active concurrently. Let's leave the program editor window open and review the properties of the ORIGIN control in the NOMADS panel designer.



Now, we are going to associate logic to this control so that when it changes (someone enters something), it will be verified if it is a letter **I** or **N**. Go to the **Logic** tab to the **On Change** event:



If we select the yellow folder icon (shown in blue), we will go to the program editor where we can add the following lines (***please do not do it yet***):

```
VALIDATE_ORIGIN:
if ORIGIN$<>"I" and ORIGIN$<>"N" \
  then msgbox "The Origin must be [I]mported or [N]ational";
        NEXT_ID=ORIGIN.CTL;
        exit
return
```

The **IF THEN** command with the format "IF COND1 and COND2" basically forces an action to be executed if both are true. In this case, if ORIGIN$ is different from "I" AND is different from "N" (that is, it is not "I" or "N"), it displays the message and returns the focus to the ORIGIN control. In short, it will not let it go from there as long as the content is not "I" or "N".

Once the recording has been done, it would be nice if the Multi-Line controls were cleared, and the focus returned to the CODE control.

```
! Clear the controls: ORIGIN$="";DESCRIPTION$="";CODE$=""
! Put the focus back to CODE
next_id=code.ctl
```

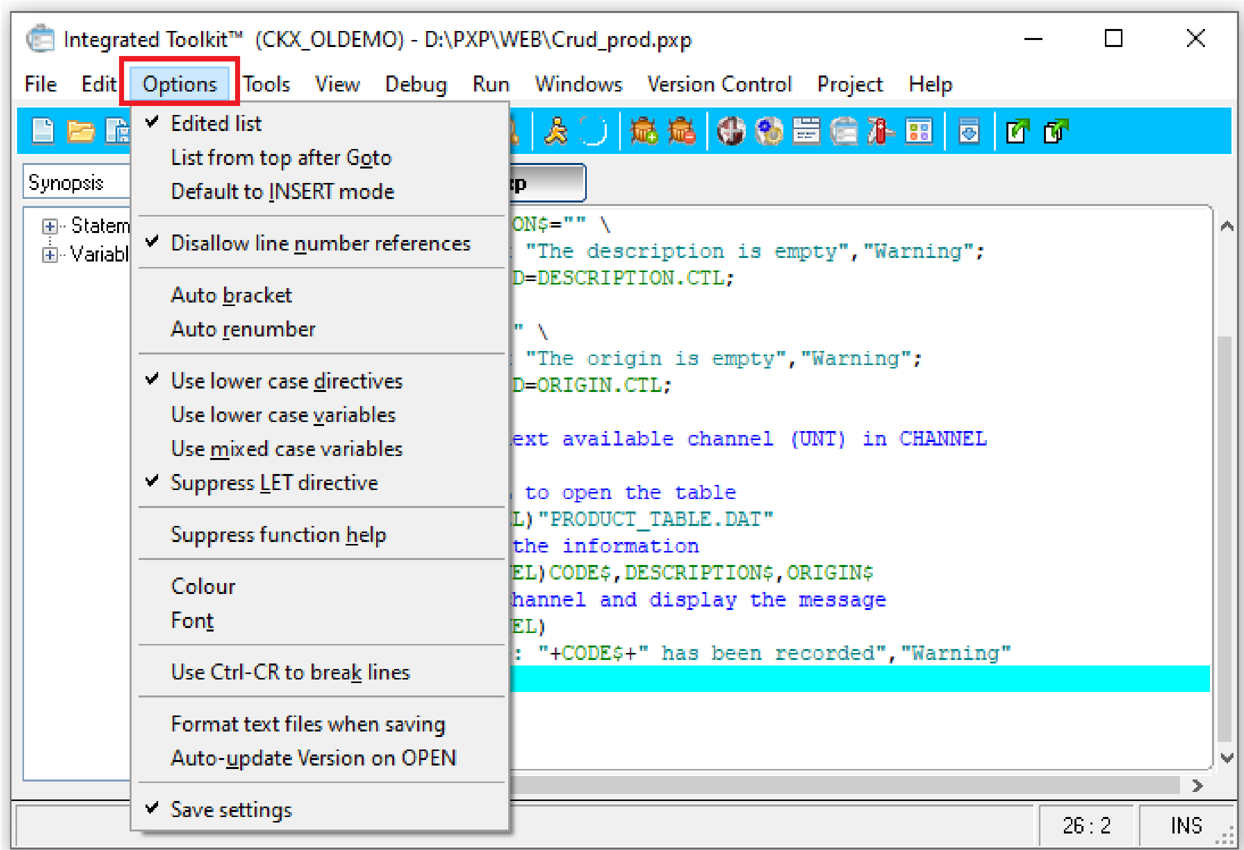
Now that we have the routine fully functional, let's look at the complete program (with some additional lines):

```
If code$="" then msgbox "The code is empty","Warning";next_id=code.ctl;exit
If description$="" then msgbox "The description is empty","Warning";next_id=description.ctl;exit
If origin$="" then msgbox "The origin is empty","Warning";next_id=origin.ctl;exit
! Use the variable UNT to assign the next free channel to CHANNEL
channel=unt
! Use CHANNEL to open the table
open(channel)"PRODUCT_TABLE.DAT"
! Write the information
write(channel) code$,description$,origin$
! Close channel and display the message
close (channel)
msgbox "Code: "+code$+" has been recorded", "Warning"
! Clean up the controls: ORIGIN$="";DESCRIPTION$="";CODE$=""
! Put focus on CODE
next_id=code.ctl
exit
VALIDATE_ORIGIN:
if ORIGIN$<>"I" and ORIGIN$<>"N" then msgbox "The Origin must be [I]mported or
[N]ational";NEXT_ID=ORIGIN.CTL;
exit
return
```

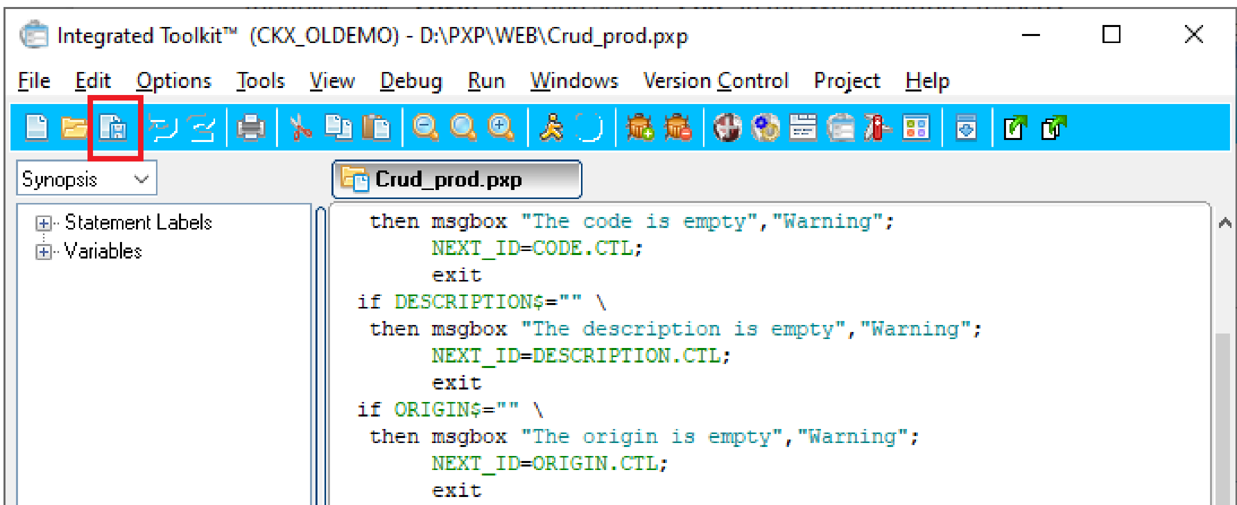
Note: Since you have the program editor open, you can delete the entire program and paste the previous routine. If you do not have it open, open it and check that the program is similar to the previous one.

Note: Remember that you can modify the appearance of the program editor. Your listing may not look exactly the same.

What options does the program editor have active? Below is a sample:



Save the program (by selecting the diskette icon marked in red on the next screen or by pressing the [**Ctrl S**] keys).



Run the panel (using the [**Test**] button in the **Library Object Selection** window) and test the program.

Your program is not working? Verify that the table **PRODUCT_TABLE.DAT** has been created and check that you have selected the [**Update File**] button. Did you specify the same name? Sometimes, we mistype the name; for example, **PRODUC_TABLE.DAT** instead of **PRODUCT_TABLE.DAT**. (This happens frequently.)

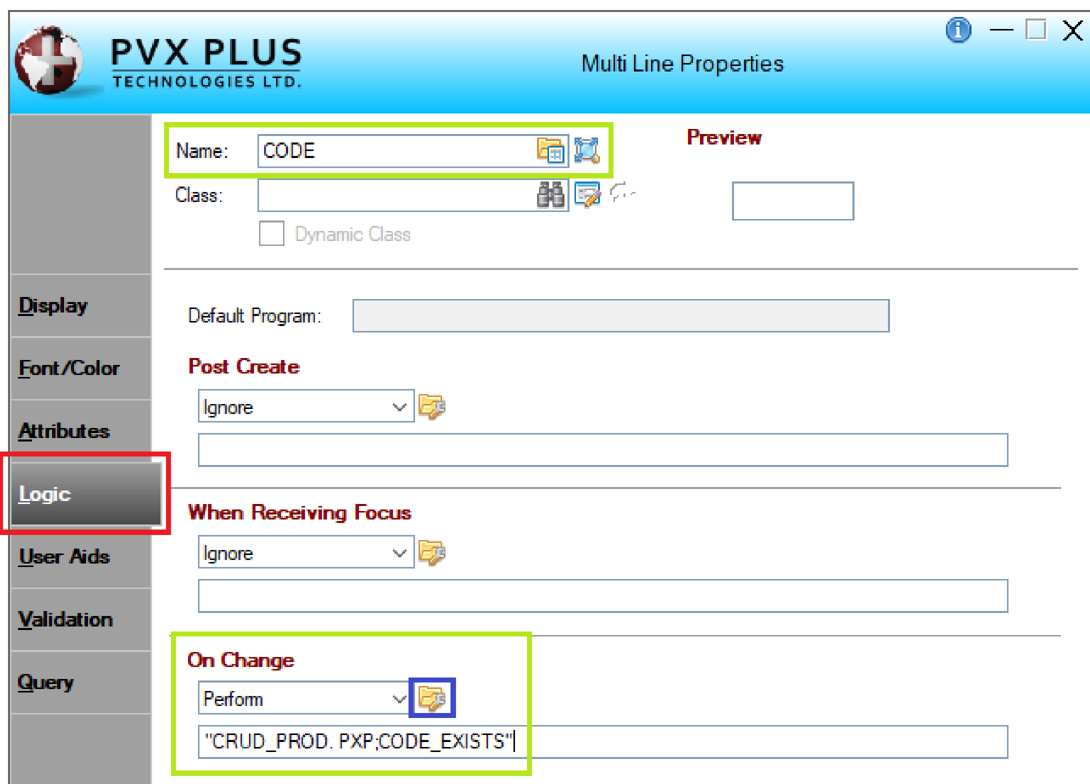
Is the [**Suppress .VAL**] attribute active? Is the [**Auto Refresh**] attribute activated? Do the **MULTI_LINE** controls (text entry boxes) have the correct names: CODE, DESCRIPTION and ORIGIN?

Did you check the syntax of your program? A line with an error is most likely in red.

To find information on many of these cases and how to correct them, review previous chapters in this book or use the search function of the PxPlus online Help.

Once our program is working, we can improve it very easily. When the user enters a code that exists, it should display it at once so that it is not replaced when we do it again. For this, we need to add a "code exists" check routine when it is entered. In the properties of the CODE control, in the **Logic** tab for the **On Change** event (when the value changes), we must select the action **Perform** and enter the routine: "**CRUD_PROD.PXP;CODE_EXISTS**".

The change should look like this:

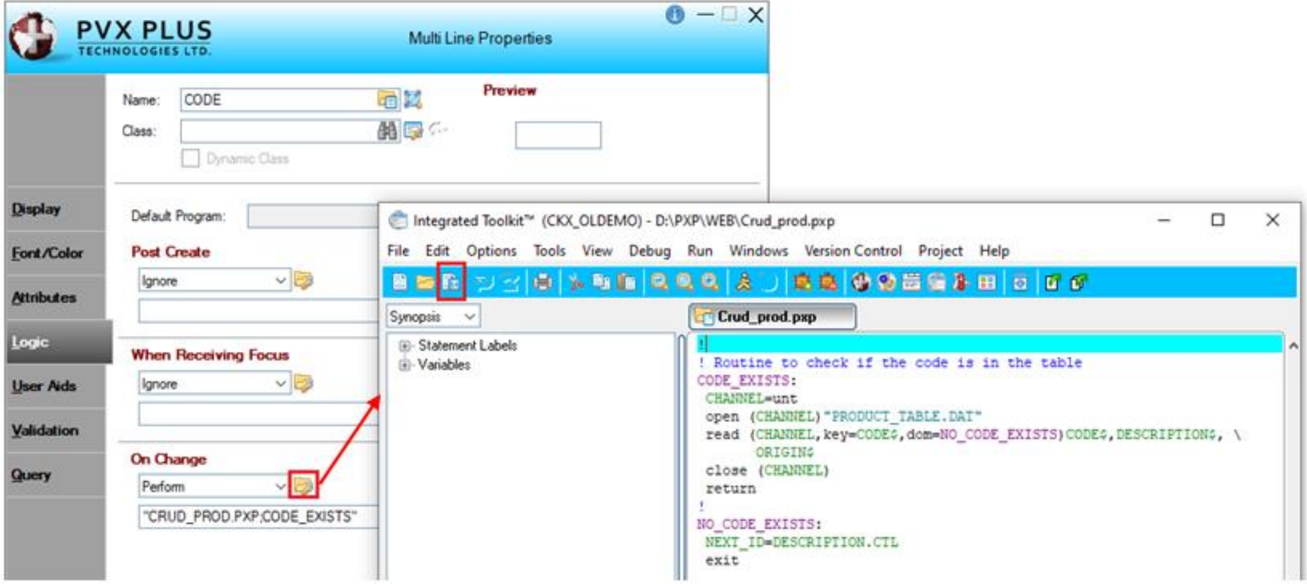


Now, select the yellow folder icon located to the right of the **Perform** action (marked in blue) to call the program editor window.

```
! Routine to check if the code is in the table
CODE_EXISTS:
    CHANNEL=unt
    open (CHANNEL)"PRODUCT_TABLE.DAT"
    read (CHANNEL,key=CODE$,dom=NO_CODE_EXISTS) CODE$,DESCRIPTION$,ORIGIN$
    close (CHANNEL)
    return
!
NO_CODE_EXISTS:
    NEXT_ID=DESCRIP.CTL
    exit
```

Remember the **,DOM=** clause in the **READ** command, which means "**Duplicate or Missing?**". It is used to trap the program flow in case a read with a specified key is not found.

Basically, the routine opens the table on the channel CHANNEL and reads it using the value of CODE$ as the key (KEY=CODE$). If a record with the specified key exists, as the [**Auto Refresh**] parameter is active, it automatically displays the values of the CODE$, DESCRIPTION and ORIGIN$ multi-line controls on the panel. If the specified key does not exist, the program branches to the routine NO_CODE_EXISTS, which simply puts focus on the DESCRIPTION control to continue entering data.

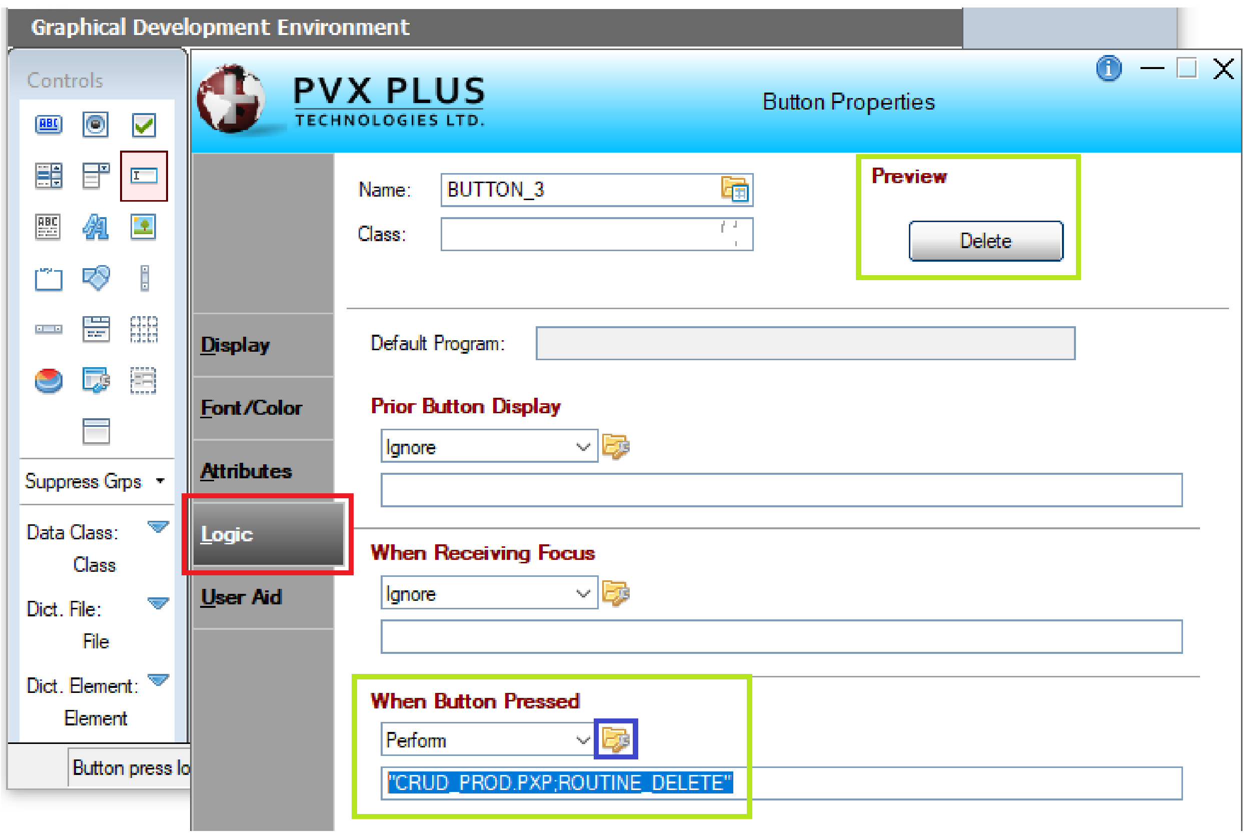


Save the program (using the diskette icon marked in red above or the [**Ctrl S**] keys). We must also save our panel and test it using the [**Test**] option.

If everything goes well, we should complete the **CRUD** (**C**reate, **R**ead, **U**pdate, **D**elete) process with the deletion of the record. We are going to modify the panel to place a **Delete** button in the middle of the other two. It will have associated logic to delete the record.

We can create the button in several ways. At this time, we are going to duplicate the **Save** button and change the text of the button and the logic so that for the **When Button Pressed** event, a routine called **ROUTINE_DELETE** is executed (**Perform**). Let's select the folder icon to the right of the **Perform** action (marked in blue) to call the program editor.

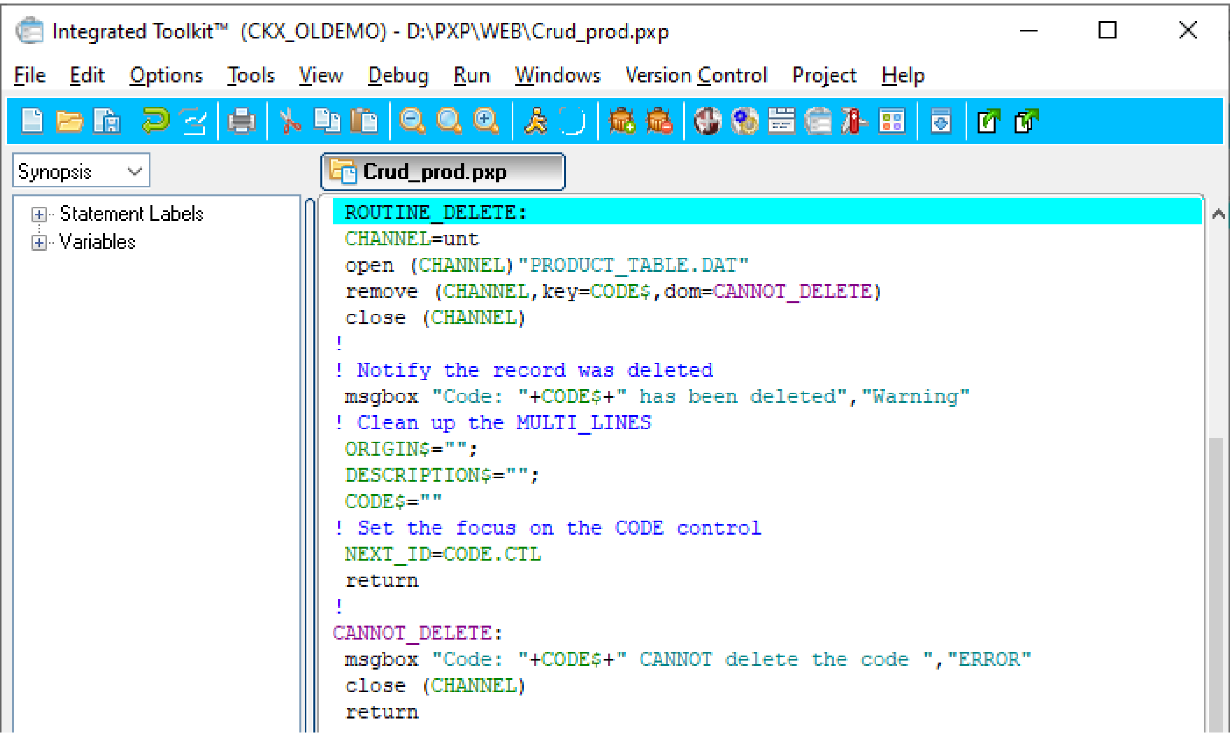
We will add a routine:




```
ROUTINE_DELETE:
    CHANNEL=unt
    open (CHANNEL)"PRODUCT_TABLE.DAT"
    remove (CHANNEL,key=CODE$,dom=CANNOT_DELETE)
    close (CHANNEL)
  !
  ! Notify the record was deleted
    msgbox "Code: "+CODE$+" has been deleted","Warning"
  ! Clean up the MULTI_LINES
    ORIGIN$="";
    DESCRIPTION$="";
    CODE$=""
  ! Set the focus on the CODE control
```

```
   NEXT_ID=CODE.CTL
   return
 !
 CANNOT_DELETE:
  msgbox "Code: "+CODE$+" CANNOT delete the code ","ERROR"
  close (CHANNEL)
  return
```

Let's see it in the program editor:



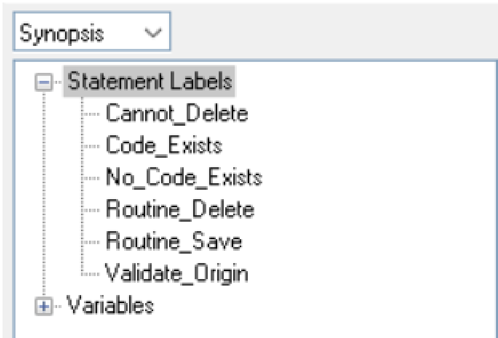
The routine looks for a free channel and assigns it to the variable CHANNEL, opens the table using it, and tries to remove the record. If it finds an error, it branches the flow to the routine CANNOT_DELETE where the message is and closes the CHANNEL.

If the record is deleted, the user is notified, CHANNEL is closed, all multi-lines (text entry boxes) are cleaned, and the focus is placed back on the CODE control.

Note: We can already see that there is a bad programming practice because we open and close channels every time we are going to perform an operation on the table. The ideal is to have a channel/table opening routine at the beginning and a closing routine at the end of the execution.

Let's proceed to save and test. Remember that you must save the program and the panel separately as they are two different entities.

Note that the program we have used, **CRUD_PROD.PXP**, is currently a collection of small routines; that is, you should not necessarily view a program file as a single entity that does "actions" for a PAYROLL system or INVENTORY. In PxPlus, it is possible to perform many tasks with routines from a single program. You could also have all the routines/programs of a project in a single entity; PxPlus does not put any limitations in this regard.



This brings with it the possibility of reusing routines for multiple projects, panels or systems. If you can make a routine sufficiently flexible and versatile, it could save you many hours of work.

Using Folders on Panels

A ***folder*** or tab type control is actually a control "container" of panels; that is, the folder control contains two or more panels that will be accessible from each of its tabs. The idea is to bring the user several panels that will be available through special buttons that simulate tabs or folders.

To create a folder control, you must have a panel with a control that is large enough to contain all the panels that will be part of it, and it is suggested that all the panels that will be part of the folder be (more or less) the same size. **Example:** If we have three 40x15 panels, the main panel must accommodate a folder type control of 46x18 so that it contains the three sub-panels without problems.

It is important that the panels to be used in the folder ***do not contain the same control names***, since - as we will see later - when the panels are grouped in a folder type control, they actually become part of a single control, and having controls with duplicate names could complicate everything.

We will create three panels (**PANEL_A**, **PANEL_B** and **PANEL_C**) with approximate dimensions of 50 x 16. In each one, we are going to place controls that do not have common names. **Example:** For **PANEL_A**, we used BUTTON_a1 as the Button name. We used the Copy and Paste object functions (on the [**Edit**] menu or in the context menu by using the right mouse button). NOMADS modifies the names of the controls so that they follow that sequence (BUTTON_a2, BUTTON_a3 and so on). We did something similar in the other panels and with the other controls.

Note: For the purposes of this exercise, it is not necessary for the panels to have any function, but only that they be different (to be able to appreciate the change of panel) with similar dimensions and that they do not have controls with the same name.

Tip: Use this opportunity to practice creating controls and using NOMADS editing, alignment and layout features.

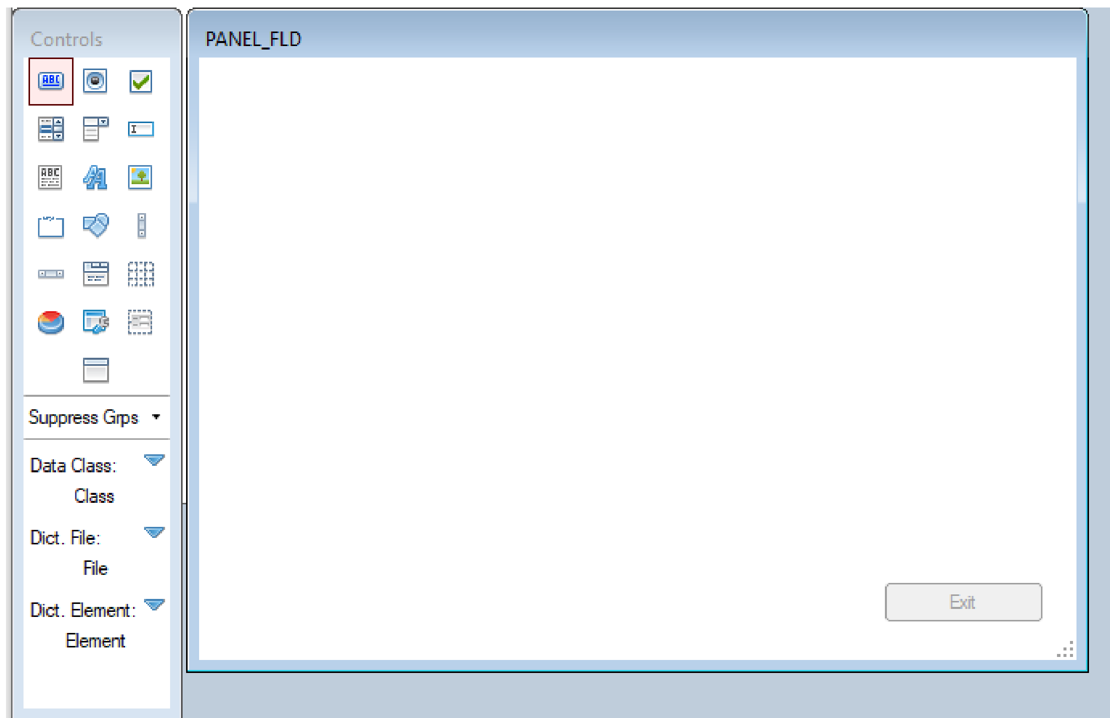
Below are the three panels we made to practice the folder control. Notice that none of them have any working options. We only placed buttons, text and some controls that do not share names.

Now that we have three panels (**PANEL_A**, **PANEL_B** and **PANEL_C**) that have approximate dimensions of 50x16, we are going to create a new panel, **PANEL_FLD**, with slightly larger dimensions. In the lower right corner, we will place an [**Exit**] button with the **End** function associated to the **When Button Pressed** event.

From the NOMADS **Library Object Selection** window, create the **PANEL_FLD** panel. Adjust its size so that it is 78 x 25 (using the [**Header**] option at the top). Then, create the button, remembering to

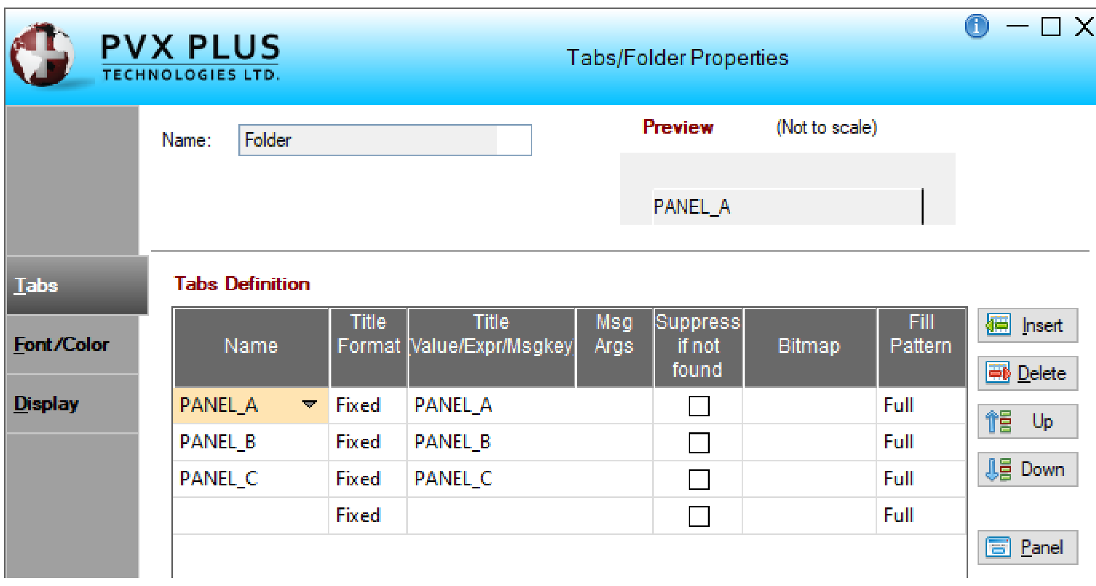
associate the logic so that the panel ends when the button is pressed.

The new panel now looks like the one shown below:



Select the [**Folder**] control in the **Controls** box on the left and draw a rectangle that occupies approximately 75% of the panel. We placed it in Column 2, Line 1 with dimensions of 58 x 22.

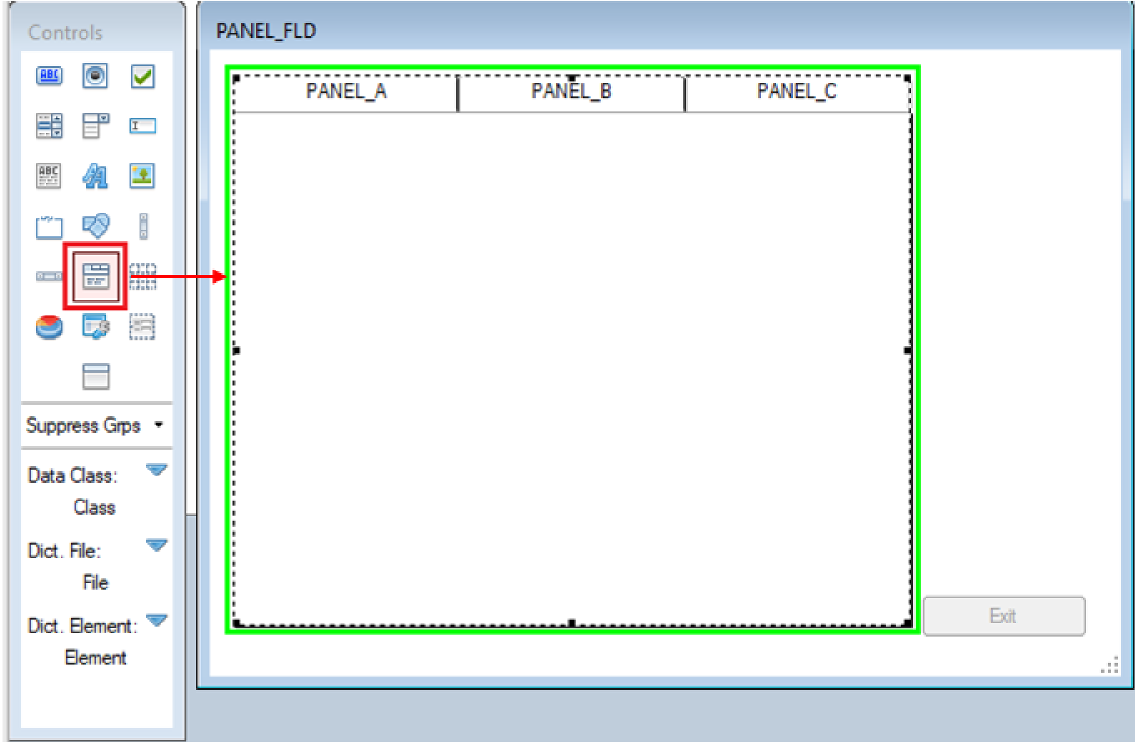
After drawing, the **Tabs/Folder Properties** window will appear for selecting the panels that will make up the folder control:



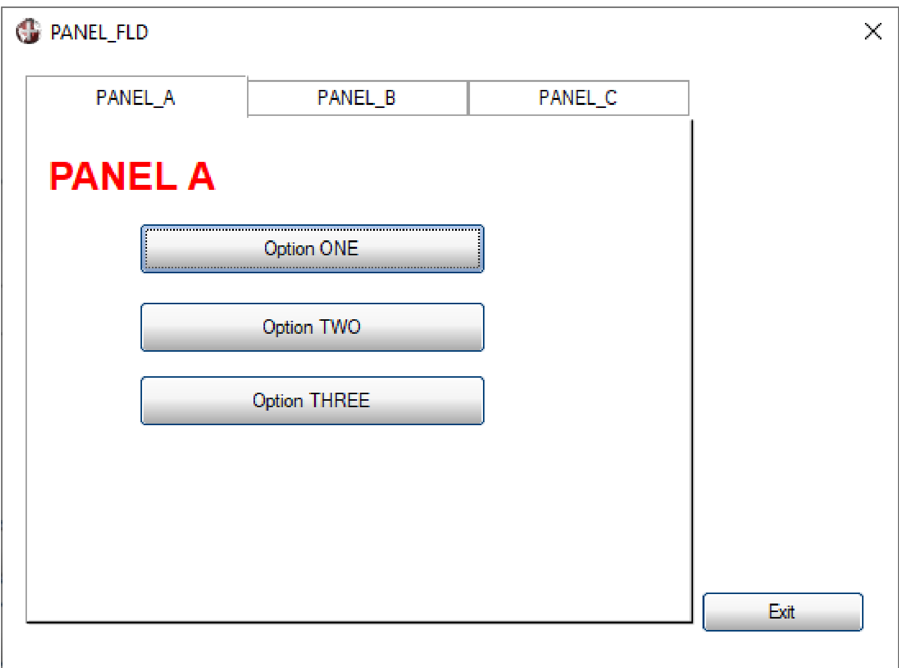
In this grid, we must select the name of the panel for each of the control tabs. We can change the title,

assign an image and select a fill pattern. For now, we're just going to choose the three panels (**PANEL_A**, **PANEL_B**, and **PANEL_C**) and leave the other options as they are.

The panel should look similar to the one below:



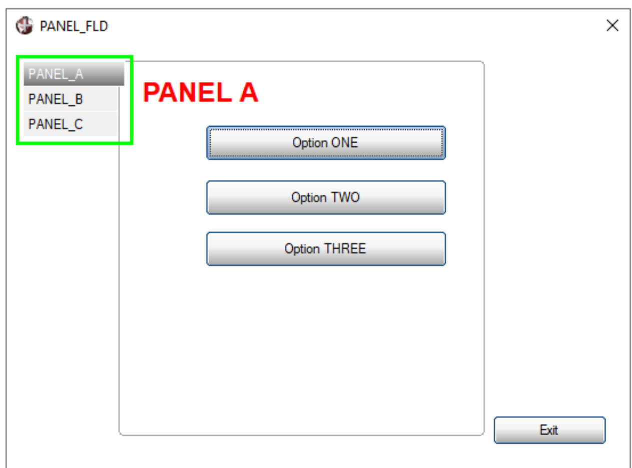
All that remains is to save our panel and then test it (using the [**Test**] option at the top):



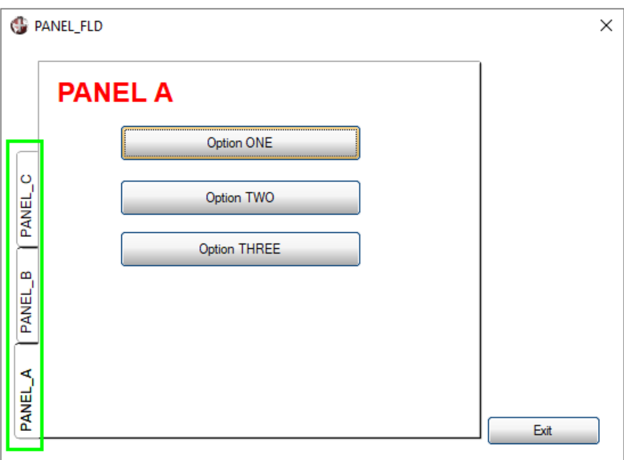
If everything went well, you will be able to move among the top three tabs of the panel. But, if this doesn't work, it could be that the panels created (**PANEL_A**, **PANEL_B**, and **PANEL_C** share some control name). It could also be that the dimension of the Folder control is too small to contain the other panels, so please check.

NOMADS allows the creation of different visual forms of this control with tabs at the top, bottom, left side or right side. To do this, go to the **Display** tab and select [**Tab Position**].

Examples:



Left Sidebar Tabs



Left (Rotated) Tabs

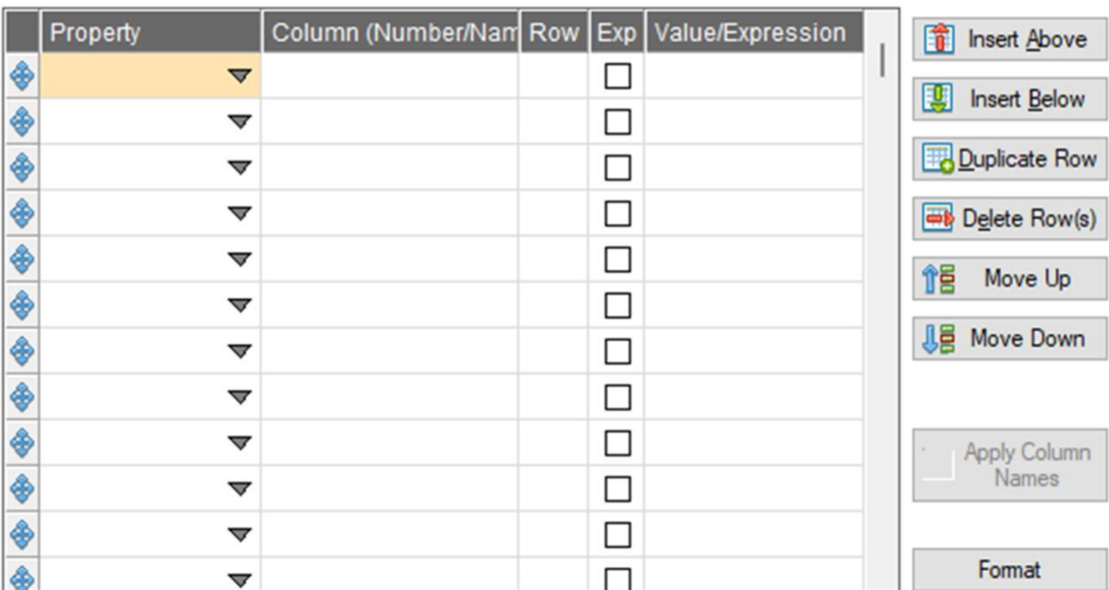
NOMADS also allows you to color code each tab and set the size of the tabs so that it appears to "stand out" from the rest of the panels. There is a variation of the control that is "without tabs" (Tabless) that allows you to make panel changes programmatically or instruct PxPlus to do them itself and to be able to make, for example, a wizard-type form (assistant) where the system takes you by the hand in entering information.

Refer to **Folder Control** in the PxPlus Help documentation.

GRID Control

Perhaps the most complex PxPlus control is the **GRID** control, which simulates a spreadsheet or two-dimensional array of cells. Each cell can be or contain a control (type can be a MULTI_LINE, BUTTON, LIST, etc.).

The main purpose of the **GRID** is the presentation/loading of multiple data elements in a highly visual manner (where multiple data elements can be viewed and modified at once). A **GRID** control looks like this:



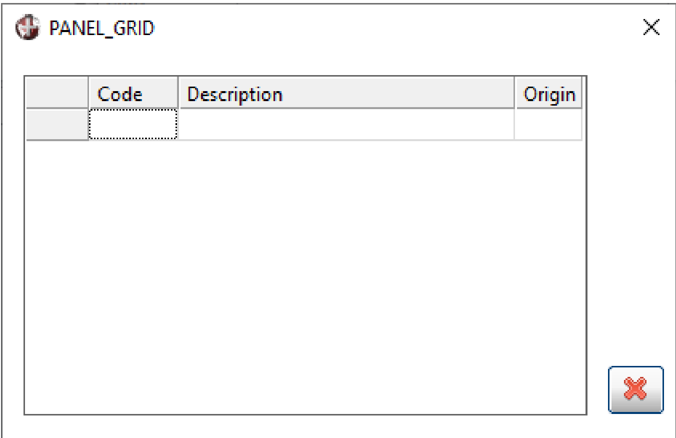
As we can see, it is made up of columns, each representing a field, which can be of different types. In this example (taken from the **GRID** control properties), there is a column with an image, another column that is the **DROP_BOX** type, others that are the text entry box or **MULTI_LINE** type and another that is the **CHECK_BOX** type.

Properly managing this control involves mastering the dynamic properties of the controls, so we will do an initial review by creating a new panel with a **BUTTON** and a simple **GRID** control, and then we will see how it works in more detail.

Refer to **Grid Control** in the PxPlus Help documentation.

Exercise: Creating a GRID Control

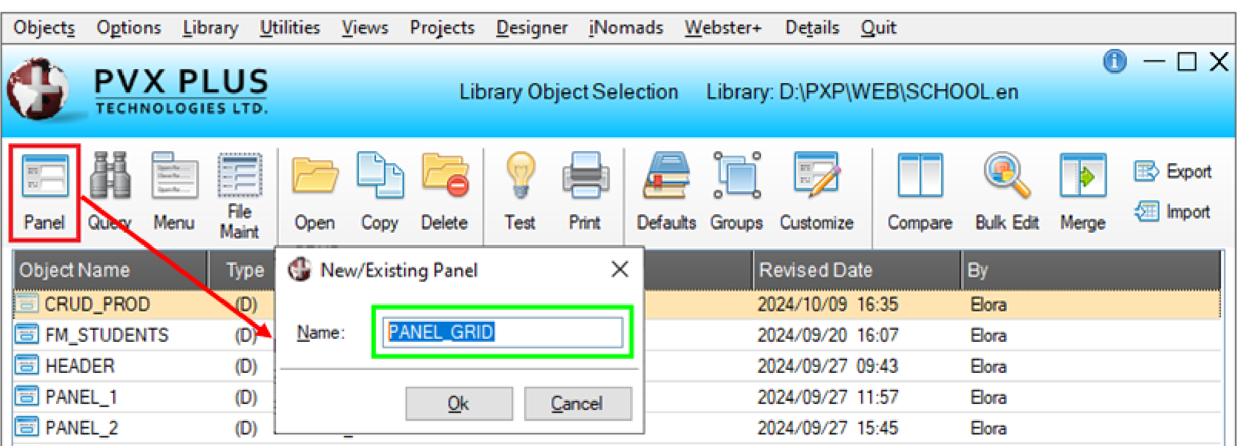
Initially, we will create a Grid with the data from the table **PRODUCT_TABLE**. We will use the previous settings to format the grid; that is, specify the titles and width of each column. We will enter the title **Code** in column 1 with a width of 8, the title **Description** in column 2 with a width of 30, and the title **Origin** in column 3 with a width of 6. Let's first see what this will look like:



Let's begin by creating a new panel from the NOMADS **Library Object Selection** window. (If this is not open, go to the PxPlus IDE main menu, open the **Graphical Application Builder (NOMADS)** category and select **Open Application Library**.)

Note: Remember that our library is called **SCHOOL.EN** and that a library is a structure or table in the NOMADS graphical designer that contains the name of several panels or objects grouped with this name.

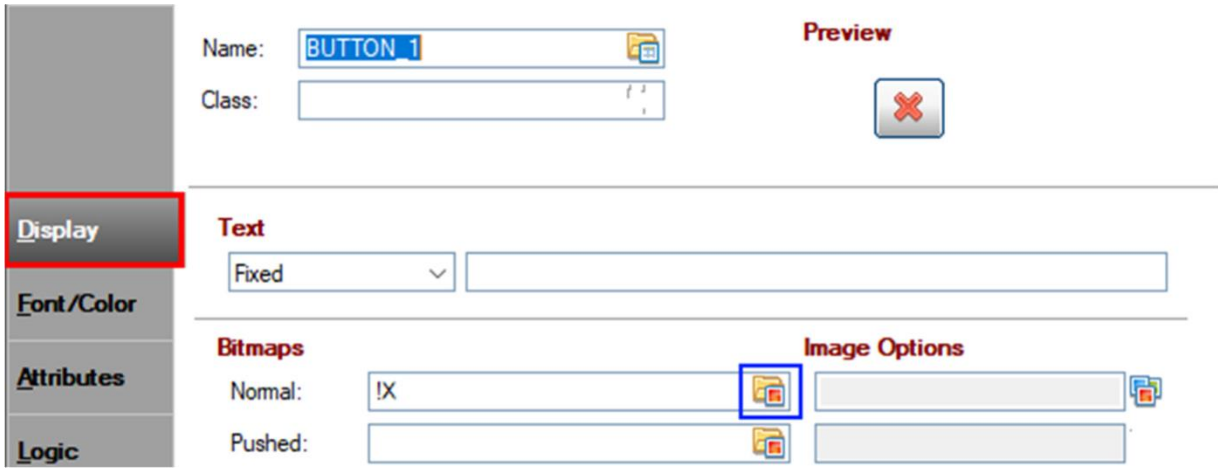
Specify the name of the new panel as **PANEL_GRID**:



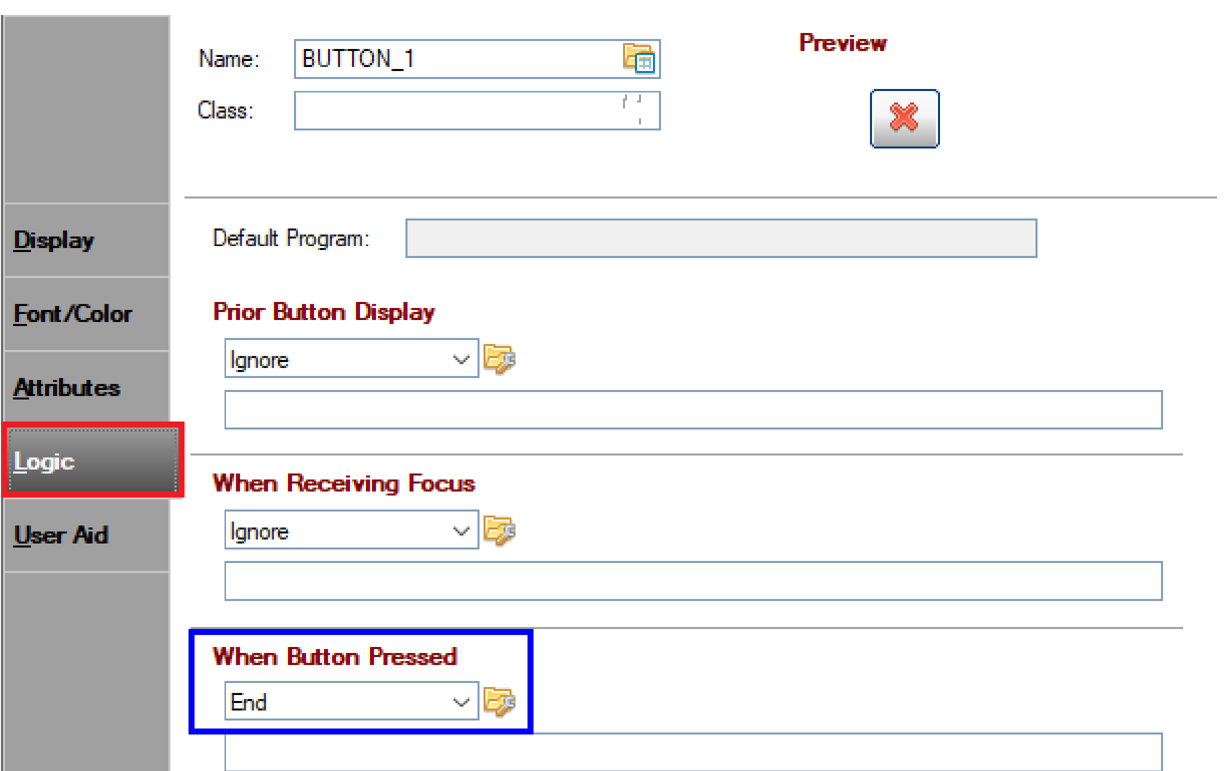
Once the name of the new panel has been specified, the NOMADS panel designer displays. Create this panel with dimensions of 60 columns wide x 16 lines high. Activate the [**Auto Refresh**] attribute.

In the lower right part of the panel, create a **BUTTON** (position is column 54, line 13 with dimensions of 5 columns wide x 2 lines high).

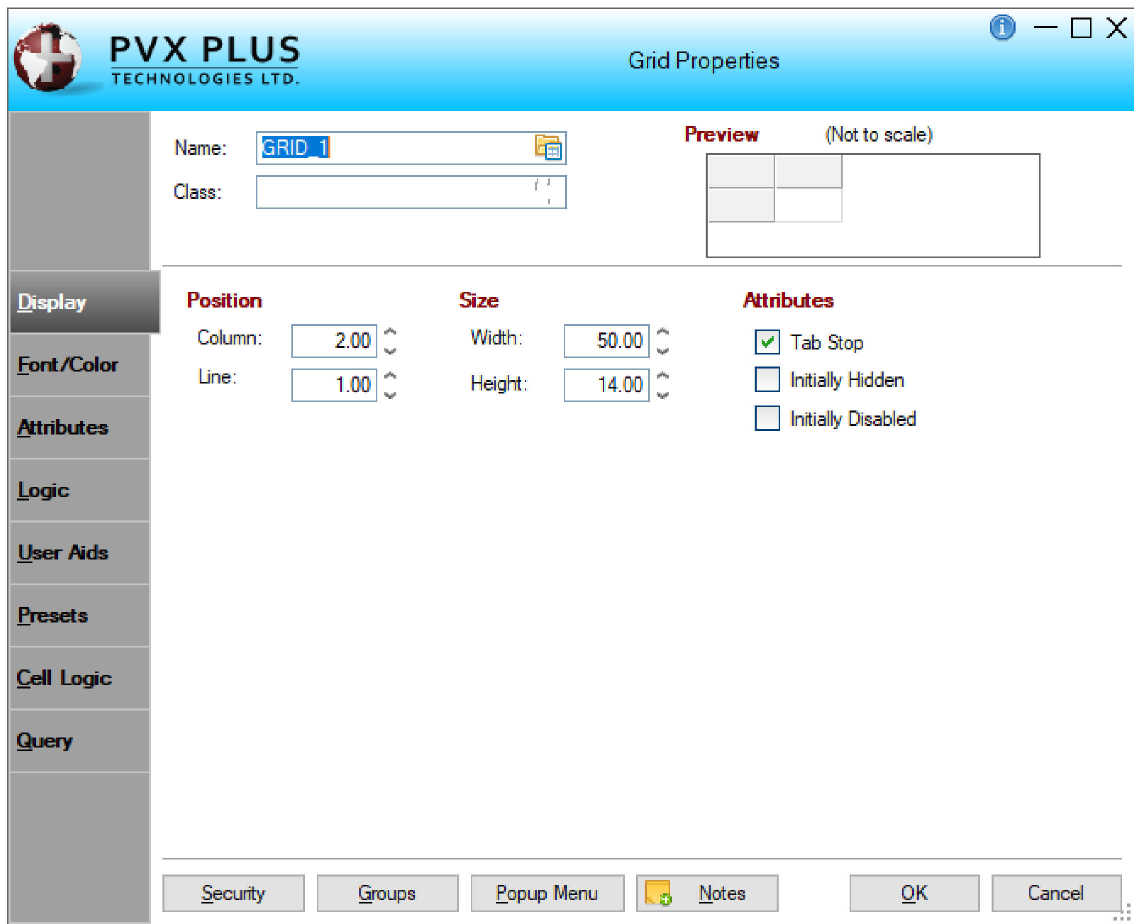
In the **Display** tab, under the **Bitmaps** section, select the red **X** bitmap by clicking the button (marked in blue).



In the **Logic** tab, for the **When Button Pressed** event, select the **End** action.

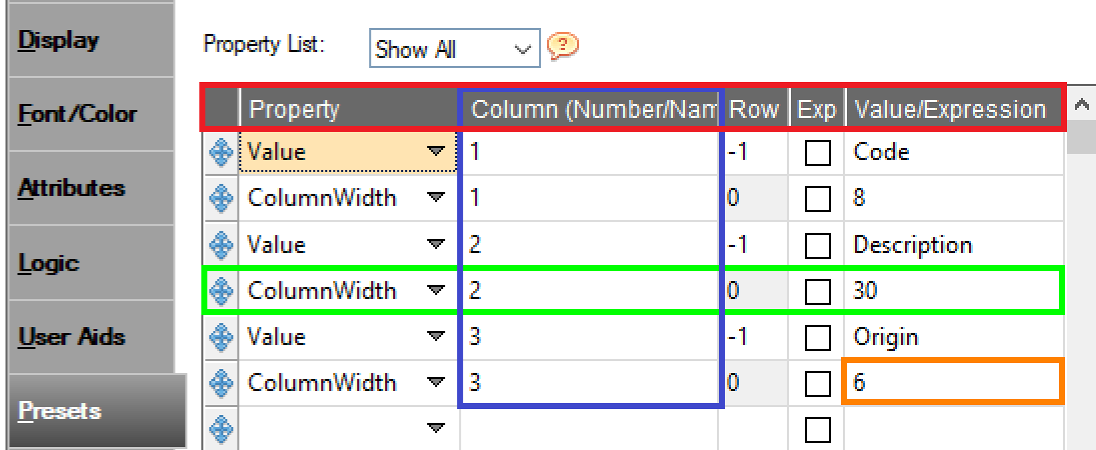


Once the buttons have been defined, we draw the **GRID** control. This control will be called **GRID_1** (default name, we can change it later). Its position will be in column 2, line 1 and will have dimensions of 50 columns wide x 14 lines high.



As you can see, the **GRID** control has many tabs to manage and to change attributes and properties. We have the visual part (**Display** and **Font/Color**). Then, we have the **Attributes**, the **Logic** part for associating actions with different events, **User Aids**, and the **Presets** settings. We also have **Cell Logic** to manage the logic of individual cells, and finally, the **Query** settings.

We will go directly to the **Presets** tab where we are going to enter the following information:



How do Presets Work?

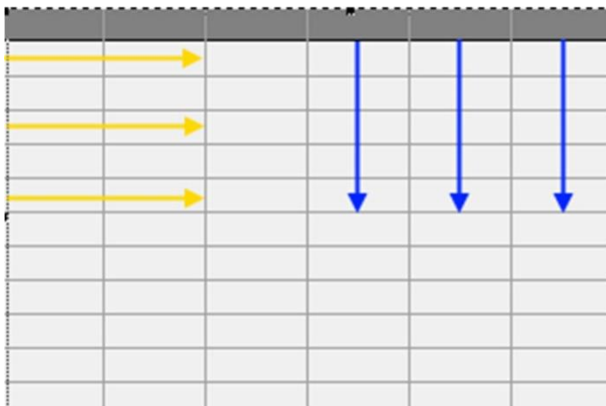
Before starting to use a grid, it is common to adjust or format it according to the use it will have. Normally, this is adding titles to the columns, specifying the type of column, adjusting the width and adjusting some other required parameters.

In the grid, a ***column*** is each of the vertical strips (column is marked in blue in the above example). A ***row*** is each of the horizontal strips (row is marked in green in the above example). A ***cell*** is the intersection between column and row (cell is marked in orange in the above example).

The ***title row*** (title row is marked in red in the above example) is a special row normally used to place the names or titles of each column. Notice that this row is composed of several cells that contain titles to represent the type of information in the columns, which, in this example, are "Property", "Column", "Row", "Exp", "Value/Expression", etc.

Since it is very important that we master these concepts, we are going to show another example.

In the grid below, the ***columns*** go in the direction of the blue arrows and the ***rows*** go in the direction of the yellow arrows:



When we are working with a grid, we can think of a matrix where each "part" (cell) will be referenced by two values.

Example:

"Cell 4,2" actually means the space where column 4 intersects with row 2. It is assumed that the upper left corner, "cell 1,1", is the origin. As we move to the right, the ***columns*** increase, and as we move down, the ***rows*** increase:

Columns →

Rows ↓

1,1	2,1	3,1	4,1	5,1	6,1
1,2	2,2	3,2	4,2	5,2	6,2
1,3	2,3	3,3	4,3	5,3	6,3
1,4	2,4	3,4	4,4	5,4	6,4
1,5	2,5	3,5	4,5	5,5	6,5
1,6	2,6	3,6	4,6	5,6	6,6

In the case of PxPlus and specifically the **GRID** control, we can refer to the entire row or to the entire column. If we want to specify the entire row 7, for example, specifying "**GRID 0,7**" will indicate that it is the entire row 7. A special case is "**GRID 0.0**" where we refer to the ***entire grid***.

On the previous page, we said that there is a special title row, which is denoted as "**GRID 0,-1**". The **-1** indicates that it will not be part of the data and that it is only used as a label.

Knowing this, we can begin to take a look at some commands related to the **GRID**. Well, actually, we will study the **GRID** command (yes, there is a command that is called the same as the control), and its general syntax is:

grid action control, column, row, value$

Where:

grid: This is the command and will always appear like this

action: Action to be performed on the control: LOAD, FIND, READ, CLEAR, LOCK, etc.

control: The ObjectID or identifier of the GRID control that we are working on

column: The column on which we will do the action

row: The row on which we will do the action

value$: The value written or read as a result of the action

In addition to these commands, we have properties, which, as we remember, are values that we can read/modify for better manipulation of our **GRID** control.

As it is very important to know the properties of the **GRID** control, we have decided to include a table that lists its main properties.

GRID Object Properties

Some of the principal **GRID** control properties are:

Cell Type	Description
"BarChart"	Used to display horizontal bar charts based on the contents of the 'Text$ property.
"Button"	Cell works like a button if clicked.
"CheckBox"	Check box with no 3D effect ('Value=0 or 1,'Text=label).
"CheckMark"	Check box that uses a check mark.
"DropBox"	A text cell that, when editing, allows the user to make a selection from a drop-down list.
"DropBoxHideBtn"	Same as "DropBox", but the drop-down button is shown only when focus is on the cell.
"Ellipsis"	Normal text cell that will show three dots (…) if text exceeds the displayable area.
"EllipsisDrop"	Same as "Ellipsis", but forces the cell to drop vertically only during an edit.
"FlatButton"	A flat button that shows no visible change when clicked.
"FlatTextButton"	Similar to "TextButton", but the button is flat and shows no visible change when clicked.
"FlatTextButtonInOnly"	Similar to "TextButton", but the button looks flat only when not depressed.
"Lookup"	Small button at the right side of the cell used to invoke a lookup.
"LookupHideBtn"	Same as "Lookup", but indicates that the button is hidden when the cell is not selected.
"Multi_line"	Normal text cell that can contain multiple lines of text.
"Normal"	Normal text cell that contains one line of data.
"Query"	Same as "Lookup", but the user is unable to edit the cell.
"QueryHideBtn"	Same as "LookupHideBtn", but the user is unable to edit the cell.
"SingleLine"	Normal text cell that allows for a single line of entry, expanding the cell horizontally according to the overlap rules.
"TextButton"	Similar to "Button", except that the text on the button adheres to the 'Align$ property and will wrap.
"VarDropBox"	A text cell that, when editing, allows the user to make a selection from a drop-down list or enter any other value.

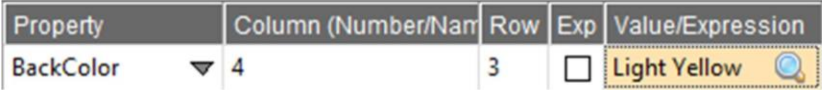
A lot of information to digest? Don't worry - you don't need to learn all of this. For now, we are going

to continue with our explanation, and you will see that you will learn it one step at a time.

Let's analyze what we are doing in the **Presets** definition of our grid. Here, the grid is "prepared" to give format, appearance and functionality to our control. In this case, the settings are broken down into four main areas: **Property**, **Column**, **Row** and **Value.** Basically, we have to change what we want by specifying which property we will change, which cell we will apply it to and what value we will put for the specified property.

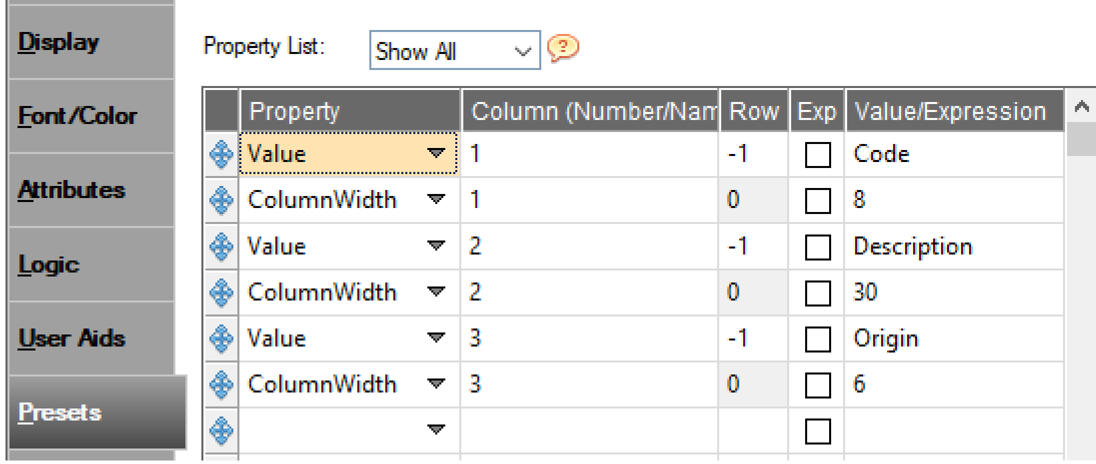
Example:

If we want to change the background color (**Backcolor$** property) of cell 4,3 (Column 4, Row 3) to the Yellow color, we would enter the following values:



In this case, NOMADS is smart enough that, when a color property is selected, the magnifying glass icon on the right allows us to select the color. But in other cases, we must specify the value we want.

Now, we should understand this much better. Let's select the **Presets** tab.



Specify the content (using the **Value** property) of cell 1,-1 (title of the first column) by entering the text **Code**.

Then, specify the width (using the **ColumnWidth** property) of column 1 as **8** characters. (Note that we have set **column=1,row=0** to refer to the ***entire column***.)

Enter the title for column 2 as **Description**.

Specify the width of column 2 as **30** characters.

Enter the title for column 3 as **Origin**.

Specify the width of column 3 as **6** characters.

At this point, we have used two dynamic properties: **Value** and **ColumnWidth**. Reviewing the **GRID**

dynamic properties table, we see there are several properties that initially look "similar", such as "**Col**" and "**Cols**" or "**ColumnWide**" and "**ColumnWidth**". We suggest that you familiarize yourself with these properties as you need them. Some will be very obvious from the beginning (**Example: BackColor** - cell background color), and others, perhaps not so much (**Example: hWnd** - Windows control handler).

Refer to **GRID Properties** in the PxPlus Help documentation.

Some properties work without having any type of prior "preparation", such as **Line**, which indicates on which line of the screen the **GRID** control is placed. However, other properties need to be set or "prepared" or have certain conditions met. **Example:** The **CellType** property needs to know where the focus is within the control; that is, which cell the program refers to in order to determine or change the type of sub-control that is the cell.

Important Note: The **GRID** control is a complex control, which we can initially see as an array of text entry box controls (**MULTI_LINE**) but are able to change so that a cell is not only a multi-line but can be a drop-down box where you can have multiple values in a single cell. We could also change a cell to behave like a button so that pressing it executes an action.

The cell types supported by the **GRID** can be one of the following:

Cell Type	Description
Normal	Cell for normal text (one data line).
Multi_line	Cell for normal text (several data lines).
Ellipsis	Cell for normal text; will show "…" if the text is bigger than the display area.
Button	A cell like a button.
CheckBox	A check box type, no 3D.
CheckBoxRaised	A check box type, 3D, looks unpressed.
CheckBoxRecessed	A check box type, 3D, looks pressed.
CheckMark	A check mark type.
CheckMarkRaised	A check mark type, unpressed.
CheckMarkRecessed	A check mark type, pressed.
DropBox	A drop box style cell, the list's values loads from **'Value$** attribute.
DropBoxHideBtn	Similar to "DropBox", but the button is hidden until cell has the focus.

Since the **GRID** control is a data entry structure (in most cases), as programmers, we must be responsible for saving all its contents to preserve it. If we delete the control or close the panel or something similar, all data contained in the **GRID** control will be lost, as will the contents of a multi-line or any memory structure on the computer.

Refer to the **GRID** command and **GRID Properties** in the PxPlus Help documentation.

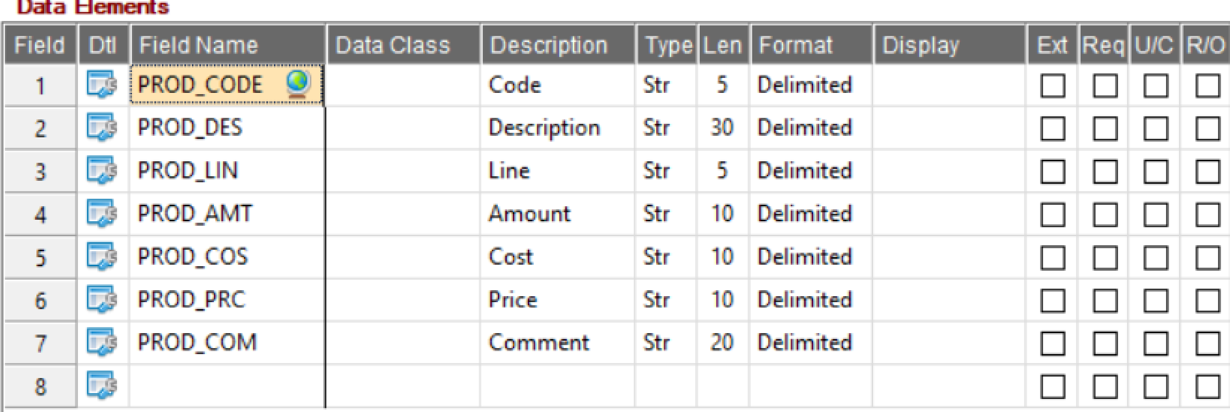
Exercise: Defining a Table with Sample Data

We are going to define a new **PRODUCTS** table and enter some sample data. We will use this information to learn about creating data sources and views in the next section.

From the PxPlus IDE main menu, open the **Data Management** category and select **Data Dictionary Maintenance**.

For the [**Name**] of the table, enter PRODUCTS. For the [**Description**], enter PRODUCT LIST. Enter the name of the [**Physical File**] as PRODUCTS.

Click the **Elements** tab and define the elements that make up this table:



Data Elements

Field	Dtl	Field Name	Data Class	Description	Type	Len	Format	Display	Ext	Req	U/C	R/O
1		PROD_CODE		Code	Str	5	Delimited		☐	☐	☐	☐
2		PROD_DES		Description	Str	30	Delimited		☐	☐	☐	☐
3		PROD_LIN		Line	Str	5	Delimited		☐	☐	☐	☐
4		PROD_AMT		Amount	Str	10	Delimited		☐	☐	☐	☐
5		PROD_COS		Cost	Str	10	Delimited		☐	☐	☐	☐
6		PROD_PRC		Price	Str	10	Delimited		☐	☐	☐	☐
7		PROD_COM		Comment	Str	20	Delimited		☐	☐	☐	☐
8									☐	☐	☐	☐

Once all the elements have been entered, we must click on the [**Define Keys**] option at the top of the panel and select PROD_CODE$ as the primary key.

Once the table elements and its keys have been defined, we can proceed with the creation or physical update of the file by clicking the [**Update File**] button.

Now that the **PRODUCTS** table has been defined and created, we are going to run the File Maintenance Generator tool to create a panel. From the PxPlus IDE main menu, open the **Graphical Application Builder (NOMADS)** category and select **File Maintenance Generator**.

A window will open asking for the name of the library and the panel. Select the library **SCHOOL.EN.** For the name of the panel, enter **FM_PRODUCTS.**

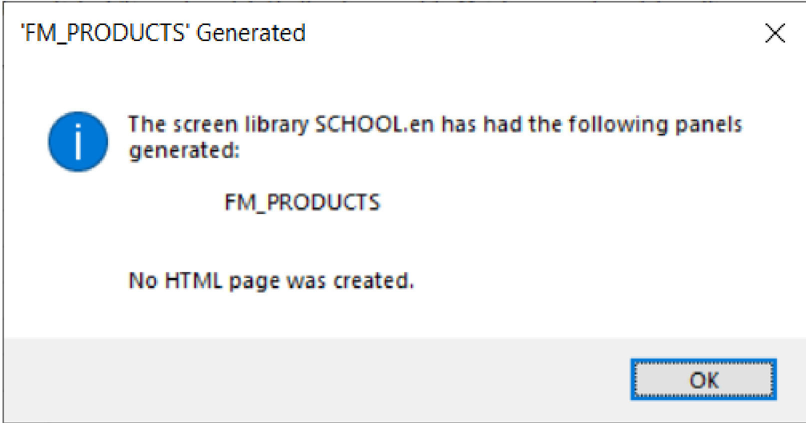
When the **Welcome** panel displays, click the [**Next**] button.

For the [**Table Name**], select **PRODUCTS**. The [**Panel Title**] displays as Products Maintenance. We will leave that as is and click the [**Next**] button to advance to **Step 2.**

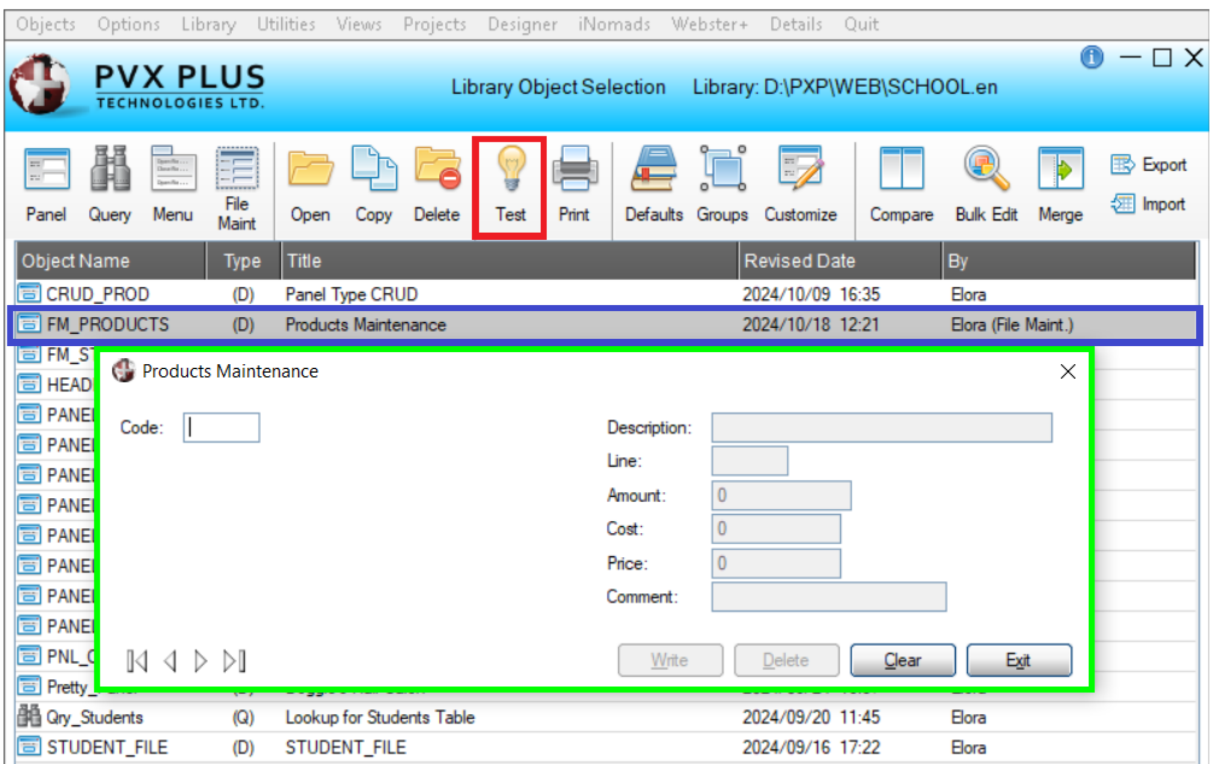
Since we will not be changing any of the options, click the [**Next**] button four more times until you reach **Step 6** where we will select the elements to be shown in the file maintenance panel.

For our panel, place all the fields on the right side, with the exception of the key, which is the Code field:

Once the field selection is complete, click the [**Next**] button to advance to **Step 7** where a summary screen will appear. Click the [**Finish**] button, and a message similar to the one shown below will display:



To test our panel, we must load the library in NOMADS (**SCHOOL.EN**), and a panel similar to the one shown below will display:



We are going to use our test panel to enter the sample data below. After each record, click the **Write** button to save the record to our **PRODUCTS** table and then click the **Clear** button to clear the fields and enter the next record.

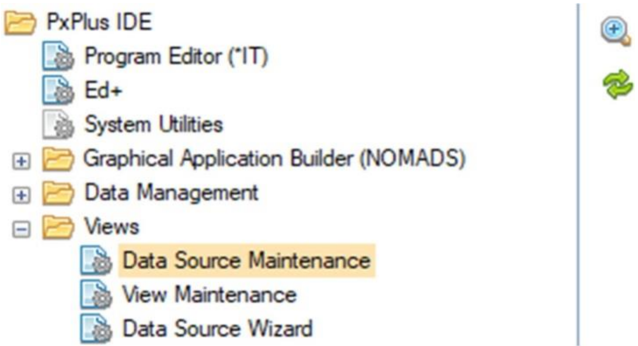
Code	Description	Line	Amount	Cost	Price	Comment
00001	Hammer_SM	0101	10	15	22	
00002	Hammer_LG	0101	8	25	35	
00003	Pliers_SM	0102	15	30	40	
00004	Pliers_LG	0102	10	40	50	
00005	Screwdriver_FL	0103	5	10	12	
00006	Screwdriver_ST	0103	8	12	15	
00007	Screwdriver_SQ	0103	7	15	20	
00008	Saw_WD	0104	9	50	75	
00009	Saw_MT	0104	4	60	85	
00010	Saw_TB	0104	2	200	400	
00011	Drill_WD	0105	4	100	200	
00012	Drill_CT	0105	3	150	250	

8. Introduction to Data Sources

Until now, we have seen, as a source of our information, a "simplified" unit called (for our understanding) a ***table***, which is basically a file containing fields or elements; that is, a unique type of information for each field. But a data source goes further by allowing fields from one table to be linked to another to access information not contained in a table.

We will simplify everything for now and imagine a data source that is just a table to begin to know data sources and then, the views.

For now, we are going to execute **Data Source Maintenance** (go to the PxPlus IDE main menu and open the **Views** category):

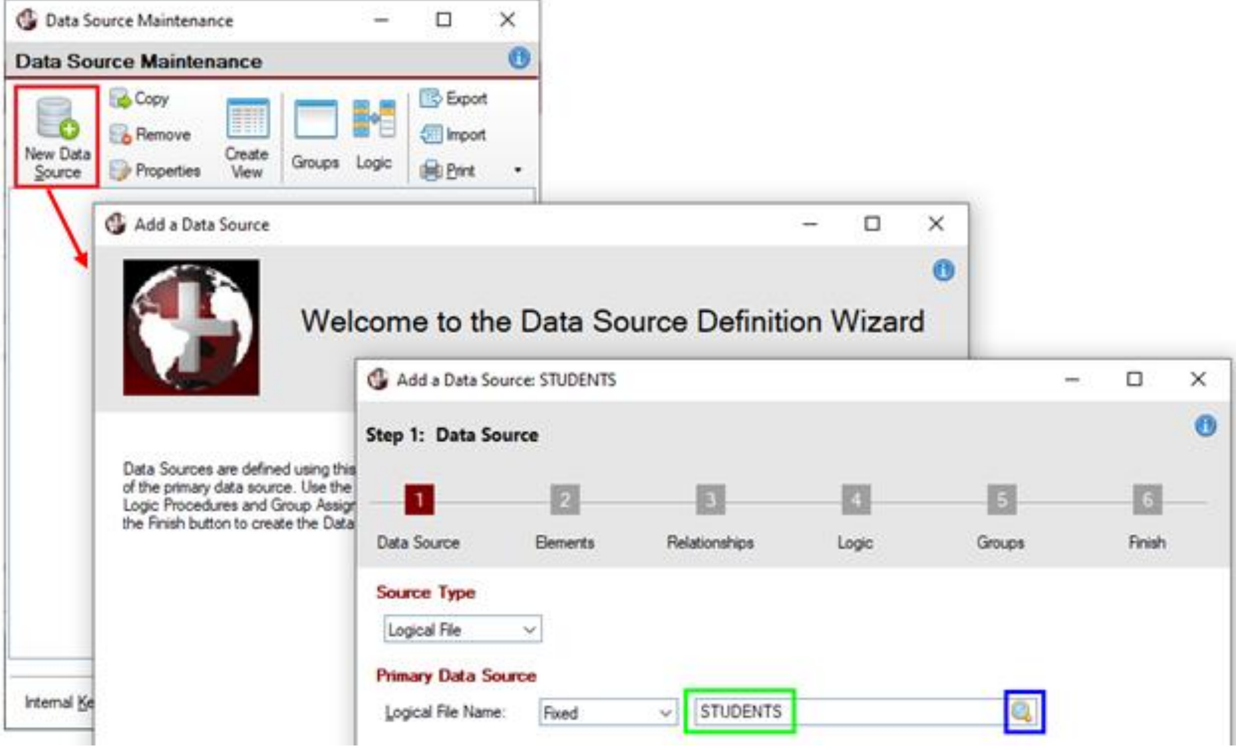


If the Data Sources file does not exist, a message will display to ask if you wish to create it. Answer **Yes**.

The **Data Source Maintenance** window will display. Click the **New Data Source** button at the top. The **Welcome** panel of the **Data Source Definition Wizard** will display.

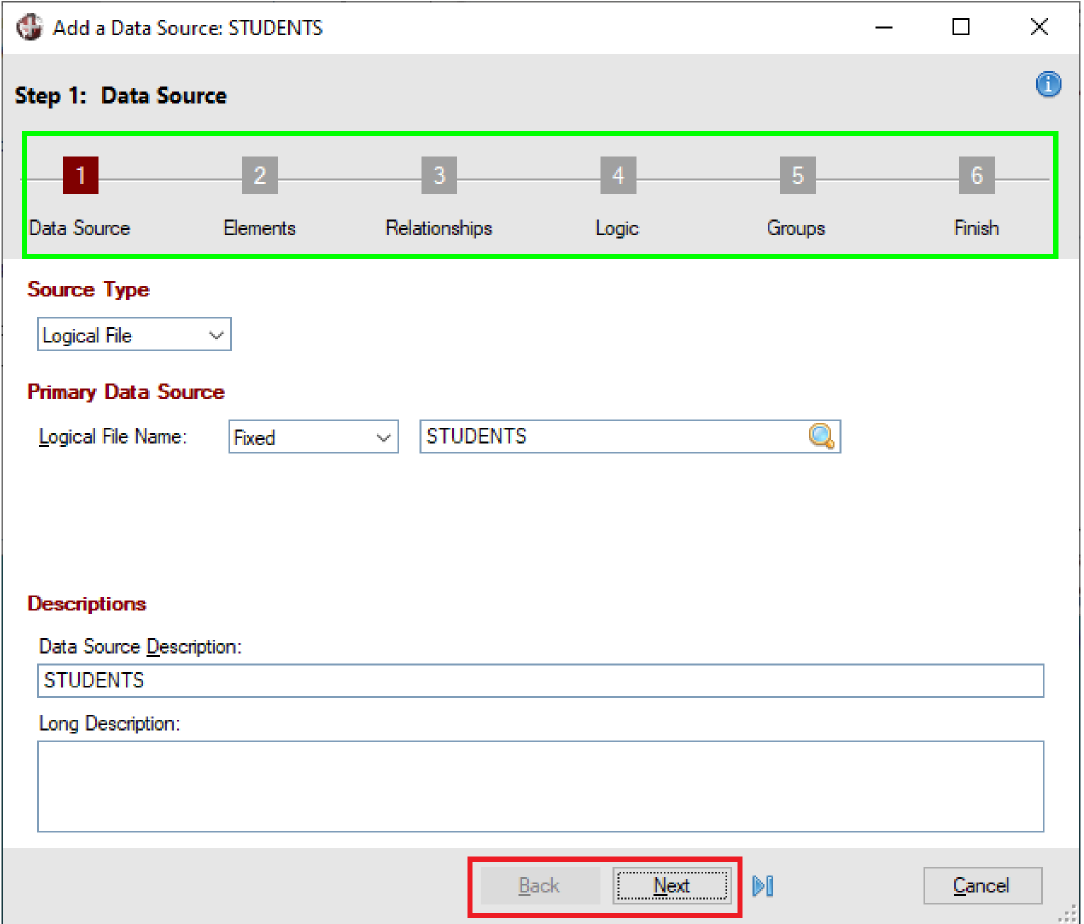
Click the [**Next**] button to advance to **Step 1: Data Source**.

For the **Logical File Name** field, enter or select (using the magnifying glass icon) the **STUDENTS** table.



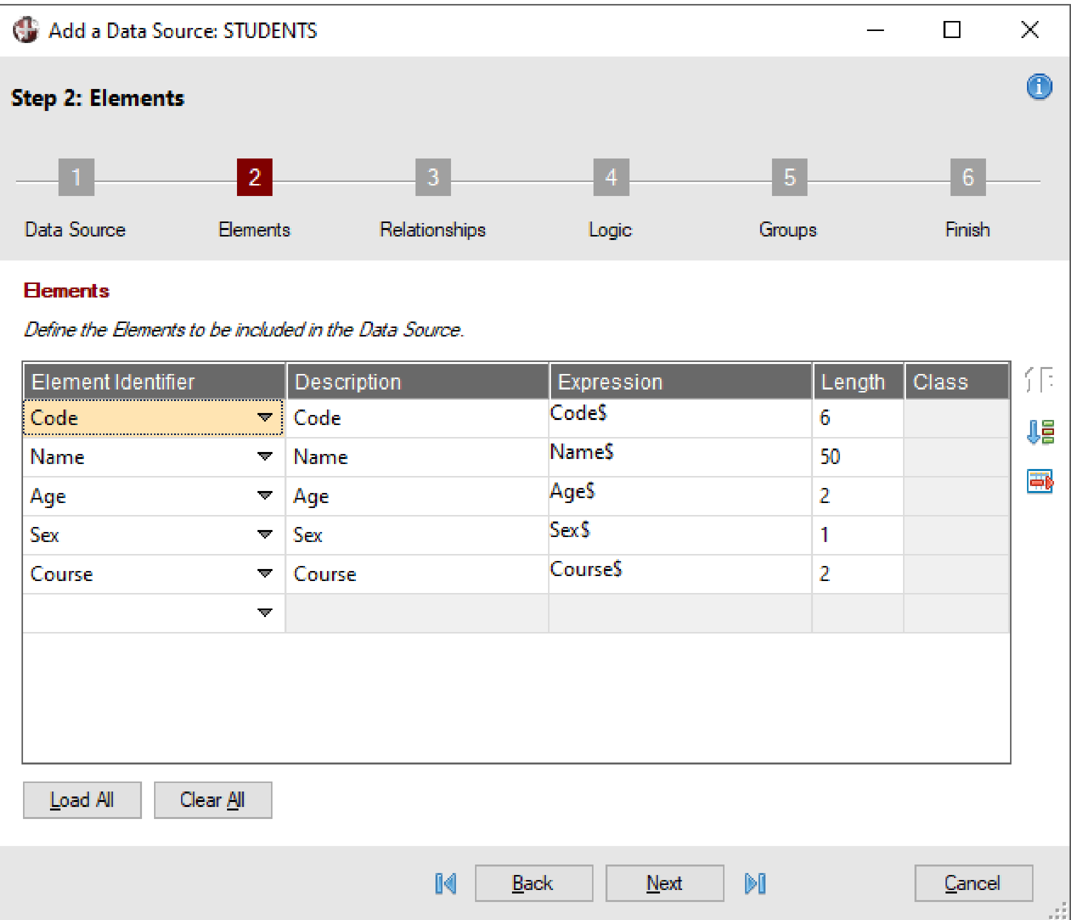
We can see that this panel is actually a wizard-type panel where we will go step-by-step to enter the necessary information to define the new data source (origin). This panel has a series of steps at the top (marked in green) and [**Back**] and [**Next**] buttons at the bottom (marked in red) to navigate through each step.

Refer to [Data Source Definition Wizard]() in the PxPlus Help documentation.



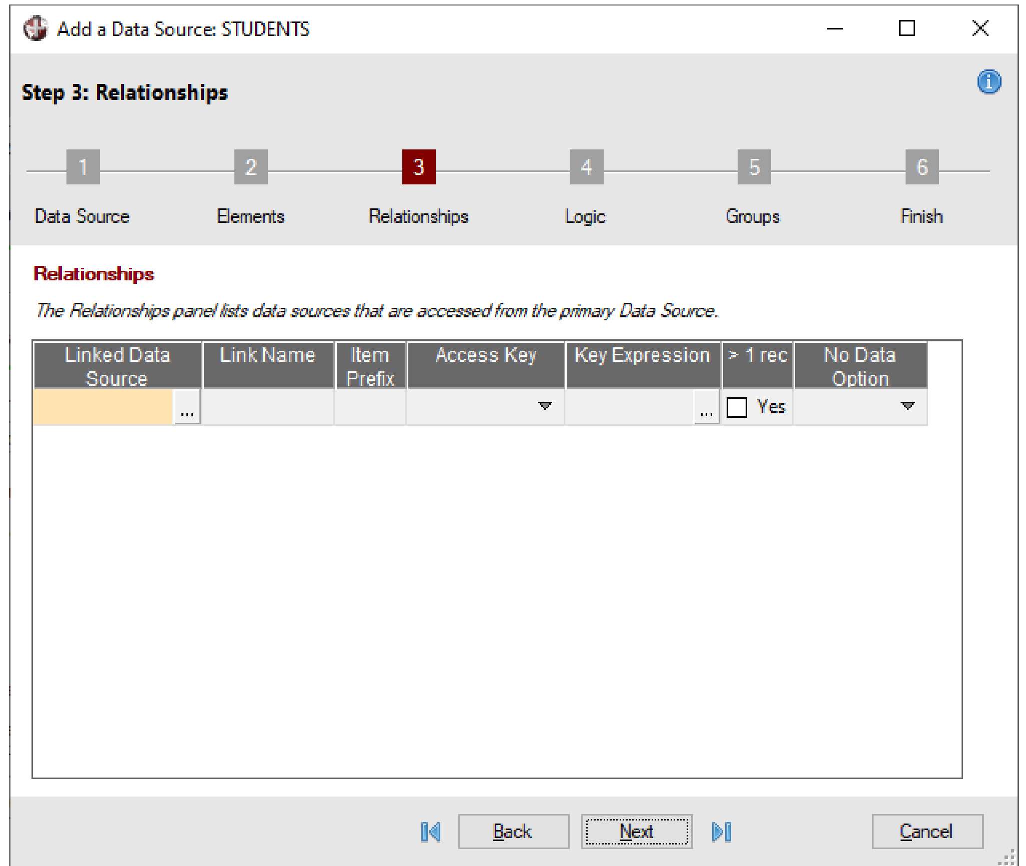
Click the [**Next**] button.

The **Step 2: Elements** panel is where we can select which fields/elements we are going to show in the view. We want to load all of them so click the [**Load All**] button located at the bottom left of this panel.



Click the [**Next**] button to continue.

The **Step 3: Relationships** panel is where we can establish links between the elements of this table and those of another. For now, we are going to leave this panel as is, without the additional elements.



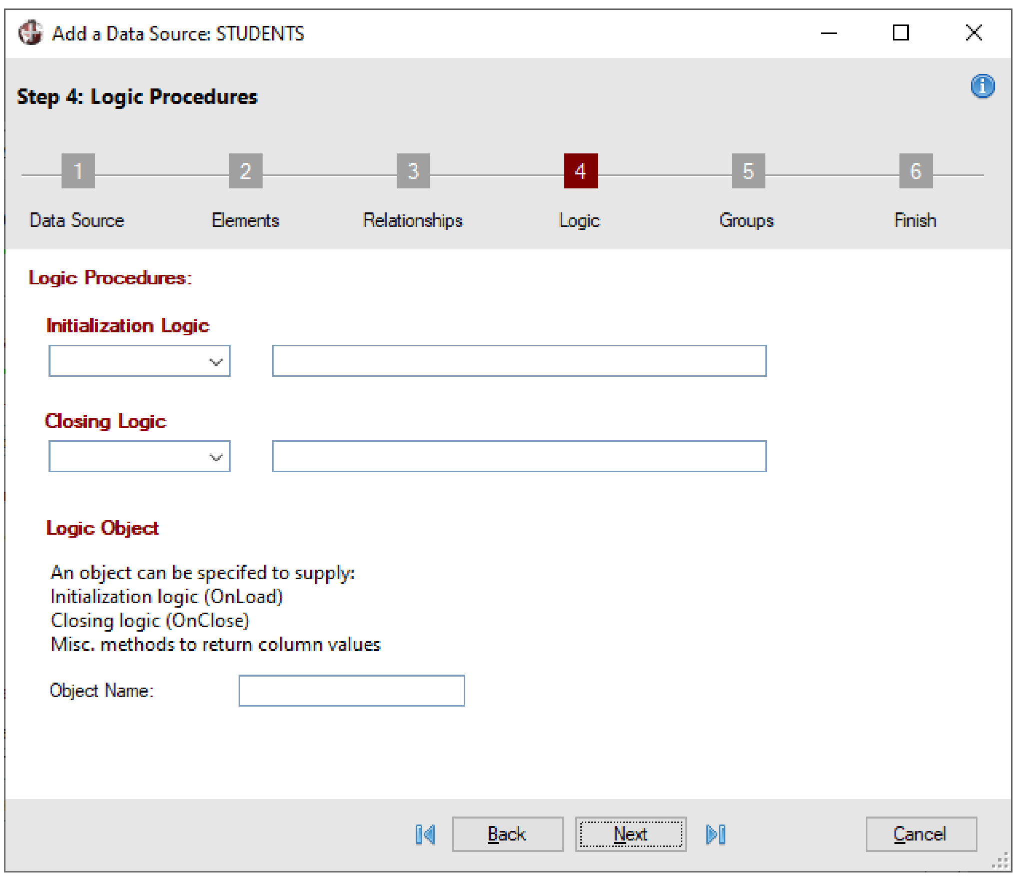
Click the [**Next**] button to continue.

The **Step 4: Logic Procedures** panel is where we can specify routines (programs) that will be executed before showing the view.

The first is the **Initialization Logic** and then the **Closing Logic**. In both events, it is possible to perform any of the following actions: **None** (does nothing); **Call** (call a routine or program) and/or **Execute** (execute a PxPlus command).

It is also possible to supply the name of an object that is responsible for managing both actions (or just one of them).

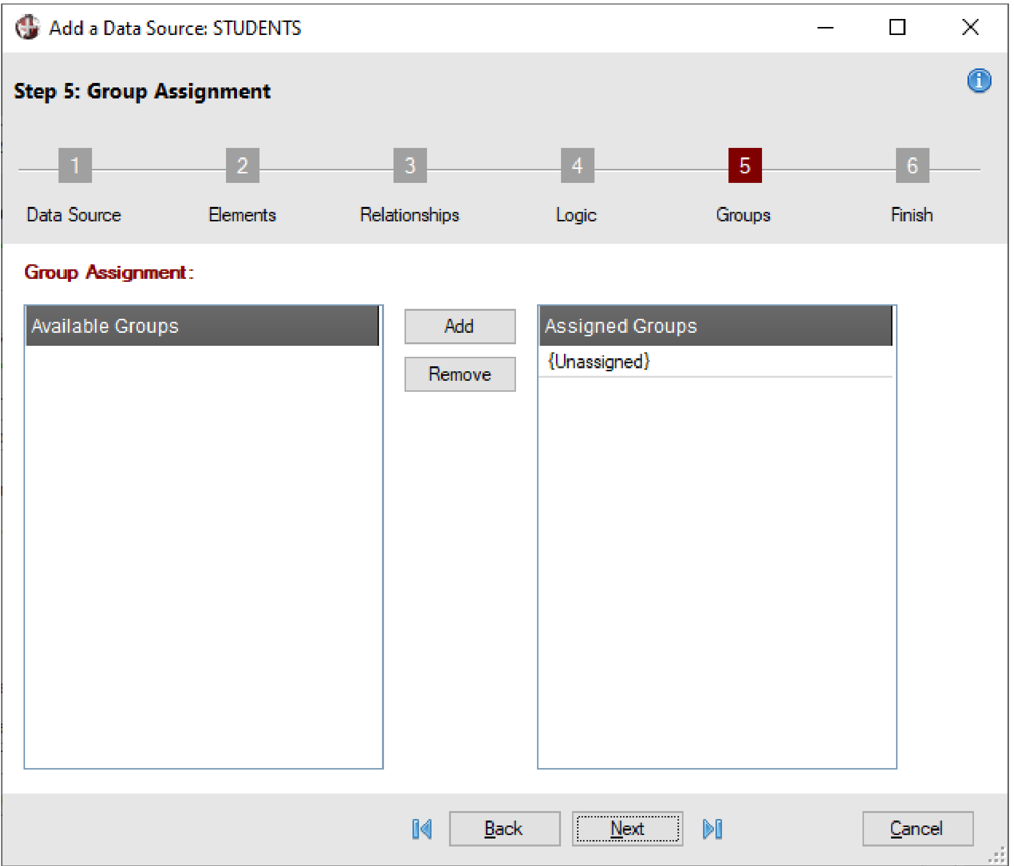
For now, we are going to leave this panel as is.



Click the [**Next**] button to continue.

The **Step 5: Group Assignment** panel allows the user to organize his/her panels/views logically; that is, at an operational level. There is no impact on whether or not the data source belongs to a certain group, just so that the user has an easier way to search/manage your data sources.

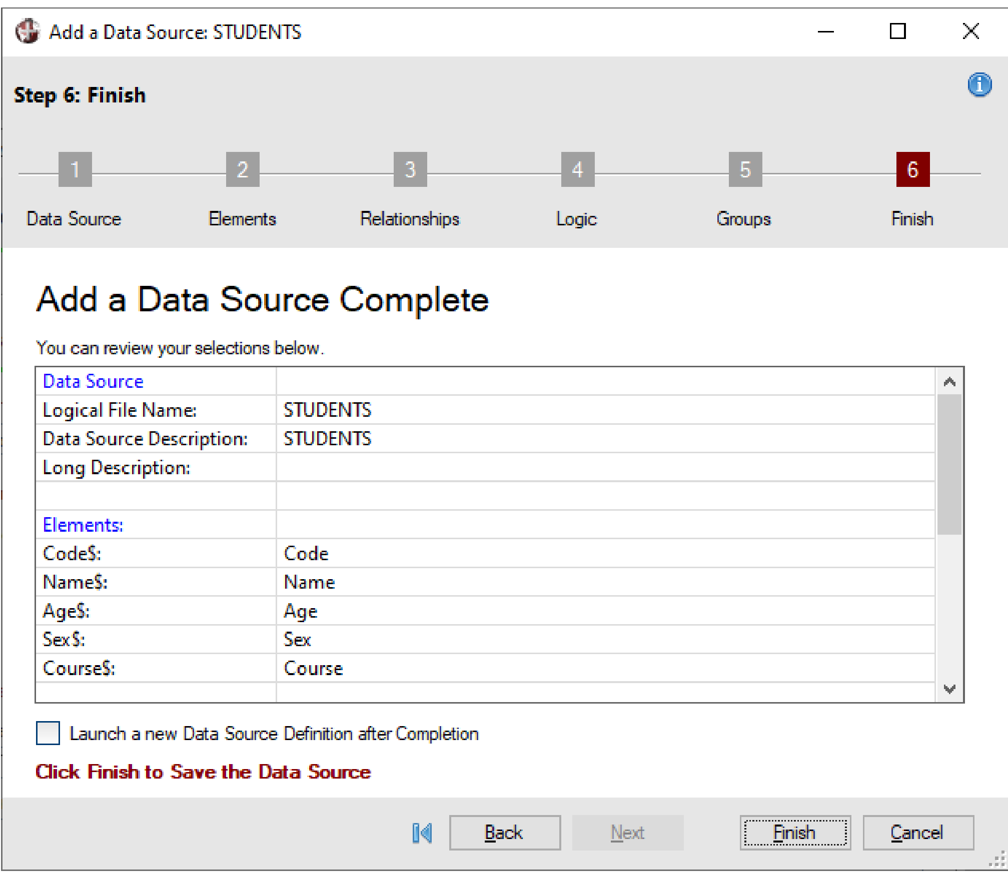
For now, let's leave this panel as is.



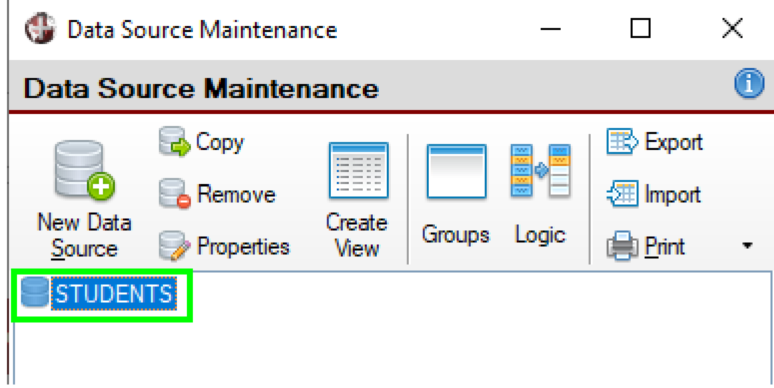
Click the [**Next**] button to continue.

The **Step 6: Finish** panel displays a summary of what was selected during the creation of the data source. View the results. You can go back to previous steps to make any necessary changes.

Click the [**Finish**] button at the bottom of this panel. This step completes the creation of the data source.



The **Data Source Maintenance** window will display where we should see the newly created data source listed.



Refer to **Data Source Maintenance** in the PxPlus Help documentation.

Let's create another data source. Select **Data Source Maintenance** (go to the PxPlus IDE main menu and open the **Views** category).

Select the [**New Data Source**] toolbar button. When the **Data Source Definition Wizard** displays, click the [**Next**] button to advance to **Step 1** where we will enter or select the **PRODUCTS** table we created in a previous exercise.

Click the [**Next**] button to advance to the **Elements** panel. Click the [**Load All**] button to load all the elements:

Elements

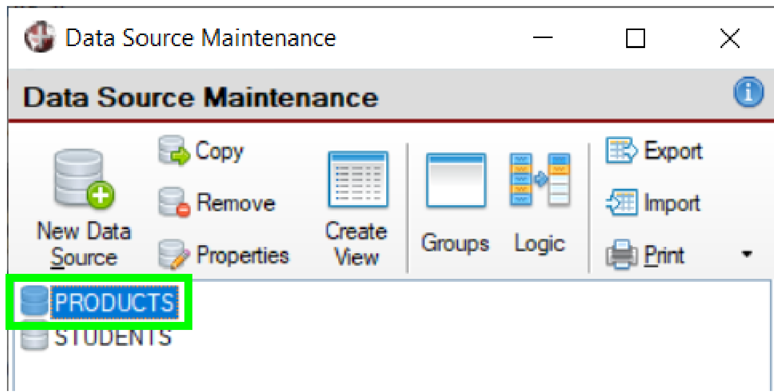
Define the Elements to be included in the Data Source.

Element Identifier	Description	Expression	Length	Class
PROD_CODE	Code	PROD_CODE$	5	
PROD_DES	Description	PROD_DES$	30	
PROD_LIN	Line	PROD_LIN$	5	
PROD_AMT	Amount	PROD_AMT	10	
PROD_COS	Cost	PROD_COS	10	
PROD_PRC	Price	PROD_PRC	10	
PROD_COM	Comment	PROD_COM$	20	

Click the [**Next**] button to advance to the **Relationships** panel. We will leave this panel as is.

We will also leave the **Logic** and **Groups** panels as is. Click the [**Next**] button three more times to advance to the last step and then click the [**Finish**] button.

The **Data Source Maintenance** window is shown with the new data source listed:



Defining a View

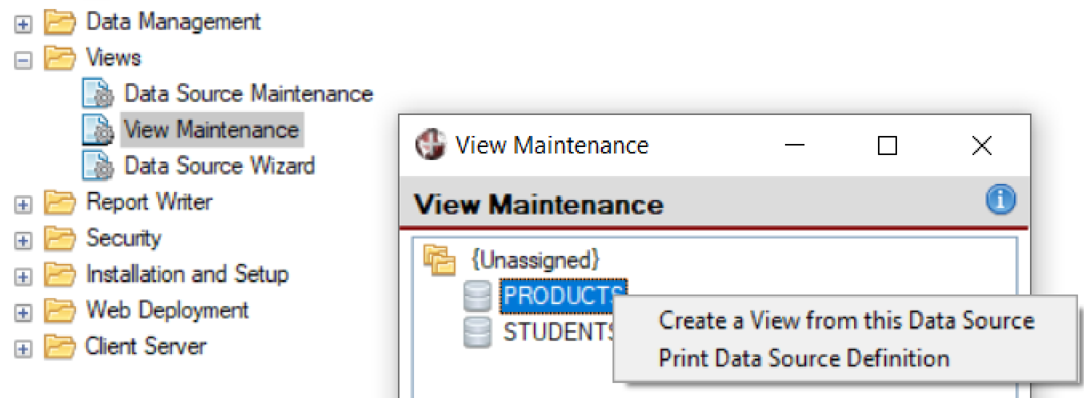
A ***view*** is a custom way to display the content of a data source, providing the ability to specify calculations, data selection criteria, logical procedures and more. As usual, we will start by defining a basic view based on the data source that we have just defined, and then, we will see how to best take advantage of this tool.

Refer to **View Maintenance** in the PxPlus Help documentation.

Exercise: Defining a View

To better illustrate creating a view, we are going to select **PRODUCTS** as the data source instead of using the **STUDENTS** data source.

We are going to execute **View Maintenance**. Go to the PxPlus IDE main menu and open the **Views** category:

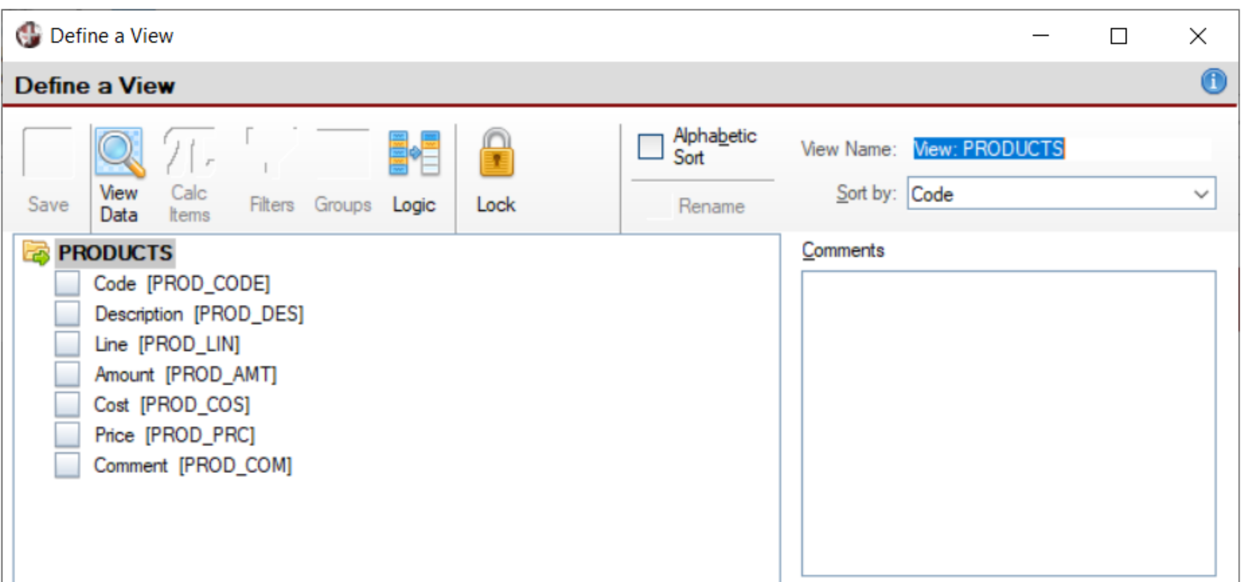


The **View Maintenance** panel is rather simple. To create a view, we select the data source (identified with the little drum or container icon).

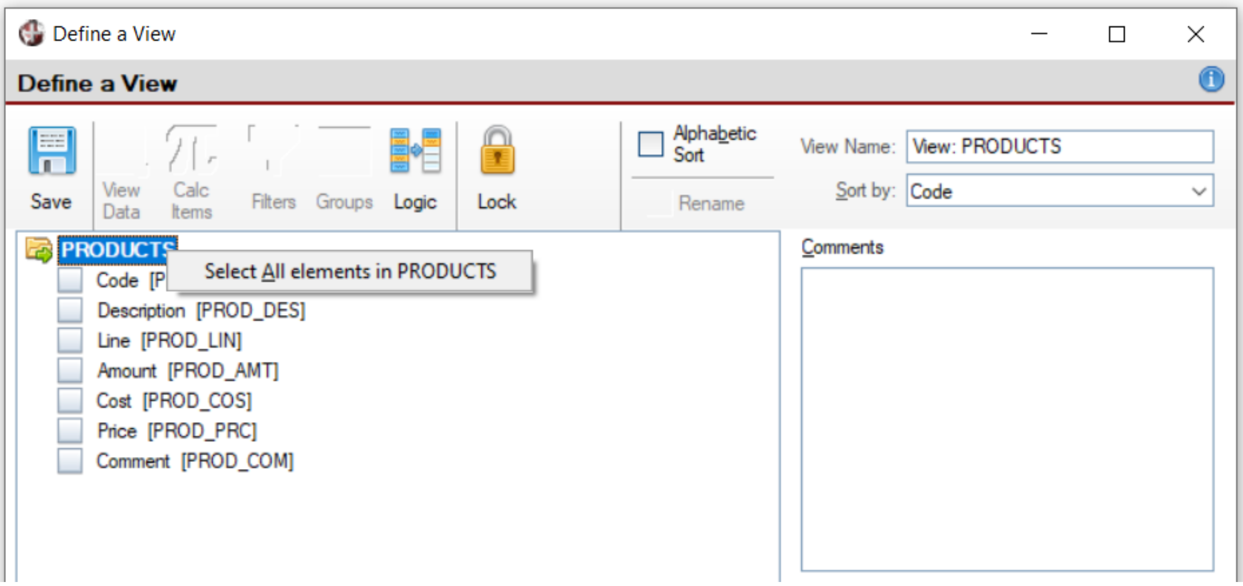
To see our two data sources listed, first double click on the **{Unassigned}** folder to open it. Then, double click or right click on **PRODUCTS**.

Double clicking displays a message that asks if we want to create a new view. Right clicking opens a contextual menu that allows us to choose the option [**Create a View from this Data Source**].

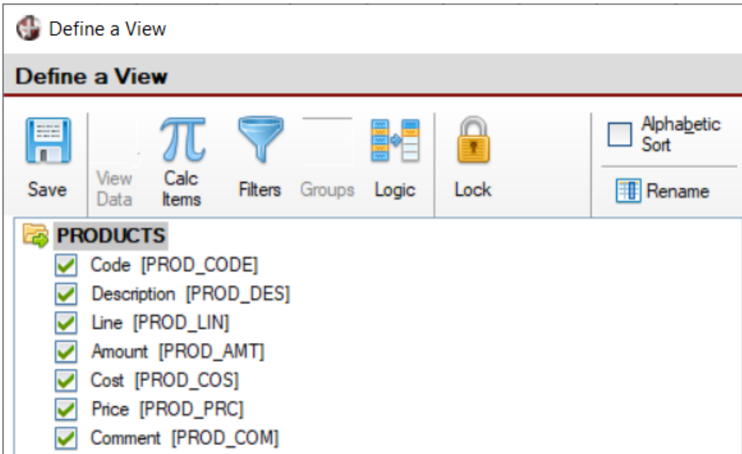
In both cases, the **Define a View** window displays where we must initially select all the elements (fields) of the data source.



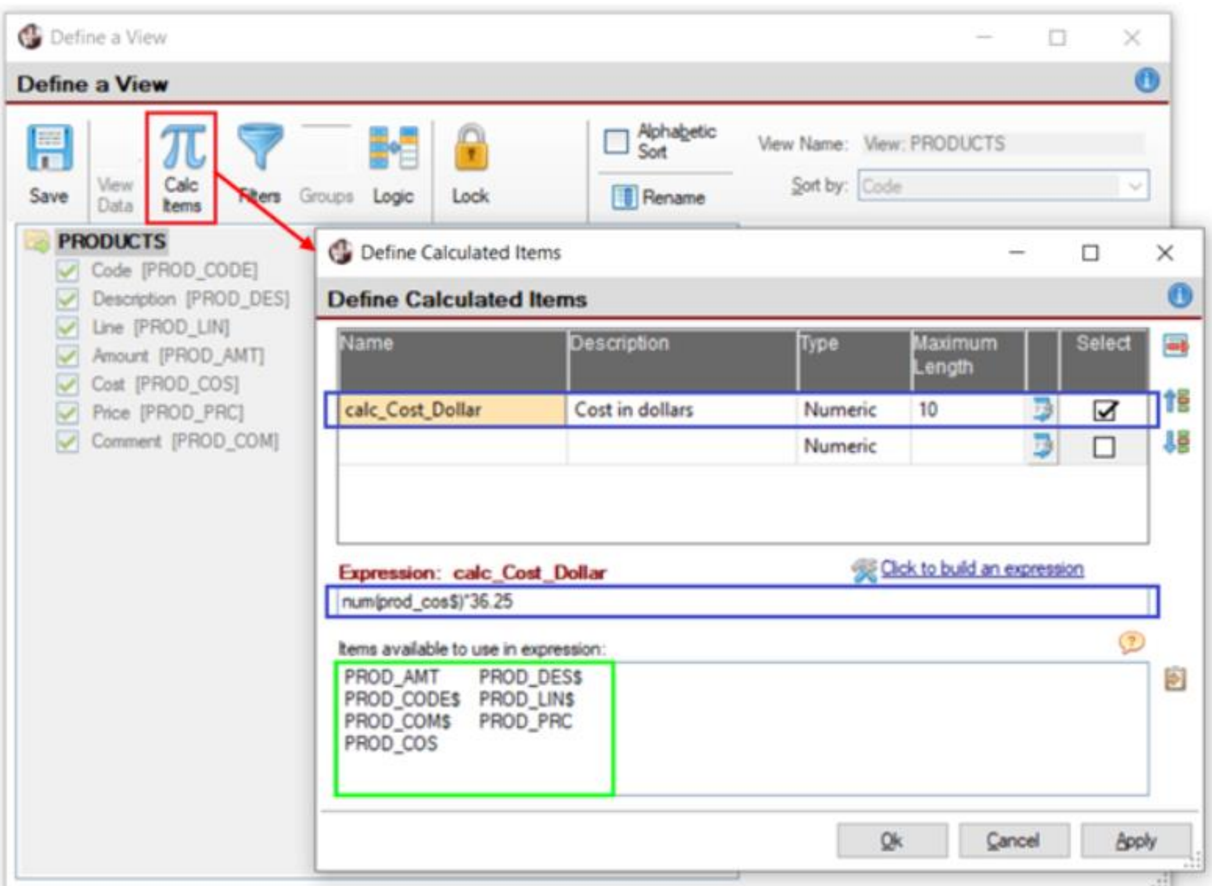
If there are many elements, we can right click on the name of the data source (in this case, **PRODUCTS**), and a context menu will appear with a [**Select All elements**] option.



We select this option to select them all. This places a check mark in the check box beside each element:



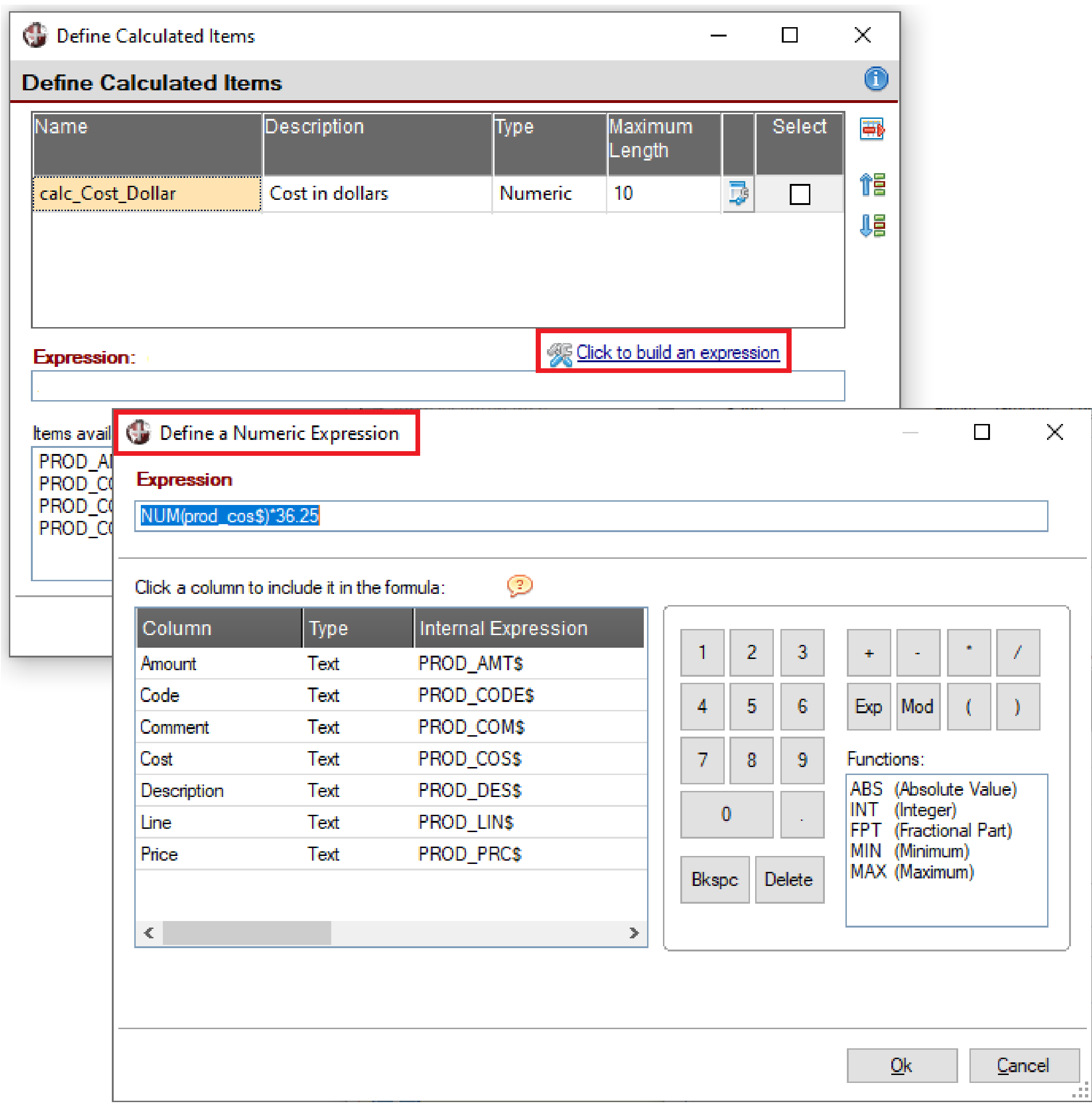
Now that all the elements have been selected, we are going to define a calculated element. Click on the [**Calc Items**] button located at the top. This opens the **Define Calculated Items** window, which is used to define fields that are not part of the data source but will be created from a specified calculation. This is an optional field.



We enter a field name (**calc_Cost_Dollar**) with a description (**Cost in dollars**), a type (**Numeric**) and a **Maximum Length** (**Example:** 10). This calculation will be based on the elements that make up the data source and appear in the box (marked in green above). The formula that will be used to calculate the new element (**calc_Dollar_Cost**) is written in the [**Expression**] input box (marked in blue).

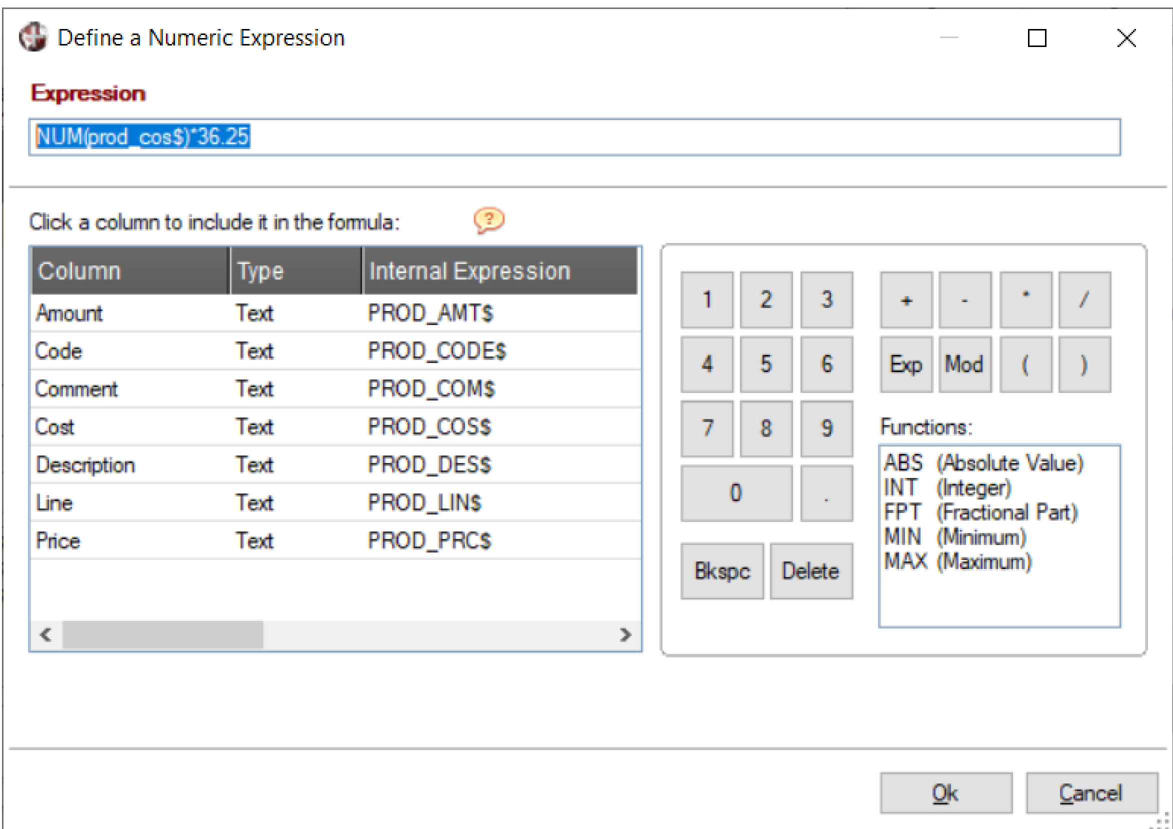
In this example, we are defining an element that is the multiplication of the element **PROD_COS$**, converted from literal to numeric with the function **NUM(),** by the value 36.25. This is used to calculate the cost of the product in another currency.

In some cases, we need to fine tune the way we are going to build the calculation. To do this, we must select the [**Click to build an expression**] option, which will bring up an additional panel with information and functions relevant to the construction of the new expression:

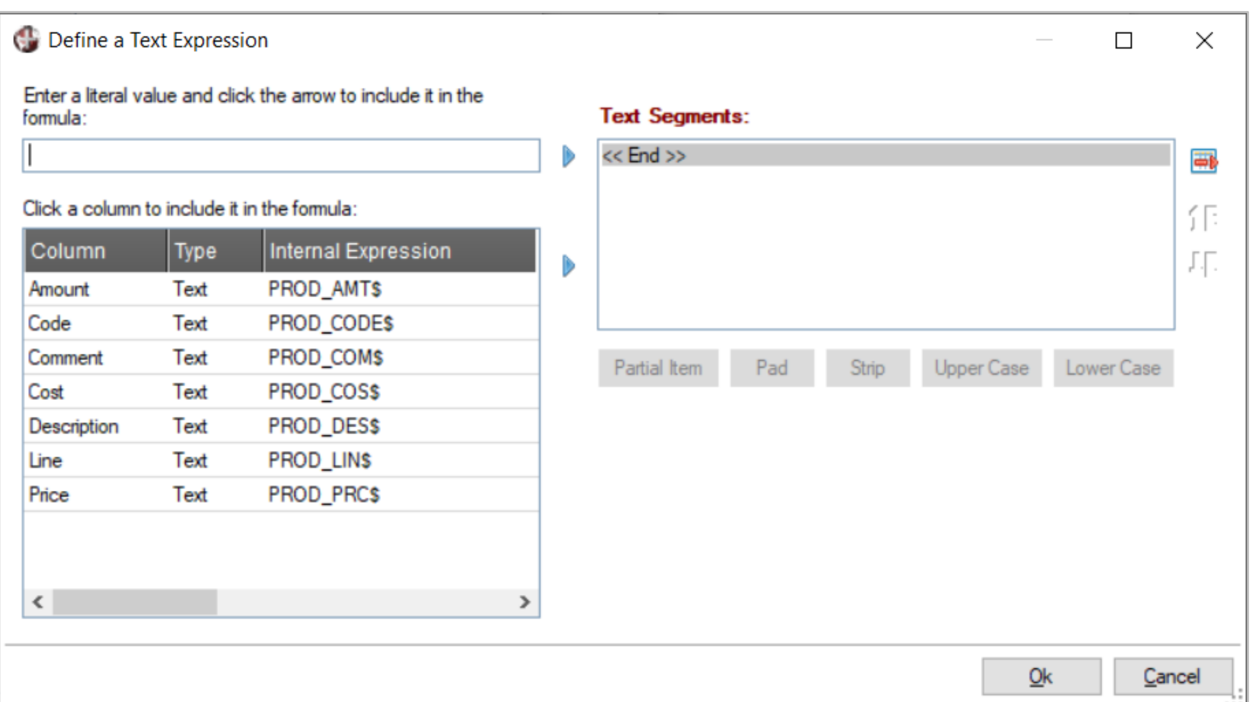


This panel will change depending on whether the calculated item is defined as numeric or text. Let's look at both screens.

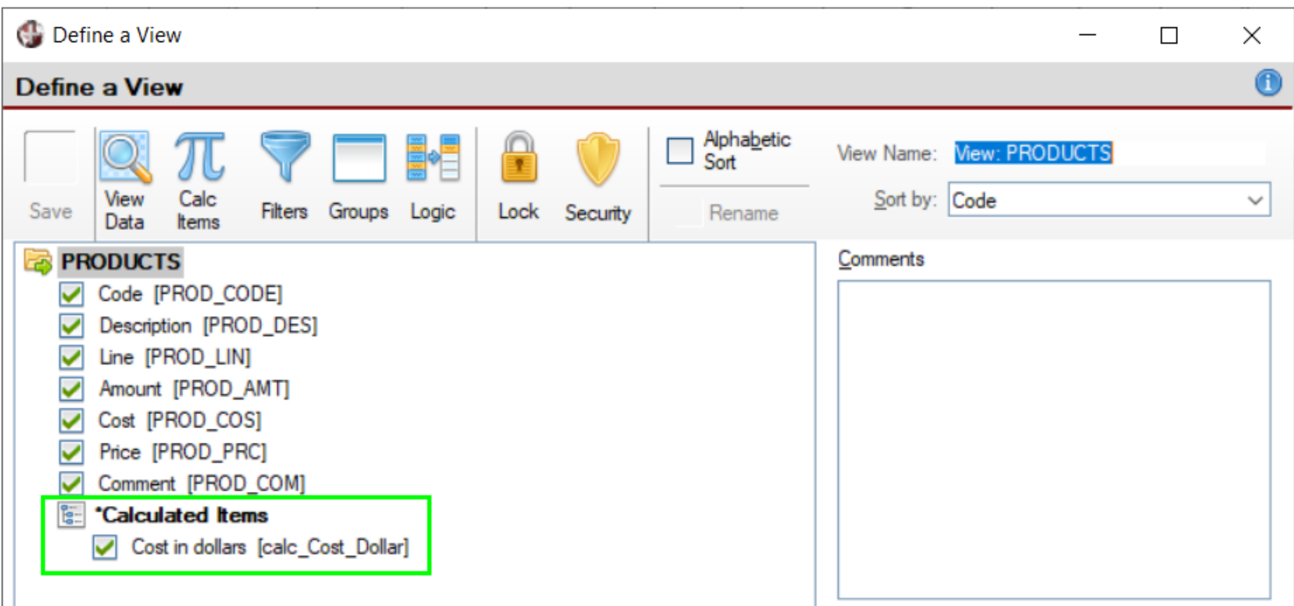
Definition of numeric expression



Definition of literal expression

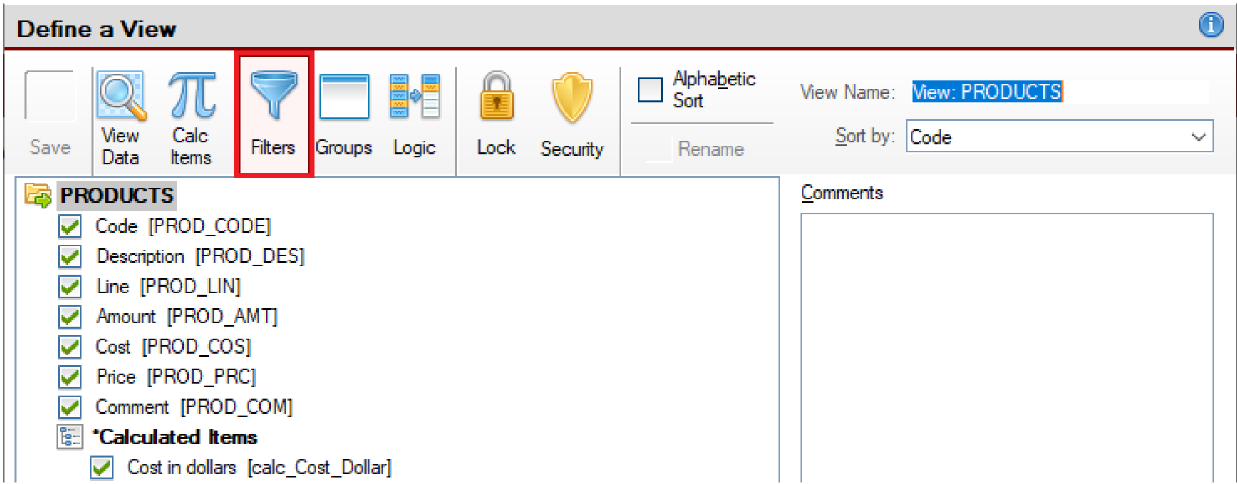


Once the calculated element(s) are defined, they will become part of the view and can be used in it.

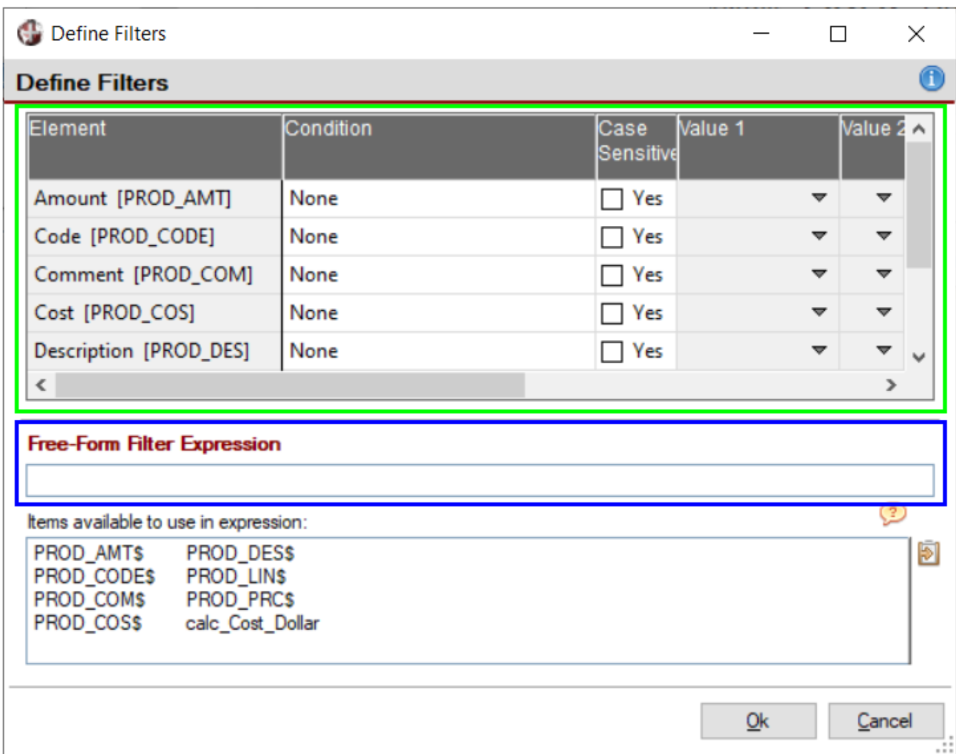


Note: The order of appearance/definition of the calculated elements is important if any of the calculated elements depends on another calculated element. In this case, it is necessary that the calculated element that will make use of the other is defined ***after the element to be used***; that is, calculated elements that make use of other calculated elements must be defined at the end of the list.

Once the calculated elements are defined, we can define filters or record selection criteria. This screen is executed by selecting the [**Filters**] button in the **Define a View** window.

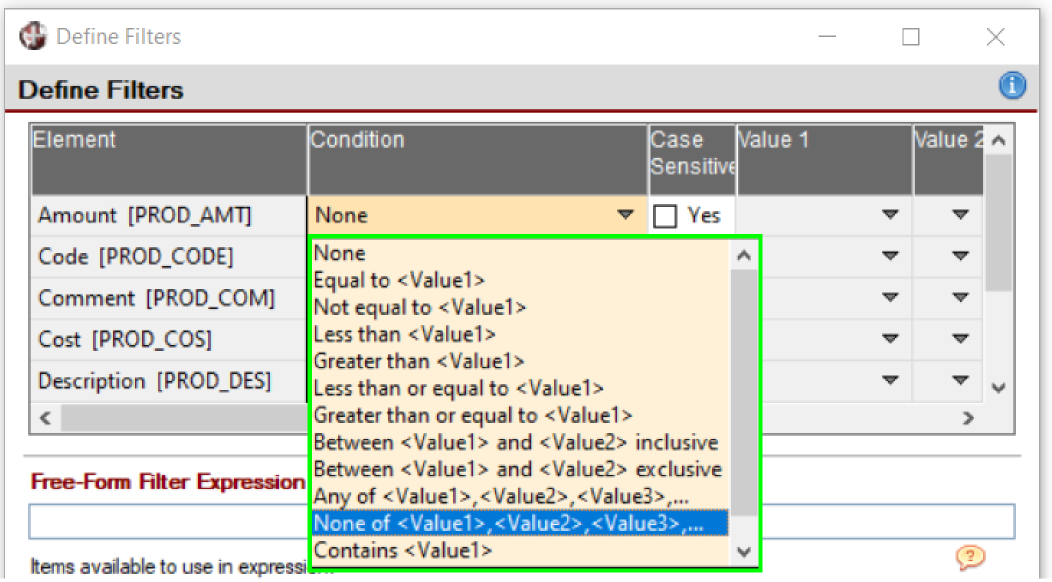


The **Define Filters** window basically has two parts to filter: one part where you can establish conditions for the different elements (marked in green) and one part where you can write a condition or filter expression by using the [**Free-Form Filter Expression**] option (marked in blue):



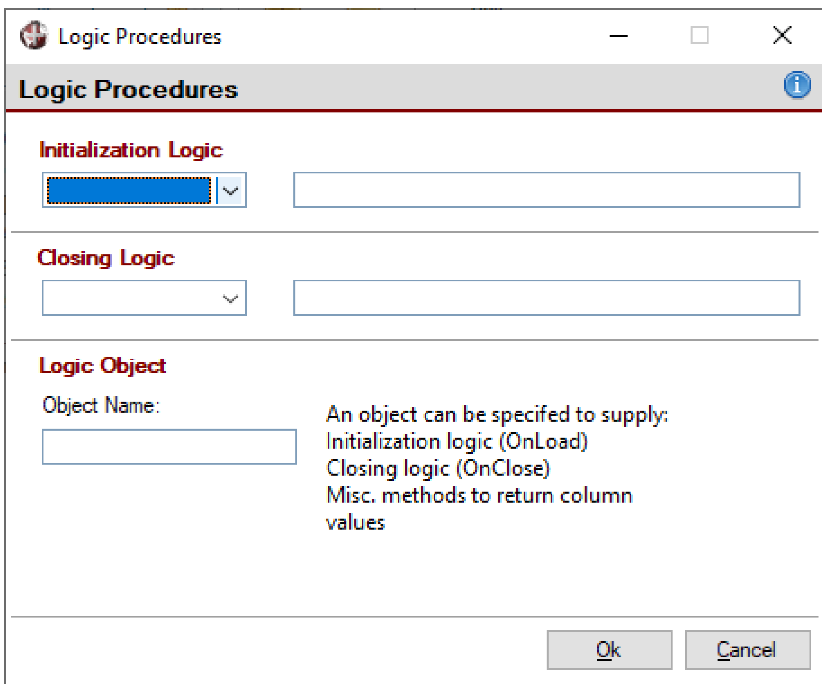
The top part depends on an element, a condition and one or more comparison values.

The conditional tests would be: None, Equal to <Value1> , Not equal to <Value1>, Less than <Value1>, Greater than <Value1>, Less than or equal to <Value1>, Greater than or equal to <Value1>, Between <Value1> and <Value2> inclusive, Between <Value1> and <Value2> exclusive, Any of <Value1>, <Value2>, <Value3>.., None of <Value1>, <Value2>, <Value3>…, Contains <Value1>.

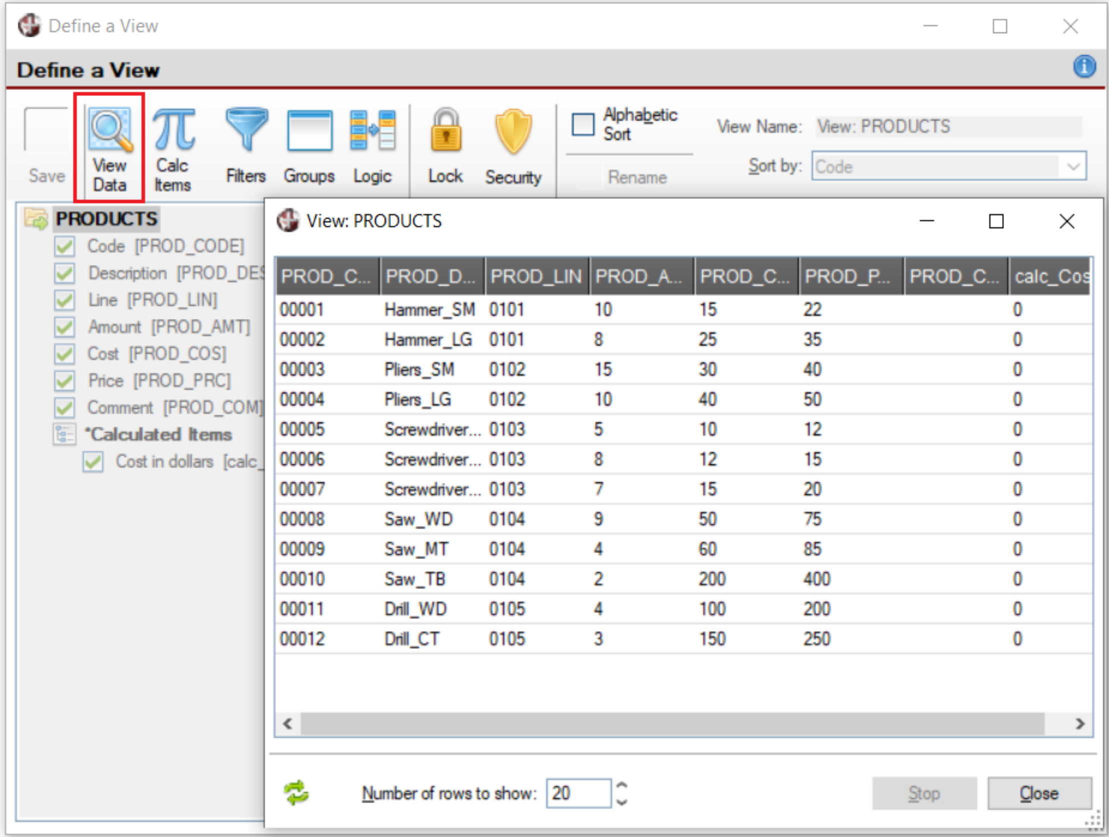


This group of actions should be enough to establish many record selection criteria, but if not, it is possible to write a criterion to our requirements.

As with data sources, it is possible to specify some actions by using the [**Logic**] button, which is located at the top of the **Define a View** window, to open the **Logic Procedures** window where we can initialize and/or close logic when executing our view. We will leave this option as is.

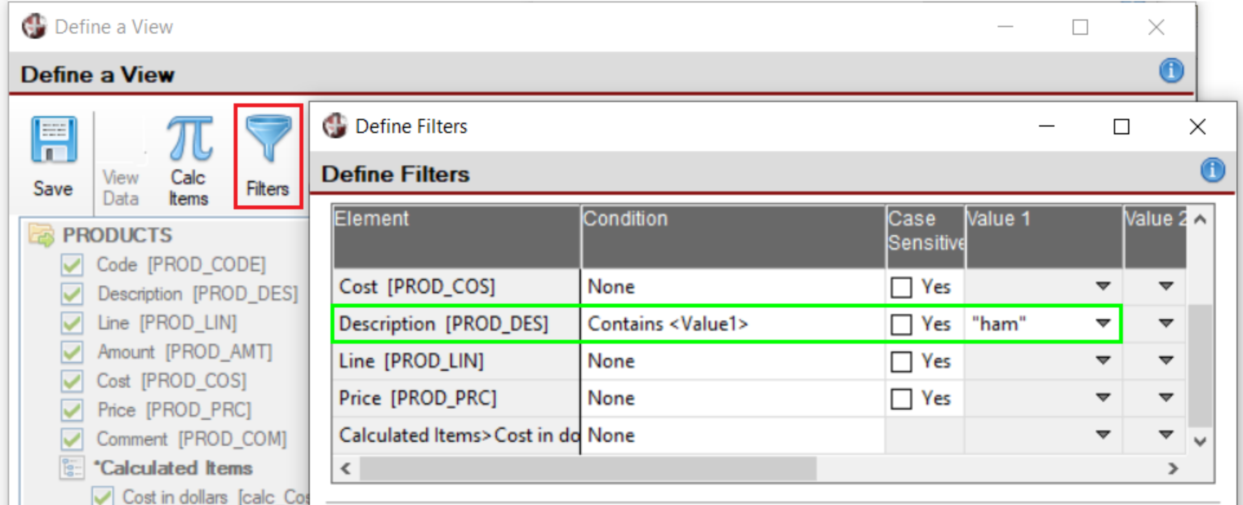


Once we have defined our view, we can save it and test it by using the [**View Data**] button at the top (marked in red):

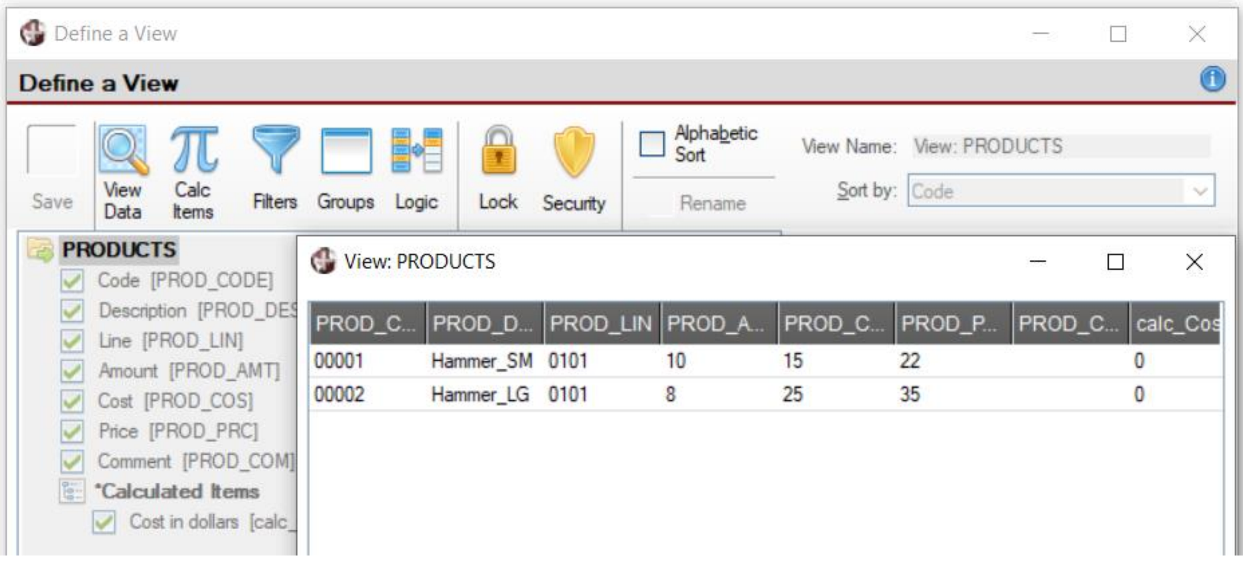


Note: We have cropped the screen to save space. We have done the same thing in many of the screens shown.

We have defined a filter so that it shows only product descriptions that begin with the letters "ham", such as "Hammer":



To see the results, save the view by clicking the [**Save**] button at the top and then click the [**View Data**] button. We see two records with product descriptions beginning with "ham":



Remember that there is an object interface to perform these operations programmatically. For now, we are going to continue seeing the powerful tools that PxPlus offers.

9. The Report Writer

The Report Writer is used for the design and generation of reports. Both the report designer and the end user can create and manipulate the report content with the simplicity and functionality of a spreadsheet.

Some bold features are drag and drop data, resizing in columns and rows, calculated values, cell formatting (font, color, alignment, borders), image support, sorting methods, grouping and breaking routines, data filtering and other settings.

The data input can be any native PxPlus file or any data source. The output can be to a printer, a PDF (Portable Document Format) file, the internal viewer, the Windows Clipboard, an HTML file, or some other output object (previously defined).

PxPlus offers two methods to design a report: the **Report Designer** for complete control or the **Report Wizard** for quickly and easily generating a report.

The **Report Designer** is the primary user interface for designing the report, its structure, the data, sorting rules, selection criteria for data filtering, etc. This format is saved in report definition files, which are then used to generate the report.

The **Report Wizard** provides a simple way to create a new report definition by guiding you through a series of eight interactive steps.

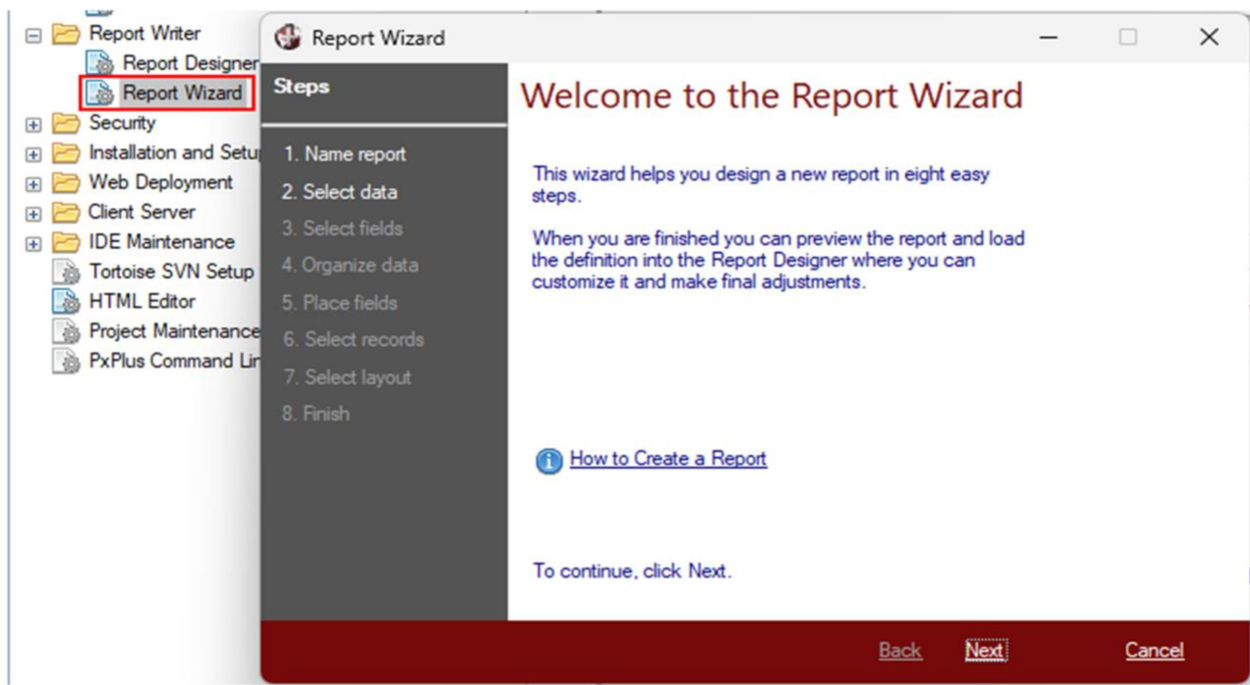
A report can be generated directly from the Report Designer itself or with a special program or through the **pvxreport** object using the **RunReport()** method.

Refer to **Report Designer**, **Report Wizard** and **Generating a Report** in the PxPlus Help documentation.

Exercise: Using the Report Wizard

The **Report Wizard** is available from the PxPlus IDE main menu by opening the **Report Writer** category and selecting **Report Wizard.**

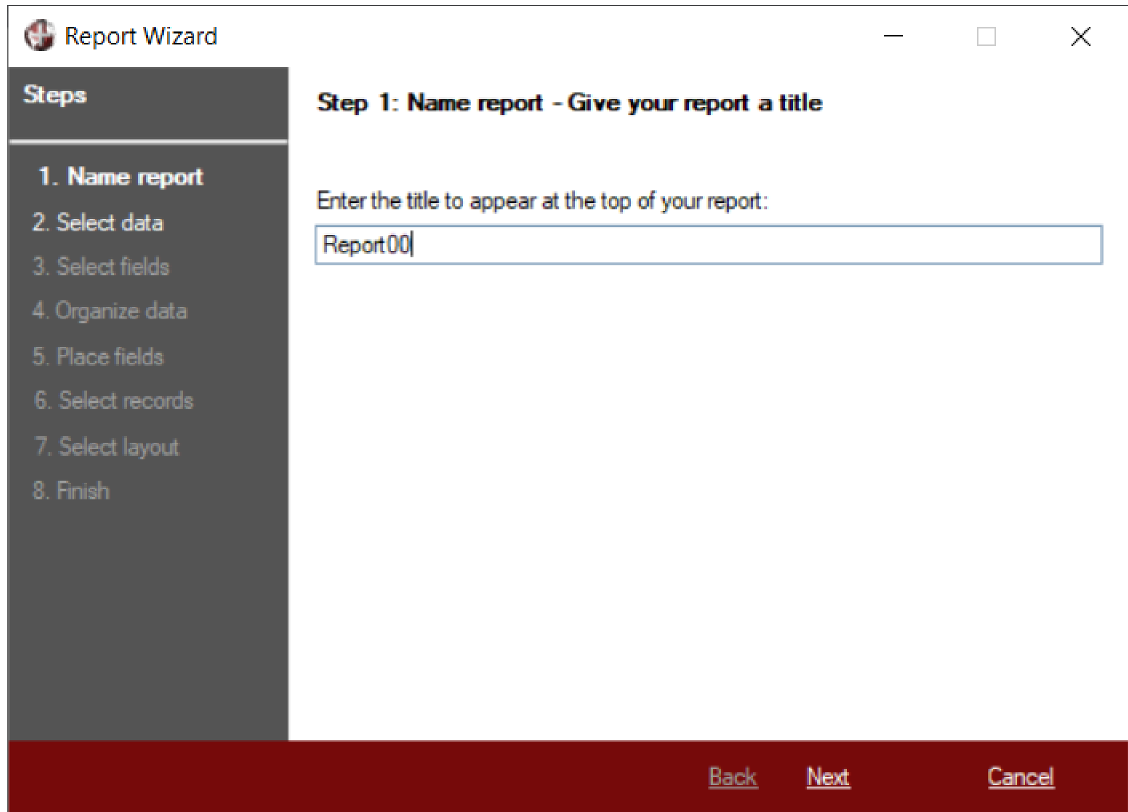
Selecting this option displays the **Report Wizard Welcome** panel:



Click the [**Next**] button to advance to **Step 1: Name report.**

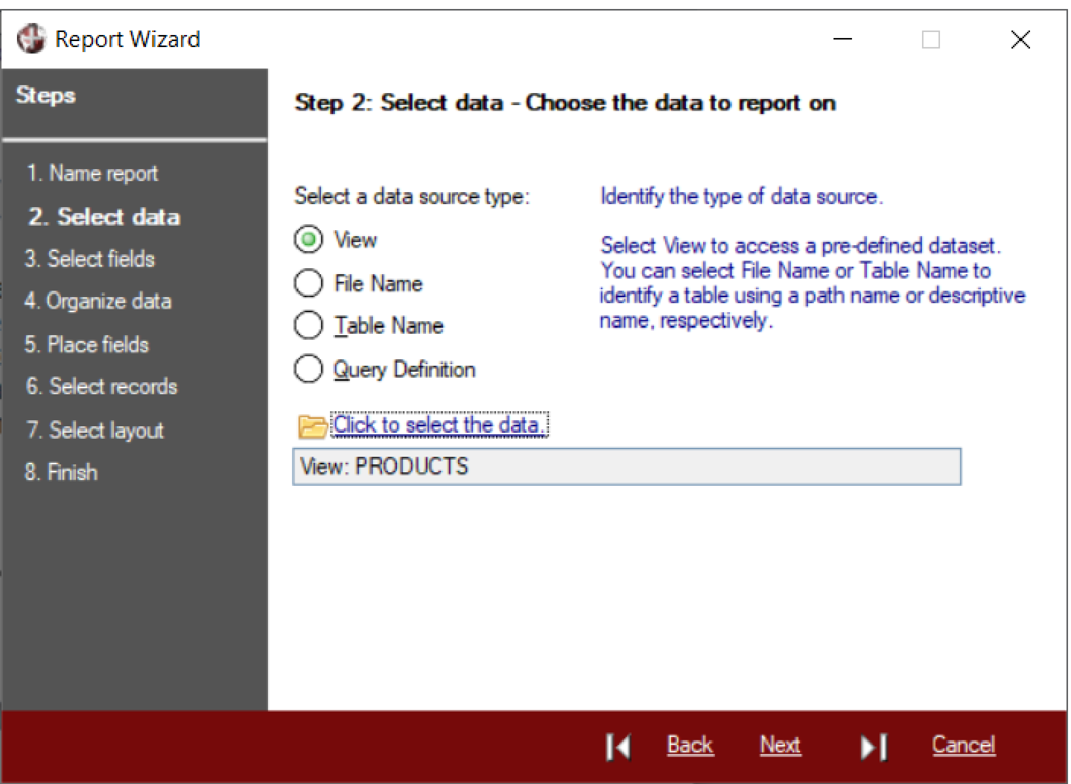
The first step in designing a report is to specify its name.

Enter **Report00** and then click the [**Next**] button to continue to the next step.



Step 2: Select data is the selection of the origin or source of the data. Remember that a report can be supplied with data that is in a view, in a file, in a table or in a Query definition. For now, let's select the **Product** view we just defined.

The **View** option is selected by default, so click the blue hyperlink option [**Click to select the data**] and select the **Product** view.



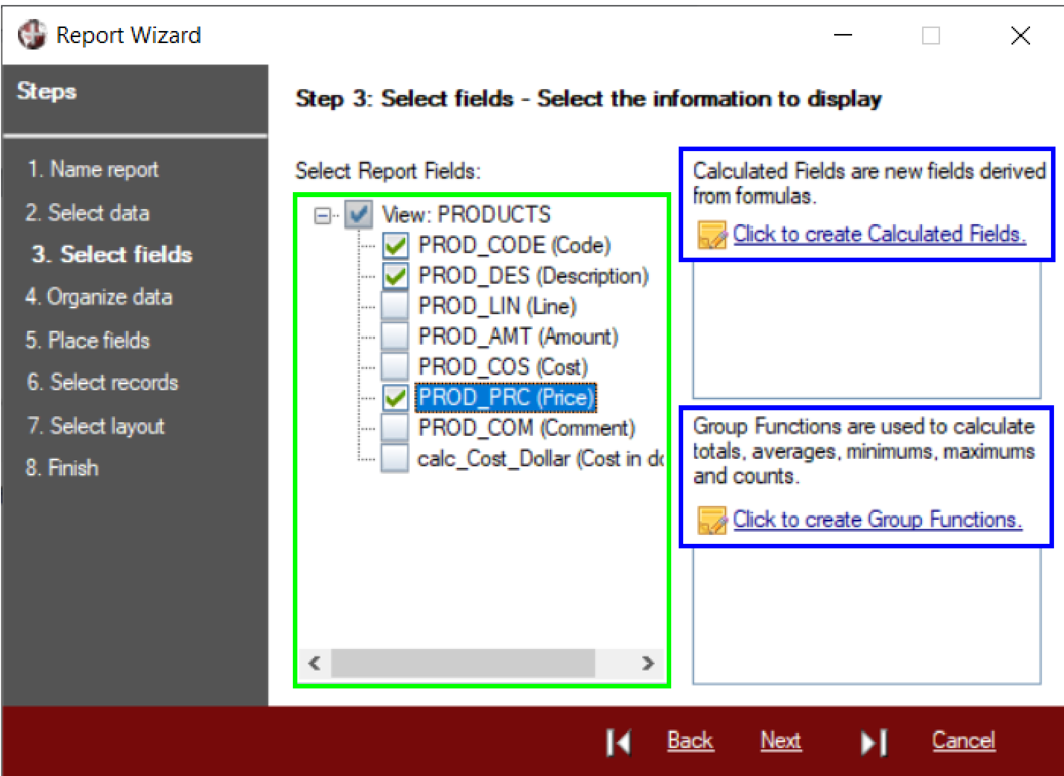
You can scroll forward or backward using the scroll buttons at the bottom.

Click the [**Next**] button to continue.

Once the data source has been selected, **Step 3: Select fields** is the selection of the table elements or fields that will display in the report (marked in green). In our case, we have selected only three fields: PROD_CODE, PROD_DES and PROD_PRC.

This step also gives us the possibility to create **Calculated Fields** (marked in blue) to incorporate values created from the application of a formula or the manipulation of some data.

We can also take advantage of **Group Functions** (marked in blue) to calculate sums, averages, minimums, maximums and counts.



The generation of a simple report may not require Calculated Fields, but we are going to review the creation of one of them.

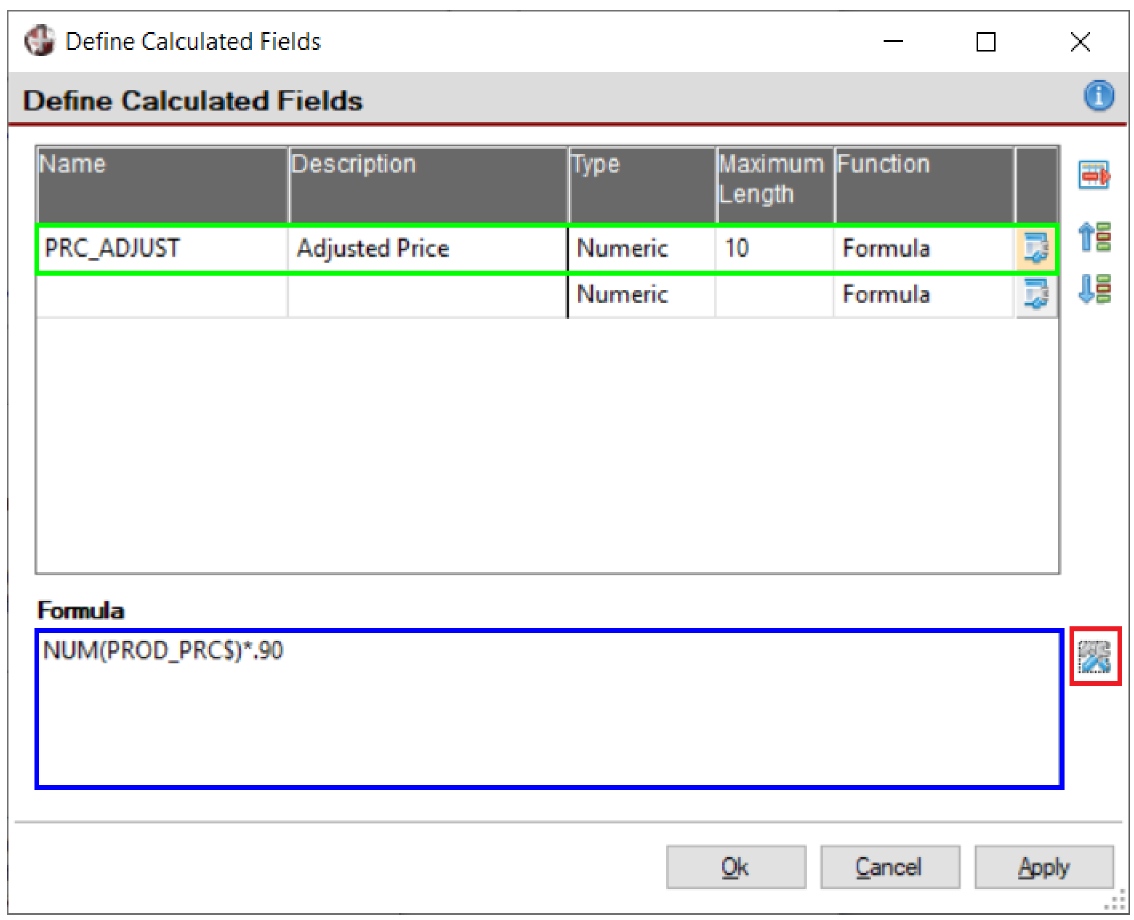
Exercise: Defining a Calculated Field (Numeric)

Click the blue hyperlink option [**Click to create Calculated Fields**]. We are already familiar with the **Define Calculated Fields** window!

At the top of this window (marked in green below), we specify the name of the field, its description, the type of calculation (numeric or text), its length and whether it will be a formula or a translation.

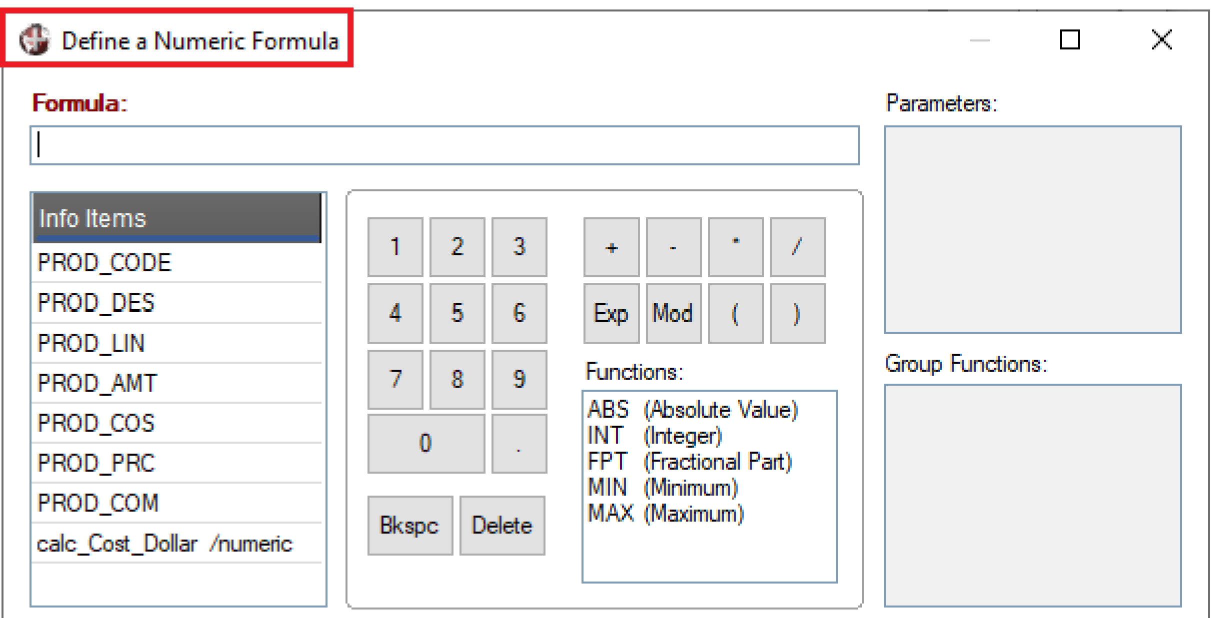
Enter the following details:

Name: PRC_ADJUST
Description: Adjusted Price
Type: Numeric
Length: 10
Function: Formula



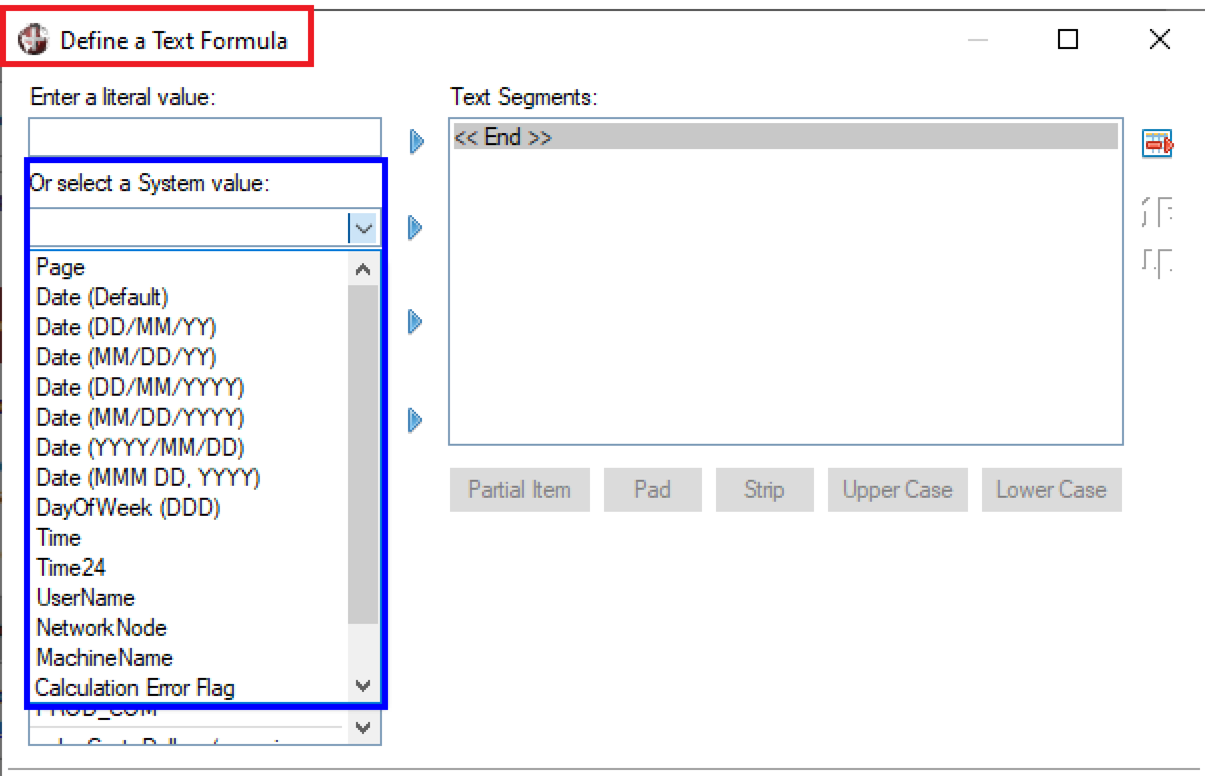
In the area at the bottom (marked in blue), we can enter the formula that will give us the value of the calculated field, or we can click the button with the tools icon located on the right (marked in red) to open the expression definition window.

In this case, because we had specified that the **Type** of calculated field is **Numeric**, clicking the tools icon opens the **Define a Numeric Formula** window. This window shows the data source elements, the calculated elements, and a series of functions and parameters (the parameters can be requested from the user at the time of running the report), as well as group functions to apply if available.



Our numeric formula to calculate the field will be 90% of the sales price of the product. Since we have the price in literal format (text), we must use the **NUM()** function to convert it to numeric and be able to perform calculations with it.

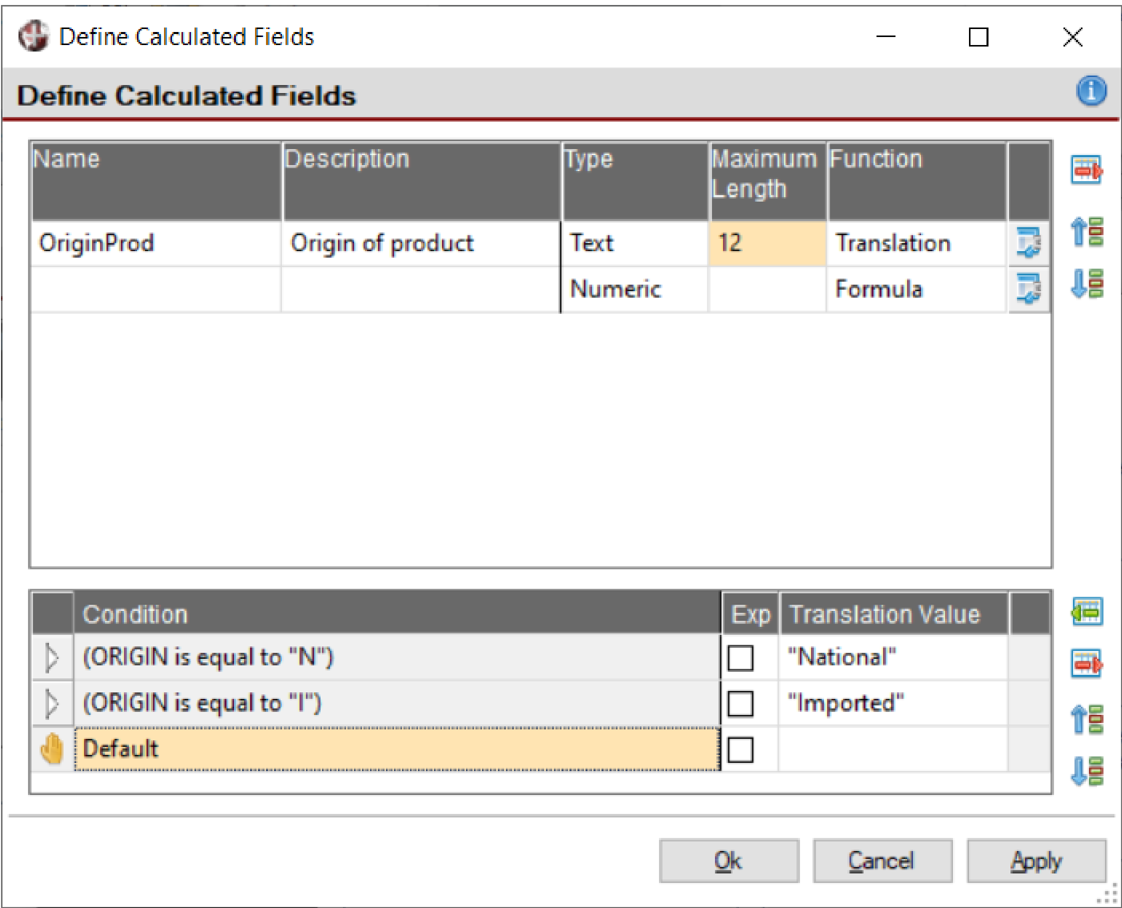
If we had specified that the type of calculated field is **Text**, clicking the tools icon would have opened the **Define a Text Formula** window. This window would have changed to incorporate appropriate functions for the handling of literals, such as substrings (use only part of a text), fill, convert to upper or lowercase, as well as use a series of System values, which can be selected from the **select a System value** drop-down box on the left. Among the System values that can be selected are page number, date, day of the week, time, user name, network node, machine name, and so on.



Now, let's return to the **Define Calculated Fields** window. We can change a default value for another. **Example:** If we have an ORIGIN field that has the values "I" for an "Imported" product and "N" for a "National" product, we can create a translation table where we change the value of that value for another literal. With this, instead of having an element in our report that is "I" or "N", we will have "Imported" or "National", which will definitely make our report much clearer.

Example: Defining a Calculated Field (Text)

Let's look at an example definition (not applicable to this table; only used to illustrate how it works):



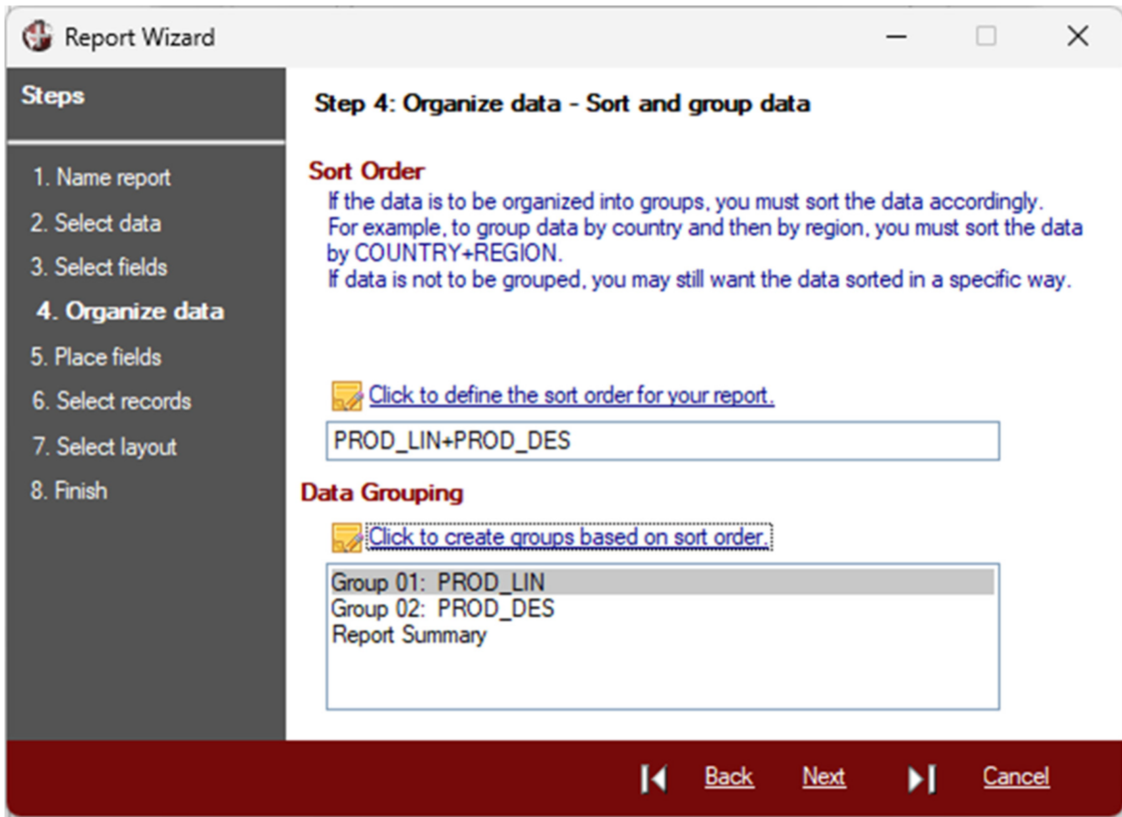
Basically, with this type of translation, we have an important part covered in case our data has information that needs to be explained or expanded. **Example:** Another example could be that the data has "F" or "M" registered, but our report could show "Female" or "Male".

Once the element selection part of the report is completed, click the [**Next**] button.

In **Step 4: Organize data**, we continue with the data sorting where we can make different breaks. For this, we can select the behavior of the report if it find other types of data.

Example: If we have a "Province" field, we can tell the report to do a break and a total when the Province code changes. For this, it is necessary that there be a key or sorting method with the "Province" field.

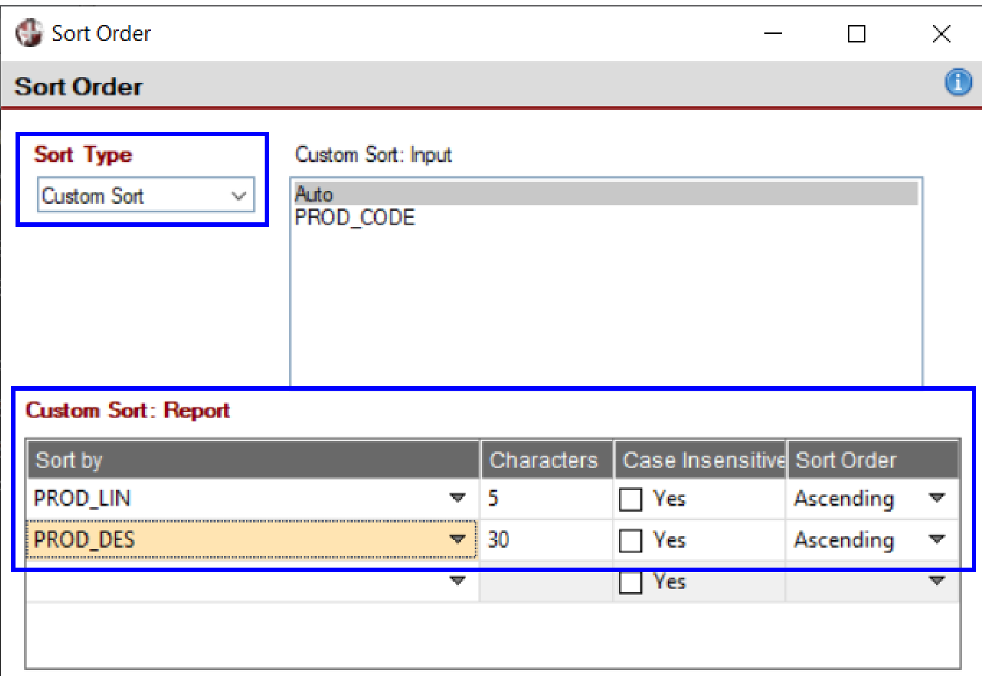
In our View example, we have several sorting methods defined, and we can select different groups (each group can have its header and footer, at the beginning and at the end, with sums, averages and other values).



Click the blue hyperlink option [**Click to define the sort order for your report**].

In the **Sort Order** window, change the [**Sort Type**] option (marked in blue in the upper left part of this window) to **Custom Sort**.

Then, in the bottom part of this window, select the elements to define a **Custom Sort** sequence. First, select PROD_LIN and then select PROD_DES. Click the [**OK**] button.

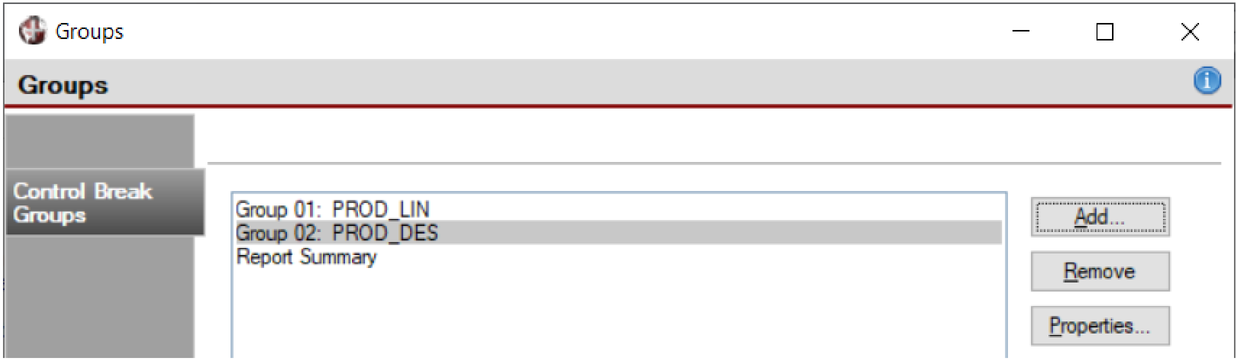


With the **Sort Order** defined, click the blue hyperlink option [**Click to create groups based on sort order**] located on the **Step 4: Organize data** panel.

In the **Groups** window, we see the **Control Break Groups** tab on the left. Click the [**Add**] button.

This opens the **Group Definition** window. First select Group 01: PROD_LIN and click the [**OK**] button. Then, click the [**Add**] button again and select Group 02: PROD_DES and click the [**OK**] button.

We see the two data groupings defined:



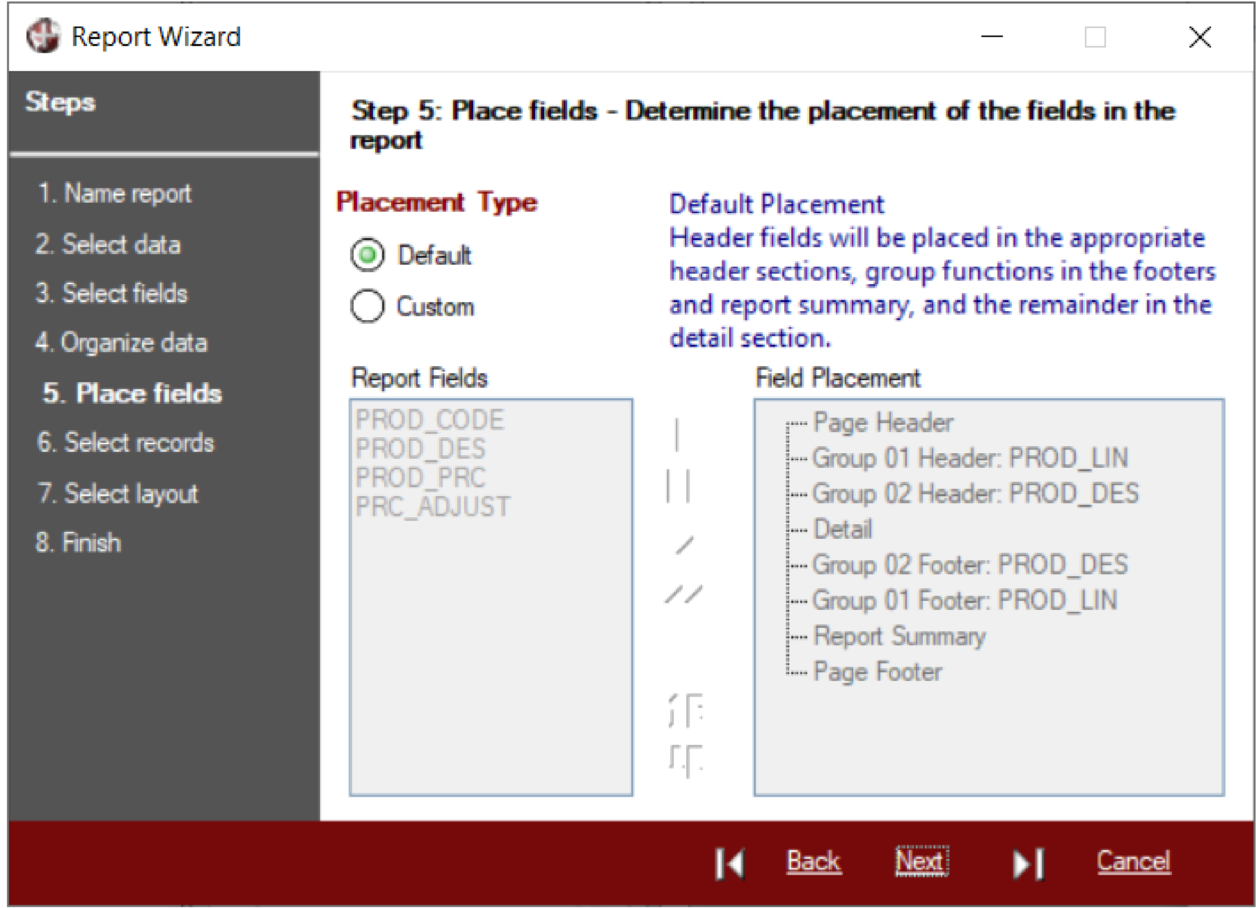
Click the [**OK**] button to close the **Groups** window. We are returned to the **Step 4: Organize data** panel.

Click the [**Next**] button to continue.

The **Step 5: Place fields** panel allows us to determine the location of each field in the report, according to the ordering and grouping scheme defined in the previous step.

In this case, we can leave this in the hands of the report generator (with the **Placement Type** option set as **Default**) or change it to our convenience by selecting **Custom**.

We are going to leave this set as **Default**.



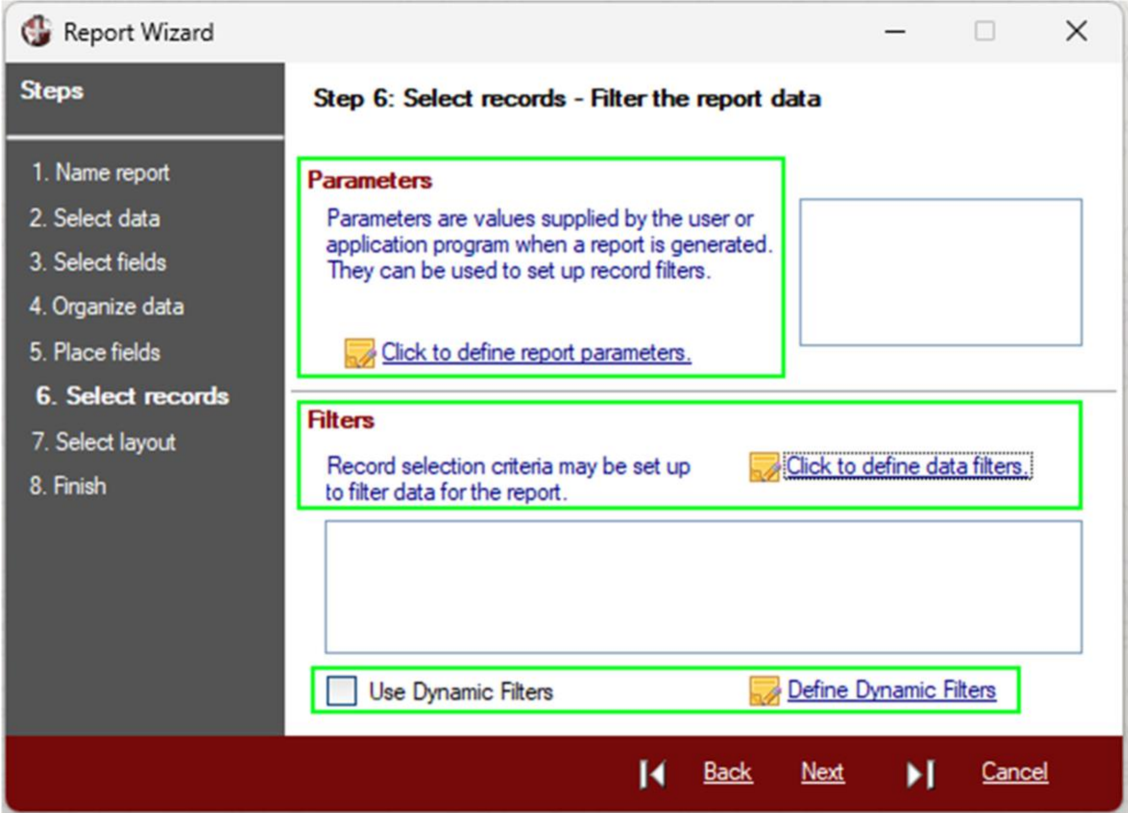
Click the [**Next**] button to continue.

The **Step 6: Select records** panel allows us to specify which records will be captured in the report and allows the designer to ask the user for information when executing the report (for generation/printing).

In addition, it is possible to define filters to select the information to be displayed. These filters can be **Static** (defined by the report designer) or **Dynamic** (defined by the user at the time of generating the report).

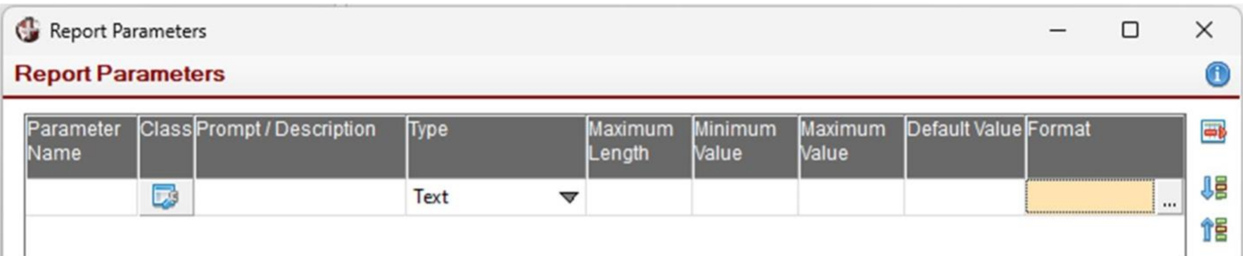
Example: A parameter could be asking the user the region to which the report will refer or to which year they want the sales statistics to be carried out. These parameters can be numeric or literal (text), have minimum and maximum values and even have an input format.

Note: As we will see later, the validation and characteristics of the parameter could be given by the (previous) definition of a data class, so that by saying that the parameter in question belongs to the mentioned class, it will "inherit" all the aspects of it.



Click the blue hyperlink option [**Click to define report parameters**] to define the report parameters (first box marked in green).

Clicking this link opens the **Report Parameters** definition window, which is composed of the following values:



Value	Description
Parameter Name	Name of the parameter. This value could be used in the report.
Class	Class of data. If any class was defined, we could indicate that the data is of that type.
Prompt/Description	Display message for the user when asking for the parameter.
Type	Internal type, numeric or text (literal).
Maximum Length	Maximum length in characters.
Minimum Value	*(Optional)* Minimum value.
Maximum Value	*(Optional)* Maximum value.
Default Value	Value to accept if the user does not enter anything.
Format	Input format.

These parameters can be expressions and PxPlus variables, so that they are totally dynamic.

Example: You can make the maximum length as **=%valuemaxparam** and thus the report generator will change the maximum value according to the content of the global variable (**%valuemaxparam**).

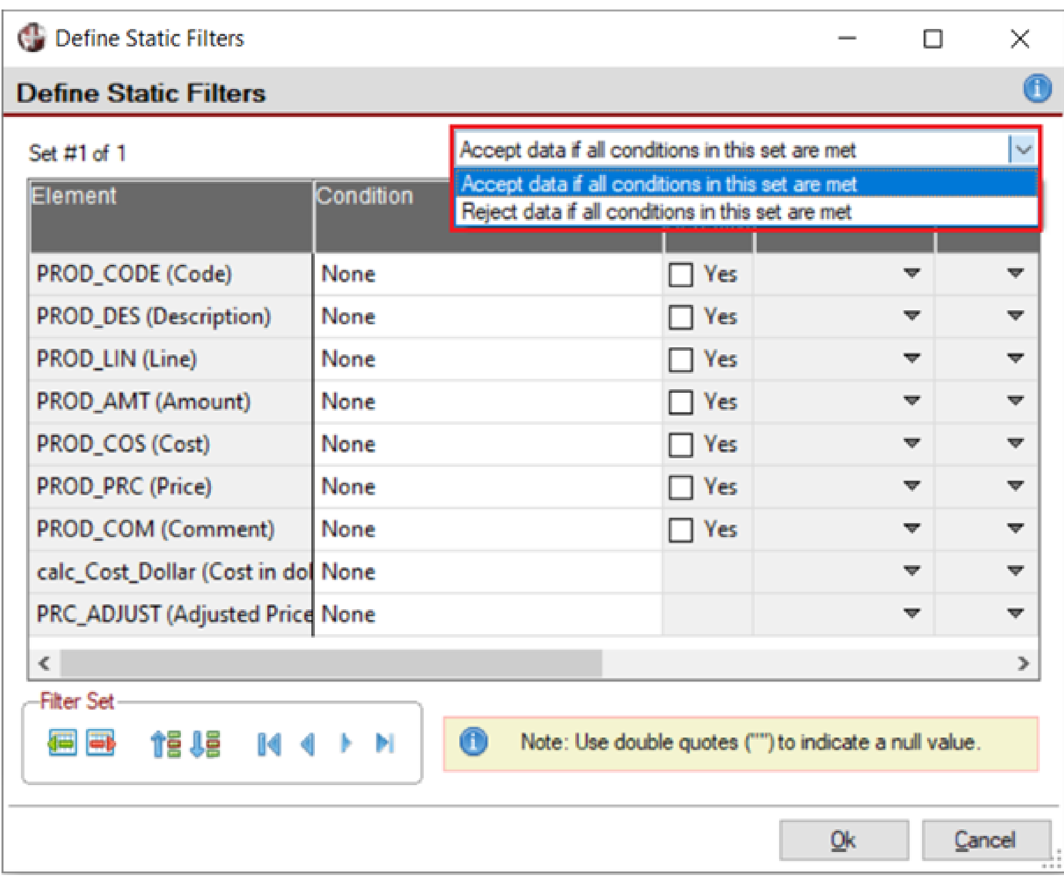
The format allows you to filter the entry. **Example:** Specifying a format such as 00/00/0000 will only accept numeric digits for the date.

Important Note: Beyond the minimum, maximum, type and length, there are no other validations on the user's input. You must be explicit in this or obtain the information from the application that the user is running to avoid possible failures in the information generated as a result of a wrong parameter at the time of executing the report.

The parameters can be supplied programmatically through the reports object (***rpt/pvxreport**) or by defining a Custom Interface.

Once the system parameters have been defined, we can define the **Static Filters** where we can decide what type of condition certain elements of the list must meet before being processed or displayed in it. This interface is also known to us, and we can basically choose an element, a possible condition, and one or more values to compare.

Tip: Note that many of the components of PxPlus are common to different tools, so we suggest that you become very familiar with some of them in order to get the most out of that tool.

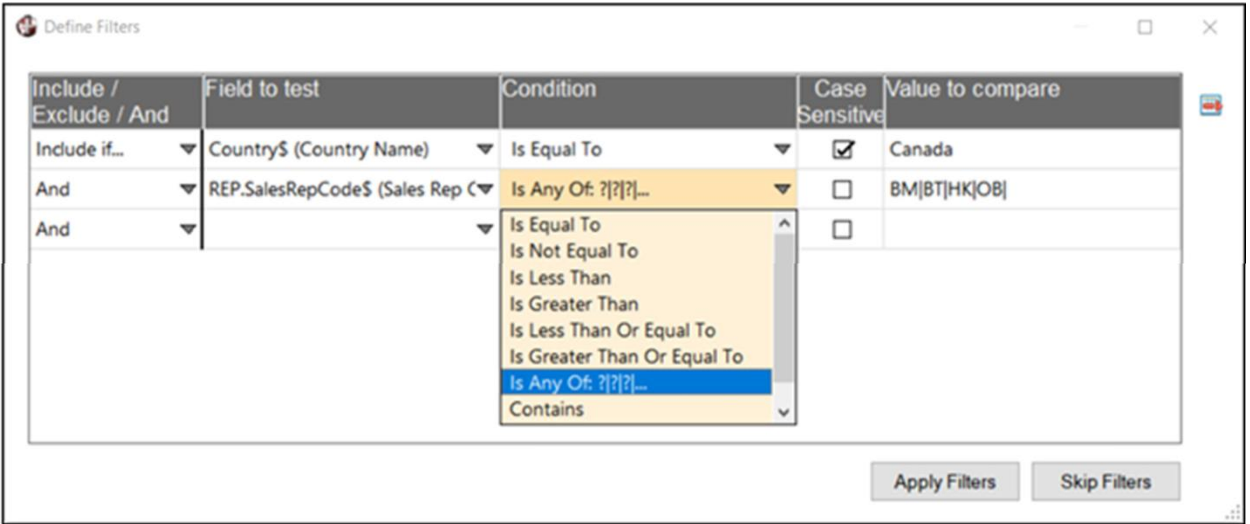


It is important to know that you can have different sets of filters and select different conditions that they must meet (or should not meet) in order to select the data. From the drop-down list at the top (marked in red above), you can select [**Accept data if all conditions in this set are met**] or [**Reject data if all conditions in this set are met**]. It is possible to specify one set of filters that ***must be met*** and another set of filters that ***must not be met***.

It is also possible to specify two sets of filters that ***must both be met*** (**Example:** You could have a first set that specifies the product line between 1000 and 1999, and a second set that specifies the product line between 7000 and 7999).

You can define up to eight sets of filters, and the report generator will test each filter. If any of the filters has a condition that is met, that data will be incorporated into the report; otherwise, the next set will be evaluated for filters.

Dynamic Filters are basically similar to Static filters. Only the designer specifies which elements can be filtered, and at execution time, the system shows the user a **Define Filters** window similar to the one shown below:



The user will be able to select the elements to filter (within those previously selected by the report designer) and apply conditions and nesting of conditions (IF one condition AND/OR this other condition AND/OR this other condition, and so on).

Refer to **Data Filters** in the PxPlus Help documentation.

To execute the report, we have several ways. One way is:

```
call "*rpt/runreport;Run_Call","c:\system\myreport.pvr"
```

Where:

"***rpt/runreport;Run_Call**" is the name of the PxPlus utility that will execute the report. The first argument, "**c:\system\myreport.pvr**", is the name of the previously defined report.

For this exercise, we will not be defining any filters, so we will leave **Step 6** as is. Click the [**Next**] button to continue to the next step.

The **Step 7: Select layout** panel provides options for choosing the appearance of the report, such as **Orientation** (*Portrait* or *Landscape*) and **Layout** (*Simple View* or *Tabular View*).

We will leave **Step 7** as is and click the [**Next**] button to continue to the final step.

The **Step 8: Finish** panel provides options for selecting the output destination for the report. Prior to completing the wizard, click the blue hyperlink option [**Click to preview the report using the report viewer**] to view the results. You can go back to previous steps to make any necessary changes.

When done, click the [**Finish**] button. The resulting report definition can be loaded into the Report Designer where it can be modified as needed and saved.

To learn more about the Report Designer, refer to the section **The Report Designer: An In-Depth View** within this book and also refer to **Report Designer** in the PxPlus Help documentation.

10. BUTTON, CHECK_BOX, RADIO_BUTTON: A Deeper Explanation

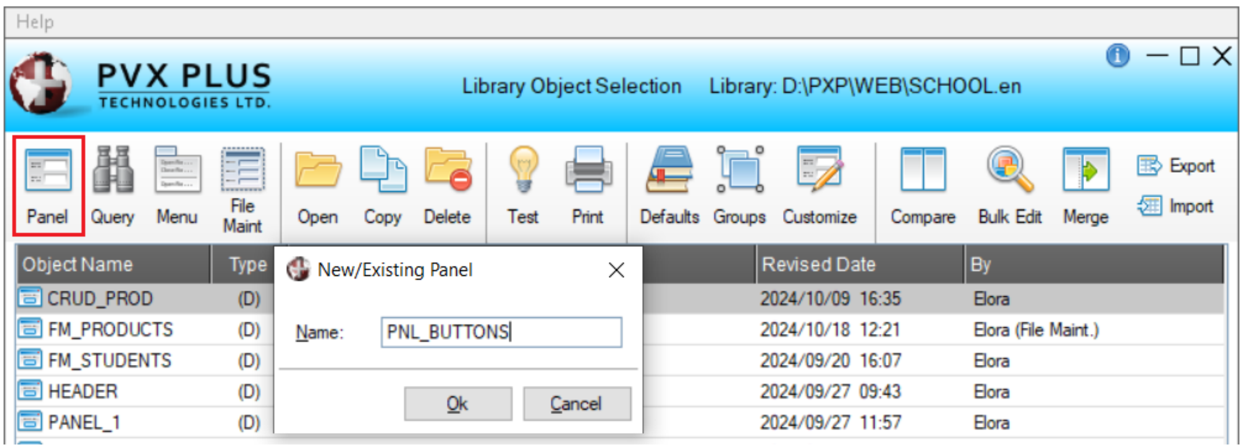
Previously, we mentioned the use of these controls:

- **BUTTON** control where we can associate an action with the use of a button.
- **CHECK_BOX** control that allows activating or marking a possible value.
- **RADIO_BUTTON** control that allows you to select one alternative among several.

In this section, we are going to explore other options and attributes of these controls to give our work additional functionality or appearance.

While the primary purpose of the **BUTTON** control, for example, will always be the execution of an action, it is possible that its appearance may be altered to hide it or make the user feel more comfortable with it.

If we want to practice creating these controls, we can create a new panel called **PNL_BUTTONS** in the **SCHOOL.EN** library.



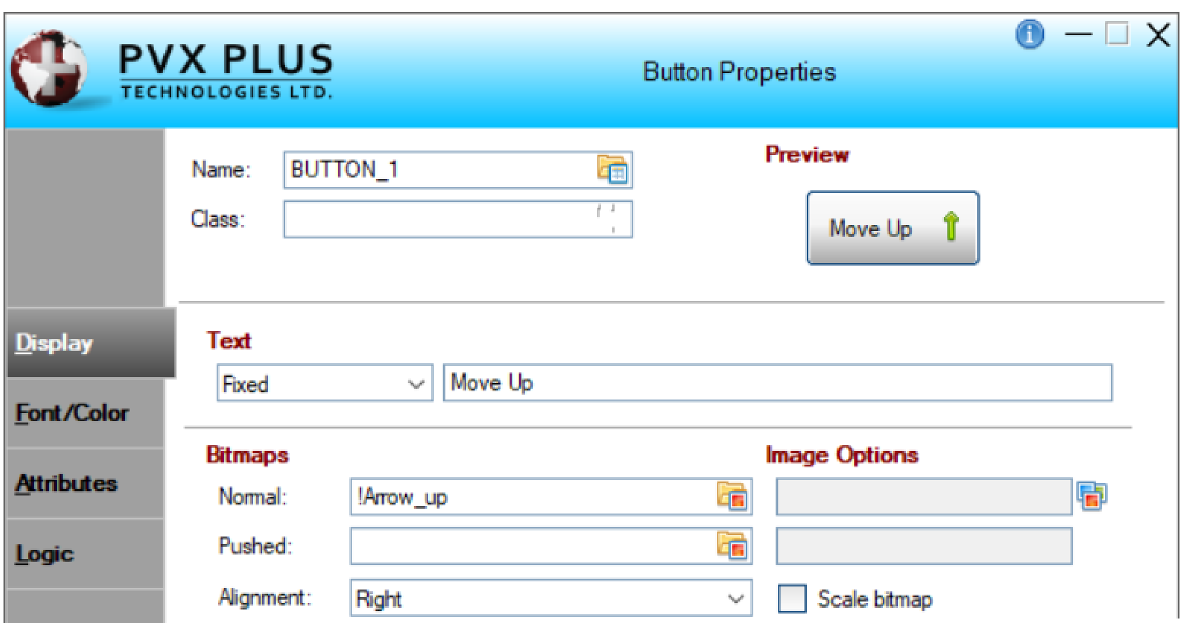
When the design panel displays, go to the panel properties window by selecting the [**Header**] option and activate the [**Auto Refresh**] attribute to guarantee that the changes made will be displayed automatically.

BUTTON Control: Other Functions and Attributes

Let's look at the characteristics of the **BUTTON** control. We will start with the **Display** tab in the **Button Properties** window since, apart from having associated text, it could also have an image (or a combination of text and image). It is also possible to assign a second image that will be shown when the button is pressed; that is, when the button is pressed, the image changes.

The image of the button at rest is assigned to the **Bitmaps** [**Normal**] option. The image that the button will display when pressed is assigned to the **Bitmaps** [**Pushed**] option. The [**Alignment**] of the image on the button can be *Left*, *Right*, *Top*, *Bottom* or *Center/Scale*.

If you specify an external image, we can tell PxPlus to scale it to the size of the button by selecting the [**Scale bitmap**] check box.

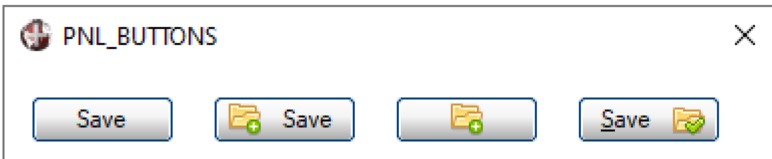


Remember that it is possible to specify a hot key or accelerator by using the **&** (*ampersand*) sign before a letter.

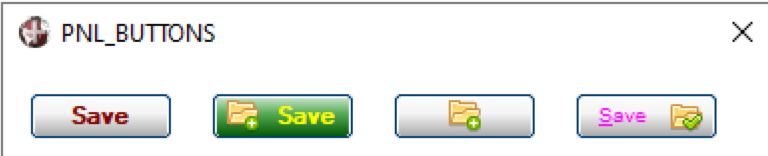
Example: "**&**Save" designates "S" as a hot key.

Example: Button Controls

This example shows a button with text only, another button with text and an image on the left, one button with an image only, and the last button has text (with a hot key) and an image on the right.



It is possible to change the font type/color of the button, as well as the color of the text, and any combination of all the attributes discussed so far.



There is also a series of attributes that allow us to change the behavior or appearance of the button. **Example:** We can decide whether or not it is a stop for the Tab key (normally used to navigate a panel), if it is hidden or disabled initially, or if it accepts focus (in this case, the button can be pressed, but it will not accept focus). Other attributes are the possibility that the button can be moved (during execution the user can drag and move the button, but the new position will not be registered or saved, this is limited to movements within the panel).

It is also possible to change the cursor type (Hover Cursor), as well as create a Web hyperlink style button (no borders, underlined and changes color when you hover over it).

You can define flat buttons (that look like they are not buttons but look as if they are part of the panel) or define a transparent button that shows what is behind it.

It is also possible to create an image type button, which will be divided into four images, allowing us to give the appearance that we want to change this button, since the state of the button will determine which part of the image will be shown.

Finally, it is possible to define a **DROP_BOX** control (drop-down box) within the button, which will allow a drop-down arrow to be placed on the button and a contextual menu associated with it.

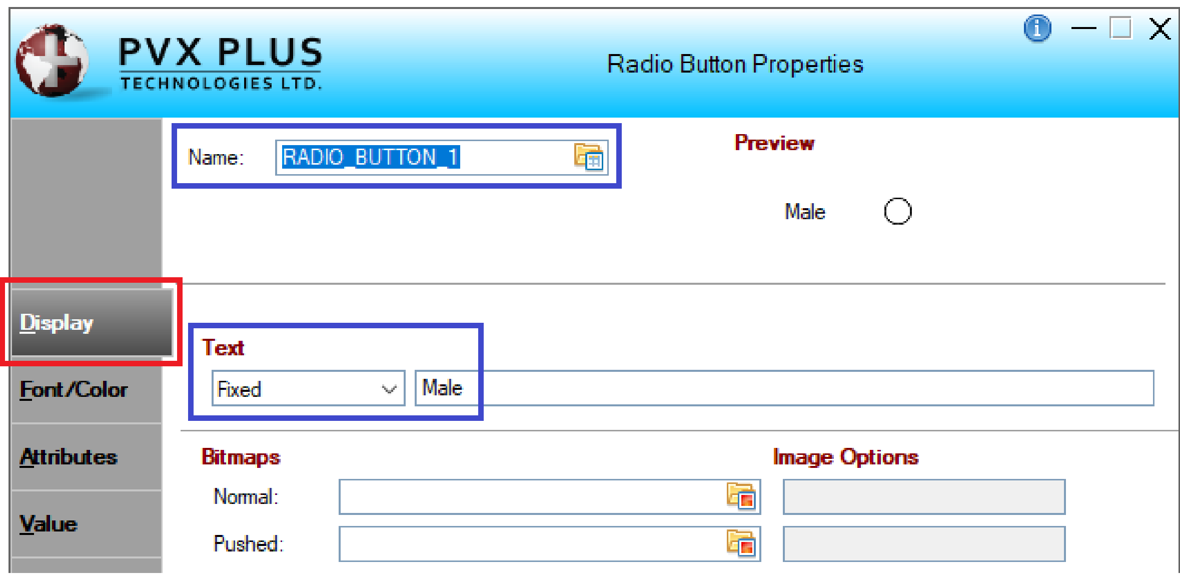
Note: It is not necessary for the buttons to have an elongated shape and the typical size of 10x1 or similar. You can use your imagination.

Refer to **Button Control** in the PxPlus Help documentation.

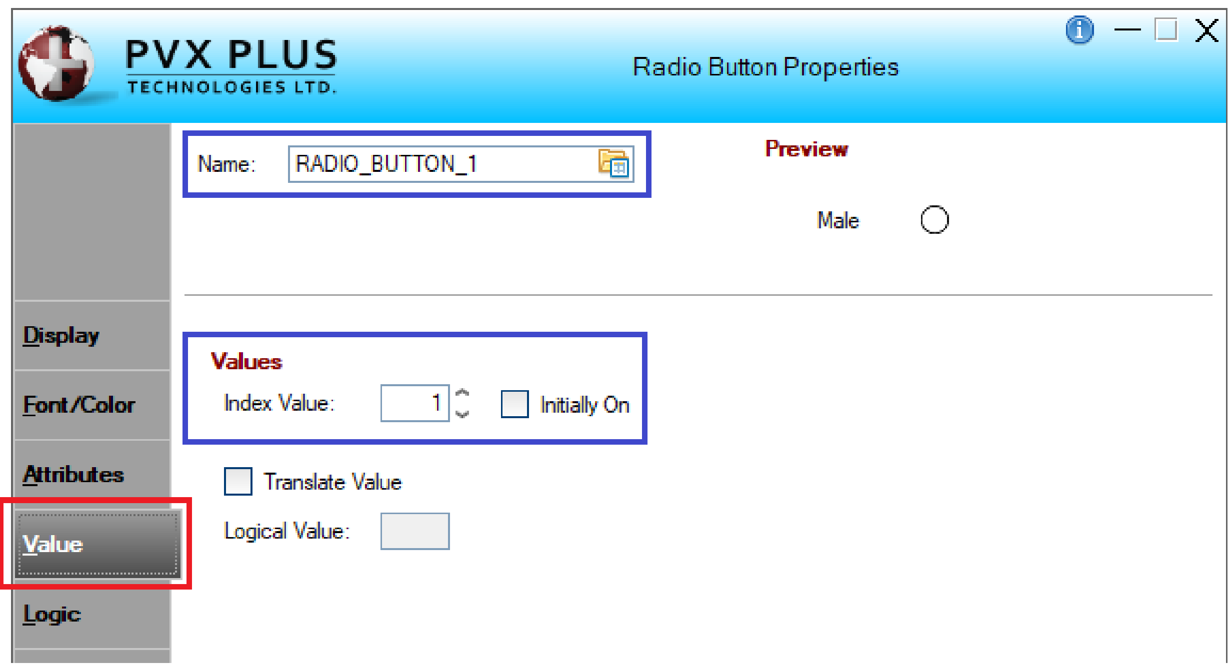
RADIO_BUTTON Control: Other Functions and Attributes

Remember that the selector or **RADIO_BUTTON** control allows you to select one and only one option among several. It is the only control that, although it must be drawn several times (as many times as there are options), it ***must share the same name***. A radio button can have up to 255 options, although much fewer are typically used. It is possible to have several radio button controls on the same panel.

We must create the first radio button, specify the name, and enter text (**Example:** Male) in the **Display** tab in the **Radio Button Properties** window.

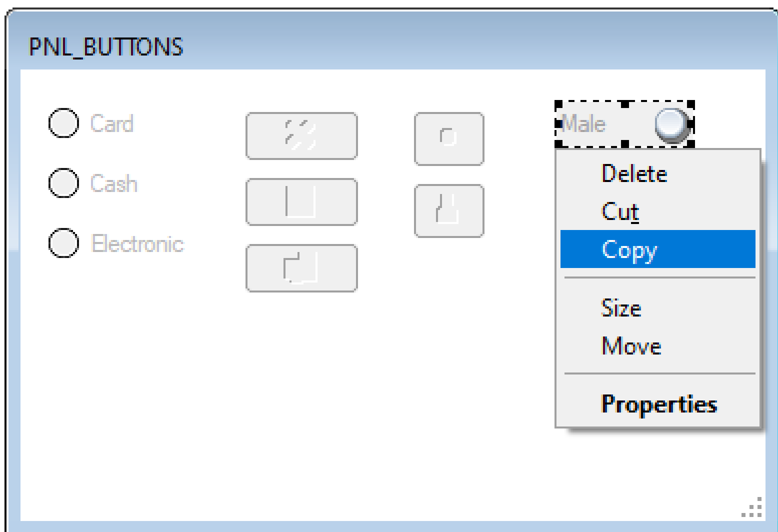


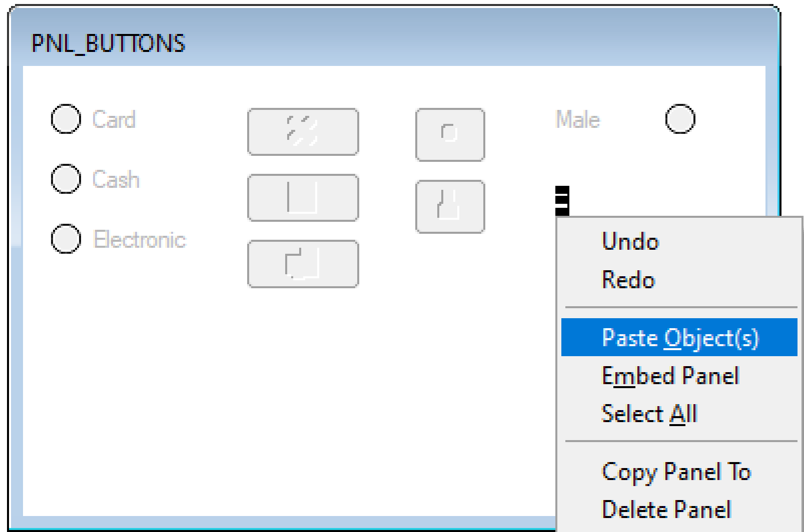
In the **Value** tab, we specify the [**Index Value**] as 1.



After we have created the first radio button, we make a copy of it. We select the control, choose [**Edit**] -> [**Copy**] from the menu at the top, and then choose [**Edit**] -> [**Paste Object**].

Another way is to select [**Copy**] from the right click context menu, go to where we want to place it, right click again and select [**Paste Object**].





We open the properties of the new control and enter ***the same control name*** (**Example: RADIO_BUTTON_1**).

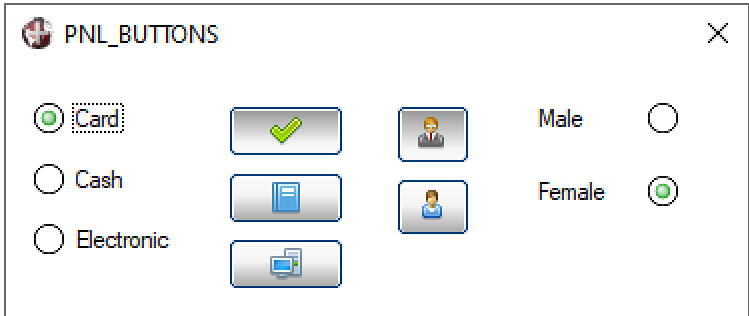
In the **Display** tab, change the [**Text**] to Female.

Go to the **Value** tab and enter 2 as the [**Index Value**].

It is possible to play with various attributes to have different aspects of Radio Button controls.

Exercise: Creating RADIO_BUTTON Controls

Below are some examples of Radio Button controls with different attributes:

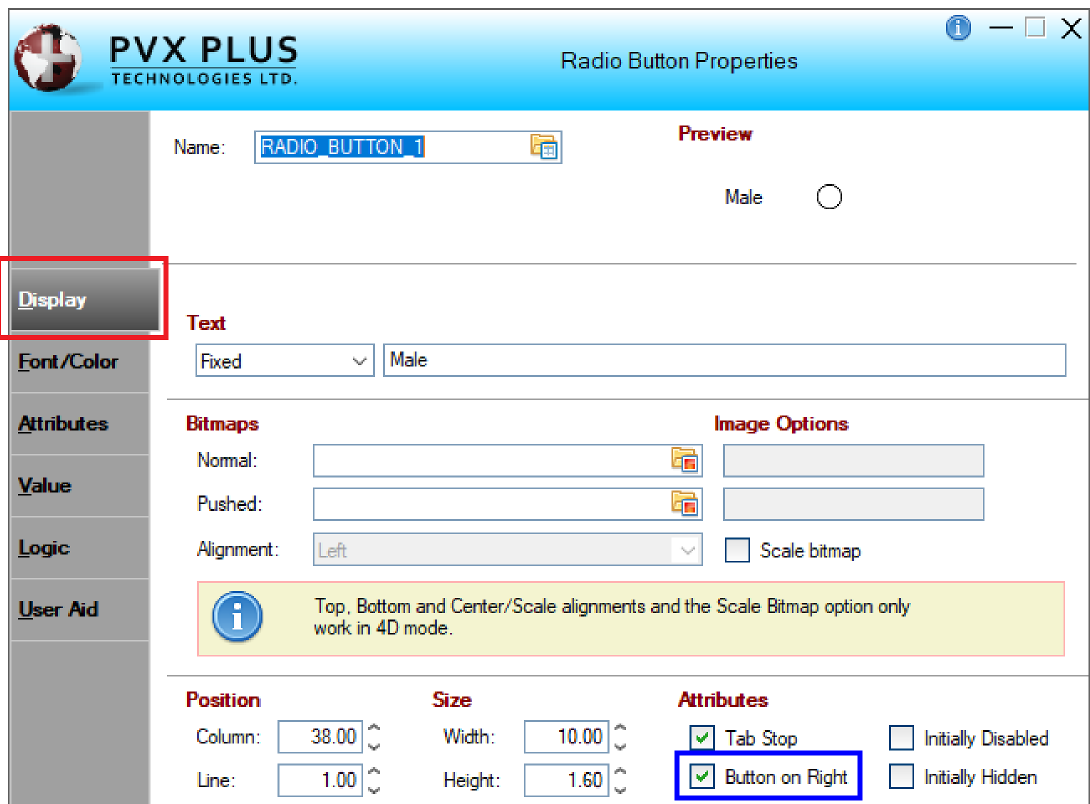


In the first example, the control is a "traditional" radio button with text in three options.

In the second example, the control has three options that are rectangular buttons (because an image was added (in the **Display** tab). When the button is "pressed", a green check mark is activated.

In the third example, the control has two options and allows you to select a man or a woman.

In the fourth example, the control has two options with the round dial button on the right instead of on the left, which is set using the [**Button on Right**] attribute (marked in blue below) in the **Display** tab:



We know that the Radio Button control has an associated event when one of the options is selected or changed, and it is possible to trigger or execute some action. We have already seen this. But, how do you know which option the user has selected?

Important Note: A Radio Button control can have several options, but when assigning the logic for the events, it should only be assigned to the first option; that is, the one with the [**Index Value**] equal to 1.

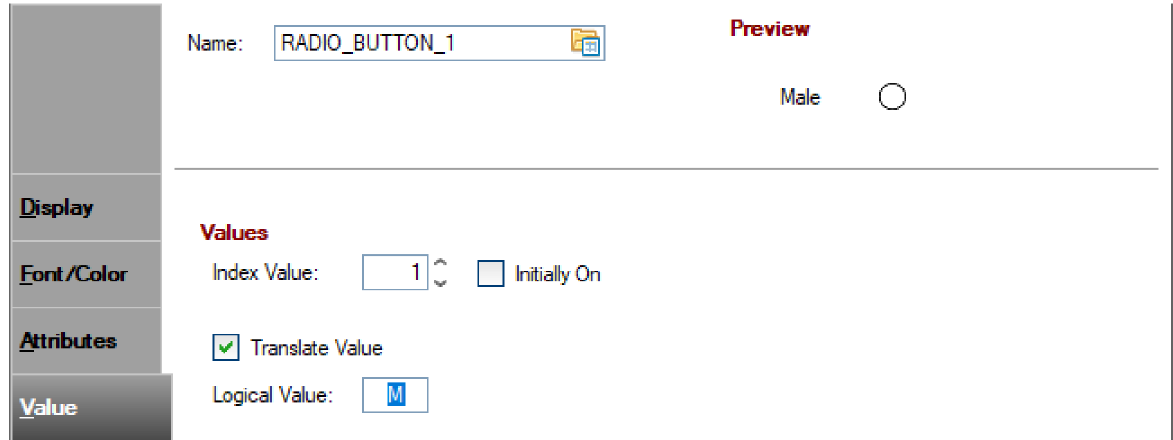
Since a Radio Button control has a name, that name will store the selection in literal or text form. If the option is the first one of the control (Index Value = 1), then "1" is returned in the variable **name_radio_button$**.

Note: Remember that, by default, the [**Suppress.VAL**] attribute must be active; otherwise, the variable **radio_name_button.val$** will be used.

Referring to the previous example, if the "Male" option is selected, the variable RADIO_BUTTON_1$ will be equal to "1", and if "Female" is selected, it will be equal to "2".

In the **Value** tab of the **Radio Button Properties** window, it is possible to define a conversion value called [**Translate Value**] where it is possible to apply equivalences.

Example: The first option returns the value "M" instead of the value "1":

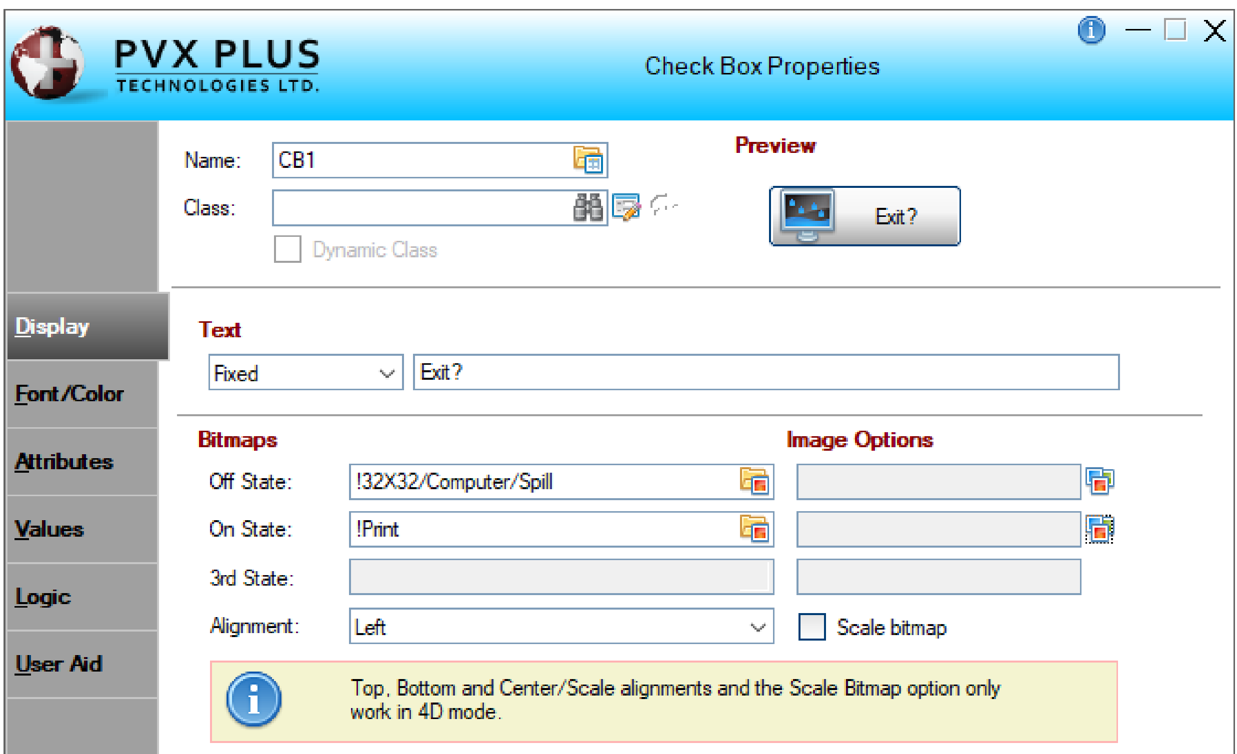


In general, the **RADIO_BUTTON** control and the **BUTTON** control (as well as the next control, **CHECK_BOX**) have many similar attributes and functionality. Mastering one of these controls will most likely make you more skilled in the use of the other two.

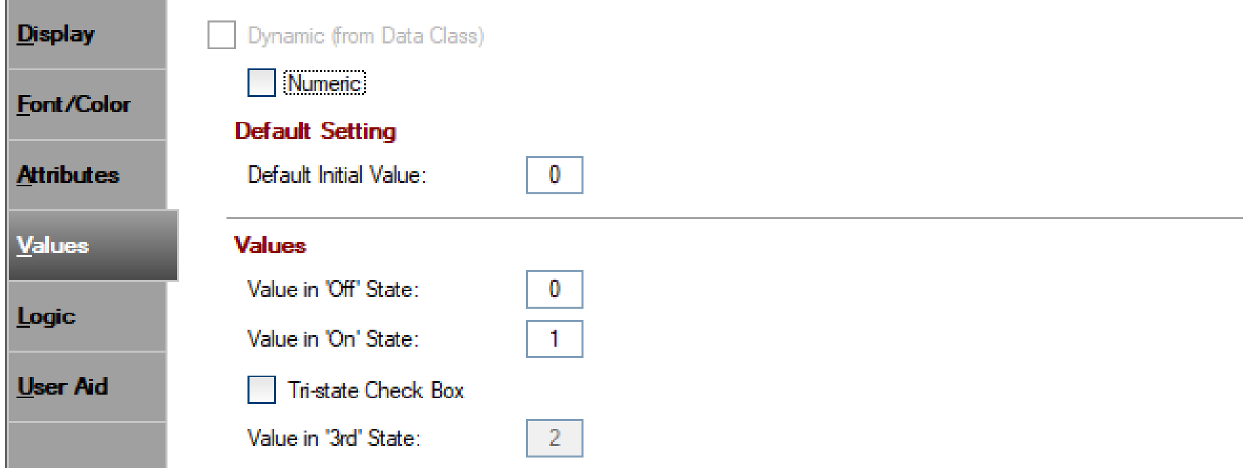
CHECK_BOX Control: Other Functions and Attributes

This control is actually called a **CHECK_BOX and TRI-STATE control** because it can have two values (On/Off, In/Out, Checked/Unchecked, etc.), or it can have three possible states. By default, the values in the check box are literal (text); that is, "0" for Off and "1" for On, although it is possible to define the check box values as numeric using the [**Numeric**] attribute. Apart from the Name, the main options to display for this control are the associated text and two or three images (the first for when the control is Off or unchecked, the second when the control is On or checked, and the third [optional] for the third state.

It is possible to define some options for these associated images (as with the Button and Radio Button controls), such as transparency. Remember that we can supply our own images.

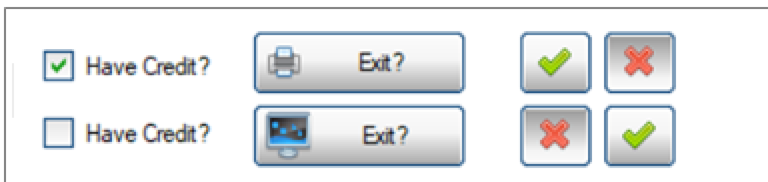


This control shares many attributes with the Button control with the difference being in the part of the **Values** tab where it is possible to specify whether the control will be text (literal) or numeric, in addition to providing different values for the On and/or Off States (in addition to the Third State):



Exercise: Creating CHECK_BOX Controls

Below are some examples of the **Check Box** control. They are the same controls captured at two different moments: one with the control On, and the other with the control Off.



The first example is the classic Yes/No on the control.

In the second example, the control is similar to a Button because an image was added (in the **Display** tab), but it is a Check Box that changes the image on the left.

In the last example, the control has an image that changes, but the control remains as "pressed" (has the [**Sticky Button**] attribute activated in the **Attributes** tab).

Using a RADIO_BUTTON or CHECK_BOX Control in a Program

With a Button control, we have no problems in our programs. The user presses the button, and the associated logic for the **When Button Pressed** event is executed.

In contrast, the other controls assume the user's selection in the form of the value of a variable.

Example:

If we have a Check Box called CREDIT on a panel that will determine whether the client has credit or not, we can do a routine similar to the one shown below. We assume the Check Box as literal or text values.

```
if CREDIT$="1" then gosub credit_authorized
msgbox "The customer has no credit","Notice"
exit
credit_authorized:
! Here continues the program
! …
```

With the condition IF CREDIT$="1", we are already determining whether the Check Box is on or not.

We can use this variable in the **IOLIST** command. Remember that it is used to define the list of elements or fields of a table:

```
list: iolist code$,name$,credit$,balance,address$
… read(channel,iol=list)
```

In this way, the program knows to which variables it is going to place each value read. We will go over the management of tables and files later.

In the case of a Radio Button control, it is the same.

Example:

Suppose that we have a Radio Button control to select whether you will pay by cash, card or electronically. Our control is called PAYMENT:

```
!
if payment$="1" then gosub pay_cash else
    if payment$="2" then gosub pay_card else
    gosub pay_other
!
```

One advantage of Radio Button controls is that we know we will only have one of the possible values; that is, there is no way for the user to enter a value other than one of the control options.

We can also record directly to a file.

Example:

Suppose that we have a Radio Button control called Civil_status$:

```
!
list_empl: iolist code$,name$,age$,civil_status$
…
!
write(channel,iol=list_empl)
!
```

In this second example, we place the variable corresponding to the Radio Button directly in the table write operation. These controls do not require the content to be read (happens with others, as we will see later).

In short, the user's selection for Check Box or Radio Button controls will be stored in a variable with the same name as the control itself. **Example:** A non-numeric Check Box control called CREDIT is created. By default, CREDIT$="1" if the Check Box is checked and CREDIT$="0" if the Check Box is unchecked.

Remember that, in some cases, a control may be defined as numeric and the variable associated with these controls will also be numeric (no dollar sign at the end); in this case, CREDIT=1 (checked) or CREDIT=0 (not checked).

11. Files and Tables: An In-Depth View

A ***file*** is a container that is usually stored on the computer, such as a document, image or information about a company. We call the files that contain the organized information, ***tables***. A ***table*** is simply a special type of file or a file with a special structure.

We already know that the regular form of communication between PxPlus and a table requires the use of a logical channel (an integer between 1 and 65535) that will serve to carry out the actions. The idea of this is to abstract the input/output operations from the file name.

Suppose we have a routine that reads and processes information from a table called "datafile", and we make our (pseudoformal) code like this:

```
Read information from datafile
Process information
Record information in datafile
…
Read datafile and count number of records
…
Update datafile
```

Now, if we must do that routine for another datafile, we must duplicate all the lines of the program:

```
Read information from datafile
Process information
Record information in datafile
…
Read datafile and count number of records
…
Update datafile
Read information from another datafile
Process information
Record information in another datafile
…
Read another datafile and count number of records
…
Update another datafile
```

On the other hand, if we abstract the name of the program, as PxPlus actually does, we could do:

```
Use 1 for filedata
Read information from 1
Process information
Record information in 1
…
Read 1 and count number of records
…
Update 1
```

This second approach means that by simply changing the file or table open in channel 1, we can leave the rest of our routine the same. That is definitely a big advantage, being able to use the same routine to handle files, tables, databases and almost any data storage structure.

One of the most powerful functions of PxPlus refers to the management of files or tables, the fundamental basis of any commercial application. We can initially conceive a table as part of a database, with some advantages and other disadvantages, which we will see throughout this book.

Note: PxPlus offers the possibility of working transparently with the main existing databases. We will begin by looking at table management (the very heart of PxPlus storage), and then we will see operations with databases.

As we already know, we will use channels to communicate with tables and files (as well as with other devices). We can use the **UNT** variable (and the **HFN** variable) to obtain the number of the next free channel. Remember that in ***that*** channel, only one file can be open at a time.

To release a channel in use, the **CLOSE** command is used. If the channel is closed and the **CLOSE** command is used, it will not return an error, unless otherwise indicated with the **ERR=** clause.

Example:

Let's look at several examples with the **CLOSE** command:

```
close(10)
! The channel 10 is not in use, but no error is generated
close(10)
! In this line, the error will be generated (we specify an err=error_routine clause)
close (10,err=error_routine)
! If channel 10 is closed, the program will branch to the routine "error_routine"
!
```

Once a device, file or channel has been released, it can be used again. It is possible to open the same file or table more than once; that is, have a table open in two (different) channels simultaneously, although it is not a recommended practice. Any integer between 1 and 65535 can be used.

In PxPlus, channels are also used to communicate with devices, both physical (such as printers) and virtual (such as the PDF generation printer). The general considerations are the same, and that is the opening (**OPEN**), reading (**READ**), writing (**WRITE**) and closing (**CLOSE**) will be used to communicate with them.

In addition to the **READ** and **WRITE** commands to read and write respectively, it is possible to use the input and print commands to read and write. Later, we will see the differences between these.

Note: On some devices, such as printers, it is normal to use only **WRITE** commands and not **READ** commands, given the characteristics of the device.

Types of Files in PxPlus

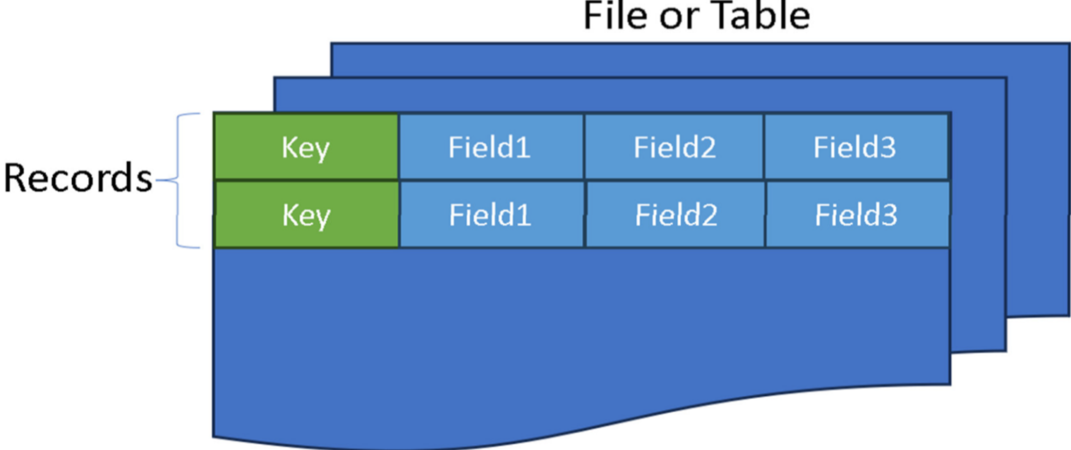
In PxPlus, there are two types of files: a "Native" file (such as a table) or a "Foreign" file (such as a text file or an image). Likewise, there are two types of devices: "Physical" (such as printers) and "Logical" (such as the PDF file printer). All the considerations about channel management are similar.

In PxPlus, the types of files are:

Text or ASCII (SERIAL)	A file with un-encoded information; also called a flat file.
Indexed (INDEXED)	A file whose information is referenced by the index number, where each record is accessed by an increasing number: 1 for the first, 2 for the second and so on.
Direct Multi-Index (MKEYED/DIRECT/SORT)	PxPlus work file, a file whose content is referenced by a Key, which allows direct access to any record instantly.
Program (PROGRAM)	A file containing program instructions.
Directory	A special file containing program names.
Databases	A specialized storage structure.
Virtual Devices	Specialized PxPlus structures to view information, create links, create files in PDF format, among many others.
Binary	Any other type of file; also called foreign.

We can access and work with all these types of files, some with specialized commands (such as programs and directories, for example), and the others with the same commands: **OPEN**, **READ**, **WRITE** and **CLOSE**. That is one of the great advantages of PxPlus, a unique simplicity in file processing.

For now, we are going to concentrate on the "Native" files of PxPlus, which we will refer to interchangeably as "files" or "tables", although formally, a file is any storage space and a table has a more formal structure.



Let's look at certain definitions related to file management.

A ***record*** is a block of bytes of a known length or with a mark, and is normally divided into fields of fixed length or with an end mark.

We can summarize (although not valid for all cases) that to read a field-oriented file, we can use the command:

```
Read (channel, key=key$)field1$,field2$...fieldN$
```

And to read a record-oriented file:

```
Read (channel, key=key$)records$
```

It is actually possible to read many of the PxPlus files and tables with either of the two commands. The first will read each field of the record and save it in the variables field1$, field2$ and so on. In the second example, everything will be stored to the registry in a single variable called registry$.

We can say that the records are read and written with the **RECORD** option in the **READ/WRITE** commands, **READ RECORD** or **WRITE RECORD**. PxPlus will transfer the record to and from a literal (text) variable without caring about the contents of the record. A field is a string of bytes ending with an end-of-field mark. A record is also a string of bytes terminated by an end-of-record marker.

We can change the list of fields: field1$,field2$...fieldN$ and use an input and output list:

```
Read (channel, key=key$)field1$,field2$...fieldN$
! Versus:
list:iolist field1$,field2$...fieldN$
Read (channel, key=key$)iol=list
```

The second approach allows you to change any value of any field in a centralized way, whereas with the first example, you must do it in each Read or Write instruction.

When we define a PxPlus table or file, we must specify its "dimensions", basically maximum length of the key (or keys), maximum length of the record and (optionally) maximum number of records.

When a file is created in PxPlus, it must be created in a specific directory or folder using one of the commands **KEYED/INDEXED/CREATE TABLE**, among others. We can do it "manually" or through the IDE environment, which is responsible for doing all this for us.

In any case, we should consider using the **PREFIX** command to specify the file/table search path to tell PxPlus where to locate our files/tables. This command specifies a series of directories or folders for PxPlus to search for files or tables.

This way, it is easy to have our systems organized in folders and activate the search in them (or not) at our convenience. The **PREFIX** command allows you to establish search paths for programs. It supports several options.

Example:

Some examples of the **PREFIX** command are:

```
prefix "C:\SIS\DATA\ C:\SIS\PGMS\"
prefix data "/u/payroll/ /u/payroll/data/ /u/payroll/util/"
prefix program "/progs/1/ /progs/2/"
```

It is also possible to create replacement directories where the system will automatically replace one content with another.

Example: If we have a system that has ***multiple references*** to a path "**/home**" and is going to be installed on a system that references another directory such as "**/usr/system**", the **PREFIX** command could be defined like this:

```
prefix "/home=/usr/system"
```

PxPlus will do the replacement automatically every time it finds this search specification.

The **PREFIX** command is very versatile, allowing you to have file names inside other files, add wildcards, add search suffixes, etc.

Refer to the **PREFIX** command in the PxPlus Help documentation.

READ/WRITE Clauses and Operations

To have more complete control over input and output operations, PxPlus offers certain clauses or options when reading/writing. These clauses are added to the command, normally separated by a **,** (*comma*).

The main ones are:

DOM=label	If the record in question is not found (if it is a READ) or if it is duplicated (if it is a WRITE), control of the program is transferred to the routine with the name label (or the one supplied by you).
END=label	If the end of a file is reached, control is transferred to the routine **label**.
ERR=tag	If an error occurs during I/O, control is transferred to the **tag** routine.
IND=num	The record identified with the index **num** will be accessed (read).
KEY=key$	The record identified with **key$** will be accessed.
KNO=num	The record identified with the alternate key number **num** will be accessed.
KNO=name$	The record identified with the key **name$** will be accessed.

With these clauses, whose handling we have already seen, it is possible to have more detailed control over the input/output operations of the files.

Example:

```
! Write the record identified with the key CODE$
Write(channel,key=CODE$)
! Read the file with the key CODE$, if you reach the end of the file,
! Go to the END_FILE routine
Read(channel,key=CODE$,end=END_FILE)
! Read with DNI$ key, if it does not exist, go to the NEW_CLIENT
! routine
Read(channel,key=DNI$,dom=NEW_CUSTOMER)
```

Along with these options or clauses, you can find out some conditions and information about the keys and status of the file opened on the channel:

FID()	Returns information about the characteristics of the file.
FIB()	Same as the **FID()** function but in a fixed format.
END()	Physical information about the file.
IND()	Returns the index of the next record in the file.
KEC()	Returns the current registry key of the file.
KEF()	Returns the key of the first record of the file.
KEL()	Returns the key of the last record in the file.
KEN()	Returns the key of the next record in the file.
KEP()	Returns the previous registry key of the file.
KEY()	Returns the key of the next record in the file.
RCD()	Returns the full record.
RNO()	Returns the number of the current record.

We can use this routine to read a file backwards (starting from the last record):

```
channel=unt
open(channel,err=file_error)"FILE"
!
! Find the last key
key$=kel(channel)

! Define the read loop
read_cycle:
find(channel,key=key$)field1$,field2$
! Process the info…
...
! Find the previous key
key$=kep(channel,end=end_cycle)
! Back to label to continue the loop
goto read_cycle

! End of routine
end_cycle:
close(channel)
exit

file_error:
msgbox "An error occurred accessing the file"
exit
```

This routine has several new things in case the file generates an error when opening it. In that case,

the program goes to the routine **file_error** and displays an error message. We also have the **KEL** function that allows us to read/locate the last key of the "FILE" file.

We have changed the **READ** command to the **FIND** command. The reason is that the **READ** command automatically advances the file pointer to the next record, and we read the record with the **KEP** clause (previous key). The **READ** command will read and advance one record, leaving us in an infinite cycle.

Then, we have used the **,end=** clause to branch control of the program to the **end_cycle** routine when the last record of the file is found (in this example, it will actually be the first record).

Using a Table with an Embedded Data Dictionary

We know that the utility for defining data dictionaries, **Data Dictionary Maintenance**, can be selected from the **Data Management** category of the PxPlus IDE main menu. Once a table has an embedded data dictionary, instead of doing the opening and reading like this:

```
List: IOLIST CODE$, NAME$, AGE$, SEX$, COURSE
channel=unt
open(channel)"STUDENTS"
read(channel,key=key$)IOL=list
```

We would do the following:

```
channel=unt
open(channel,iol=*)"STUDENTS"
read(channel,key=key$)
```

We include the clause **,IOL=*** to indicate that the IOList is embedded. We do the **READ/WRITE** commands without specifying the list of variables because PxPlus already knows which data dictionary is associated with that table.

To know the list of the elements that make up that data dictionary, from our program, we do:

```
LIST$=LST(IOL(channel))
```

Note: We do not need to do this! Simply placing and using the names of the elements is enough. PxPlus keeps the name of each variable in memory.

Using the Structure SELECT FROM..NEXT RECORD

There are reading alternatives in PxPlus to read the tables. One of the popular and powerful ones is the **SELECT** command and its associated commands known as **SELECT** loop or **SELECT** structure because, in reality, they are several commands that make up a complete routine.

The general syntax is:

```
SELECT FROM "FILE"
! PROCESS THE READ INFO
…
! END OF THE READING LOOP
NEXT RECORD
```

This loop reads records from a table and will repeat the loop (up to the **NEXT RECORD** statement) as long as it finds records that meet the given condition. A basic loop to display all records from a table called "FILE" with fields would be like this:

```
! We define the list of fields
LIST: IOLIST FIELD1$,FIELD2$
! We start the SELECT cycle
!
SELECT IOL=LIST FROM "FILE"
! Here we process or show FIELD1$ and FIELD2$
…
! The NEXT RECORD command sets the end of the loop
NEXT RECORD
```

Note that it is ***not necessary*** to open the file. This structure does not need a channel to establish communication with the table. In addition, the file will be processed completely because we have not placed any additional filters or options.

It is possible to use the **SELECT** command on an open file. We simply replace the file name with the channel.

Example: We are going to imagine the table open at channel 200:

```
! Table previously opened on channel 200:
OPEN(200)"FILE"
…
! We define the list of fields
LIST: IOLIST FIELD1$,FIELD2$
! We start the SELECT cycle
!
SELECT IOL=LIST FROM (200)
! Here we process or show FIELD1$ and FIELD2$
! The NEXT RECORD command sets the end of the loop
NEXT RECORD
```

The **SELECT** command also allows you to set limits or the start and end of records. The syntax would be:

```
SELECT FROM "FILE" BEGIN "BEGINNING" END "END"
..
NEXT RECORD
```

These clauses allow the initial PxPlus to read the record associated with the key "BEGIN" and read until it finds the key "END".

Example: To read the records that begin only with the letter "M", we could do:

```
SELECT FROM "FILE" BEGIN "M" END "N"
..
NEXT RECORD
```

In addition to the filtering of establishing the beginning and end of records, you can also establish a condition; for example, that some of the fields be equal to a value:

```
SELECT FROM "file" BEGIN "BEGINNING" END "END" WHERE city$="Panama"
..
NEXT RECORD
```

In this case, all records would be read from the key "BEGIN" to the key "END" and that also contain "Panama" in the city$ field.

Note: There is a way to force PxPlus to do a scan with a physical sequential read (that is, records one after the other, but without any sorting) that results in a much faster read. To do this, specify the **KNO=** clause *****:

```
SELECT FROM "file",kno=*
..
NEXT RECORD
```

In the **SELECT** command, you can also perform more complex searches using conditions, such as the logical condition **OR**.

Example: We will search for records with the cities "Panama" or "Caracas":

```
SELECT FROM "file" BEGIN "BEGIN" END "END" WHERE city$="Panama" OR city$="Caracas"
..
NEXT RECORD
```

Note: PxPlus allows the **SELECT** command to be used to access foreign databases such as MySQL, MariaDB or any other. This command (like the other PxPlus table reading commands) is not limited to its native tables.

The **SELECT** command allows you to perform advanced queries where several tables are specified, and filtering and linking are done between them. For this purpose, the **SELECT** command can be used in a "nested" manner; that is, one **SELECT** command serves as input or feed to the other.

Example: Suppose you need the Name and Department of the employees of branch "17". However, that information is in two tables: Employee table and Department table. The Employee table has the

Employee Name and Department Code. The Department table contains the Department Code and Name.

The **SELECT** command allows you to JOIN both tables with the Department Code and filter the results so that they are those of branch "17":

```
SELECT *,REC=empl$ FROM "employees" WHERE empl.branch$="17" WITH SELECT
*,REC=dept$ FROM "department" WHERE empl.cod_dept=dept.cod_dept
PRINT template.name$ + " " + dept.name$
NEXT RECORD
```

This command is very powerful and versatile. Its use and practice are recommended.

Refer to the **SELECT** command in the PxPlus Help documentation.

Erasing Records from a Table

Part of the processing of the tables is definitely the possibility of deleting records. We do this by using the **REMOVE** command, which allows, by specifying a key, to delete or delete the entire record:

```
REMOVE(CHANNEL,KEY=KEY)
```

Example:

Since there is nothing better than an example to clarify, let's look at one to remove a client:

```
! We locate a free channel and store it in CHANNEL
  channel=UNT
  OPEN (CHANNEL)"CUSTOMER"
 ! Routine to ask the client's code
 ask_customer:
  INPUT "Which client do you want to delete: ",CLTE$
  IF CLTE$="" \
   THEN GOTO ask_customer
  IF CLTE$="*EXIT*" \
   THEN GOTO exit
  REMOVE (channel,KEY=CLTE$,DOM=cannot_find_it)
  MSGBOX "Client "+CLTE$+" removed"
  GOTO ask_customer
 cannot_find_it:
  MSGBOX "I cannot find the client "+CLTE$
  GOTO ask_customer
 exit:
  CLOSE (CHANNEL)
  END
```

This example uses the **INPUT** and **PRINT** commands for keyboard input and display in the PxPlus Command console window. We combine the **,KEY=** clause to locate the record belonging to the client with the **,DOM=** clause to branch to a routine in case the same one is not found. (Remember that **,DOM=** means "Duplicate Or Missing", detect duplicate when writing or missing when reading.)

Note: How do we replace a particular record or field? Well, when performing a **WRITE** operation on an existing record, PxPlus will replace the fields that have changed.

How to Use PxPlus Special Devices

Although, in PxPlus, it is perfectly possible to access any file or device directly through a logical channel, such as a printer in Linux …

```
channel=unt
open(channel)"/dev/lpt0"
```

… in many cases, it is preferable to refrain from directly manipulating it and do it through "regular" means.

Example: If the port "/dev/lpt0" is being used by a printing system (Spooler) and we access it directly, we could interfere with its proper functioning and have catastrophic consequences.

For these cases and others that require that the information sent have special treatment, a series of special devices are included as part of PxPlus called **logical** or **special devices**. Because they are not physical devices, they are like drivers that will help us to handle many situations and convert data.

Let's initially look at the list of devices, and then we will delve into some of them:

Device	Purpose/Description
BITMAP	Generate an image or bitmap in memory.
HTML	Special device file used to generate HTML formatted output.
MEMORY	Memory-resident logical file; create and use memory file.
PDF	Output interface for generating PDF-compatible files.
PLUSFAX	Special device file that provides fax output using email and internet access.
QUERY	Access the contents of a Query like a standard read-only file.
STDIO	Special logical device allows Windows applications to access the logical files "stdin" and "stdout" that are generally only available when run from Windows Command mode.
TBRED	Special device file that provides remote access to Thoroughbred® data files.
VIEW	Access the contents of a View like a standard read-only file.
VIEWER	(Win) Special device file that allows you to preview print jobs.
WINDEV	(Win) Special device file that provides access in raw or pass through mode to the Windows print sub-system.
WINPRT	(Win) Special device file that provides standard API access to the Windows print sub-system.

These files have different functionalities, depending on each one, but the general use of them is that they will be treated as devices. That is, they require to be "opened" through a logical channel with the **OPEN** command, and we will communicate with them with the customary commands: **READ**, **WRITE**, **INPUT**, **PRINT**, etc.

BITMAP Device

The first logical device or special device is the ***BITMAP*** device (the * are part of the name). This device is used to create an image in memory and later save it to disk or print it.

Example:

```
!
! Start a new workspace
Begin
! Open logical device to create an image on channel 12
open (12)"*bitmap*"
! Draw a rectangle
print (12)'rectangle'(@x(1),@y(1),@x(10),@y(5)),
! Put some text
print (12)'text'(@x(12),@y(2),"Hello!"),
! Send the picture to screen
print 'picture'(@x(40),@y(0),@x(79),@y(24),"#12",0),
! Close channel
close (12)
```

Refer to ***BITMAP*** in the PxPlus Help documentation.

HTML Device

The ***HTML*** device is used to create pages or documents in HTML format, which is the format used for Web pages and documents. We can open this device to generate a page compatible with Web browsers.

The basic device syntax is:

open(channel,opt="FILE=name.htm;options")"*HTML*

Where:

channel will be the channel used to create the document or page. (In this example, it will be called "name.htm" and may have some options, such as font, color, title, etc.)

Example:

```
channel=unt
open (channel,opt="FILE=content.htm;SHOW;FONT=Courier New;TITLE=Title of Example Content;BACK=FFFFFF;TEXT=000000")"*HTML*"
print (channel)"Content line"
close (channel)
```

This example will use the channel (assigned through the **UNT** variable that determines the next free channel), create a page called "content.htm" with a font "Courier New", a title "Example Content Title", the color background will be "FFFFFF" (White, according to RGB format), and the text color "000000". The **SHOW** parameter indicates that the output will be displayed in the system browser. Then, the text "Content Line" will be sent, and finally, the channel is closed.

Refer to ***HTML*** in the PxPlus Help documentation.

MEMORY Device

This device simulates a PxPlus table in memory, ignoring the physical aspect of disk access and therefore being much faster. In addition, its content WILL BE DELETED WHEN CLOSING THE LOGICAL DEVICE ('file' ***MEMORY***).

The ***MEMORY*** file can use access with indexes (**,IND=**) or with keys (**,KEY=**). This will be determined by the first WRITE operation to the device.

In this type of logical device, you can use all the input/output commands that are used in regular files:

READ
READ RECORD
WRITE
WRITE RECORD
REMOVE
CLOSE

It is possible to use the ***MEMORY*** device to dump information read from another table/file (physical).

Example:

This example combines it with the **SELECT** command:

```
! We define the list of fields
LIST: IOLIST CODE$,NAME$,AGE,CITY$
! We create the *MEMORY* file
channel_mem=unt
open(channel_mem)"*MEMORY*"
! We start the SELECT cycle
!
SELECT IOL=LIST FROM "FILE" WHERE CITY$="Cartagena"
! Here we copy the content to the *MEMORY* file type
write (channel_mem,key=CODE$)iol=list
! The NEXT RECORD command sets the end of the loop
NEXT RECORD
!
! We already have the content of the file "FILE" in the file type
! *MEMORY* filtered according to the conditions of the SELECT command
```

Since the first WRITE operation to ***MEMORY*** was with the **KEY=** clause, the ***MEMORY*** file will be accessed via KEY. But, we can make it so that it records with **INDEX**:

```
! We define the list of fields
LIST: IOLIST CODE$,NAME$,AGE,CITY$
! We will create a counter type variable that will be incremented to use
! as index of *MEMORY* file
index=0
! We create the file *MEMORY*
```

```
channel_mem=unt
open(can_mem)"*MEMORY*"
! We start the SELECT cycle
!
SELECT IOL=LIST FROM "FILE" WHERE CITY$="Cartagena"
! Here we copy the content to the *MEMORY* file type
write (channel_mem,ind=index)iol=list
! We increase the counter or index:
index++
! The NEXT RECORD command sets the end of the loop
NEXT RECORD
!
! We already have the content of the file "FILE" in the file type
! *MEMORY* filtered according to the conditions of the SELECT command
```

Note: Logical device names can be in upper or lowercase; therefore, ***memory*** and ***MEMORY*** are acceptable, but you ***must*** place the ***** (*asterisk*) as part of the name. MEMORY (without the asterisks) is not a valid name to refer to a logical device.

Tip: Remember that the Indexed file type is a file whose content (the records) will be referenced by the index or consecutive number, and the first record will be **IND=1** and so on.

Refer to ***MEMORY*** in the PxPlus Help documentation.

PDF Device

A feature that many users like when they get started with PxPlus is the ease of creating documents compatible with the Portable Document Format (PDF) format, thanks to the ***PDF*** logical device.

The basic syntax of the ***PDF*** logical device is:

open(channel)"*PDF*;FILE=name.pdf"

This command will create a PDF-type document that will have the name "**name.pdf**". This is one of the most complete logic devices in PxPlus and offers great flexibility through many options.

Example:

The basics to create a document would be:

```
! Example to convert the content of a table to a PDF
! We define the list of fields
LIST: IOLIST CODE$,NAME$,AGE,CITY$
!
! We create the *PDF* file
channel_pdf=unt
open(channel_pdf)"*PDF*;FILE=LIST.PDF
! We start the SELECT cycle
!
SELECT IOL=LIST FROM "FILE"
! Here we copy the content to the PDF document
print(channel_PDF)IOL=LIST
! The NEXT RECORD command sets the end of the loop
NEXT RECORD
!
! We already have the content of the "FILE" file in the PDF file
```

Although the previous example works, the list will be "ugly". The data will appear together, since there is no type of organization in the list.

For this, we are going to incorporate the PRINT coordinates through the **@()**. We can specify in which column we want a piece of data to be located.

Example:

```
print(channel)@(10),CODE$
```

This command will send the content of the variable CODE$ to the channel, but it will be located from column or position 10. We can modify the previous program a little so that the list of fields (CODE$, NAME, AGE and CITY$) will be placed in different columns, giving it more of a report appearance:

```
! Example to convert the content of a table to a PDF
! We define the list of fields
LIST: IOLIST CODE$,NAME$,AGE,CITY$
```

```
!
! We create the *PDF* file
channel_pdf=unt
open(channel_pdf)"*PDF*;FILE=LIST.PDF
! We start the SELECT cycle
!
SELECT IOL=LIST FROM "FILE"
! Here we copy the content to the PDF document
print(channel_PDF)@(2),CODE$,@(12),NAME$,@(50),AGE,@(65),CITY$
! The NEXT RECORD command sets the end of the loop
NEXT RECORD
!
! We already have the content of the "FILE" file in the PDF file
```

The key change was the PRINT line that now has coordinates and will place the Code in column 2, the Name in column 12, the Age in column 50 and the City in column 65.

In the first example, the content of the PDF file/document will be like this:

```
00001JohnHenderson40Caracas
00002AnaPerez40Cartagena
00010LuciaSmith50BuenosAires
00012AndresRamirez60Medellin
00026MiguelMoreno30Medellin
00100RaulSanchez25Asuncion
00150CristinaRojas21Bogota
```

With the column specifications to make the report more readable, it would be:

00001	John Henderson	40	Caracas
00002	Ana Perez	40	Cartagena
00010	Lucia Smith	50	Buenos Aires
00012	Andres Ramirez	60	Medellin
00026	Miguel Moreno	30	Medellin
00100	Raul Sanchez	25	Asuncion
00150	Cristina Rojas	21	Bogota

We mentioned that the ***PDF*** logical device offers a large number of options. The syntax to include many of these options is:

open(channel,OPT="option;option")"*PDF*;FILE=name.pdf"

We can also do it:

open(channel)"*PDF*;FILE=name.pdf;option;option"

This table lists some of the main options that we can provide to the PDF document that we are generating:

Option	Description
AUTHOR=*text*	Add an "Author" label with given text.
CREATOR=*text*	Add an "Creator" label with given text.
ENCRYPT	Encrypts the document. By default, a password is required to be specified.
FILE=*name*	Name of file to create.
FORM=*form*	Specify the form. (LETTER by default, 8.5" x11")
MARGINS=*top*:*left*:*right*:*bottom*	Define margins.
NOCOPY	Disable the copy.
NOEDIT	Disable the editing.
NOPRINT	Disable the printing.
OPENPSWD=*passwd*	Set password to read/open the document.
ORIENTATION=LANDSCAPE | PORTRAIT	Set the page orientation.
OWNERPSWD=*passwd*	Set the owner password.
SUBJECT=*text*	Add a Subject field to the document.
TITLE=*text*	Add a Title field to the document.
VIEW	Opens the document when closing.
WATERMARK=*xxx*	Define a watermark.

Apart from these options, there are many others, as well as being able to use "mnemonic" controls to modify the content of the document. These mnemonics will be used to give a different appearance to the text or information that we are recording.

Example:

```
print (channel_pdf)'font'("Arial,-10"),"Document content"
print (channel_pdf)'F3',"Print with color number 3"
```

The list of mnemonics is quite long, and each one accepts different values:

'TEXT', 'FONT', 'PEN', 'FILL', 'PICTURE', 'PIE', 'POLGON', 'ARC', 'RECTANGLE', 'LINE', 'CIRCLE', 'OFFSET', 'LPI', 'CPI', 'MODE', 'F*n*' and 'B*n*' (Foreground and Background Color), 'WHITE' and '_WHITE' and other named colors, 'COLOUR' and 'COLOR', 'LF', 'CR', 'LM', 'PM' and 'FF'.

Mnemonic	Color Name
F0 / B0	Black
F1 / B1	Light Red
F2 / B2	Light Green
F3 / B3	Light Yellow
F4 / B4	Light Blue
F5 / B5	Light Magenta
F6 / B6	Light Cyan
F7 / B7	White
F8 / B8	Dark Gray
F9 / B9	Dark Red
F: / B:	Dark Green
F; / B;	Dark Yellow
F< / B<	Dark Blue
F= / B=	Dark Magenta
F> / B>	Dark Cyan
F? / B?	Light Gray

Example:

To place a title in 20-point Arial font with Light Blue and Dark Gray in the background, we could do:

```
print (channel_pdf)'font'("Arial,-20"),'F4','B?',"Title"
```

Note: You must put the font size in negative numbers. If you put it in positive numbers, the size will be related to the previous size.

This logical device has many functions and possibilities to adapt the document according to our needs.

Refer to ***PDF*** in the PxPlus Help documentation.

PLUSFAX Device

This logical device allows information to be sent via fax. This device will not be discussed in this book.

Refer to ***PLUSFAX*** in the PxPlus Help documentation.

QUERY Device

Remember that we previously created queries using the "Query" tool to define a series of filtering, sorting and selection criteria.

We can take advantage of all that and make that information available as if it were a file or table (we cannot write to a *QUERY* type device).

Basically, we must open the device specifying the name of the Query and the library that contains it.

Example:

```
open(channel_qry)"*QUERY*;MIQUERY;LIBRARY.EN"
```

Once we have opened the channel, we can read it in the regular way as we read any other table in PxPlus.

Example:

Let's do a little routine:

```
channel=unt
open(channel)"*QUERY*;QRY_PROD;SCHOOL_AB.EN"
!
! We define the reading cycle
read_cycle:
read(channel,end=end_cycle)code$,description$,line$,cost$,pvp$
! Information processing routine
...
key$=kep(channel,end=end_cycle)
! We return to the label to continue the cycle
goto read_cycle
! End of the routine
end_cycle:
close(channel)
exit
```

Refer to ***QUERY*** in the PxPlus Help documentation.

STDIO Device

In Linux, it is normal to make routines that redirect input and/or output using the symbols "|", ">" and "<". In Windows, this is not possible. PxPlus offers this device to help programmers fill this gap. For example, to capture the standard input:

Reading standard input (**stdin**):

```
dir a* | pxplus prog01
```

In the program "prog01", you can read the output of the **DIR** command:

```
open (1)"*stdio*"
while 1
    read (1,err=*break)R$
    print R$,
wend
msgbox "ready"
```

Writing to **stdout**, the following will output a sorted directory, taking advantage of the Windows "sort" command:

```
pxplus -mn prog02 | sort
```

The program "prog02" is like this:

```
set_param 'SD' ! Include directory delimiter
open (1)"*stdio*"
select F$ from "." where mid(F$,1,1)<>"."
  print (1)F$
next record
close (1)
```

Refer to ***STDIO*** in the PxPlus Help documentation.

TBRED Device

This device is an interface to read and write Thoroughbred BASIC files. This device will not be discussed in this book.

Refer to ***TBRED*** in the PxPlus Help documentation.

VIEW Device

This device is an interface to read information from a previously defined View as if it were a read-only file. The considerations are similar to the "***QUERY***" logical device with some small changes and considerations:

```
open(channel_qry)"*VIEW*;MIVISTA"
```

Review the ***QUERY*** logical device in this book, and refer to ***QUERY*** in the PxPlus Help documentation.

Refer to ***VIEW*** in the PxPlus Help documentation.

VIEWER Device

This is one of the most used logical devices in PxPlus. Basically, it allows you to have an on-screen view of a list or output of a report.

Like the other logical devices, the basic syntax is very simple:

open(channel)"*VIEWER*"

Like almost all PxPlus tools, we can include many options to adapt it to our needs, prioritizing simplicity of use and sharing many common aspects with other tools.

Starting to use the VIEWER is as simple as:

```
!
! Routine to display the content of a table on the screen
!
! We define channel for the viewer
channel=unt
open(channel)"*viewer*
!
! Let's read the file and send it to the viewer
!
! We start the SELECT cycle
!
SELECT IOL=LIST FROM "FILE"
! Here we copy the content to the on-screen viewer
print(channel)@(2),CODE$,@(12),NAME$,@(50),AGE,@(65),CITY$
! The NEXT RECORD command sets the end of the loop
NEXT RECORD
!
! We close the *viewer* channel
close (channel)
! We now have the content of the "FILE" file in the viewer!
```

As you can see, using the viewer is very simple. However, it allows great functionality: specify fonts and letter sizes, capture a print queue (Spooler), view the document at various zoom levels and page formats, send the report we are seeing to a printer or to a document or PDF file.

The generic syntax to add options that will change the behavior of ***VIEWER*** can be included through the **,OPT=** clause, as we can see in the general syntax:

open(channel,OPT="option;option;...")"*VIEWER*

This table lists some of the main options that we can use when opening the ***VIEWER*** logical device:

Option	Description
COPIES=*num*	Number of copies (if prints)
FONT=*fontspec*	Default Font for reports ("Courier New, -10")
FONTSIZE=*num*	Font size
MARGINS=*top :left:right:bottom*	Set margins
ONCLOSE	Show the viewer when the channel is closed
ORIENTATION=LANDSCAPE | PORTRAIT	Landscape or portrait orientation
PAPERSIZE=*num*	Set paper size
SUPPRESSWATERMARKS	Suppress watermark
WATERMARKIMAGE=*image*	Set watermark

This logical device has many functions and possibilities to adapt the document according to our needs.

Refer to ***VIEWER*** in the PxPlus Help documentation.

WINDEV Device

In MS-Windows environments where the operating system is normally in charge of managing all printing with the printing sub-system, it is difficult to gain direct access to many devices to have finer and more detailed control of devices such as printers.

This logical device serves as an interface or bridge between PxPlus and allows commands to be sent directly to the printer without any driver or manager to interpret them, thus giving full access to the printer's functions.

When you start using the logical device, you can either specify the name of the printer (or print queue) at once or allow a printer selection box to open for you to select.

```
! Open the logical device and printer selection box
open (channel)"*WINDEV*"
! Open the "HP LaserJet" printer
open (channel)"*WINDEV*;HP LaserJet"
! Open the "HP LaserJet printer located on that server"
open (channel)"*WINDEV*;HPLaserJetON\\Main_Server\P Laser"
! Open the LP device on port LPT1
open (channel)"*WINDEV*;LP ON LPT1"
```

Once the channel is open, we can begin to send information to the printer, normally with the **PRINT** command.

You can use the **OPEN INPUT** command to inquire about the printer's current settings without sending a print job or disturbing its operation:

```
open input (30)"*WINDEV*;ASIS"
winprt_setup read properties PROPERTIES$
close (30)
RANGE=ALL;COLLATE=AUTO;COPIES=1;ORIENTATION=PORTRAIT;PAPERSIZE=1;SOUR
C E=276;QUALITY=-3;MARGINS=-1:-1:-1:-1;OFFSET=0:0;COLOR
=YES;DUPLEX=1;TRUETYPE=1;DRIVER=winspool;TITLE=
```

In short, if you want to make any special settings or send any commands or change any special features of a printer, you must use this logical device. But, it will not accept any of the mnemonics to make "cosmetic" changes to the print, such as colors, fonts, attributes, etc. You will need to do this by sending the command sequences (known as ESC/P sequences). If you want to use the features provided by the printer driver, you must use the ***WINPRT*** logical device.

Refer to ***WINDEV*** in the PxPlus Help documentation.

WINPRT Device

Using the ***WINPRT*** logical device, you have access to the Windows printing subsystem API, and all print jobs and commands will be interpreted and processed by the printer driver. Low-level command sequences (such as ESC/P sequences) cannot be sent. This device shares syntax with the ***WINDEV*** device, and you can specify at once the name of the printer (or print queue) to use or allow a printer selection box to open for you to select.

```
! Open the logical device and printer selection box
open (channel)"*WINPRT*"
! Open the "HP LaserJet" printer
open (channel)"*WINPRT*;HP LaserJet"
! Open the "HP LaserJet printer located on that server"
open (channel)"*WINPRT*;HPLaserJetON\\Main_Server\P Laser"
! Open the LP device on port LPT1
open (channel)"*WINPRT*;LP ON LPT1"
```

If the printer supports fonts, it is possible to send forms with the mnemonic 'FONT' and take full advantage of the printer's features.

You can open the specified default printer using the command:

```
! Open default printer
open (channel)"*WINPRT*;DEFAULT"
```

You can also open and use the printer specifications and latest settings to take advantage of them:

```
! Open the default printer with its current settings
open (channel)"*WINPRT*;ASIS"
```

You can also instruct PxPlus to select the printer but allow the user to make some changes and settings, such as number of copies, paper size, etc., or a more limited option from the same Print properties box. Both commands are:

```
! Open the printer with basic settings
open (channel)"*WINPRT*;NORMAL"
! Open the printer with complete settings
open (channel)"*WINPRT*;SETUP"
```

PxPlus allows on this logical device full access to the printer features:

```
SOURCE$="MS Sans Serif"
channel=unt; open (channel)"*WINPRT*"
print (channel)'font'(FONT$,-12),'DF', ! Set Font 10 CPI
print (channel)@(0),"Customer",@(45),"Balance"
print (channel)@(0),"Test Client ONE",@(45),103.21:"####,##0.00-"
print (channel)@(0),"DOS Test Client",@(45),327.14:"####,##0.00-"
```

Example:

This example comes from ***WINPRT* / *WINDEV* Printing Examples** in the PxPlus Help documentation and illustrates more complete functionality:

```
! Color Printing
    FONT$="MS Sans Serif",COLS_REQD=80
    CHAN=unt;
    open (CHAN)"*WINPRT*"
    print (CHAN)'font'(FONT$,-12),'DF', ! Base set to 10 CPI
    CUR_COL=mxc(CHAN)+1;
    if CUR_COL=0 \
          then CUR_COL=80
    INCHES_WIDE=CUR_COL/10
    print (CHAN)'font'(FONT$,CUR_COL/COLS_REQD),'DF',
    print (CHAN)'cpi'(COLS_REQD/INCHES_WIDE),'lpi'(6),
    for Z=0 to 15
          INTENSITY$=tbl(Z>7,"","Light ")
          COLOUR$=tbl(mod(Z,8),"Black","Red","Green","Yellow","Blue","Magenta","Cyan","White")
          COLOUR_MNEM$=esc+"F"+chr(asc("0")+Z)
          print (CHAN)@(0),COLOUR_MNEM$,INTENSITY$,COLOUR$
    next
```

Refer to ***WINPRT*** in the PxPlus Help documentation.

As you can see, PxPlus offers a variety of logical devices to complement its tools while providing simple and easy usage. There are no external tools or anything additional to install in an integrated way, and with the same commands that we already know, we can take full advantage of the functionalities of different physical devices. You may be surprised to learn that all of these logical devices are actually PxPlus programs executed in a special way but available for you to make your own.

So, if you think you've seen everything about this powerful development environment, go grab a coffee or tea. There's still a lot to explore!

Using Special Access TAGS

PxPlus incorporates special labels that will be part of the name (as well as a series of parameters for each occasion) that will indicate to PxPlus that the file must have special treatment.

Below is a short explanation of each of these access labels. Refer to **Special File Command Tags** in the PxPlus Help documentation.

Label/TAG	Description
[ADO]	Microsoft SQL Server (Win)
[DB2]	DB2 Support
[DDE]	Dynamic Data Exchange (Win)
[DLL]	Dynamic Link Library (Win)
[HTTP / HTTPS]	Remote Call to Apache Server
[LCL]	Access User's Local Machine (Wdx)
[LIB]	Program Library (Win)
[MYSQL]	Open MySQL Native Database Link
[OCI]	Connect to Oracle Server
[ODB]	Open Database (Win)
[RPC]	Remote Process Control
[TCP]	Transmission Control Protocol
[WDX]	Direct Action to Client Machine (Wdx)

The first label tells PxPlus that an **MS SQL Server** database is being opened, so it must send/process all operations through the server in question. The generic syntax to open it is similar to this instruction:

```
open (1,iol=*,opt="CONNECT='Provider=SQLOLEDB'")"[ADO]DataSrc;Customer"
```

Where:

"DataSrc" would be the server and "Customer" the table.

There are many parameters and options to allow you to contact and process information in an **MS SQL Server** database. Refer to the **[ADO]** command tag in the PxPlus Help documentation.

The other tags, **[DB2]**, **[MYSQL]**, **[OCI]** and **[ODB]**, also refer to external databases. Many of these tags will likely need access to a server with the databases in question. Each one has its particularities, and you will need to check the references for more information.

The label **[DDE]** (Dynamic Data Exchange) tells PxPlus that they are connecting to an application that handles this type of information exchange, such as **MS Excel** for example. By opening the application and with the appropriate commands, it is possible to read and/or directly write native files

of that application and take advantage of its commands.

The **[DLL]** (Dynamic Link Library) tag is used to access a file of external libraries, normally programmed in a language such as C, and allow the use and connection with them from your PxPlus program.

The **[HTTP]** and **[HTTPS]** tags are prepended to the **CALL** command (not the **OPEN** command) to establish communication with a Web server and exchange information through the SOAP protocol and XML packets. Thus, directly from our PxPlus application, we can request or send information (XML encoded) to a Web server.

The **[LIB]** tag is a special tag that allows you to set a special file as a program repository. Instead of having many small files on disk, each corresponding to one program, you can create one file containing all the programs, and it's as simple as specifying the name tag before the program when saving one.

The way to save (from memory) our program to disk is through the **SAVE** command like this:

```
SAVE "PROGRAM.PXP"
```

This will create a program file with the name PROGRAM.PXP.

Note: Remember that you can use any extension for PxPlus programs. You do not need to use the ".PXP" extension.

To speed up the process and facilitate the administrative part, a program library can be specified:

```
SAVE "[LIB:/usr/pxp/MYLIBRARY]PROGRAM.PXP"
```

A file "/usr/pxp/MYLIBRARY" will be created that will contain all the programs that we want to put there. All PxPlus commands for manipulating programs, such as SAVE, LOAD, RENAME, ERASE, etc., are compatible with this format.

You can also define a special prefix so you don't have to write the library every time:

```
Prefix program "[LIB:/usr/pxp/MYLIBRARY]"
```

The **[RPC]** tag allows us to execute a command or open a file that is located on another server so that all the processing management or the content of the file itself is located on another server, and the communication will be done through the network.

Example:

The syntax of the **CALL** command or the **OPEN** command would be similar to:

```
CALL "[RPC:SERVER_NAME]PROGRAM;ROUTINE",argument$
OPEN (channel)"[RPC:SERVER_NAME]FILE"
```

In this example, PxPlus will locate the remote server "SERVER_NAME" and will tell it to execute the "ROUTINE" routine of the "PROGRAM" program and send it the **argument$** (optional). If the server "SERVER_NAME" is found, it will locate and execute the program in question, and when it finishes, it will send a response to the computer that initiated the action (where the **CALL "[RPC]"** command was executed).

The second example will tell the server "SERVER_NAME" to open a file "FILE".

The server "SERVER_NAME" must be configured and connected to the machine where the instruction will be executed.

Refer to the **PROCESS SERVER** command in the PxPlus Help documentation.

The **[TCP]** label indicates that a channel will next be opened for communications with the TCP/IP networking protocol. PxPlus allows you to open multiple network channels (sockets) simultaneously, as well as have ports listening and many other functions.

The **[LCL]** and **[WDX]** tags allow you to specify that a local resource should be used and not one from the server. Let's clarify this a little.

One of the great advantages of PxPlus is that it regularly operates in client-server mode where one machine provides its resources (Disks, Processor, Memory, etc.) in a unified manner with several others. The one that provides the resources is known as the "Server", and the one who requests or uses them is called the "Client". This way, all files, tables, and programs are in a single location that will be shared with dozens or hundreds of users, all reading and writing information simultaneously.

For these communications, a PxPlus license is installed on the server and a special program called WindX is installed on the clients, which will always point to the server. It will behave like PxPlus, but the resources it will use are those of the server.

Example: If on a client machine with WindX, we do this command:

```
open(99)"FILE"
```

The WindX program will open the file (at the server side), which provides access to it. The same thing happens if you want to create a file or use a printer:

```
KEYED FILE_NAME$ ! Create a file with name FILE_NAME$
open(90)"PRINTER" ! Open a printer
```

In all cases, WindX makes the request to the server, and it allows it (or not) to establish communication or create the file. But sometimes, we need to access a local resource; that is, a file located on the client machine or print to the local printer and not to the network or server printers. In that case, we must use the label **[WDX]** or **[LCL]**.

If we want to use the Windows printing subsystem, on our local machine, we can do:

```
! Open the LOCAL printer called "PRINTER"
open(90)"[WDX]*WINPRT*;PRINTER"
! Open the local DATA file
open (9)"[WDX]C:\tmp\DATA"
```

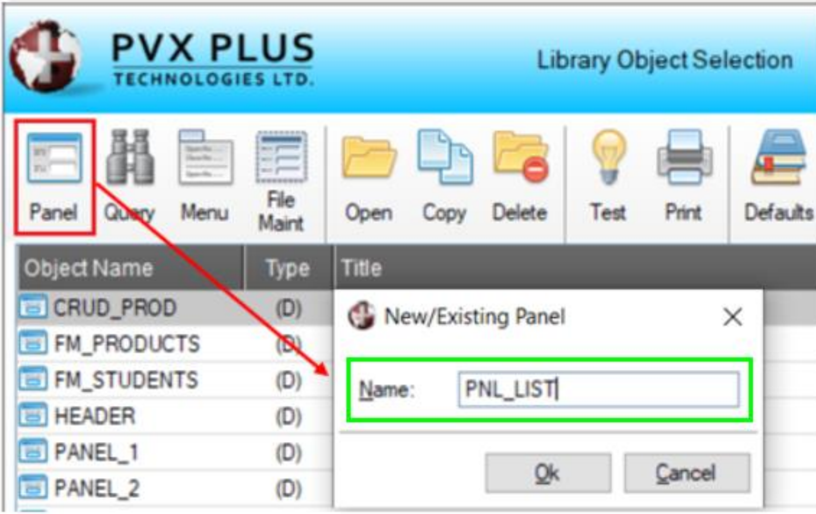
The difference between the tags is that if you are not on a client-server connection and using the **[WDX]** tag, an error will be generated, while the **[LCL]** tag will be ignored.

Later, we will take a look at the client-server part of PxPlus.

Using List-Type Controls (LIST_BOX, DROP_BOX)

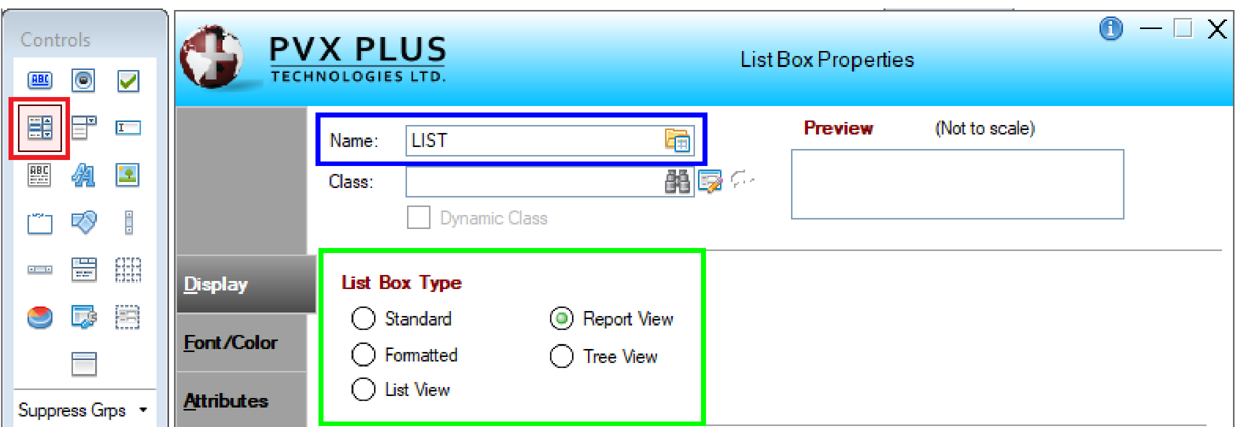
When we saw the introduction to these list-type controls, we surely wanted to go deeper, feeling that there were many things left in the pipeline. Now, after knowing better the commands for manipulating files and tables, we can return to these controls.

Let's create a panel again in the NOMADS graphical application designer, and we will do the entire exercise from scratch.

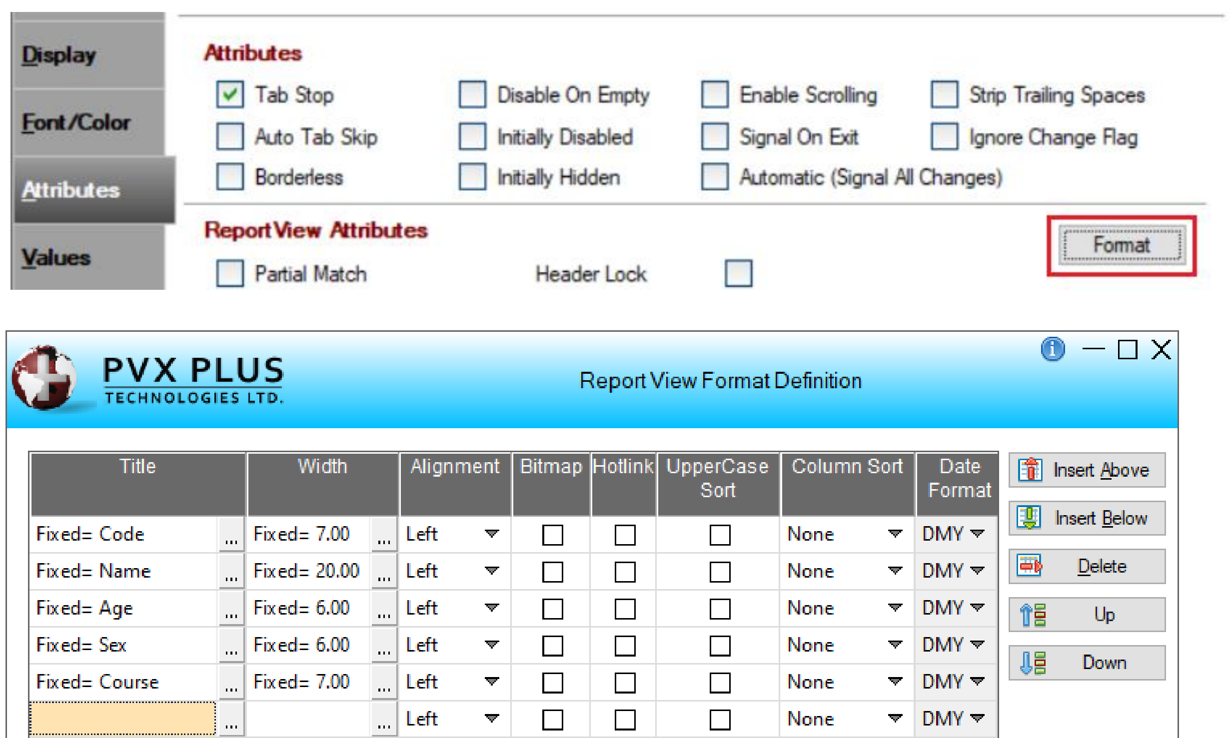


Once the panel is created, activate the [**Auto Refresh**] attribute. Create an **Exit** button with logic associated to the **When Button Pressed** event that ends the panel when the button is pressed.

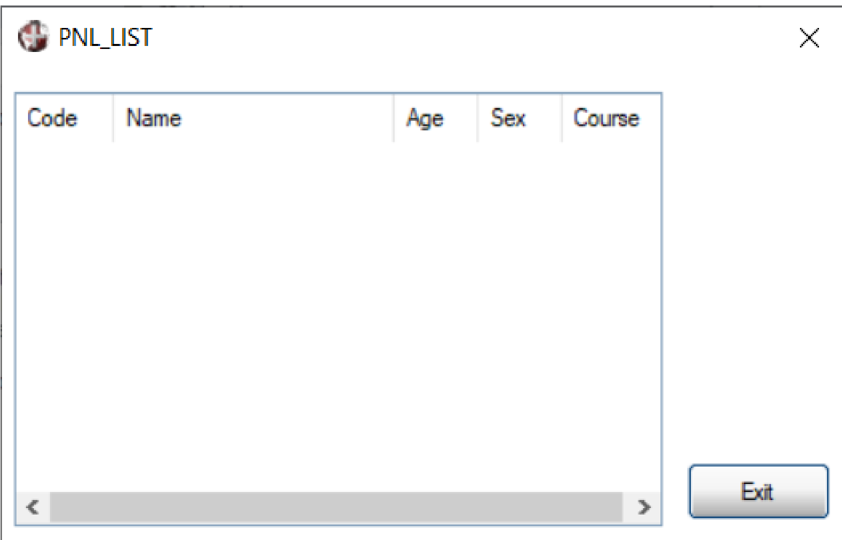
Next, we are going to draw a **LIST_BOX** called **LIST**. Change the [**List Box Type**] to **Report View**. The size is 46 columns x 14 lines:



Now, go to the **Attributes** tab. Click the [**Format**] button located on the right side and enter the following details:



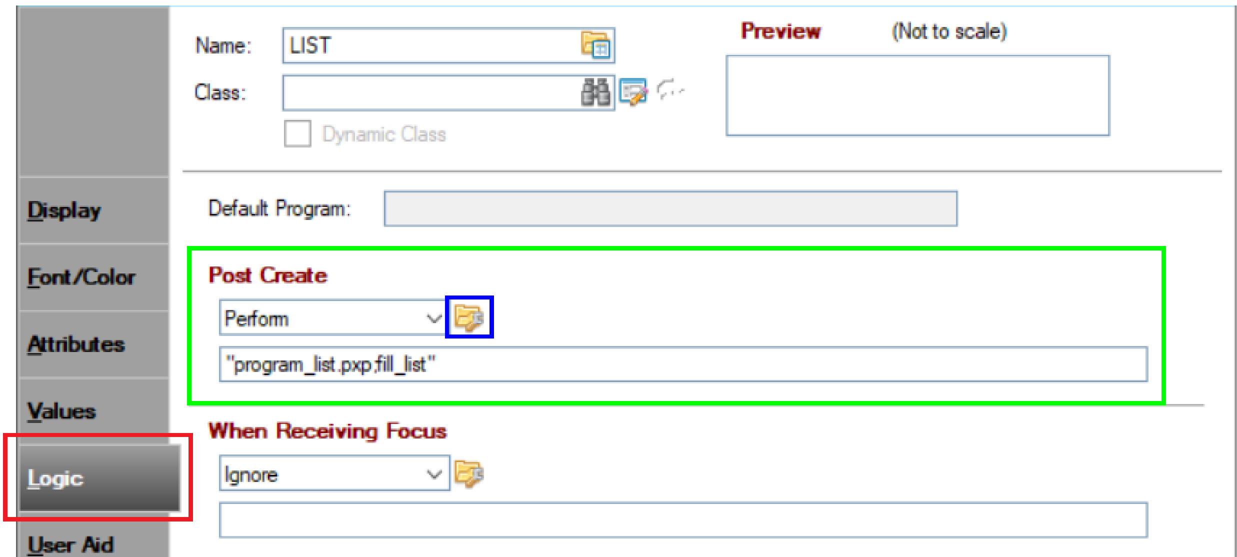
Save and test the panel. The panel should look similar to the one shown below:



Let's fill the list with the contents of the **STUDENTS** table.

To do this, go to the properties of the LIST_BOX control. Select the **Logic** tab and associate the **Post Create** event with the **Perform** action, which will execute a **Fill List** routine in a program called **program_list.pxp**. The argument would look like this:

"program_list.pxp;fill_list"

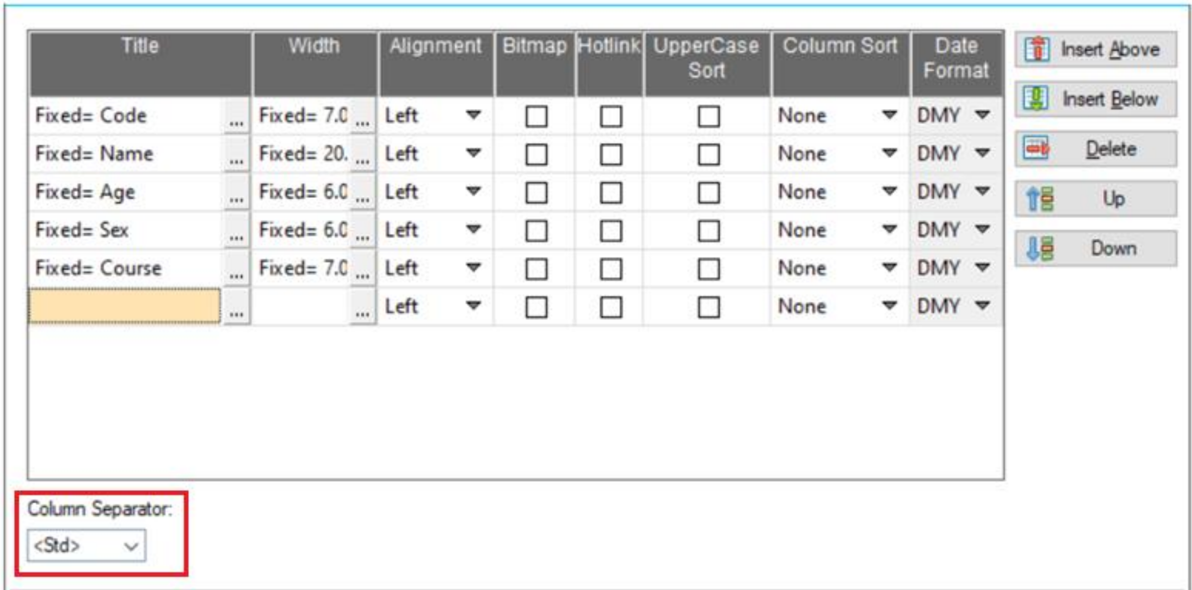


Click on the yellow folder icon (marked in blue) to the right of the **Post Create** event action to call the program editor. We are going to write this routine:

```
FILL_LIST:
    select * from "STUDENTS"
    list_box load LIST.CTL,0,CODE$+sep+NAME$+sep+AGE$+sep+SEX$+sep+COURSE$+sep
    next record
    return
```

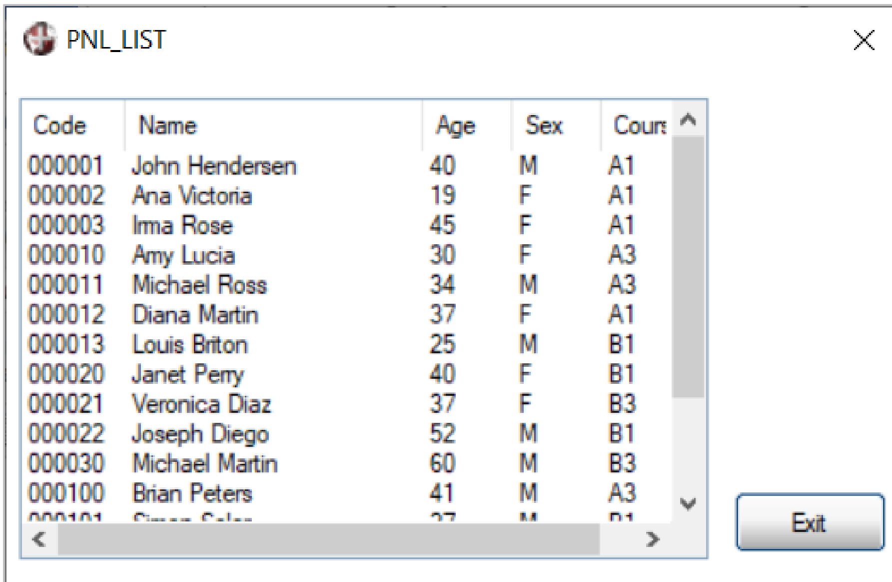
Basically, the routine uses the **SELECT** command with an * (*asterisk*) to indicate that they are all the fields of the **STUDENTS** table. The **LIST_BOX** command loads the **LIST.CTL** control with the values read from the table. We end the cycle with the **NEXT RECORD** command.

We use a special variable **SEP** (SEParator) when we are loading the fields to separate one from another. We can change the separator in the **Report View Format Definition** window. In the **Attributes** tab, click the [**Format**] button.



You have to select very carefully which character to use as a separator because, if we choose, for example, a comma, and there is some data that contains the comma, the results may not be as expected.

Save and test the panel. The result should be similar to the one shown below:



You didn't get the expected results? Let's look at a small checklist:

a) Was the panel created with the [**Auto Refresh**] attribute selected (using the [**Header**] option)?

b) Is the control a **LIST_BOX** called LIST and has the specified size 46x14?

c) Is **Report View** selected as the **List Box Type** and has the format defined (using the [**Format**] button in the **Attributes** tab)?

d) Does the **STUDENTS** table contain data? You can verify this by testing the FM_STUDENTS maintenance panel and scrolling with the arrow buttons.

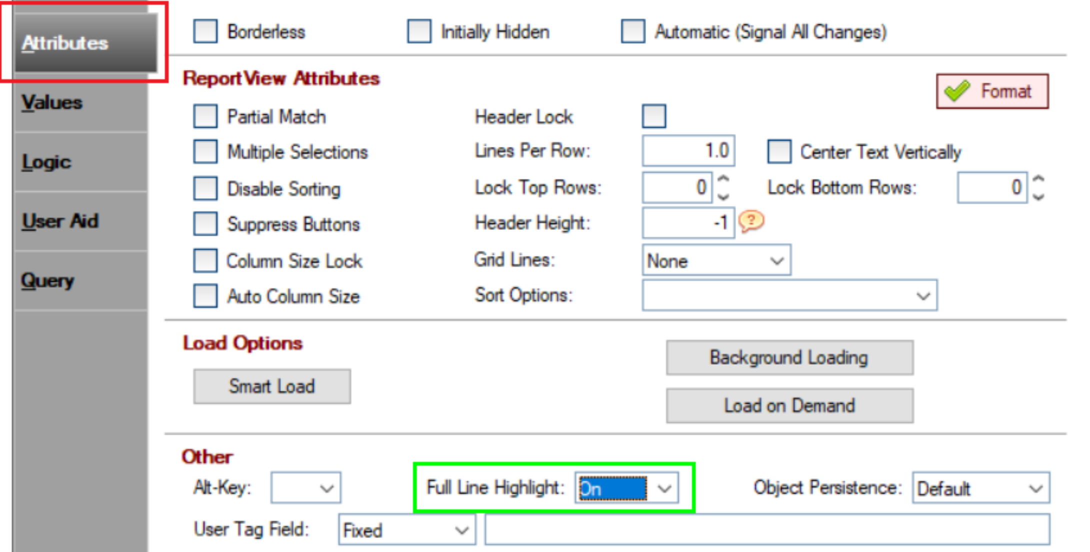
e) Is the FILL_LIST routine saved in the program editor?

If the panel shows some data but it is not displaying correctly, it could be the way you defined the format of the LIST_BOX control. If it does not show information in the List Box and the table does contain information, check the loading routine. Use only one field for now (the character after LIST.CTL is a zero):

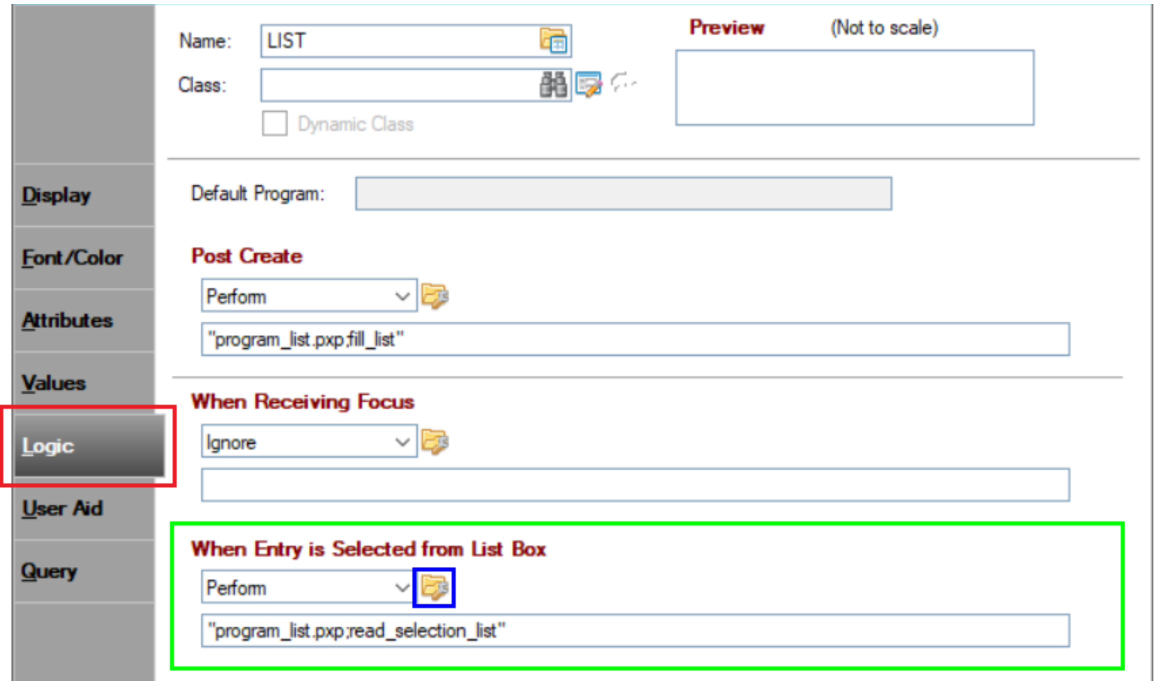
```
list_box load LIST.CTL,0,CODE$+sep
```

If everything goes well, you will see that clicking on the headers of each of the columns allows you to sort them.

You will also see that you can select an entry by clicking on it. If the record is only partially highlighted, you can change the [**Full Line Highlight**] attribute in the **Attributes** tab:



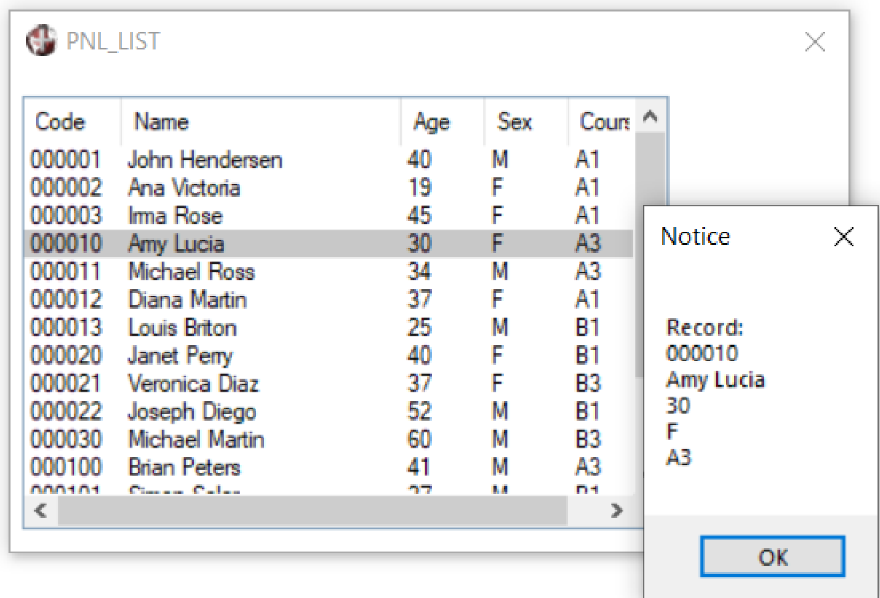
We can add a routine that will be added to the **When Entry is Selected from List Box** event. We enter a routine called **Read_Selection_List** (in the same previous program **program_list.pxp**):



Click on the yellow folder icon (marked in blue) and enter this routine:

```
READ_SELECTION_LIST:
list_box read LIST.CTL,SELECTION$,err=NO_SELEC
msgbox "Record: "+$0A0D$+SELECTION$,"Notice"
return
NO_SELEC:
msgbox "Nothing selected"
return
```

This routine should open a message with the user's selection when the user double clicks on any of the records:



Note: We can leave the program editor window open to facilitate our work of editing and correcting programs, although, if you close it and click on the yellow folder icon, PxPlus will open the editor window again.

Tip: Remember to save ***both*** the panel you are editing in the NOMADS graphical designer ***and*** your program.

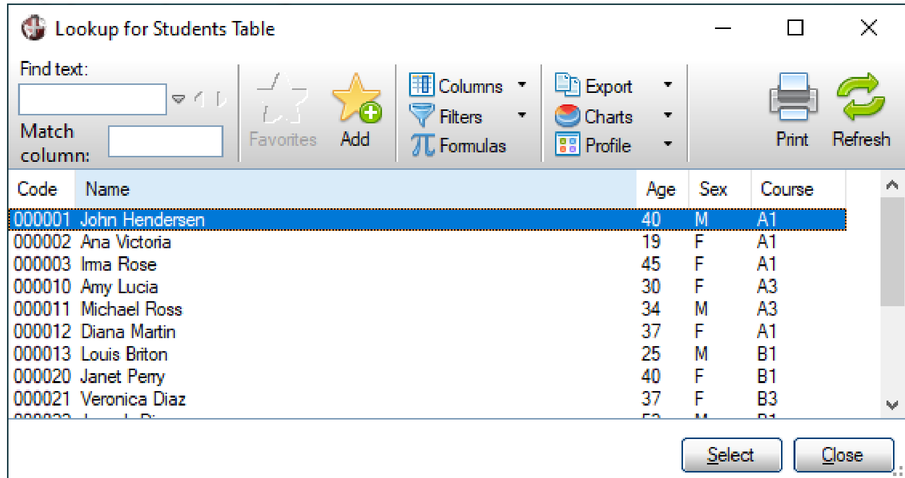
This same **SELECT...NEXT RECORD** routine allows the filling of the other list-type controls. You only have to change the name of the command, DROP_BOX, and change the identification of the control according to its name.

Example:

```
select * from "STUDENTS"
drop_box load list.CTL,0,CODE$+sep+NAME$+sep
next record
```

As we have emphasized in several parts of this book, one of the great advantages of PxPlus is that by knowing a control or an aspect of a component, we can take advantage of that knowledge for other controls. If we master the LIST_BOX control, we have already mastered other similar controls: DROP_BOX, TREE_VIEW, REPORT_VIEW, etc.

In an earlier chapter, we defined a Query called **QRY_STUDENTS** for the **STUDENTS** table:



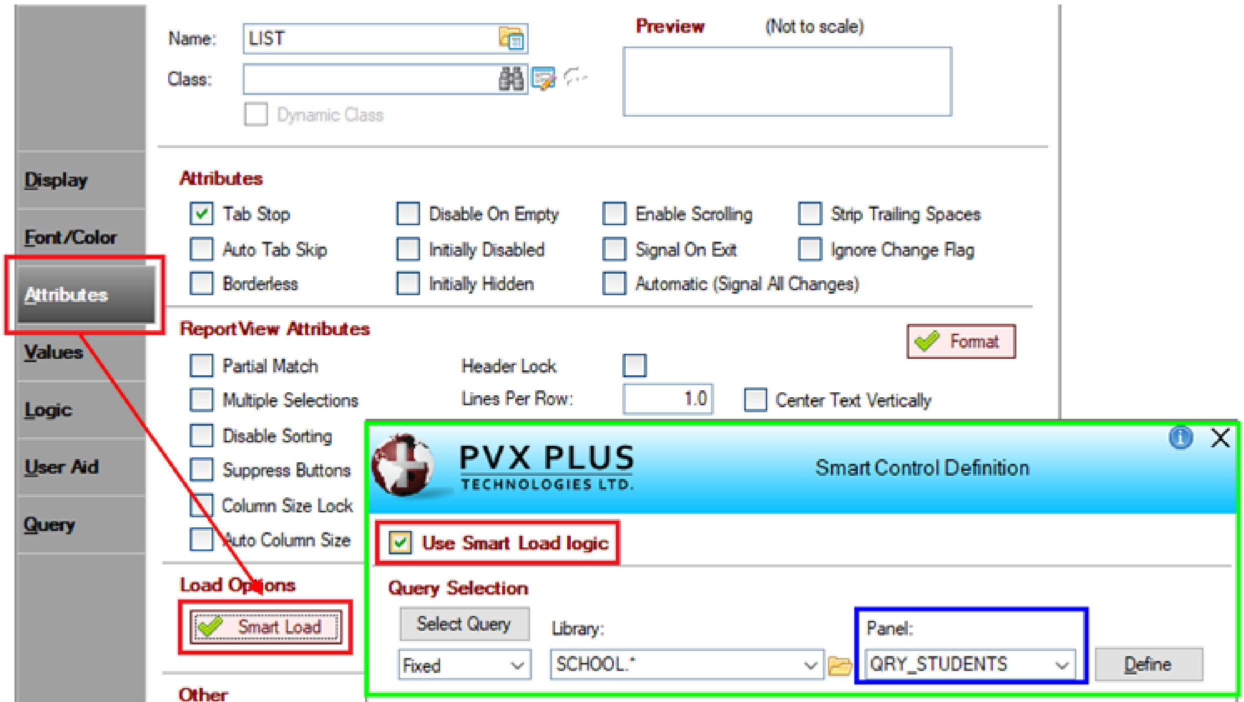
We also saw that PxPlus allows views, queries and more to serve as data sources for other controls and objects. How do we make this allow us to optimize our work even more?

In NOMADS, some controls can be created with the **Smart Controls** feature, an automatic load data function done by associating a query or other object.

Before we begin to learn more about Smart Controls, go to the **Logic** tab in the **List Box Properties** window for the **LIST** control. Remove the logic associated with the **Post Create** event to prevent it from being loaded automatically and change the action back to **Ignore**.

After removing the logic associated with this event, go to the **Attributes** tab and click the [**Smart Load**] button. This opens the **Smart Control Definition** window.

Select the [**Use Smart Load logic**] check box at the top of this window. For **Query Selection**, go to the **Panel** drop-down list and select **QRY_STUDENTS**.



We will be using a **Standard Query.** There is another query type called **Query List**, which can also be used. It is intended for use only by Smart Controls to auto-load a list of records from a data file or database table.

Each of these query types can be defined from the **Library Object Selection** window by using the **Query** toolbar button or **Objects > Query Object** from the menu bar. They can also be defined from the PxPlus IDE main menu by opening the **Graphical Application Builder (NOMADS)** category and selecting **Query Definition**.

For now, we will use the **Standard Query**. Accept these selections in the **Smart Control Definition** window, and then accept the **List Box Properties** window. When we reach the NOMADS panel designer, save and test this panel.

It doesn't work for you? Let's look at a small checklist:

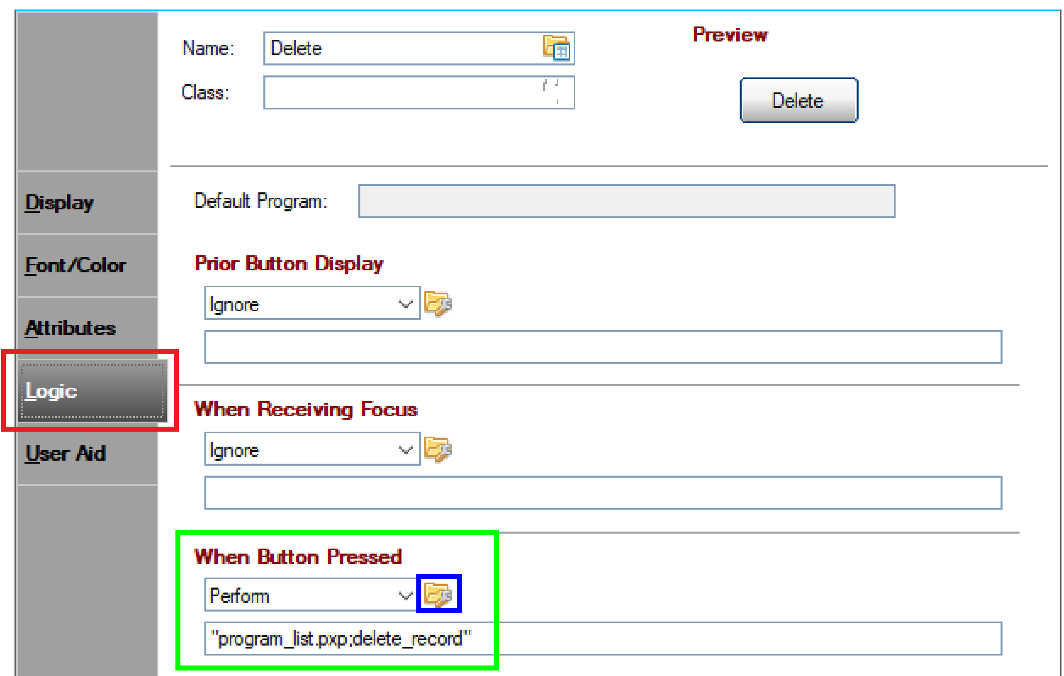
a) Verify that the query (**QRY_STUDENTS**) has information by selecting and testing it directly from the **Library Object Selection** window.

b) If the query works, verify that the panel has the [**Auto Refresh**] attribute selected (in the **Attributes** tab).

c) Verify that **Smart Load** is activated. In the **Smart Control Definition** window, the [**Use Smart Load logic**] check box should be selected.

Let's continue by adding a delete function to our panel so that the user can delete a record from the LIST_BOX.

Return to the NOMADS panel designer, and add a [**Delete**] button to the panel. We will associate a routine to this button for the **When Button Pressed** event. Add the **Delete_Record** routine (in the same previous program **program_list.pxp**):



Note: Remember that you are free to enter the names you want, and if you want to use a different program, you can do so.

Once the program and routine have been entered, click on the program editor button to create/modify our routine:

```
!
DELETE_RECORD:
!
! Check there is a selected record
list_box read LIST.CTL,SELECTION,err=NO_RECORD_SELECTED
! Erase the record
list_box load LIST.CTL,SELECTION,*
msgbox "Record erased","Warning"
return

!
NO_RECORD_SELECTED:
msgbox "No record selected"
return
!
```

This routine reads from the **LIST.CTL** control, but instead of reading the contents of the list, it reads

the index (that is, the position of the record in it) and uses that index to do the deletion, which is by instruction "**list_box load list.ctl,index,***". Loading an * (*asterisk*) deletes the control record.

Note: This routine does not delete the information from the table, only from the **list_box.** We can add a delete routine to the file, something like:

```
!
DELETE_RECORD:
!
! Check there is a selected record
list_box read LIST.CTL,SELECTION,err=NO_RECORD_SELECTED
! Erase the record from LIST_BOX only
list_box load LIST.CTL,SELECTION,*
gosub delete_from_file
msgbox "Record erased","Warning"
return

!
NO_RECORD_SELECTED:
msgbox "No record selected"
return
!
! Delete record from file
delete_from_file:
channel=unt
open(channel)"STUDENTS"
remove(channel,ind=selection,err=Cannot_delete_record)
close(channel)
return
! Cannot erase the record from file
Cannot_delete_record:
msgbox "Error trying to erase the index "+str(selection)
return
```

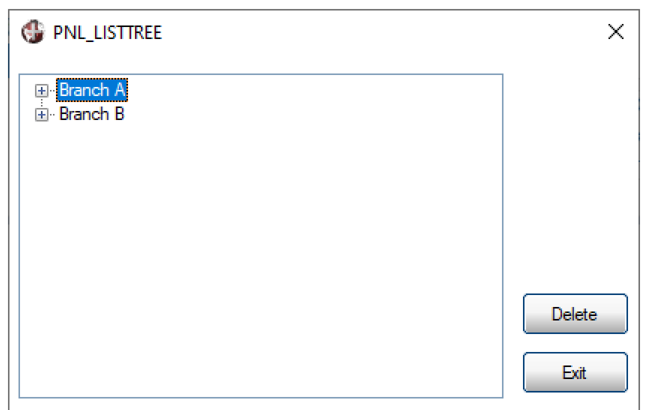
In this case, we are deleting using the file index. We can also modify it to delete the record with the key.

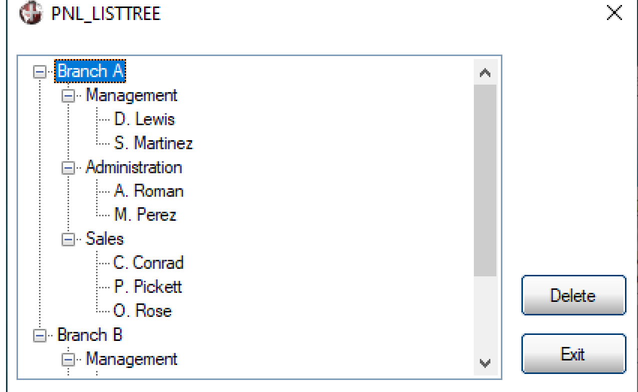
Given the large number of functions and controls that PxPlus has, it is essential that you practice, read, study and experience the different functions and components, giving priority to the controls and functions that you consider most convenient for the type of development you will carry out.

TREE_VIEW Control

The **TREE_VIEW** control offers a different functionality, since it is a collapsible, tree-type structure where there are several levels, and each level can be collapsed or expanded to show other options inside.

Example: Each level that has a **+** (*plus*) sign or a **-** (*minus*) sign indicates that it is an option that has other options associated with it.



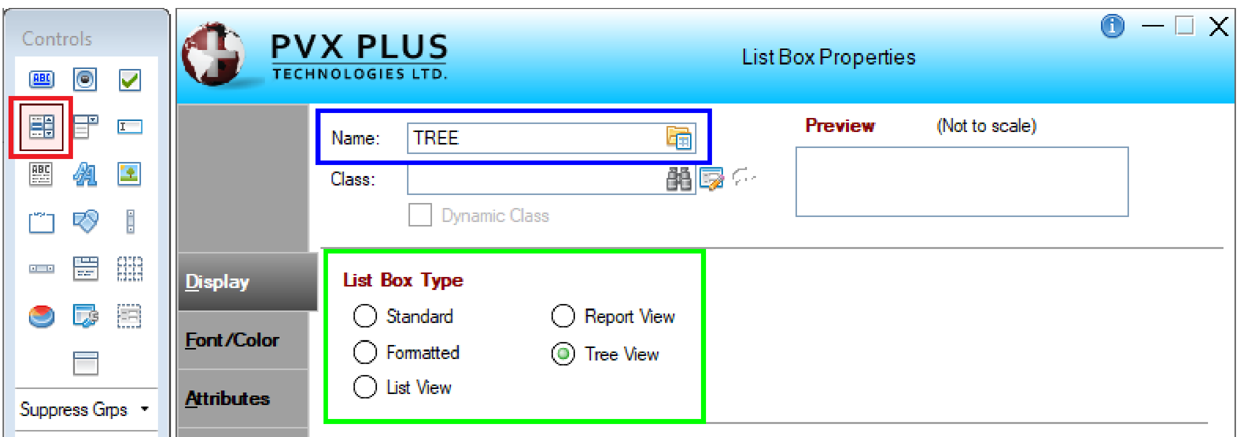


The example on the left shows a **TREE_VIEW** control with all its options (branches) collapsed. The example on the right shows the control with the options expanded. In this example, the control has three levels, but it can have more levels.

The **TREE_VIEW** control is a modification or sub-type of a LIST_BOX control where the options have a certain structure that allows PxPlus to know which part should be contracted or expanded. The options can have images or bitmaps associated with them.

Exercise: Creating a TREE_VIEW Control

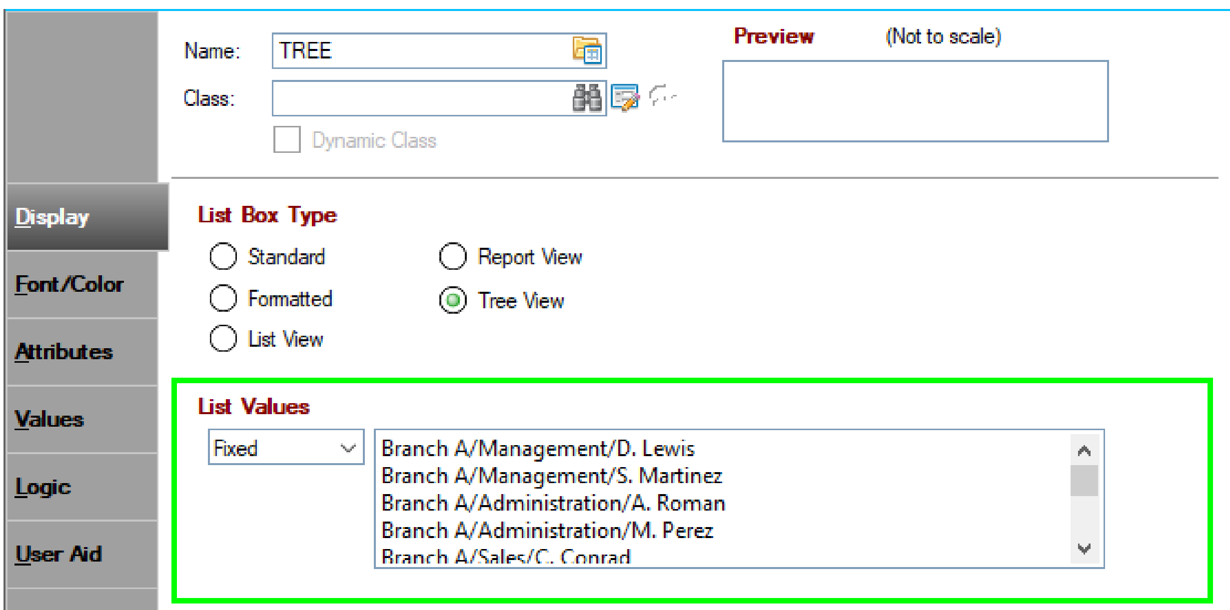
We are going to create a panel (remember to activate the [**Auto Refresh**] attribute) and draw a **LIST_BOX** control called **TREE**. For **List Box Type**, select [**Tree View**]. The size is 46 columns x 14 lines.



In the [**List Values**] box in the **Display** tab, enter this sample data, including the ***/*** (*forward slashes*):

Branch A/Management/D. Lewis
Branch A/Management/S. Martinez
Branch A/Administration/A. Roman
Branch A/Administration/M. Perez
Branch A/Sales/C. Conrad
Branch A/Sales/P. Pickett
Branch A/Sales/O. Rose
Branch B/Management/I. Hammond
Branch B/Management/U. Thomas
Branch B/Administration/J. Martinez
Branch B/Administration/O. Ronald

Once the sample data has been entered, the [**List Values**] box will look like this:



This format must be respected. Basically, it should have a structure similar to this:

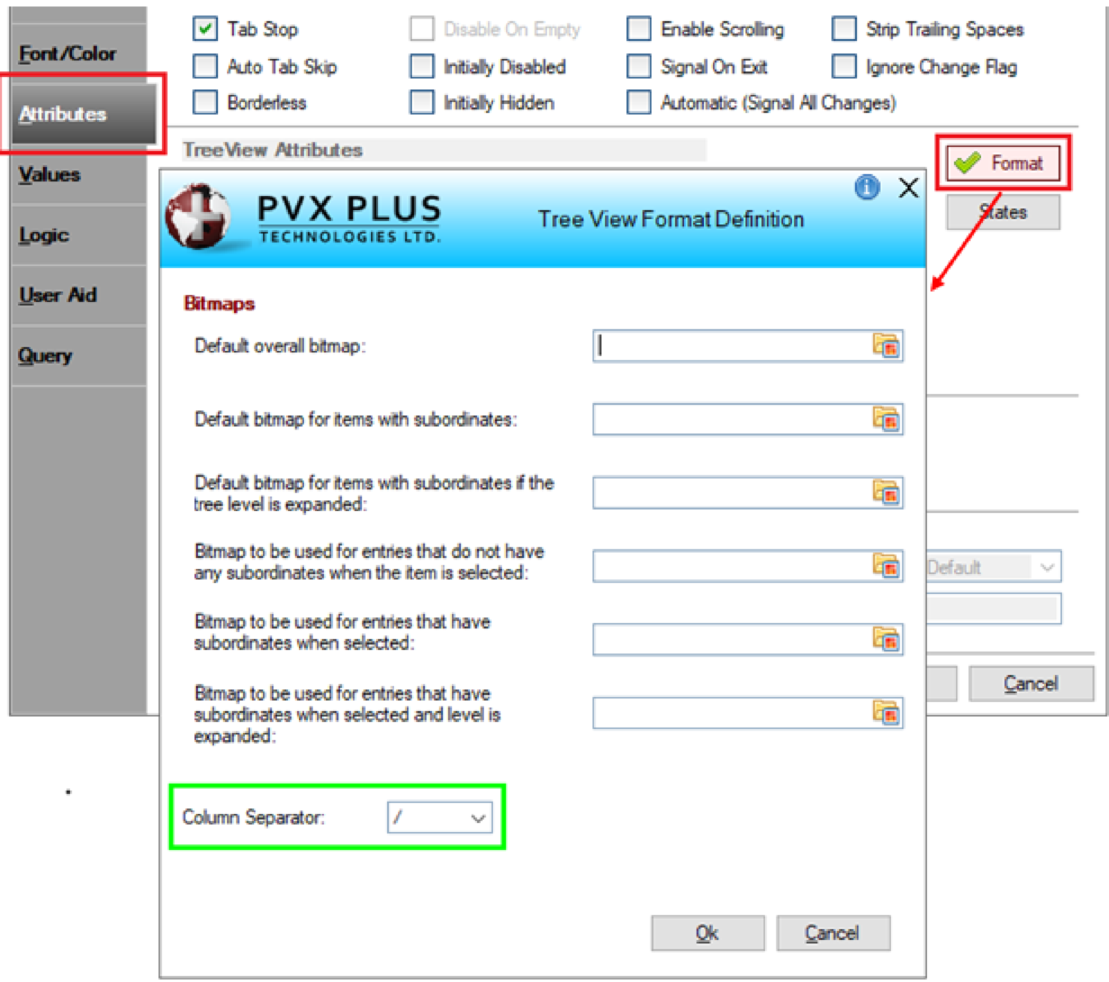
Option1<Sep>SubOption1<Sep>Element1
Option1<Sep>SubOption1<Sep>Element2
Option1<Sep>SubOption2<Sep>Element1
Option1<Sep>SubOption2<Sep>Element2
Option2<Sep>SubOption1<Sep>Element1
Option2<Sep>SubOption1<Sep>Element2
Option2<Sep>SubOption2<Sep>Element1
Option2<Sep>SubOption2<Sep>Element2

Where:

<Sep> must be a single character, allowing PxPlus to determine the content of the control precisely. In our example, we used the ***/*** (*forward slash*) as an option separator.

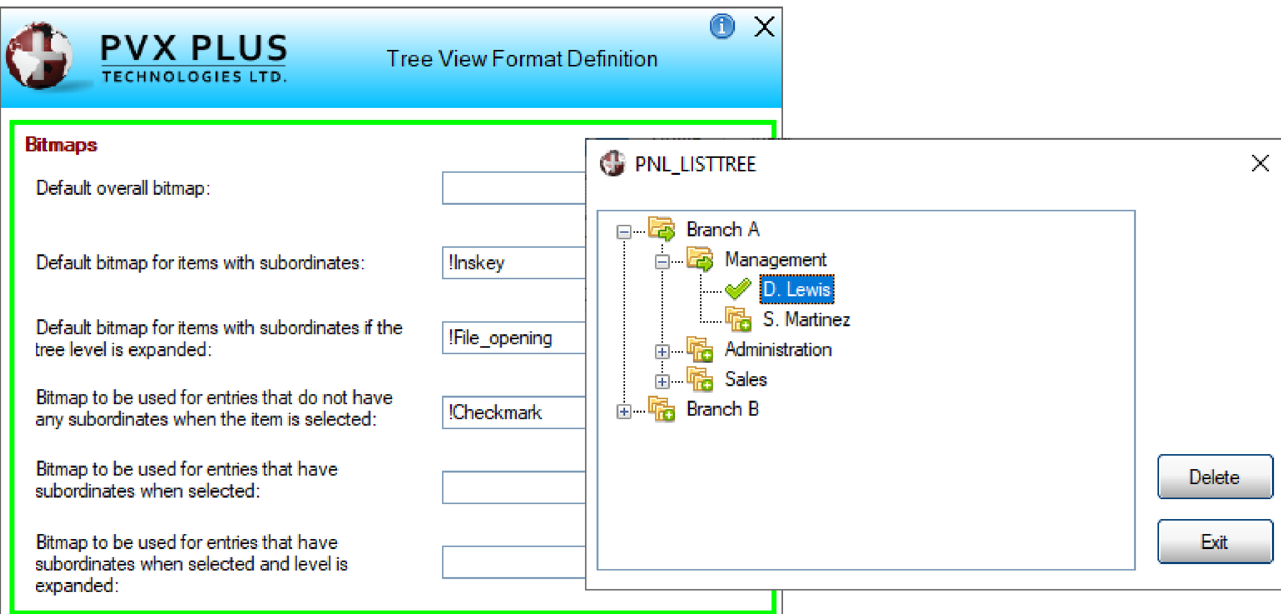
Once the values have been loaded, go to the **Attributes** tab. On the right side, click the [**Format**] button. The **Tree View Format Definition** window opens.

For the [**Column Separator**] option (marked in green), select the ***/*** (*forward slash*) character from the drop-down list.



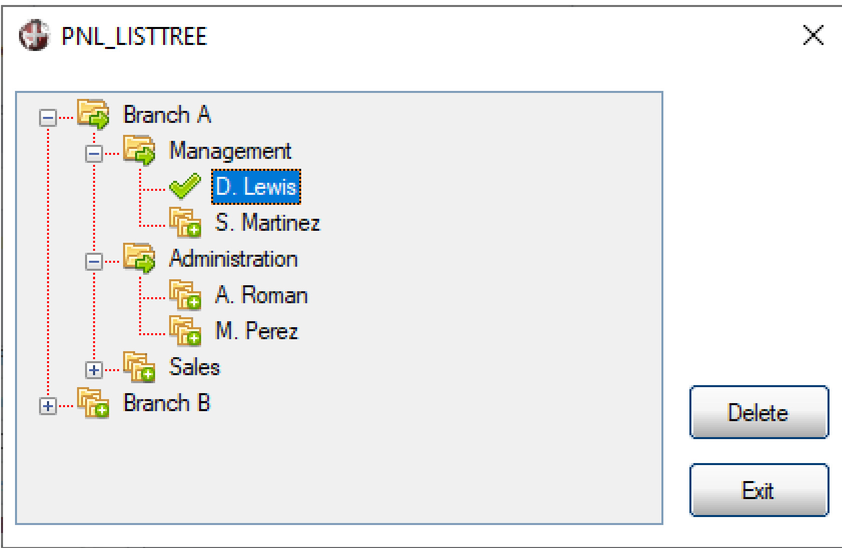
This type of control also allows you to assign images to the different states of the options.

Example: This shows the images that were added in the **Tree View Format Definition** window:



In addition to being able to assign images for the different states, other attributes allow you to draw connecting lines between entries.

In the **Font/Color** tab, the options for [**Line Color**] and [**Background**] color can also be changed (as shown below), among other things.



Although it physically looks like a different control, in reality, it is the same type of control with some differentiating characteristics, but we can use the same commands and considerations that we have with the other LIST_BOX type controls.

GRID Control

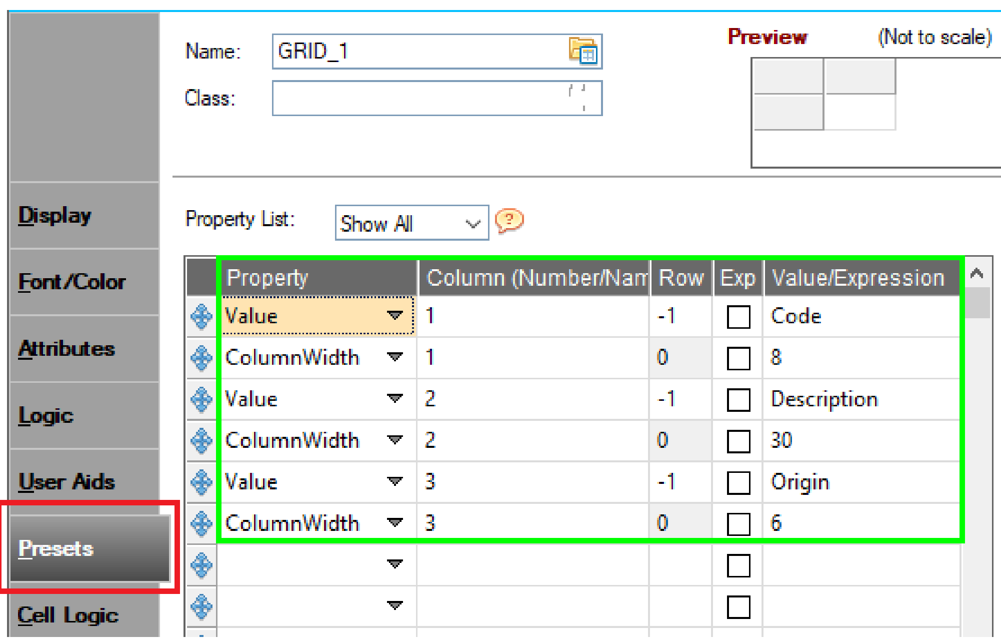
The **GRID** is perhaps the most complete control and also the most difficult to manage, since its functionality will be similar to having a spreadsheet and will be a collection of controls (text entry box type initially).

To manage the grid, we must know the dynamic properties of the control and define settings (**Presets** tab) that will help us have better control over our grid. The important thing is that managing a 4x4 cell grid is similar to managing one with 1,000 cells. We will start by defining a simple grid, and then, we will give it some additional functions.

Exercise: Defining a Grid Control

Let's look at the panel we previously created called **PANEL_GRID** where we defined a **GRID** control and added a Button with **End** logic for the **When Button Pressed** event.

The **Presets** settings are:

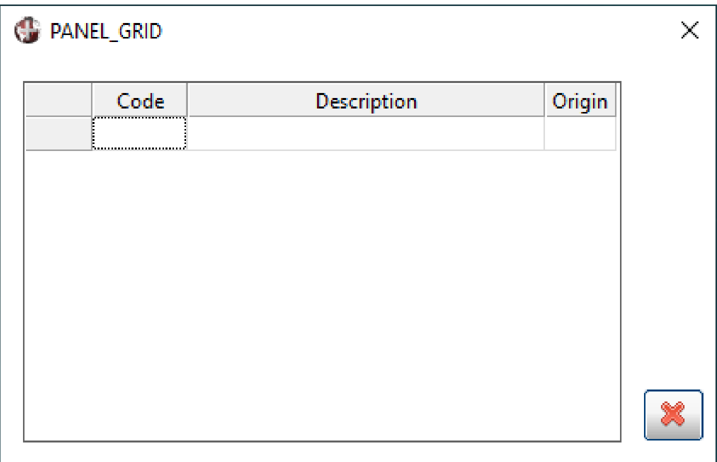


Column 1 is the Code and has a width of 8.

Column 2 is the Description and has a width of 30.

Column 3 is the Origin and has a width of 6.

When tested, the panel should look similar to the one shown below:



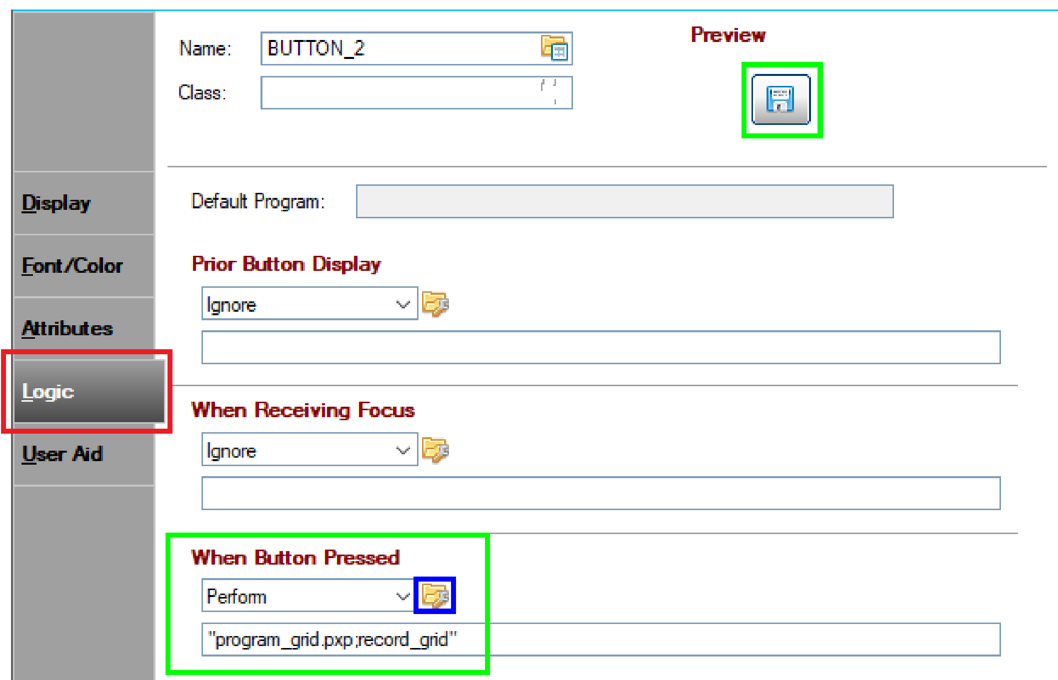
We can fill this GRID control from the keyboard or from a table. We will look at filling it from the keyboard.

When we run the program, we should determine where the user is, and if it is in the last column (Origin), then add a row at the end. So, if the user is in row 1, add a row 1+1 (row 2) and then continue loading. When the user gets to column 3 (Origin) and enters the data, we must add row 3, and so on:

```
! Cell value change check routine
CELL_CHANGE:
! We save in CX the current column (where we are in)
CX=GRID_1.CTL'CURRENTCOLUMN
! We save in CY the current row
CY=GRID_1.CTL'CURRENTROW
! We save in NUM_FIL the number of rows of the grid
NUM_FIL=GRID_1.CTL'ROWSHIGH
! If we are in column 3, let's add a row
if CX=3 then gosub ADD_ROW
return
! Routine to add row
ADD_ROW:
! We add a row after the last one
grid add GRID_1.CTL,0,NUM_FIL+1
return
```

Add a new Button control that allows you to record the contents of the grid. This button will have a diskette icon. (The previous button did not have text, so we will not put text on this one also.) Feel free to experiment and change the button appearance.

Once that is done, we must associate a **Record_Grid** routine to the **program_grid.pxp** program. We do this in the **Logic** tab of the **Button Properties** window. Remember that we can use the name we want as long as we are consistent with the change:



To open the program editor, click on the yellow folder icon (marked in blue).

For now, we are not going to save anything in any file. We are just going to mention the dimensions of the GRID control and the values:

```
! Routine to record GRID content
RECORD_GRID
! We initialize a variable data_list$
DATA_LIST$=""
! We save the number of rows in CAN_ROWS$
NUM_ROW=GRID_1.CTL'ROWSHIGH
NUM_ROW$=str(NUM_ROW)
for R=1 to NUM_ROW
  grid find GRID_1.CTL,1,R,CODE$
  grid find GRID_1.CTL,2,R,DESCR$
  grid find GRID_1.CTL,3,R,ORIG$
  DATA_LIST$+=CODE$+"/"+DESCR$+"/"+ORIG$+$0D0A$
Next R
msgbox "We have read: "+NUM_ROWS$+$0D0A$+DATA_LIST$,"Notice"
return
```

This routine begins by initializing or deleting a variable **DATA_LIST$**, which we will use to accumulate the data and display it on the screen. Then, we save the number of occupied rows (actually, we do it in two variables: a numeric one, **NUM_ROW**, and a literal one, **NUM_ROW$**). This is for convenience. We need one in ***numeric*** format for the **FOR..NEXT** loop and another in ***literal***

format (or text) to use in the **MSGBOX** command.

Then, we do a **FOR..NEXT** loop that will take a variable "R" from 1 to the number of rows that our GRID contains. In this loop, we are going to read the content of columns 1, 2 and 3, corresponding to CODE, DESCRIPTION and ORIGIN. We will store that data in the variables CODE$, DESCR$ and ORIG$.

Then, we accumulate this data in the variable DATA_LIST$ in the form:

DATA_LIST$+=CODE$+"/"+DESCR$+"/"+ORIG$+$0D0A$

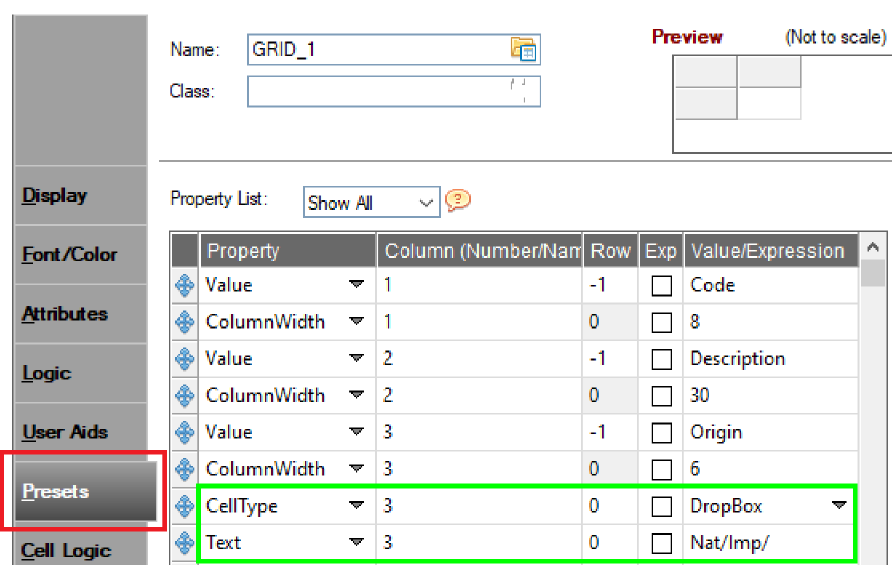
This indicates that the variable is going to accumulate content, a variable of the form A$=A$+B$ (or A$+=B$) concatenates or adds the content of the variable B$ to its current value. In the end, we use special characters that are $0A$ (Hexadecimal for the **Line Feed** character) and $0D$ (Hexadecimal for the **Carriage Return** character, which allow us to place another line in the message box of the **MSGBOX** command).

When the loop ends, marked with the command **NEXT R**, a message is displayed with the number of rows processed and the content of each one.

A grid can be read and processed in various ways. You are free to experiment and see which is most convenient or efficient for you.

Change the Origin so that it is a drop-down box where the user selects **Nat** for **National** and **Imp** for **Imported**. We are also going to validate the CODE field (Column 1) so that the user cannot enter more data than numbers and have a padding of 0s; that is, if the user enters "1", the system will fill in to complete "000001".

To change the type of cell located in Column 3 (corresponding to the Origin field) so that it is a **DROP_BOX** type cell, we must go to the properties of the **GRID** control and select the **Presets** tab. We will change the cell type and then load the DROP_BOX with the preset values (Nat/Imp/):



First, we must change the property **CellType** to Column 3, Row 0 so that it applies to the entire Column 3, and select the cell type as **DropBox**.

Then, we change the **Text** property for all of Column 3 (Column=3, Row=0) also, and prefix the values to **Nat/Imp/**.

Note: We must add the **/** (*forward slash*) at the end of **Nat/Imp/** because it is the value separator.

With this change, we now have Column 3 defined as a **DropBox** type and loaded with two values (Nat/Imp/).

Now, validate that the value entered in Column 1 (Code) is numeric and filled with 0s. We will do this validation at the programming level. It is done in the existing routine that is executed every time a cell value is changed.

The cell value change handling routine **CHANGE_CELL** will be modified to add this condition:

```
if CX=1 then gosub VALID_CODE
```

Basically, if CX, which is equivalent to the position of the focus in the grid, is 1 (it is in the first column, which is the CODE column), we are going to execute the **VALID_CODE** routine.

The **VALID_CODE** routine contains the following:

```
VALID_CODE:
! We read the content of the cell and store it in the variable CODE$
grid find GRID_1.CTL,1,CY,CODE$
NEW_CODE$=str(num(CODE$,err=CODE_NO_NUM):"00000")
grid load GRID_1.CTL,CX,CY,NEW_CODE$+sep
exit
!
CODE_NO_NUM:
grid load GRID_1.CTL,CX,CY,"*Err*"+sep
exit
```

Let's analyze each added line.

```
grid find GRID_1.CTL,1,CY,CODIGO$
```

The **grid find** command will read cell 1,CY from the GRID_1.CTL control (remember that CY has the current value of the row and was calculated in a previous line), and the value contained in this cell is saved in the variable CODE$.

```
NEW_CODE$=str(num(CODE$,err=CODE_NO_NUM):"00000")
```

We do an operation with nested functions. We convert the literal variable **CODE$** to numeric (**NUM()** function). If an error occurs during the conversion, the program goes to the routine **CODE_NO_NUM**. If the conversion is correct, we convert back to literal but with an output mask. This line is equivalent to:

```
COD_AUX=num(CODE$,err=CODE_NO_NUM
NEW_CODE$=str(COD_AUX):"00000"))
```

We convert the variable **CODE$** to numeric and store it in the variable COD_AUX. If the conversion generates an error (it cannot be converted from text to numeric), it branches to the

routine called **CODE_NO_NUM**.

The result of the conversion to numeric (the variable COD_AUX) is converted back to text or literal format but adding an output mask. In PxPlus, you can add a mask or output format to the variables.

Example:

```
A=10
PRINT A:"0000"
Result: 0010
```

After the numeric variable has been converted to a literal and stored in the variable **NEW_CODE$**, that value is loaded into the same cell with the 0s (zeroes) on the left:

```
grid load GRID_1.CTL,CX,CY,NEW_CODE$+sep
```

What the **CODE_NO_NUM** routine does is load the cell in question with the value "*Err*" so that the user realizes that the data entered is invalid:

```
CODE_NO_NUM:
grid load GRID_1.CTL,CX,CY,"*Err*"+sep
exit
```

The complete routine will look like this:

```
CHANGE_CELL:
CX=GRID_1.CTL'CURRENTCOLUMN
CY=GRID_1.CTL'CURRENTROW
NUM_ROW=GRID_1.CTL'ROWSHIGH
if CX=1 then gosub VALID_CODE
if CX=3 then gosub ADD_ROW
return
!
```

Let's see the complete listing of these routines:

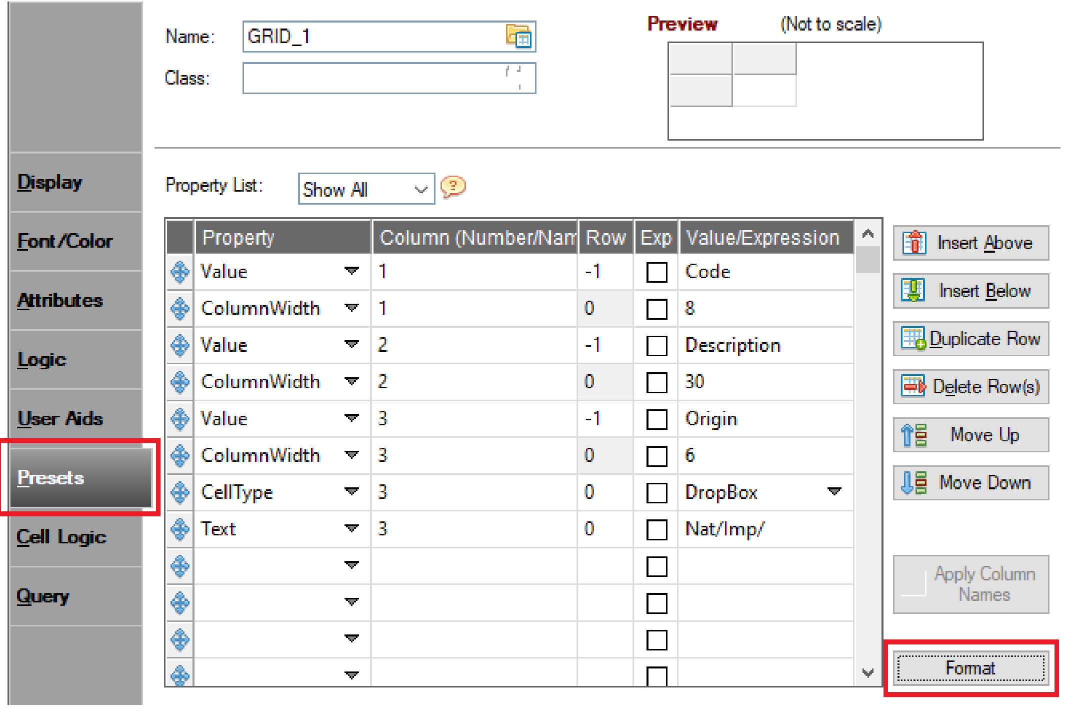
```
!
CHANGE_CELL:
CX=GRID_1.CTL'CURRENTCOLUMN
CY=GRID_1.CTL'CURRENTROW
NUM_ROW=GRID_1.CTL'ROWSHIGH
if CX=1 then gosub VALID_CODE
if CX=3 then gosub ADD_ROW
return

!
VALID_CODE:
grid find GRID_1.CTL,1,CY,CODE$
NEW_CODE$=str(num(CODE$,err=CODE_NO_NUM):"00000")
grid load GRID_1.CTL,CX,CY,NEW_CODE$+sep
exit
```

```
!
CODE_NO_NUM:
grid load GRID_1.CTL,CX,CY,"*Err*"+sep
exit
ADD_ROW:
grid add GRID_1.CTL,0,NUM_ROW+1
return
!
```

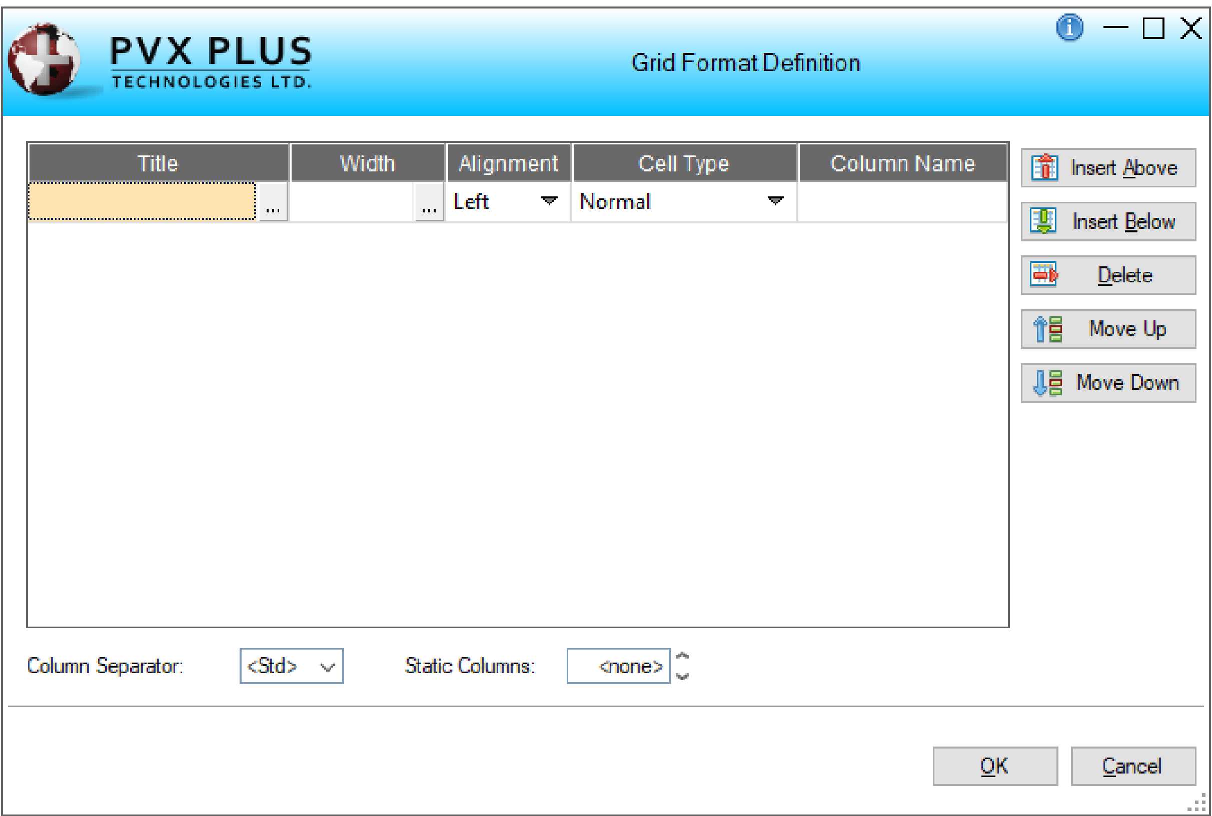
Note: Remember that the lines that begin with **!** (*exclamation point*) or with the **REM** command are considered comments and are not executed or taken into consideration.

In PxPlus, there are several ways to accomplish the same result: with commands or with dynamic properties or by means of adjustments in the NOMADS panel designer. We want to show you at least one way to do things, but remember that there is surely at least one other way to do it.

In addition to defining properties using the **Presets** panel, NOMADS incorporates a screen to define an additional format in your grid. To do this, go to the properties of the **GRID** control, and on the right side of the **Presets** panel, click the [**Format**] button.



Selecting the [**Format**] button (marked in red above) opens the **Grid Format Definition** window where you can make additional changes to your grid:

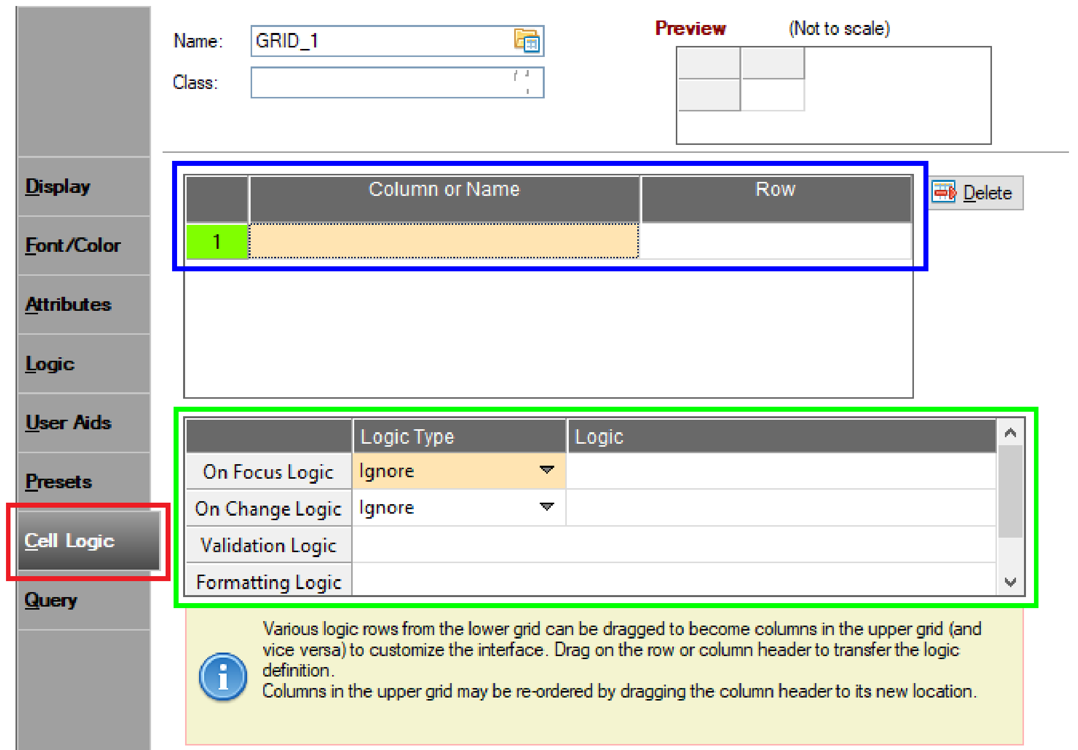


One of the things you can do in this panel is assign names to the columns. In our example, we used Column 1 for the Code, Column 2 for the Description and Column 3 for the Origin.

As it is a simple control (due to its number of columns), it is easy to associate each column with its data. But, if the grid had 20 or more columns, this identification would be much more complicated. For that, we can assign names to the columns, so that instead of referring to Column 3, we can refer to the **Origin** column, which is much simpler and more intuitive.

We must experiment with this control, which is the general recommendation. A **GRID** control can have text entry box or multi-line cells (normal), buttons (several styles), check boxes (several styles), drop-down boxes (several styles), queries, lookup buttons, graphics, images and many others. The style and appearance of the grid, such as colors, fills, lines, column alignment, etc., can also change.

Process the cells routine called **CHANGE_CELL**, but in reality, NOMADS allows the assignment of logical events for the cells individually. In the **Grid Properties** window, the **Cell Logic** tab is where you can assign different actions to cells individually or in groups:



We have two events: when the cell receives focus (**On Focus Logic**) and when it changes value (**On Change Logic**). We can assign an action to each event.

We can also assign **Validation Logic** and **Formatting Logic**. Notice that the **Validation** and **Formatting** logic that we did in our routine, we can do from this panel.

Note: Both the **Validation** and **Formatting** routines require a program routine with given arguments:

Validation: input$,columnid,rowid,msg_err$,label$,old_value$,eom_value$

Formatting: columnid,rowid,entry$,label$

Refer to **Independent Cell or Row Logic** for **GRID** controls in the PxPlus Help documentation.

It is also possible to associate a **Query** with a grid-type control, as well as define this control as a **Smart Control** so that its loading is automatic.

Refer to **Defining Smart Controls** in the PxPlus Help documentation.

12. Elements of Programming: Building Programs

No matter how powerful and flexible language is, in the end, to take full advantage of its power, we must fall into the manual solution, developing a routine or program that completely adapts to our needs.

An advantage of PxPlus is that, in addition to the large number of automatic tools, there is a powerful and complete backup language with many functions and commands, great versatility and flexibility. PxPlus incorporates hundreds of commands, functions, mnemonics, parameters, etc. We are not going to study them in detail, as we would have too much information that may be very difficult to process and understand.

Instead, we're going to take a one-step-at-a-time approach, making some simple programs or routines, and then adding other commands and functions. Knowing some commands and how they work makes it possible to investigate and understand the others, as well as the functions and other components of the language.

Creating a PxPlus Program

How do we create a PxPlus program?

After we have already gone through many pages of information, perhaps this question sounds out of place … but far from it. At the beginning, we wanted to demonstrate that many of the functions of PxPlus do not require programming knowledge and that it was possible to obtain results without programming. But, to obtain the best result, we must be able to adapt the behavior of the system to our requirements.

At the beginning, we said that PxPlus, when installed, creates more than 20 tool and program icons. There are two in particular that allow us to create and modify programs: the Integrated Development Environment (IDE) and the Command Line interpreter (PxPlus Command Line).

The first, which is already known to us, gives us access to all the functions and tools of the language in a friendly, complete menu with online help. The second is the "language" itself; that is, the command interpreter is the basis of everything. In fact, many years ago, this was the only program or option available.

It is actually possible to create programs using other editors, such as the **Visual Studio Code Extension**, the **Integrated Toolkit (*IT)** and the **ED+ Program Editor**. For now, we will concentrate on knowing the PxPlus Command Line environment.

Line numbers were, at one time, a common and distinctive feature of programming used to identify lines of code. While line numbers still work in PxPlus, they are now generally associated with legacy programs and not used in contemporary programming.

Note: Some code examples in this book use line numbers - this is for ***illustration purposes only***.

Introduction to PxPlus

Program Utilities Edit Debug Help

```
PxPlus-2024 Web (Ver:21.00/MS-WINDOWS) Serno:XXXXXXXXXXXXXXXX
(c) Copyright 2005-2024 PVX Plus Technologies Ltd. (All rights reserved)
  Website: http://www.pvxplus.com
->10 INPUT "Age: ",age
-:20 MONTHS=age*12
-:30 PRINT "YOU ARE: ",MONTHS," MONTHS OLD"
-:run
Age: 25
YOU ARE:  300 MONTHS OLD
-:
```

This command interpreter, as it is also known, is basically a program that interprets what you type and returns the result immediately. At this level (which is not where we traditionally create programs), interaction with the user is done with the **PRINT** command (to show) and the **INPUT** command (to ask).

Example:

```
PRINT "The multiplication is: ",3*8
The multiplication is:  24
```

The commands are executed as they are entered, giving an immediate answer, which allows the programmer to experiment in an easy way.

This immediacy presents a problem in some instances. The deferred mode or "program" mode allows us to enter a series of instructions that will not be executed until we give it the instruction to do so. The way we tell PxPlus that it should not execute the instructions is by means of a line number before the command:

```
10 INPUT "Age: ",age
20 MONTHS=age*12
30 PRINT "YOU ARE: ",MONTHS," MONTHS OLD"
```

To run this "program", we must execute the **RUN** command:

You must enter the program lines (with a number at the front) and execute them using the **RUN** command. Then, the command interpreter will execute them in increasing numeric order.

Although line numbers are used very little (in some cases, they are not used at all), it is customary to place the numbers by 10 in case you need to enter a line between them.

Example:

```
10 INPUT "Age: ",age
15 IF age=0 THEN END
20 MONTHS=age*12
30 PRINT "YOU ARE: ",MONTHS," MONTHS OLD"
```

The inserted line 15 tells the program that if you enter a value equal to zero, the program will end (and will not execute the following lines).

Note: Program line numbers must be positive integers: 10, 20, 30, and so on.

This program will be deleted from memory when we leave PxPlus, so we must be able to save it on disk to use it at another time. For this, we have the commands:

```
! To save from memory to disk:
SAVE "PROGRAM_NAME"
! To recover from disk to memory:
LOAD "PROGRAM_NAME"
```

Once the program is in memory, it can be executed. We can do the combined **LOAD** and **RUN** commands so that a program is loaded into memory and executed:

```
RUN "PROGRAM_NAME"
```

If the program does not exist (or is in another part of the disk), an Error 12 will occur, which indicates that the program was not found.

Note: Remember that some operating systems are case sensitive, so "PROG", "Prog" and "prog" could be different programs.

Remember the **PREFIX** command? This command allows you to specify the paths or folders where PxPlus will search for programs (also files) to load them from disk to memory.

If we want to clean the work area (memory) to start from scratch (delete the program), we must use the **DELETE** command or the **START** command.

To view the content of a program that is in memory, we use the **LIST** command.

To summarize:

We have the following commands:

INPUT	Allows you to enter data from the keyboard.
PRINT	Displays information on the screen.
END	Ends the execution of a program.
IF…THEN	Condition to evaluate or compare data.
SAVE	Store a memory program to disk (requires a name).
LOAD	Load a program from disk to memory.
LIST	Shows a list of the contents of the program in memory.
RUN	Run a program.
DELETE	Removes currently loaded program from memory.
START	Reinitializes the current PxPlus session.

Note: There are differences between **DELETE** and **START**. Refer to the **DELETE** and **START** directives in the PxPlus Help documentation.

We use the **REM** command or the **!** (*exclamation point*) to indicate that that line is a comment and its contents will not be executed.

In the case of assignments, such as:

```
A=B*2
```

You could have the optional **LET** command:

```
LET A=B*2
```

Routines and Branching

In the execution of a program, the execution is linear; that is, the smallest program line number will be executed first and will go in increasing order. But sometimes, we need to change the execution flow to go to another part of the program, mainly after a comparison. For this, we have two commands: **GOTO** and **GOSUB/RETURN**:

```
! If the variable COND equals 2, go to the routine called LABEL
IF COND=2 THEN GOTO LABEL
! It is also possible to go directly to a line number
GOTO 300
```

The difference between **GOTO** and **GOSUB/RETURN** is that **GOTO** changes the execution flow of the program permanently. (**GOTO** causes the next instruction to be executed, and from then on, the execution will continue, always executing the next line in ascending numeric form). On the other hand, **GOSUB** will make the program go to a routine, but when it encounters the **RETURN** instruction, it will return to the instruction following the **GOSUB** command that originated it.

Example:

```
 10 INPUT "CODE: ",COD
 20 IF COD=0 THEN GOTO 100
 30 PRINT 2*3
 40 END
100 PRINT "CODE IS ZERO, ENDING"
110 END
```

If the variable COD is equal to 0, the program will go to line 100 where it shows the message, and the program ends (line 110). If the condition is not met, the following lines 30 and 40 are executed. Since an END is found on line 40, the program ends.

Example:

If the variable COD is 0, it will go to the routine on line 100. It will be executed, and upon finding the **RETURN** command (return, line 110), it will return to the next line (line 30) where it will continue execution until finding the **END** command on line 40.

```
 10 INPUT "CODE: ",COD
 20 IF COD=0 THEN GOSUB 100
 30 PRINT 2*3
 40 END
100 PRINT "CODE IS ZERO, RETURNING"
110 RETURN
```

It is possible to place a program label so you don't have to deal with numbers:

```
 10 INPUT "CODE: ",COD
 20 IF COD=0 THEN GOSUB ROUTINE
 30 PRINT 2*3
 40 END
100 ROUTINE: PRINT "CODE IS ZERO, RETURNING"
110 RETURN
```

It is not necessary to enter line numbers, and we concentrate on the labels, which is basically a title that ends with a **:** (*colon*) at the beginning of a line:

```
Label:
PRINT "Here comes"
Label_two: PRINT "Can be here"
```

Loops and Repetitive Cycles

Another widely used structure is the ***loop structure***, the segments or program routines that are executed several times until a condition is met or a certain value is reached. In PxPlus, there are several types of loops. The most common are:

FOR..NEXT
WHILE..WEND
SWITCH..END SWITCH
REPEAT..UNTIL
TRY..END_TRY

Let's look at each one.

FOR..STEP..NEXT Loop

This loop serves to take the value of a variable from an initial value to a final value in given increments and will repeat the lines between the **FOR** command and the **NEXT** command that indicates the end of the loop instructions.

Example:

```
SUM=0
  FOR A=100 TO 200 STEP 5
    SUM=SUM+A
NEXT A
PRINT "THE SUM IS: ",SUM
```

This example initializes the SUM variable to ensure that it is equal to zero. Then, it will take the variable A from 100 to 200 in increments of 5; that is, A will be equal to 100, 105, 110, 115, etc. until it reaches (or passes) the value 200. As soon as it is greater, the loop will end, and the instruction following the NEXT A command will be executed.

```
FOR I=1 TO 100
  PRINT "*",
NEXT I
```

This second example will show 100 "*" on the screen.

It is possible to make decrements (from a higher value to a lower one) if the **STEP** value is negative:

```
FOR I=100 TO 1 STEP -1
  PRINT "*",
NEXT I
```

WHILE..WEND Loop

The **WHILE..WEND** loop is used to execute the instructions in it while a given condition is met. In this example, the instructions (marked as "…") will be executed while the variable C is different from 0.

```
WHILE C<>0
   …
WEND
```

The middle routine will have to, at some point, change the value of C to make it equal to 0 to avoid an infinite loop.

Example:

```
WHILE RESP$<>"S"
    INPUT "Age: ",age
    PRINT "His age in months is: ",age*12
    INPUT "Do you want to exit: ",resp$
WEND
```

REPEAT..UNTIL Loop

This loop is similar to the **WHILE..WEND** loop, but since the condition is at the end, it will be executed at least once. The basic syntax is:

```
REPEAT
    Instructions
UNTIL Condition
```

Example:

Let's look at a program that will calculate and subtract 10% from a given value until the rest is less than 100:

```
INPUT "Value: ",V
REPEAT
    PRCT=V*0.10
    V=V-PRCT
    PRINT "Remain: ",V
UNTIL V<100
PRINT "Ended with: ",V
```

Let's see the execution of the program:

```
0010   INPUT "Value: ",V
0020   REPEAT
0030   LET PRCT=V*.10
0040   LET V=V-PRCT
0050   PRINT "Remain: ",V
0060   UNTIL V<100
0070   PRINT "Ended with: ",V
-:run
Value: 200

Remain: 180
Remain: 162
Remain: 145.8
Remain: 131.22
Remain: 118.1
Remain: 106.29
Remain: 95.66
Ended with: 95.66
```

SWITCH..END SWITCH Loop

This loop structure is more complete than the previous ones since it allows several routines to be executed, depending on the value of a variable or condition. There's nothing better than an example to illustrate its operation.

Example:

Let's look at an example where there is a 10% commission for sellers type "A" and "D". If the seller is type "B", he has no commission, but if he is type "C", the commission will be 10% but deducting expenses incurred. All other types of sellers have 5% commission:

```
SWITCH SELL$
CASE "A", "D"
  COMMISSION=SALES*0.10
CASE "B"
  COMMISSION=0
CASE "C"
  COMMISSION=(SALES-EXPENSES)*0.10
DEFAULT
  COMMISSION=SALES*0.05
END SWITCH
```

You have a **SWITCH..END SWITCH** loop, and in the middle, a group of conditions or cases that will be evaluated, depending on the value of the variable specified in the **SWITCH** command.

TRY..END_TRY Loop

This loop is quite special and is used when we want to deal with a situation that will potentially generate errors, and a routine is provided in case it does.

The basic syntax is:

```
TRY
  [Block1] Routine1 to be executed
CATCH
  [Block2] If routine1 generates an error, the instructions in this block are executed
FINALLY [Optional]
  [Block3] This block of instructions will be executed finally
END_TRY
```

The way that the **TRY..END_TRY** loop works is by executing a routine called [**Block1**]. If the routine is executed without problems (and the **FINALLY** routine exists, routine [**Block3**] will be executed). If an error is generated when executing [**Block1**], then after the error is generated, [**Block2**] will be executed, and when it finishes (if **FINALLY** exists), [**Block3**] will be executed.

We can summarize by saying that this loop can have two or three blocks: **TRY**, **CATCH** and **FINALLY**. The first block is executed. If there is an error, the second block is executed. After executing both, the third block will be executed.

Let's look at a routine that will try to process a working file (which might eventually become unavailable) and copy its contents to another file:

```
TRY
    channel=unt
    open(channel)"WORK_FILE"
    channel_dest=unt
    open(channel_dest)"DESTINATION_FILE"
    list: iolist code$,description$,cost$,pvp$
    dest_list: iolist code$, description$, cost$, pvp$, quantity, branch$

    cycle:
    read(channel,end=end_cycle)iol=list
    write(channel_dest,key=code$+branch$)iol=dest_list
    goto cycle

    end_cycle:
     close(channel),(channel_dest)
CATCH
    channel=unt
    open(channel)"LOG_FILE"
    write(channel)"The work file was not processed"
    write(channel)"An error occurred: "+dte(0)
    close(channel)
```

```
FINALLY
   ! Some routine that we want to execute independently
   ! whether the first routine is executed or not
END_TRY
```

In this routine, we open two files, "WORK_FILE" and "DESTINATION_FILE". We will try to copy the content of the first file to the second. In case of an error, the CATCH routine will be executed where a file, "LOG_FILE", will be opened, where the incident will be recorded. If there were instructions in the FINALLY part, they would be executed whether the file was copied or the copy failed. We will comment on the most notable instructions.

See what we do:

```
channel=unt
open(channel)"WORK_FILE"
channel_dest=unt
open(channel_dest)"DESTINATION_FILE"
```

And not:

```
channel=unt
channel_dest=unt
open(channel)"WORK_FILE"
open(channel_dest)"DESTINATION_FILE"
```

This is because if the new assignment is made to the **UNT** variable, both the **CAN** and **CAN_DEST** variables will have the same value, and an error will be generated. We have defined two lists of input and output variables (**IOLIST** command). In reality, we can have any quantity defined.

In the **WRITE** statement, we are combining several variables at the time of writing. This is perfectly valid (we can do it at any time). At the end of the loop, we close both channels with a single **CLOSE** command. CLOSE (1),(2) is valid.

The last instruction to highlight is the **WRITE** in the **CATCH** section that includes a **DTE()** (Date) function. We will see the use of the functions later.

PxPlus Program Components

PxPlus is a language rich in commands, but in reality, not all of them are commands. There are also functions, parameters, variables, mnemonics, etc. It is quite possible that, at first, we feel confused about what each of these is, but we will try to simplify this so that we can process each element of information without so much stress. Let's look at a basic definition for each of them. Later, we will see some of the most notable in each case.

Directives

A ***directive*** (also called "command") is the main component of the language. Directives normally execute one or several actions; some work on files, others on programs.

The best known directives are **INPUT**, **PRINT**, **GOTO**, **GOSUB**, etc. Some commands are direct execution. They are not executed ***inside*** the program but are used to manipulate programs, such as **LOAD**, **SAVE**, **END**, **RUN**, **LIST**, etc.

Functions

A ***function*** is a language component that will modify or calculate some data and will return a value as a result. In PxPlus, most functions are three letters long, and their arguments (on which they will be executed) are enclosed in parentheses.

Examples of functions are **LEN()**, **DTE()**, **NUM()** and **STR()**.

Variables

As the name indicates, a ***variable*** is a changing value structure. PxPlus offers, for the convenience of the programmer, a series of variables that help us determine the state of some components. To differentiate them from the variables we use in our programs, these are called **System Variables**. Many of the system variables also have three letter names.

Examples of system variables are **UNT**, **HFN**, **SEP** and **CTL**.

Mnemonics

A ***mnemonic*** is a modifier of an input or output command on the screen or keyboard. Mnemonics basically serve to alter the way some data will be presented. **Example:** We can use a mnemonic to tell the printer to use a certain type of font (Bold, Normal, etc.).

Objects

An ***object*** is a self-contained entity with specific characteristics and functions that complies with a series of laws or conditions.

Examples of objects are different graphical controls, a program (not all programs can be considered an "object"), some external entities, among others. An object belongs to a type or class (**Example:** "employee class" or "button" - all buttons, although different, have similar characteristics and functions).

Parameters

A ***parameter*** is a value that will modify the regular behavior of PxPlus; for example, not generate errors in some conditions or change the way the variables and commands are presented in the lists.

Arguments

An ***argument*** is data that is supplied to a routine or function to be processed. One argument could be the value the function will use to perform the calculation, such as X in the example root=SQR(X). In this case, the X would be the argument of the SQR (Square Root) function.

Programs

A ***program*** is a collection of instructions that have a specific function or purpose. Programs are stored using the **SAVE** command and executed using the **RUN** command.

Public Program

A ***public program*** is a program that is executed from another previous program.

Introduction to Language Directives (Commands)

We will look at some of the most commonly used directives (commands) in the language.

For a complete list, refer to the **Directives** section in the PxPlus Help documentation.

ACCEPT

Allows you to read one key on the keyboard and does not show it on the screen.

Example:

Message routine:

```
PRINT @(0,22),'BLUE', "Press any key to continue: ",
ACCEPT AUX$
```

CALL

Allows you to call a program or routine that is on disk and execute it. Once the execution finishes, it will continue with the program that called it. The program called is called a "public program." You can suffix the label of the program you want to run simply by adding a **;** (*semi-colon*) and the name of the tag in question.

Example 1:

Call the "Header" routine of the program "PROG01" with the IOLIST of line 8000 as an argument. Call a public program called "DATE" with a variable as an argument:

```
0010 CALL "PROG01;Header",IOL=8000
8000 IOLIST CED$, COD$, NAME$
CALL "DATE",DATE$
```

Example 2:

Call the "Header" routine of the program "PROG01" with two arguments, X$,Y$:

```
CALL "PROG01;Header",X$,Y$
```

CASE/SWITCH

We can use structures that have a much clearer logic for the programmer and make it easier to understand the program.

Example:

```
SWITCH X$
CASE "Dog", "Parrot", "Pig"
    PRINT "Selected: ",X$
CASE "Goat"
    PRINT "Selected Goat"
CASE "Parrot"
    PRINT "Selected Parrot"
CASE > "Zebra"
    PRINT "Selected something older than Zebra"
DEFAULT:
    PRINT "Default selection"
    ! None of the above
END SWITCH
```

CLIP_BOARD

Allows you to read and write data to the Windows Clipboard (for temporary storage).

Example 1:

Write "CKX" to the clipboard and then read them and store them in A$:

```
CLIP_BOARD WRITE "CKX" ! Write data
CLIP_BOARD READ A$ ! Read data from the clipboard
```

Example 2:

Read a record based on an ID; prepare and send as data to Microsoft Excel ($0D0A$ is added to the end of each data):

```
1000   INPUT "Customer ID:",CED$
1010   READ (1,KEY=CED$)CED$,NOM$,CIU$
1020   CLIP_BOARD WRITE CED$+" "+NOM$+$0D0A$+CIU$+$0D0A$
1030   PRINT "Information copied."
```

DAY_FORMAT

We can define the output format of the DAY variable using the characters M, D, and Y to adapt it to our needs.

Example:

```
DAY_FORMAT "DD/MM/YYYY"
PRINT DAY = "06/14/1999"
```

DEFCTL

We can assign new CTL values to any set of ASCII values.

Example:

Assign CTL=40 to the sequence EscA (ESCape + "A"):

```
DEFCTL $1B41$=40
```

DIM

The **DIM** directive has three functions. It can be used to define an array, initialize a string, or define a structured string.

Use the **DIM** directive to define an array with one, two or three dimensions. The dimensions of the array are defined as [minimum]:maximum. The minimum value is optional and defaults to 0.

Example:

DIM X[10]	Creates a numeric array with 11 elements X[0] through X[10]
DIM X$[1:2,1:3]	Creates a two-dimensional string array with 6 elements X$[1,1] through X$[2,3]

DIM can be used to define a string variable of a specific value and length.

Example:

DIM A$(5,"*")	Yields the same as A$="*****"
DIM B$(4)	Yields the same as B$=" " (i.e. four spaces)

You can combine the above to pre-initialize all the elements of a string array.

Example:

DIM ACCT_ID$[10](6,"0")   Initializes the array elements with 6 zeroes.

DIM can also be used to create Composite Strings, which are compound literals formed by other variables. Refer to **Composite Strings** in the PxPlus Help documentation.

Example:

```
0010 IOLIST NAME$,ADDRESS$,BALANCE
0100 DIM CST$:IOL=0010 ! The variables will be called CST.NAME$, CST.ADDRESS$ and
CST.BALANCE
CST.NAME$="Tom",CST.ADDRESS$="Mytown",CST.BALANCE=2
```

The set of variables can be referenced by CST$, yielding:

```
Tom+SEP+Mytown+SEP+2+SEP
```

ENTER

Used in a public program, it accepts the arguments that are sent from the program through a list of variables. We can define an IOLIST as part of the input arguments in the execution of a public program.

Example:

Define the input list of arguments with the IOLIST on line 8000. Accept a list of three arguments from the calling program:

```
0010 ENTER IOL=8000
8000 IOLIST CODE$,NAME$
ENTER PROD_ID$,AMOUNT,CODE$
```

END

Ends program execution.

Example:

End execution at line 2500:

```
2500 END
```

ERROR_HANDLER

Allows you to capture errors through a public program in those programs that do not have any type of validation or error capture.

Example:

Set "MAN_ERROR" as the error handling program:

```
ERROR_HANDLER "MAN_ERROR"
```

EXIT

Ends the execution of a routine and returns to the previous program.

Example:

Terminates the execution of the routine at line 2500 and will return to the calling program:

```
2500 EXIT
```

FOR..NEXT

Establishes a repetitive loop where it takes a given variable from an initial value to a final value. The **NEXT** command marks the end of the routine that will repeat during the loop.

Example:

Take the variable CONT from 1 to 200:

```
FOR CONT=1 TO 200
! These instructions will be repeated 200 times
NEXT CONT
```

GOSUB

Temporarily changes the flow of regular execution of a program, causing it to deviate or change to another line or routine until the **RETURN** command is found, which will return it to the instruction following the **GOSUB** command.

Example:

Branch to CHANGE_DATE routine:

```
GOSUB CHANGE_DATE
```

GOTO

Changes the regular execution flow of a program, causing it to deviate or change to another line or routine.

Example:

Branch to line 2000. Branch to the routine CHANGE_DATE:

```
GOTO 2000
GOTO CHANGE_DATE
```

INPUT

Allows you to enter information from the keyboard (If no channel is specified or channel 0 is used) or read information from a file or device.

Example:

Ask for the user's name:

```
INPUT "Enter your name: ",NAME$
```

INVOKE

Allows you to execute applications (commands) external to the language: operating system commands, other programs, etc.

Example:

Run the Windows Calculator program:

```
0250 INVOKE "CALC.EXE"
```

This command can be omitted in the program lines. It has two possible modifiers: HIDE and WAIT. The first serves to minimize the application when executing it, and the second, to stop or pause your work session while the application is running.

Examples:

```
0250 INVOKE "CALC.EXE"

0430 "CALC.EXE" ! Note that the INVOKE command is missing

0810 INVOKE HIDE "CALC.EXE"

0940 INVOKE WAIT "CALC.EXE"
```

LET

This command is optional and can be disabled. It allows you to assign a value or expression to a variable.

Example:

Assign the result of the addition to variable A. Concatenate NAME$ and LAST NAME$:

```
LET A=89+D
LET COMP_NAME$=NAME$+SURNAME$
```

LIKE

Allows comparisons to be made between a word and a mask or expression according to the mask format specified to the right of the command. All regular expressions of the **MSK()** function are supported. Is also used in OOP (Object-Oriented Programming) to inherit class attributes.

Example:

```
IF A$ LIKE MASK$ THEN ...
```

For Thoroughbred® users, the **'TL'** system parameter has been added, which allows you to switch between the extended mode of the language and the traditional method of Thoroughbred®.

When **'TL'** is Off, it is used according to the expression performed by the **MSK()** function. When On, **LIKE** behaves in a simplified manner; this is similar to Thoroughbred®'s LIKE.

(Thoroughbred® is a registered trademark of Thoroughbred Software International Inc.)

LINE_SWITCH

Allows you to modify the device associated with channel zero (0) to be used with another file or device instead of the terminal.

Example:

Associate the input/output device to the COM2 port of the computer:

```
0010 OPEN (1,OPT="9600,n,8,1") "COM2" ! Open serial port
0020 CALL "*dev/ansi" ! Load ANSI device
0030 LINE_SWITCH (1)
```

LIST

Displays the content of a program in memory on the terminal screen. It can be replaced with the **/** (*forward slash*) symbol.

Example:

List the entire program. List from line 200 to 400:

```
LIST
/200,400
```

LOCAL

Allows variables to be used locally in a routine without altering its original contents.

Example:

Define the variables VAR and BALANCE$ as LOCAL:

```
LOCAL VAR,BALANCE$
```

MERGE

Allows you to copy the contents of a flat file to memory without having to open the file. It works with flat and indexed files.

Example:

Merge the program in memory with the program found in the flat file "Archive":

```
MERGE "File"
! Alternate method (Other Languages)
CHANNEL=UNT
OPEN (CHANNEL)"File"
MERGE (CHANNEL)
CLOSE (CHANNEL)
```

MESSAGE_LIB

Used to define the source file for user messages, which are accessed through the **MSG()** function.

Example:

Define the file "File.ES" as a message library:

```
MESSAGE_LIB "File.ES"
```

MNEMONIC

Used to create new mnemonics for devices not directly supported by the system.

Example:

Define the mnemonic 'JH' as ESC+A:

```
MNEMONIC (LFO)'JH'=ESC+"A"
```

NEXT

It is the end of loop indicator for the **FOR** command. This command also works without the need to specify the name of the variable corresponding to the **FOR** (that is, it can be done NEXT A or NEXT).

Example:

```
FOR I=1 TO 100
S=S+I
NEXT I ! The variable is used (declared)
FOR I=1 TO 100
S=S+I
NEXT! Without using (declaring) the variable
```

NEXT RECORD

Used in conjunction with the **SELECT** command to indicate the end of a **SELECT** loop.

Refer to the **SELECT** command in the PxPlus Help documentation.

OBTAIN

Works the same as the **INPUT** command, but the entry of the characters is not shown on the screen. Ideal for reading the keyboard, entering passwords, etc.

Example:

```
OBTAIN AUX$
```

OPEN

Allows communication to be established with a file or device through a logical channel. It is possible to open the file by specifying a mode (blocking, blanking, entry only, with cache), and it also allows specifying multiple files.

Examples:

Open FILE$ through the channel using the different alternatives:

```
OPEN LOCK(channel)FILE$ ! Open and lock
OPEN PURGE(channel)FILE$ ! Open and purge
OPEN INPUT(channel)FILE$ ! Open read only
OPEN LOAD(channel)FILE$ ! Open read only with cache
OPEN(1)A1$,(2)A2$,(3)A3$ ! Open multiple files
```

PASSWORD

Allows the assignment and removal of access keys for programs. Programs protected with this command cannot be listed or edited without first removing the password. To assign a password to a program, simply load the program (via **LOAD**), enter the command **PASSWORD** followed by the password enclosed in " (*quotes*), and save the program again.

Example:

How to protect the "NOTPRO" program:

```
LOAD "NOTPRO"
PASSWORD "MICKEY" SAVE
LOAD "NOTPRO"
LIST

Error #52: Program is password protected
```

PERFORM

The **PERFORM** command can be considered an external **GOSUB** command. All the variables are passed to the program executed via **PERFORM**. If, within this program, there is a **RETURN** and there was no previous **GOSUB**, the program will consider it as if it were an **EXIT** command.

Example:

Execution of the "Routine" routine of the "MYPROGRAM" program and also set the example to execute a program called "/PXP/CKX/PROG":

```
PERFORM "MYPROGRAM;Routine"
PERFORM "/PXP/CKX/PROG"
```

POP

The **POP** command allows you to eliminate the last level of the stack; for example, when you have an open, pending loop (**FOR..NEXT**, **GOSUB..RETURN**).

Example:

Remove the last executed GOSUB (its return):

```
250 GOSUB CYCLE
...
1000 CYCLE:
...
1250 POP
1260 GOTO ROUTINE
```

PREFIX

The PxPlus work environment provides the possibility of emulating an environment with several paths or directories through the use of the **PREFIX** command. Internally, PxPlus can maintain several different prefixes, each with a different route.

Example:

Loading and using various types of **PREFIX**:

```
PREFIX (0)"/usr/pxp/ /home/data/"
PRINT "DISK PREFIX #0: ",PFX(0)
PREFIX PROGRAM "/usr/pxp/ /usr/pgms/"
PRINT "PROGRAM PREFIX: ",PFX(PGN)
```

PREINPUT

Allows the simulation of data entry via program. It is possible to create several PREINPUTs. They will be processed according to their order of arrival (FIFO).

Example:

Simulate that the user enters "Jean Hendrickx", and then enters a modification to a program line:

```
PREINPUT "Jean Hendrickx"
PREINPUT "1100 INPUT A$"
```

PROCESS

Used to execute a panel (form) developed in NOMADS from your program.

Example:

Run the PanelA panel found in the BANKS library. We have another example where the library is found in the global variable %PROJECT$, and the language is in the global variable %LANGUAGE$:

```
PROCESS "PANELA", "BANKS.EN"
PROCESS "PANEL", %PROJECT$+"."+LANGUAGE$
```

PURGE

Deletes the content of the specified file. The file must be open and locked for this command to be executed.

Example:

Erases all the records in the file "DATA.DAT":

```
OPEN (channel)"DATA.DAT"
LOCK (channel); PURGE (channel)
CLOSE (channel)
```

QUIT

Used to terminate the execution of the PxPlus language. Other commands that end the language are **BYE** and **RELEASE**.

Example:

End the program:

```
1050 QUIT
```

READ

Allows reading the information from a file or device, normally opened by a logical channel.

Example:

Read the record identified with a key stored in PROD_ID$:

```
READ(channel,key=PROD_ID$)REG$
```

READ DATA

Allows reading the information stored in a literal and processes it (stores it) in the variables of an IOLIST.

Example:

Read the record, and if it is type "A", we process it with the IOLIST on line 8000; otherwise, we process it with the IOLIST on line 9000:

```
0100 READ RECORD (channel)REG$
0200 IF REG$(1,1)="A" THEN READ DATA FROM REG$ TO IOL=8000 ELSE READ DATA
FROM REG$ TO IOL=9000
8000 IOLIST CODE$,NOM$,BALANCE
9000 IOLIST CED$,NOM$,CIU$
```

Clear all IOLIST variables on line 8000:

```
READ DATA FROM "" TO IOL=8000
```

A variant is the following, in which the data is read from the **DATA** commands (see **DATA** and **RESTORE** commands) and is stored in DATA$:

```
DATA "Jean Hendrickx","Pedro Perez","Miguel Martínez"
CYCLE:
    DATA READ NAME$
    PRINT NAME$
GOTO CYCLE
->RUN
Jean Hendrickx
Peter Perez
Miguel Martinez

Error #2: END-OF-FILE on read or File full on write
```

REFILE

Deletes the contents of the specified file, similar to the **PURGE** command, but the file does not have to be open.

Example:

Clears the content of the "DATA.DAT" file:

```
REFILE "DATA.DAT"
```

RENUMBER

Lists the lines of the program that are in memory and it resolves all references and line changes. You can use the REM sequence **nn** to force (if possible) the next line to be number **nn**.

Example:

Renumber the program but leave the "ROUTINE" label on line 1000:

```
11 PRINT
12 PRINT NOTHING$
14 REM 990
19 ROUTINE: ! This line will stay as line 1000
RENUMBER
LIST
0010 PRINT
0020 PRINT NOTHING$
0990 REM 990
1000 ROUTINE: ! This line will stay as line 1000
```

REPEAT

Used in conjunction with the **UNTIL** command to define a loop that will be executed until the condition specified with the **UNTIL** command is met.

Example:

Run loop until MONEY<=0:

```
0090 PRINT "How much money does he give to his wife"
0100 INPUT "How much does he earn per month:", MONEY
0110 REPEAT
0120 INPUT "Money for expenses: ",EXPENSES
0130 MONEY-=EXPENSES! Equivalent to: MONEY=MONEY-EXPENSES
0140 UNTIL MONEY<=0
0150 ! Let's see if you have more money left
```

RESTORE

Used to restore the pointer to the in-memory data specification (using the **DATA** command).

Example:

Restore the pointer, depending on the language:

```
1000 DATA "One","Two","Three","Four","Five"
2000 DATA "One","Two","Three","Four","Five"
3000 INPUT "You want the Spanish/English (E/I) language: ",LANGUAGE$
3010 IF CVS(LANGUAGE$,4)="I" THEN RESTORE 2000 ELSE RESTORE 1000
```

RUN

Allows you to execute a program in memory. It is also used to execute a routine within a larger program, or a program on disk.

Example:

Execute the ROUTINE routine in the PROGRAM program:

```
RUN "PROGRAM; ROUTINE"
```

SELECT

Allows you to open, read and consult records within a file. It must end its reading cycle with the **NEXT RECORD** command. You can use the **WHERE** clause to filter or condition the reading of records. You can end the reading by making an **EXITTO** clause (the channel will be closed).

Example:

Read the records from the letter "A" to the letter "J" with CITY="CARACAS":

```
1000 SELECT IOL=0100 FROM "FILE" BEGIN "A" END "J" WHERE CITY$=1000:"CARACAS"
1010 PRINT REC(IOL=0100)
1020 NEXT RECORD
```

SETCTL

Intercepts the CTL values in the **INPUT** command and transfers control of the program (using **GOSUB**) to the specified statement. This subroutine must end with a **RETURN** command. Control of the program will return to the **INPUT** that caused the execution of the routine in question.

Example:

Program CTL=4 as screen refresh:

```
0010 SETCTL 4:2000
0230 INPUT (0,SIZ=1)"Name: ",N$
2000 ! Screen refresh
2010 PRINT 'RS',
2020 RETURN
```

SETESC

This command will intercept the interruption of a program (via the keyboard sequence <Ctrl - C> or <Ctrl - Break>).

Example:

Specify the "MABRK" program as an interrupt handler:

```
0010 SETESC "MABRK"
```

SETFID

Allows you to dynamically change the **FID** value of a file. Normally, it is used to modify the **FID** of channel zero (0), that is, FID(0), but this command allows you to change the **FID** value of any open file.

Example:

Change the FID(0) as "T44":

```
SETFID "T44"
```

SETMOUSE

Used to define areas of the screen or text strings as mouse sensitive where, if the mouse pointer is pressed or released, a CTL value is generated or a specific command is executed.

Example 1:

Make the box from @(4,4) to position @(8,8) generate a CTL equal to 10 when the mouse is pressed inside it:

```
SETMOUSE @(4,4,4,4)=4 ! The 3rd and 4th values are width and height of the region, not screen positions
```

Example 2:

The **SETMOUSE** command also accepts screen coordinates that are decimal:

```
SETMOUSE @(2.5,3,10.5,1.2)=4
```

Example 3:

Make the "COPY.PRG" program be executed via CALL when the user clicks with the mouse on the "Copy" text:

```
SETMOUSE "Copy"="CALL ""COPY.PRG"""
```

SETPARAM

Used to modify the values of system parameters. It complements the **PRM()** function and the **PRM** variable.

Example:

Modify the **'OP'** parameter so that it returns the original program name:

```
PRINT PGN
CKX
SET_PARAM 'OP'
PRINT PGN
/PXP/JEAN/CKX
! Configure command line (SL parameter) with 50 lines
SET_PARAM 'SL'=50
```

STOP

Allows ending the execution of a program. If the program is public (it was executed with the command **CALL**), it behaves like **EXIT**. If the language is in run mode (no console mode allowed), this command behaves like the **QUIT** command.

Example:

Finish the program's execution:

```
0110 STOP
```

SYSTEM_HELP

Used to activate the Windows Help subsystem. It is also used to activate different programs based on the program/file extension.

Example:

Open a PDF document (Acrobat Reader will automatically load and run):

```
SYSTEM_HELP "MANUAL.PDF"
```

TABLE

Used to define a translation table. It consists of a hexadecimal number and then pairs of hexadecimal characters. The first number will be used to condition the conversion, and the following will be of the type "character to convert" and "character converted".

Example:

Convert the letters a-j/A-J into the numbers 1-0:

```
0010 TABLE 0F303132333435363738394142434446
0020 INPUT (0,tbl=0010)A$
0030 PRINT A$
```

TRANSLATE

Allows changing all occurrences of one (or several) character(s) into another character(s) according to a conversion table.

Example:

Change all letters "A" to "@" in Text$:

```
TRANSLATE Text$,"@","A"
```

UNTIL

Used in conjunction with the **SELECT** command to define a loop that will be executed until the condition specified with the **UNTIL** command is met.

Refer to the **SELECT** command in the PxPlus Help documentation.

USER_LEX

Allows language definitions to be extended to include new symbols and directives. Typically used to simplify conversion from other languages to PxPlus.

Example:

Define the **PAD(** function as the **FILL(** function, and define SAMPLE as PRINT:

```
USER_LEX "FILL ("="PAD ("
USER_LEX "PRINT"="PRINT"
USER_LEX LOAD DEFINITION$
```

Important Note: It is strongly recommended to ***be extremely careful with this command***. It could render the language unusable. Consult your local distributor.

VIA

Used to assign values to variables whose name is stored in another variable. This type of functionality can be achieved using the **EXECUTE** command, but the **VIA** command operates much faster. The **VIN** and **VIS** functions can be used to obtain the value of a variable indirectly.

Example:

Ask for a variable, ask for its value, assign indirectly:

```
INPUT "Value: ",VALUE$
VIA NOM_VAR$ = VALUE$
PRINT NOM_VAR$
```

WAIT

Used to implement delays or waits in programs; for example, to display messages, delay events, etc. where the computer must display the data so that the user can read it. The waiting time can be tenths of seconds. The maximum length is 10 minutes (600 seconds).

Example:

Show a message for half a minute (30 seconds):

```
PRINT @(0.22),MESSAGE$
WAIT 30
```

WHILE

Used to implement a repetitive cycle. It must end with the command **WEND**. Unlike the **REPEAT** loop (and its counterpart **UNTIL**), in this command, the condition is done at the beginning so that it is not executed if the condition is not met.

Example:

Execute cycle while NAME$ is different from "CKX Network Business":

```
0100 WHILE NAME$<>"CKX Network Business"
0110 LET K$=KEY(CAN,END=END)
0120 READ (CHANNEL)
0130 PRINT NAME$
0140 WEND
```

WEND

Used to implement a repetitive cycle. It must begin with the command **WHILE**.

Refer to the **WHILE..WEND** command in the PxPlus Help documentation.

Using the Built-In Functions

A ***function*** is a language component that will modify or calculate some data and will return a value as a result. In PxPlus, most functions are three letters long, and their argument (on which they will be executed) is enclosed in parentheses.

Examples of functions are **LEN()**, **DTE()**, **NUM()** and **STR()**.

For a complete list, refer to the **System Functions** section in the PxPlus Help documentation.

@( ) - Location Function

Sets the position for placing the cursor to print or input a string. The following statement prints the date in the upper left hand corner of the screen with the time starting in column 75 of the top line:

```
PRINT @(0,0),"Date: ",DAY,@(75),TIM
```

This prompts for information on the left side, 5 lines from the top:

```
INPUT @(0,5),"Enter favorite sport:",@(30),SPORT$,@(0,6),"Thanks"
```

ABS( ) - Absolute Value

Calculates the ABSolute value (no sign, always positive or zero) of a given number.

```
0010 INPUT "Give me your first number ",X
0020 INPUT "Give me another one ",Y
0030 PRINT "The difference is ",ABS(X-Y)
-:RUN
Give me your first number 12.345
Give me another one 23.456
The difference is 11.111
? abs(-3.22)
 3.22
```

ARG( ) - Command Line Argument

Returns the ARGument passed at Command line (at operating system level) to execute PxPlus.

Given:

```
PXPLUS -SZ=20 -ARG TOM JONES
```

Then:

```
ARG(-1) yields the INI filename in use for the current session
ARG(0) yields PXPLUS (the command that launched PxPlus)
ARG(1) yields TOM
ARG(2) yields JONES
ARG(3) yields Error #41: Invalid integer encountered...
```

ASC( ) - Get Internal Character Value

Brings the ASCII code for a character.

```
?ASC("A")! yields 65
string$="a";?ASC(string$) ! yields 97
?ASC("abc") ! also yields 97 (the ASCII value for "a")
```

CHR() - ASCII Character of Value

Given the ASCII code generates the corresponding character:

```
LET X$=CHR(65) ! (Sets X$ to "A")
LET X$=CHR(33) ! (Sets X$ to "!")
```

CVS() - Convert String

Convert a string stripping some characters or converting to upper/lowercase.

```
a$="normal"
?cvs(a$,4)
NORMAL
```

DEC() - Get Binary of String

Two's complement binary equivalent of the string.

```
0020 LET A$="a"; PRINT DEC(A$),
0030 LET B$=$0040$; PRINT " | ",DEC(B$),
0040 LET C=DEC("A"); PRINT " | ",C,
0050 LET A=DEC($FE$); PRINT " | ",A,
0060 PRINT " | DONE"
-:RUN

 97 | 64 | 65 | -2 | DONE
```

DIM() - Generate String/Get Array Size

Use the **DIM()** function to generate or initialize a string.

```
0100 PRINT ":",DIM(11,"*Cat"),":"
->RUN

*Cat*Cat*Ca:
```

DTE() - Convert Date

The **DTE()** function converts a date (and time) from Julian form to a formatted string.

```
PRINT DTE(0:"%Dl %Ml %D/%Y %hz:%mz %p")
Monday July 24/1995 10:27 pm
DAY_FORMAT "MM/DD/AA"
PRINT DTE("01/01/A0":"%Y %Ml %D")
2000 January 1
PRINT DTE("01/01/A0":"%Dz/%Mz/%Y")
01/01/2000
```

ENV() - Environment Values

Use the **ENV()** function to obtain the value of an environment (Operating System) variable.

```
0010 T_TYP$=ENV("TERM")
0020 IF T_TYP$="" THEN PRINT "No terminal defined";
0020: STOP
0030 PRINT "You are using a "+T_TYP$+" terminal"
```

EVN() - Evaluate Numeric Expression

The **EVN()** function evaluates and returns the numeric value of a numeric variable or computed expression.

```
MONT=10000,FRST=2500,INT=10
INPUT "Which field (MONT,FRST,INT):",var$
PRINT evn(var$)
```

EVS() - Evaluate String Expression

The **EVS()** function evaluates and returns the value (evaluated contents) of a string variable.

```
NAME$="Janet"
ADDR$="Bolivar Avenue"
CITY$="Valencia"
INPUT "Which field (NAME,ADDR,CITY):",X$
PRINT evs(var$+"$")
```

FIB() - Return File Information Block

The **FIB( )** function returns a character string containing a file information block description for an existing open file.

```
0010 INPUT "Enter filename:",F$
0020 OPEN (1)F$
0030 IF MID(FIB(1),10,1) <> $02$ THEN PRINT "Not a Keyed file"
```

FIN() - Return File Information

The **FIN( )** function returns a character string containing details about an existing open file.

```
channel=unt
open(channel)arch$
print arch$," has ",fin(channel,"NUMREC")," records"
```

FPT() - Return Fractional Part

The **FPT()** function returns the decimal portion of the given numeric value.

```
A=FPT(4.561) yields A = .561
```

HSH() - HaSH Function

The **HSH()** function can return a hash value or an encrypted/decrypted value for a given string.

```
PRINT hta(hsh("Some text for testing a test"))
AC15
```

IND() - Return Next Record Index

The **IND()** function returns the record index of the next record.

```
0010 OPEN (13) "CUSTNO"
0020 LET I=IND(13,END=1000)
0030 READ (13,IND=I) R$
0040 PRINT "Rec#: ",I," Data: ", R$
0050 GOTO 0020
1000 PRINT "End-of-file"
1010 END
```

INT() - Return Integer Portion

The **INT()** function returns the integer portion (strips decimals) of the given numeric value.

```
A=INT(3.23) ! yields A=3
A=INT(-5.6) ! yields A=-5
A=INT(.9999) ! yields A=0
```

IOL() - Get IOList Specification

The **IOL()** function returns the variables list for either a file or a composite string variable.

```
0100 DIM CUST$:IOL=0110
0110 IOLIST NAME$,ADR1$,ADR2$,SMAN$
0120 PRINT LST(IOL(CUST$))
-:RUN
IOLIST NAME$,ADR1$,ADR2$,SMAN$
```

JUL() - Return Julian Date

The **JUL()** function is used to convert a date from year, month, day to a Julian date. The Julian date is an integer: the number of days since the system base-date.

```
0010 INPUT "Enter Date (MM/DD/YY):",X$:"00/00/00"
0020 LET M=NUM(X$(1,2))
0030 LET D=NUM(X$(3,2))
0040 LET Y=NUM(X$(5,2))
0050 LET N=JUL(Y,M,D,ERR=0100)
0060 PRINT "That is ",N-JUL(0,0,0)," days from now"
0070 STOP
0100 PRINT "Invalid date"; GOTO 0010
```

KEC() - Return Key of Current Record
KEF() - Return First Key of File
KEL() - Return Last Key of File
KEN() - Return Key After Next
KEP() - Return Prior Record's Key

These functions return the respective keys for the records in the file specified.

LCS() - Return Lowercase String

Convert a string to lowercase.

```
0010 INPUT "Enter name: ",NAME$
0020 LET NAME$(2)=LCS(NAME$(2))
0030 LET NAME$(1,1)=UCS(NAME$(1,1))
0040 PRINT "Name is",@(10),": ",NAME$
-:RUN
Enter name: SMITH
Name is : Smith
```

LEN() - Return String Length

The **LEN()** function returns an integer with the length (characters) of a string.

```
A=LEN("HELLO") ! yields 5
A=LEN("") ! yields 0
A=LEN("A"+"BC") ! yields 3
```

MID() - Return Substring

Use the **MID()** function to extract a portion of a string.

```
a$="Jurassic"
PRINT mid(a$,3,1)
r
```

NEW() - Create New Object

The **NEW()** function is used in Object Oriented Programming to create a new object based on a specified class name.

```
Cst = NEW ("Customer")
```

NUM() - Convert String to Value

The **NUM()** function returns the numeric value of a numeric expression in a string.

```
A=NUM("1.34") ! Yields 1.34
A=NUM("-1,005.") ! Yields -1005
A=NUM("A",ERR=50) ! On error, transfers to 0050 and sets ERR=26
```

PAD() - Pad/Truncate String

The **PAD()** function converts a given character string to the length specified.

```
->PRINT pad("",10,"*")
**********
name$="Jean"
name$=pad(name$,20,0," ")
? name$
    Jean
```

POS() - Scan String for a POSition

The **POS()** function scans the string$ to determine where a portion of it will satisfy the relationship with the pattern string.

```
Given A$="The quick brown fox":
POS("q"=A$) ! yields 5
POS("z"=A$) ! yields 0
```

STP() - Strip Leading/Trailing Characters

The **STP()** function returns a character string generated by stripping specified instances of a character or a mnemonic from a string expression.

```
STP(PTR$,3,$1B$) ! Strips out all escape characters
STP("Hello there",3,"e") ! Strips out every lowercase "e"
```

STR() - Convert Numeric to String

The **STR()** function converts and validates numbers to strings.

```
A$=STR(5*6) ! (yields A$="30")
A$=STR(5*6:"000") ! (yields A$="030")
A$=STR("1234567":"000-0000") ! (yields A$="123-4567")
A$=STR("AB":"00","**") ! (yields A$="**")
```

SYS() - Invoke Operating System Command

The **SYS()** function passes a given string to the operating system command processor for execution.

```
a=sys("calc") ! Executes the calculator
```

TCB() - Return Task Information

The **TCB()** function returns information for several different purposes.

```
PRINT TCB(23) ! Maximum user count
PRINT TCB("OS_GetUserUID","johndoe") ! Gets Operating System User ID
```

UCS() - Return Uppercase String

The **UCS()** function replaces all lowercase characters with uppercase.

```
0010 INPUT "Enter name: ",NAME$
0020 LET NAME$(2)=LCS(NAME$(2))
0030 LET NAME$(1,1)=UCS(NAME$(1,1))
0040 PRINT "Name is: ",NAME$
-:RUN
Enter name: rOBERT
Name is: Robert
```

VIN()/VIS() - Obtain Value of Numeric Variable

Returns the numeric/string value (contents) of a variable where the name is stored in another variable.

```
10 name$="Janet",city$="Valencia",clr$="Blue"
20 INPUT "Variable (name$, city$, clr$): ",var$
30 PRINT vis(var$)
->run
Variable (name$, city$, clr$): name$
Janet
->run
Variable (name$, city$, clr$): clr$
Blue
```

System Variables

A ***variable***, as its name indicates, is a changing value structure, but PxPlus offers, for the programmer's convenience, a series of variables that help us determine the state of some components.

To differentiate them from the variables we use in our programs, these are called ***System Variables***. Many of the system variables also have three-letter names.

Examples of system variables are **UNT**, **HFN**, **SEP** and **CTL**.

Some of the system variables in PxPlus are listed below. For a complete list, refer to the **System Variables** section in the PxPlus Help documentation.

BKG - Background Process Status

The **BKG** system variable contains the following numeric status codes:

0 = Program is connected to a terminal
1 = Program is not connected to a terminal (background process)

```
? BKG
  0
```

CTL - Control Signal Code

The **CTL** system variable contains a numeric code that represents a signal of user input from the keyboard or mouse.

```
input "Name: ",name$
if CTL=4 then GOTO USER_PRESS_F4
```

DAY - Return Current System Date

The **DAY** system variable contains the current system date (**Example:** 11/15/00) in a format based on the date style set in the **DAY_FORMAT** directive (MM/DD/YY by default).

```
0100 PRINT DAY
0110 DAY_FORMAT "DD/MM/YYYY"
0120 PRINT DAY
->run
11/15/99
15/11/2000
```

ERR - Last System-Detected Error Value

The **ERR** system variable contains a numeric value (integer) that indicates the last system-detected error. It is reset by the **BEGIN**, **CLEAR**, **END**, **RESET**, **STOP** and **START** directives.

```
0100 OPEN (5)"TEST"
Error #14: Invalid I/O request for file state
?ERR
  14
```

ERS - Line Number of Last Error

The **ERS** system variable contains the line number that generated the last error detected in the program.

```
0030?LET X$="" PRINT "Reset"
Error #20: Syntax error
->run
?ERS
  30
```

GFN - Highest Available Global Channel

The **GFN** system variable contains a numeric value (integer) representing the highest available (i.e. not open) global file channel.

```
set_param 'xf'
?gfn
  65000
SET_PARAM -'XF'
?GFN
  127
```

HFN - Highest Available Local Channel

The **HFN** system variable contains a numeric value (integer) representing the highest available local channel/file number.

```
?HFN
  63
SET_PARAM -'XF'
?HFN
  32767
```

HWD - Starting/Home Directory

The **HWD** system variable contains the name of the directory that was current at start up (when PxPlus was initialized). This is considered the home directory.

```
print hwd
D:\PxPlus
```

LFO - Last File Number Opened

The **LFO** system variable indicates the channel/file number of the last file opened.

```
0100 OPEN (HFN) "TESTFILE"
0110 CHANNEL=LFO
```

LWD - Current Working Directory

The **LWD** system variable contains the full pathname of the current working directory.

```
? lwd
D:\PxPlus
```

NAR - Number of Arguments in PxPlus Start_up

The **NAR** system variable contains a numeric value (integer) representing the number of arguments in the PxPlus command that launches PxPlus (**Example:** in using a batch file or from a command statement).

NID - Network Identifier

The **NID** system variable contains the network node or identifier.

```
IF NID="CKPARIA" then GOSUB authorized_node
```

OBJ - Object Handle for the Object

If running as an object, the **OBJ** system variable contains the object handle for the object.

```
IF OBJ=0 then MSGBOX "There isn't a class/object active","Notice"
```

PFX - Current Pathname Prefix for PREFIX (0)

The **PFX** system variable indicates the current settings of the **PREFIX** directive (for PREFIX entry 0).

```
PRINT "The current prefix is: ",pfx
```

PGN - Name of Currently Loaded Program

The **PGN** system variable contains the name of the currently loaded program, complete with its full operating system pathname.

```
?pgn

load"date.pvc
?pgn
D:\pxp\date.pvc
```

RND - RaNDom Number Generator

The **RND** system variable contains a different random number each time you use it to return a value. The value will be in the range from 0 to 1.

```
LET SECRET=INT(RND*100)
```

TIM - System Time

The **TIM** system variable returns the current system time in hours past midnight.

```
print tim
settime 1.10
print "Current time ",tim
->run
19.341613
Current time 1.100502
```

UID - Current User ID

The **UID** system variable contains the current User ID. On systems without User registration, it returns the value of the environment variable USER or the Network User ID.

```
IF uid$<>"admin" THEN GOTO ONLY_ADMINS
```

UNT - Lowest Available Channel

The **UNT** system variable contains the lowest available channel/file number.

```
CHAN_NUM=unt
OPEN (CHAN_NUM)"FILENAME"
PRINT CHAN_NUM 1
```

Introduction to Mnemonics

A ***mnemonic*** is a modifier of an input or output command on the screen or keyboard. Mnemonics basically serve to alter the way some data will be presented or transmitted.

Example:

We can use a mnemonic to tell the printer to use a certain type of font (bold, normal, etc.):

```
PRINT (can_prt)'EP',"This title will be double width",'EF'
PRINT 'RED',"These letters on the screen will be red"
```

Some mnemonics can also send a command to a device, such as a display or printer:

```
Print 'CS' ! Clear Screen, this mnemonic will clear the screen
```

The mnemonics are defined out of necessity to tell the language how to modify some aspect of an output.

Example:

```
PRINT (channel) "The employee is called: ",emp$
```

In this case, the output does not have anything that highlights the value of the emp$ variable above the rest of the text. We could tell it to put that data in bold.

Example:

```
PRINT (channel) "The employee's name is: ",emp$
PRINT (channel) 'SB',"The employee's name is: "','SF',emp$
```

The result would be like this:

The employee's name is: Miguel
The employee's name is: **Miguel**

We can indicate other "attributes" or aspects, such as font type and size, orientation, etc. **Example:** When we are making a list on the printer or through the viewer or when we create a PDF file, it is normal to use mnemonics.

Mnemonics can be used on screen.

Example:

```
PRINT 'BLUE',"Data in blue ",'GREEN',"This will be in green"
```

```
Data in blue This will be in green
```

We can get text in various colors. Note that the mnemonic 'BLUE' sets the letters in Blue, while '_BLUE' will set the background in Blue. So, '_GREEN','YELLOW' will set a Green background to Yellow letters.

Some mnemonics only have a purpose on certain devices, such as 'CS', which is used to clear the screen.

Note: The word **mnemonic** comes from "memory"/"reminder" and refers to the fact that most of the names of mnemonics are derived from their use or function:

```
'BB'=Begin Blinking
'EB'=End Blinking
'CS'=Clear Screen
'FF'=Form Feed (advance one form/page)
```

As we previously mentioned, one case where mnemonics are used a lot is for print or output to PDF:

```
PRINT (pdf_chan)'BB',trade$,'EB',
PRINT (pdf_chan)'SB',"Process date: ",'SF',proc_date$
PRINT (pdf_chan),'font'("Arial,-20"),Title$
```

There are several mnemonics that have the same function, in some cases, to maintain compatibility with previous versions of the language, in others, to give it greater flexibility.

Example:

```
PRINT 'RED',"Text in red"
PRINT 'F9',"Text in red"
```

Mnemonics are very widely used in PxPlus. They were originally used for a few things and have been extended to offer very complete functionality. Among the outstanding functions of mnemonics, we have:

```
PRINT 'PICTURE'(cx,cy,size,size,image$) ! Show an image
PRINT 'WINDOW'(5,5,100,40,"Title",opt="S") ! Define a window
INPUT 'UC',name$ ! Convert input to uppercase
PRINT (c)'PEN'(1,1,2) ! Green color, 1 pixel thickness, 'normal' mode
PRINT (x)'RECTANGLE'(200,200,600,600) ! Defines a rectangle
```

In short, the number and uses of mnemonics is very broad. You should read and research to learn more about them. You will see that some require additional information. Others can only be used on certain devices, and others will change their function or behavior depending on the device on which they are used.

Note: The mnemonics are not case sensitive. '**EB**', '**Eb**' and '**eb**' are all the same mnemonic.

Using User Parameters in PxPlus

A ***parameter*** is a value that will modify the regular behavior of PxPlus; for example, do not generate errors in some condition or change the way variables and commands are presented in the lists. A typical case of a parameter is division by zero. Mathematical division by zero is not allowed, and in the case of PxPlus, this generates an Error 40 (Divide check or numeric overflow). But, it is possible to turn off this error so that it would simply be ignored.

To change a parameter, you have the **SET_PARAM** command:

```
SET_PARAM 'D0'=0 ! Parameter off
PRINT 2/0 ! Generates an ERR=40
SET_PARAM 'D0'=1 ! Parameter on
PRINT 2/0 ! Returns 0 as a result of division
```

As the language has evolved over more than 40 years, many parameters have become obsolete. There are also parameters that only work in certain circumstances, and others simply alter some internal behavior of the language or serve only to condition other parameters.

A classic example of parameters is the **'AC'** parameter that tells the system to try to convert from literal to numeric or vice versa.

```
10 MSGBOX "client "+client+" owes us $"+amount
```

This instruction, by default, will generate an Error 26 (Type mismatch, type error), but if the **'AC'** parameter is activated before entering it, the same language will add the functions:

Example:

```
->SET_PARAM 'AC'=0
->PASTE
->10 MSGBOX "The client "+client+" owes us $"+amount
Error #26: Variable type invalid ...+" owes us...
-:LIST 10
0010?MSGBOX "The client "+client+" owes us $"+amount
-:DELETE
->SET_PARAM 'AC'=1
->PASTE
->10 MSGBOX "The client "+client+" owes us $"+amount
-:/
0010 MSGBOX "The client "+STR(client)+" owes us $"+STR(amount)
-:
```

Note: The **LIST** command can be substituted with the **/** (*forward slash*). The **PRINT** command can be substituted with the **?** (*question mark*), and the **EDIT** command with the **`** (*back apostrophe*).

Example: Another typical example is the case of mask overflow with a numeric mask:

```
->PRINT 2000:"000"
Error #43: Format mask invalid
->SET_PARAM 'FI'=1
->PRINT 2000:"000"
2000
->|
```

By default, PxPlus reports that three zeros (000) are insufficient to contain a four digit quantity, so it generates an Error 43 (Mask overflow); however, this can be turned Off.

Typically, parameters are appropriated (enabled or changed) in an initial boot program and are not touched again, although it is possible to change them dynamically. It will be your experience and your needs that will help you make the decision about what to change and when.

This table lists some of the main system parameters in PxPlus.

Parameter	Description
'!S'	Suppress Error Flags on Serial Save
'!T'	'DP' or Decimal for Numerics
'3D'	3D in Windows
'AD'	Auto-DIM Array
'AH'	Alternative 'WINDOW'/'BOX' Heading
'B0'	Base Zero for Level/Window
'BF'	Common File Buffers
'BT'	Binary Test: 1st Read
'BX'	BBx® Emulation
'BY'	Base Year
'CD'	Check Current Directory
'CF'	Bypass Console Flush
'CS'	Coloured Syntax
'CT'	Character Time-out
'CU'	Currency Symbol
'D0'	Divide by Zero
'DC'	Destructive Cursor
'DD'	Convert DOS Directory Delimiter
'DP'	Decimal Point Symbol
'DT'	Device Time Out
'DW'	Delay Time After 'WI'

Parameter	Description
'EG'	End Generation of Error #29
'EO'	Embedded 'EO' Mnemonics
'ES'	Display OS Errors in Command Mode
'EX'	Apply Execute Level Selector
'F4'	Return CTL=4 for Exit
'FB'	Dedicated File Buffers
'FC'	Force File Commit
'FF'	File Format
'FI'	Ignore Format Mask Error
'FL'	Filename in Lowercase
'FN'	Filename As Is: No Case Conversion
'FP'	Floating Point
'FS'	Default Field Separator
'FT'	Trapping the F10 Key
'FU'	Filename in Uppercase
'FX'	Force EXTRACT
'IC'	Ignore Case Sensitivity for Scan
'IS'	CTL for Input Ending on SIZ=
'KR'	Keyed File I/O Emulates BBx®
'LB'	Colour for Line Number in Break Points
'LC'	List Variables in Lowercase
'LD'	List Directives in Lowercase
'LE'	SAVE/LIST Indent Statements
'LS'	Colour for Line with Syntax Error
'LU'	Lock Unnecessary: Serial Files
'MF'	Multi-Line Size Factor
'MP'	Returns Positive Modulus Value
'NE'	Subprogram Error Report
'NI'	Ignore Blanks in Numeric Fields
'NK'	Null Key Stripping
'NL'	Suppress LET Directive in Listings
'NR'	No Intermediate Rounding on Division
'OC'	Commit Prior to OPEN Directive

Parameter	Description
'OF'	Maximum Size Before Output Flush
'OL'	Maximum Buffers for OPEN LOAD Description
'OM'	Old Style Mask
'OP'	Return Original Program Name
'OR'	Full OS Path for Rename
'OW'	Owner Application Code
'PC'	Program Caching
'PO'	Path Original
'PU'	Uppercase Prefix
'Q^'	Highest Task Priority
'Q_'	Lowest Task Priority
'QF'	Task Priority Factor
'QS'	START, Not Initialized
'QT'	No Prompt in Command Mode
'RN'	Rounding Control
'RP'	Raw Print for *WINDEV*
'RR'	Reset on RUN
'RS'	Round STR( ), Formatting and Functions
'SB'	Self-Block Extracts
'SC'	Show Cursor
'SD'	Sub-directory Slash
'SF'	Short Form Variables
'SK'	Shrink Keyed Files
'SZ'	Maximum Memory Size for Session
'TC'	Tip Colour
'TH'	Thousands Separator
'TL'	LIKE Emulates Thoroughbred®
'TN'	Strip Trailing Nulls
'TT'	Timed Trace
'TU'	WindX Turbo Mode
'VP'	Variable Pitch
'VR'	Verify Read
'VW'	Verify Write

Parameter	Description
'WB'	WindX BREAK Recognition
'WD'	Defer File Writes
'WF'	Force Windows Screen Update
'WH'	Delay Retry: Locking File Headers
'WK'	Keep Window
'WL'	Use Write Locks
'WT'	Number of Retries
'XF'	Extended File Channels

BBx® is a registered trademark of BASIS International Ltd.
Thoroughbred® is a registered trademark of Thoroughbred Software International, Inc.

You can modify the parameters manually or by using the configuration utility "***UCP**". In any case, you must keep in mind that the changes made to the parameters are not permanent. When you exit the language and enter it again, the default values of each parameter are restored.

Refer to **System Parameters** in the PxPlus Help documentation.

13. NOMADS: Other Functions and Tools

Themes and Visual Classes

Themes and Visual Classes provide the easiest way to define and maintain a unified look to panel controls within your application.

Each type of control has its own set of unique properties for setting colors, fonts and other attributes. You can create Themes and Visual Classes to define display settings, which can be applied to a group of controls or to individual controls.

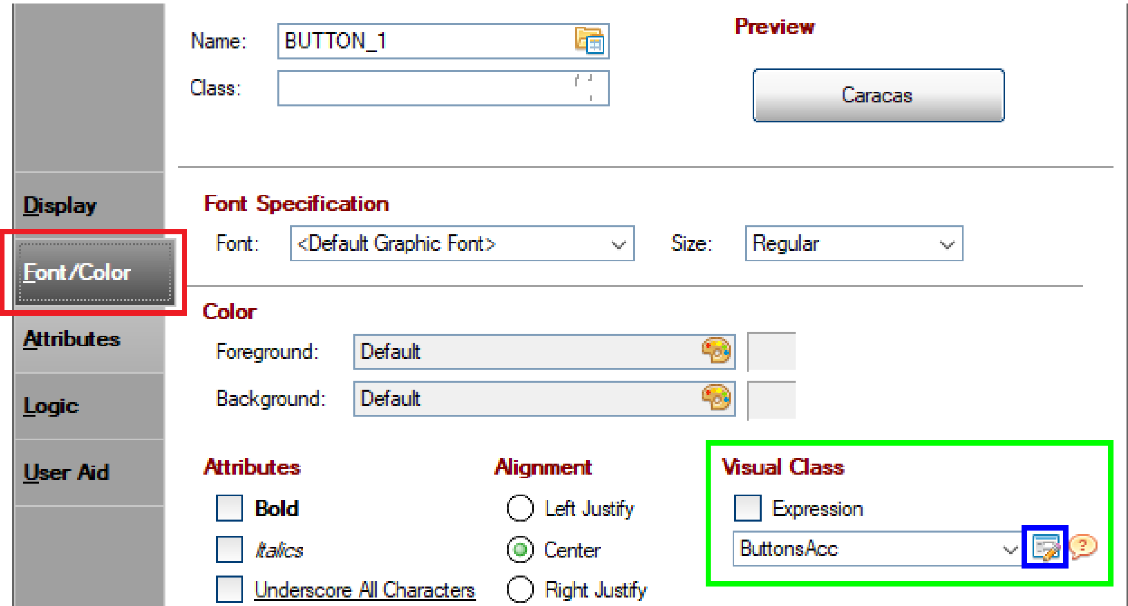
A ***Theme*** is a tool that allows you define and apply the same appearance to a ***group*** of controls that are of the same type. Themes can be applied to controls at different levels - application wide, to specific libraries or to specific panels. (**Example:** You can create a Theme for all "Button" type controls to make the button flat, set the Background property to "Light Blue", set the Border Color property to "Dark Blue", and set the Active Text Color property to "Light Red". You could apply this Theme system wide so that all "Button" type controls within your application have this same appearance.)

Refer to **Themes** in the PxPlus Help documentation.

A ***Visual Class*** is a tool that allows you to define and apply the same appearance to ***individual*** controls on selected panels. Visual Classes can be applied to controls at different levels - individually (one control at a time), to a panel (all controls on the same panel) or to the entire NOMADS library. A Visual Class overrides a Theme and is useful in cases where unique settings are required.

Refer to **Visual Classes** in the PxPlus Help documentation.

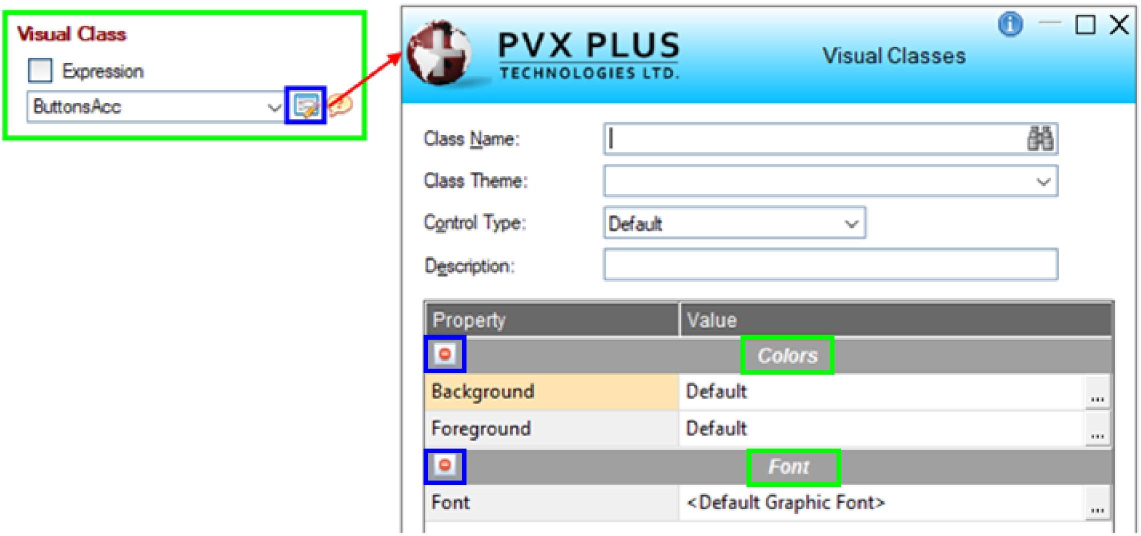
The easiest way to define a new Visual Class is from the Properties window of any control (**Example:** for a BUTTON control) by selecting the **Font/Color** tab and clicking the icon with the pencil (marked in blue below).



This opens the **Visual Classes Maintenance** utility. This utility can also be opened from the PxPlus IDE main menu by opening the **Graphical Application Builder (NOMADS)** category, expanding the **Setup** category and selecting **Visual Classes**.

To select a previously defined Visual Class, click the drop-down arrow.

The definition window for a Visual Class has common elements, such as the name, the class theme, the description and the type of control it affects. Next comes a list of properties and attributes that you can change to create your Visual Class. **Example:** You can create a Button with a Gray Background color and Blue Calibri font.



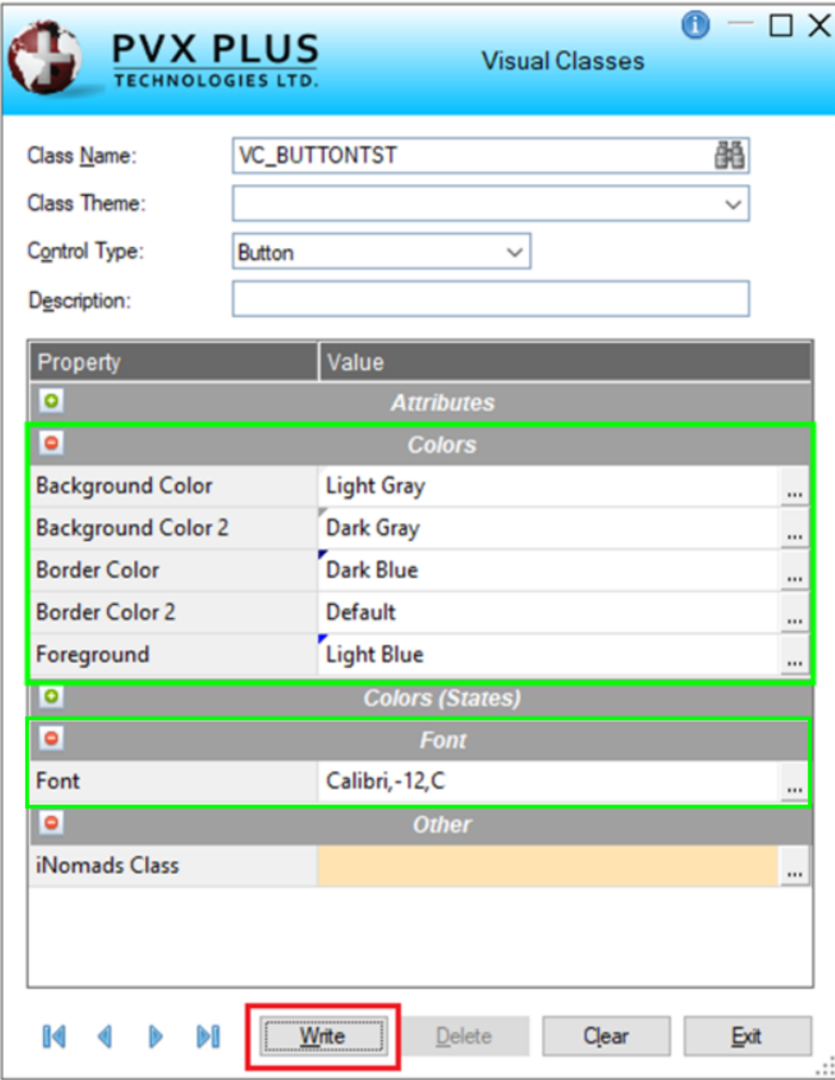
The list of properties will vary, depending on the type of control selected from the **Control Type** drop box.

The properties are grouped into categories, such as **Attributes**, **Colors**, **Colors (States)**, **Font**, etc., and are sorted alphabetically within each group. These categories can be expanded or collapsed by clicking the **+** (*plus*) or **-** (*minus*) button to the left of the category name (shown above). Only the categories that apply to the selected **Control Type** are displayed.

Exercise: Creating a Visual Class

We will use the **Visual Classes Maintenance** utility to create a Visual Class for a Button control type.

The name of the Visual Class will be **VC_BUTTONTST**, but you can call it anything. Define the **Colors** and **Font** properties (as shown below), and then save the Visual Class using the [**Write**] button.

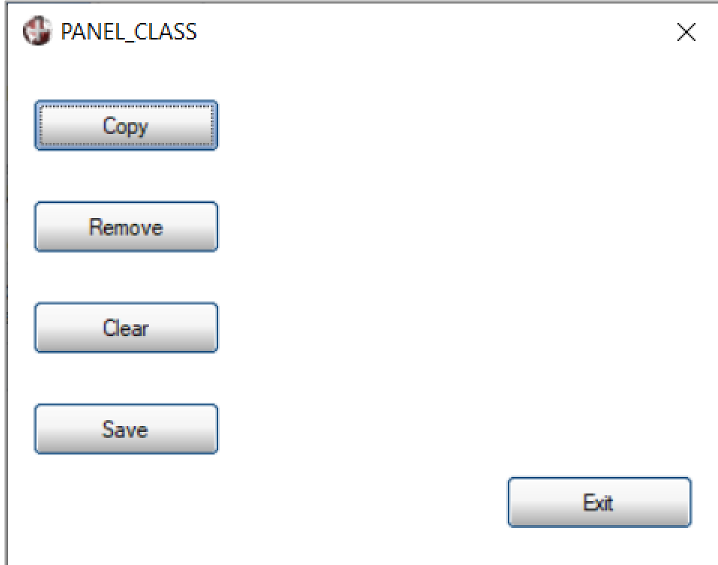


Once the Visual Class is defined, we can apply it to the desired controls.

Refer to **Visual Classes Maintenance Utility** in the PxPlus Help documentation.

Exercise: Applying a Visual Class

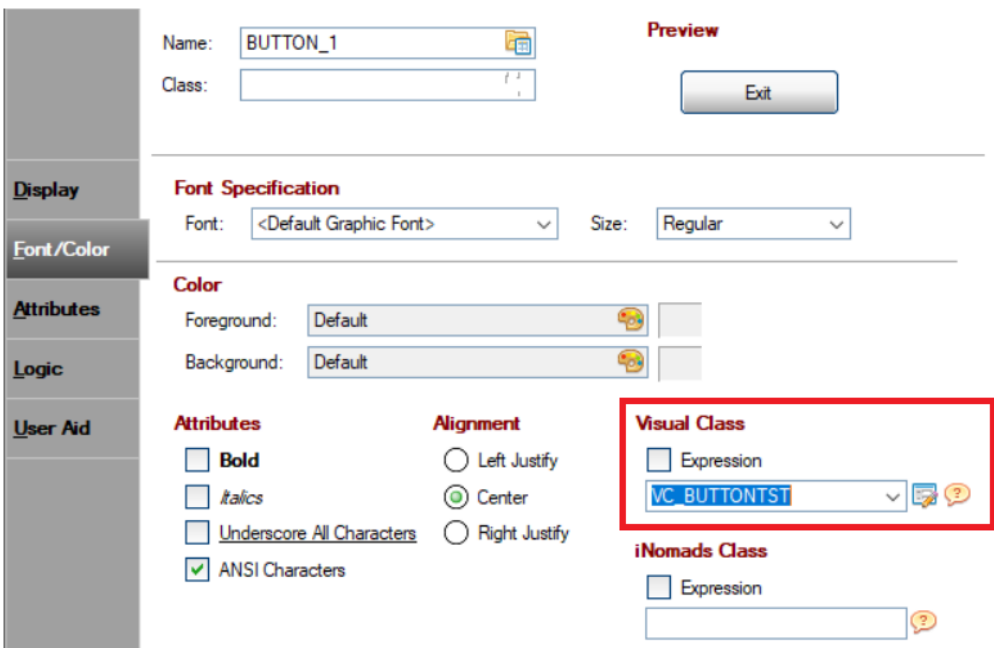
Create and save a new panel called **PANEL_CLASS** with the five buttons shown below (without any associated logic):



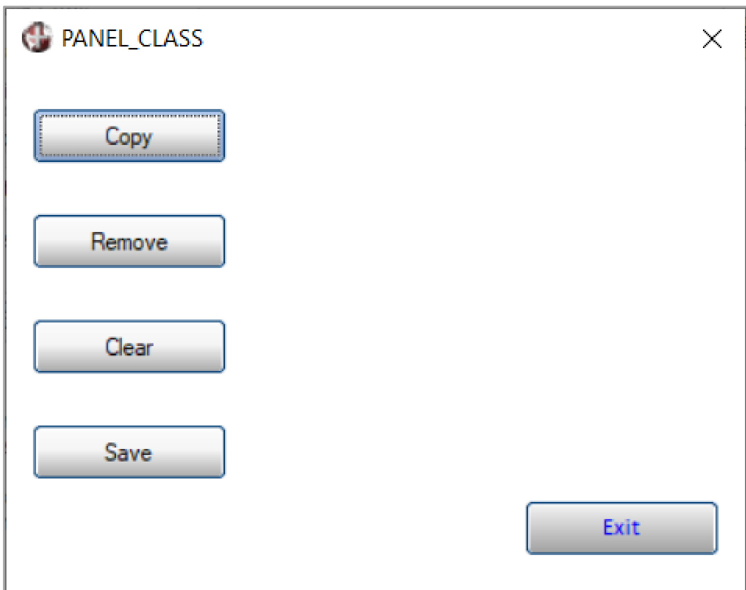
We can apply the Visual Class to a specific control or from the panel level.

To apply the Visual Class to a specific control, we open the **Button Properties** window, select the **Font/Color** tab and change it there as if it were another attribute of the control.

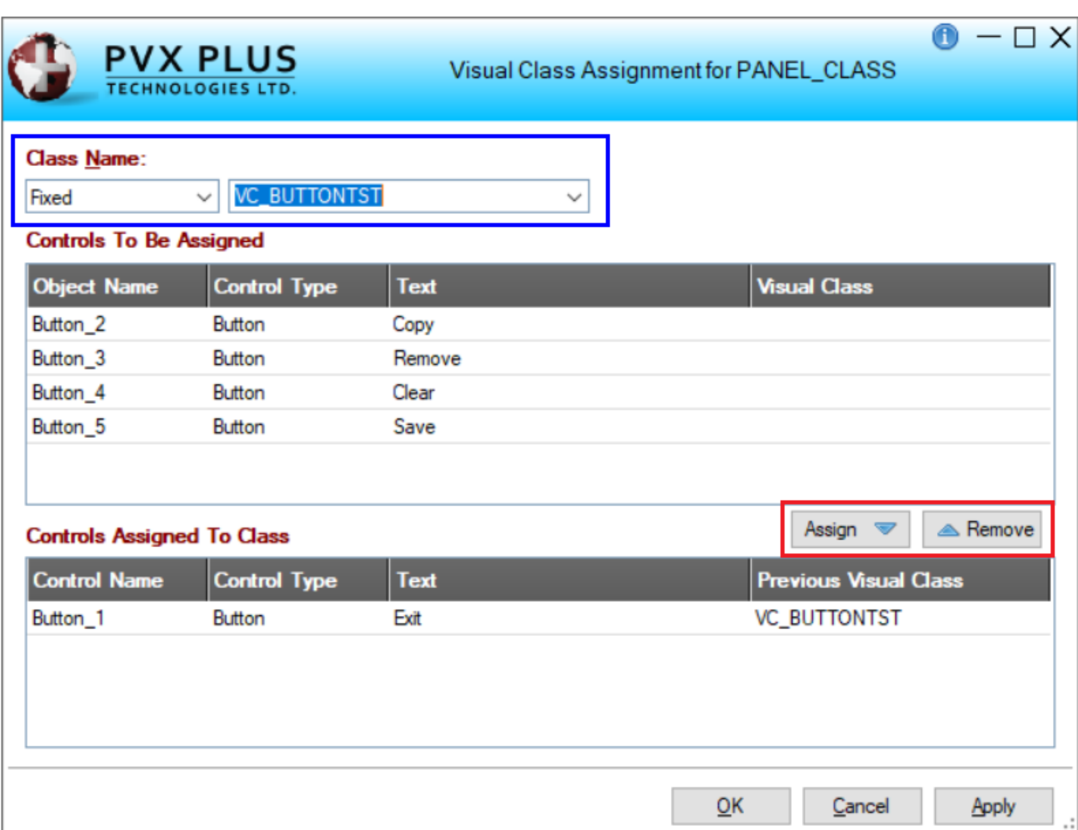
We will apply our Visual Class, **VC_BUTTONTST**, to our **Exit** button.



Save and test the panel. It should look similar to the one shown below:



To apply the Visual Class at the panel level (that is, apply it to all the controls of a panel or a group of them), in the top menu bar of the NOMADS panel designer, select the [**Utilities**] -> [**Visual Classes Assignment**] option. This opens the **Visual Class Assignment** window where we select the name of the Visual Class to be applied.

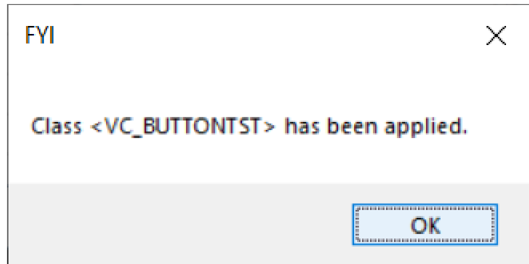


We also see the different panel controls (in this case, Buttons), which includes those that have an assigned Visual Class and those that do not. The [**Assign**] and [**Remove**] buttons are used to remove or assign the Visual Class to a control.

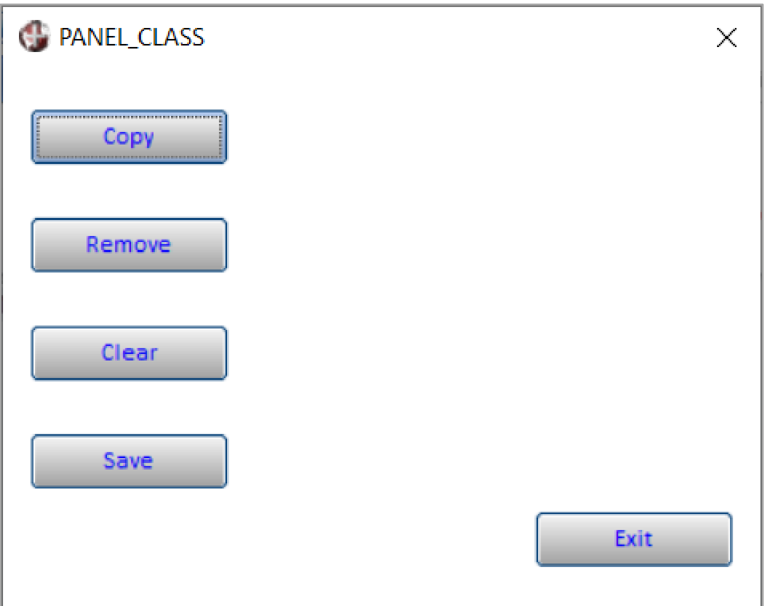
At this moment, our panel only has one Button control with the Visual Class, VC_BUTTONTST. The other Buttons do not have it. This is also reflected in the **Visual Class Assignment** window.

We will assign the same Visual Class to all of the other Button controls. We can select several controls at the same time by using [**Shift Click**]. Then, click the [**Assign**] button.

Once the Visual Class has been assigned, click the [**Apply**] button. A message displays to indicate that the Visual Class has been applied.



Save and test the panel to see the difference:



As you can see, it is a very simple operation to perform and easily reversible in case you make a mistake.

Refer to **Visual Class Assignment** in the PxPlus Help documentation.

Note: One of the advantages of Visual Classes is that, if you modify the Visual Class AFTER IT IS APPLIED to the controls, they will be updated with the new visual scheme.

Important Note: If you want, you can also use the **Library Bulk Edit and Search Utility** to add or

remove Visual Classes. It can be invoked from the graphical designer's **Library Object Selection** window by selecting [**Utilities**] **-** > [**Library Bulk Edit**] from the top menu bar. You can also make changes to the Visual Classes. Remember to make a copy of your work before using this utility.

Refer to **Library Bulk Edit and Search Utility** in the PxPlus Help documentation.

Data Classes

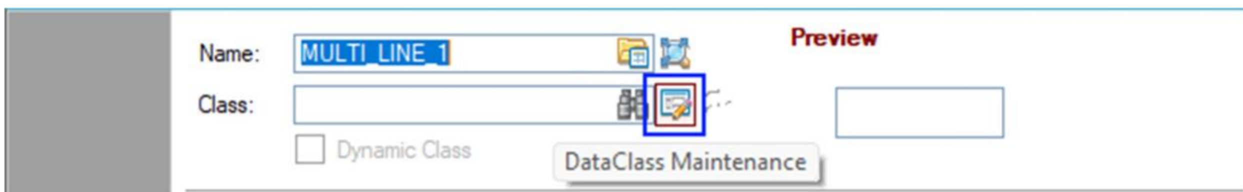
A ***data class*** is data that has special characteristics, such as dates, a group of fixed values (provinces of a country), etc. A data class is normally assigned or applied to a data entry control, which is a control that allows information to be entered, either by entering data through the keyboard or by selecting one of several options. Controls that use data classes are **Multi-Lines**, **Drop Boxes**, **List Boxes**, **Radio Buttons** and **Check Boxes**.

Once a data class is defined, there are several ways to assign it to a control, as we will see later.

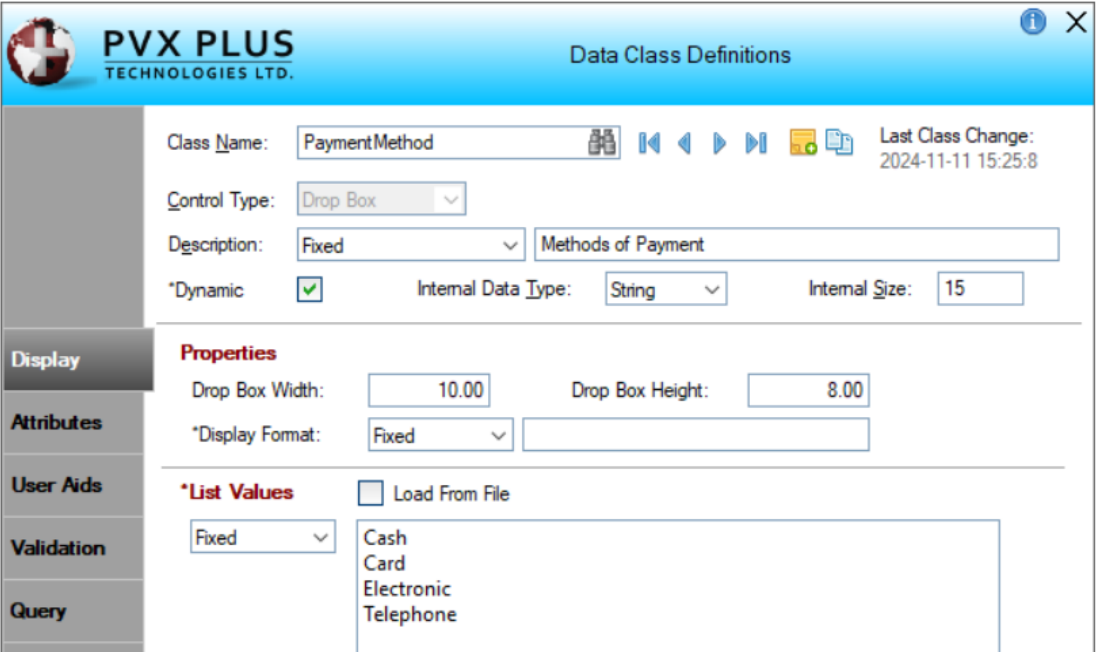
Data classes can be static or dynamic. We will see the differences later.

Start by defining a data class. **Example:** We will consider the different payment methods: Cash, Card, Electronic or Telephone. We will assign this class of data to a drop-down box type control (**DROP BOX**).

A data class can be created or defined from several places. From the PxPlus IDE main menu, open the **Data Management** category and select **Data Class Definitions**. From the NOMADS Session Manager (enter **nom** from the PxPlus Command line), select [**Dictionary**] -> [**Classes**] from the top menu bar. You can also define a data class from the **Data Dictionary Maintenance** utility by selecting [**Utilities**] -> [**Data Class Definitions**] from the top menu bar. Lastly, a data class can be defined from any control by selecting the icon with the pencil located to the right of the [**Class**] option.



The **Data Class Definitions** window looks like this:

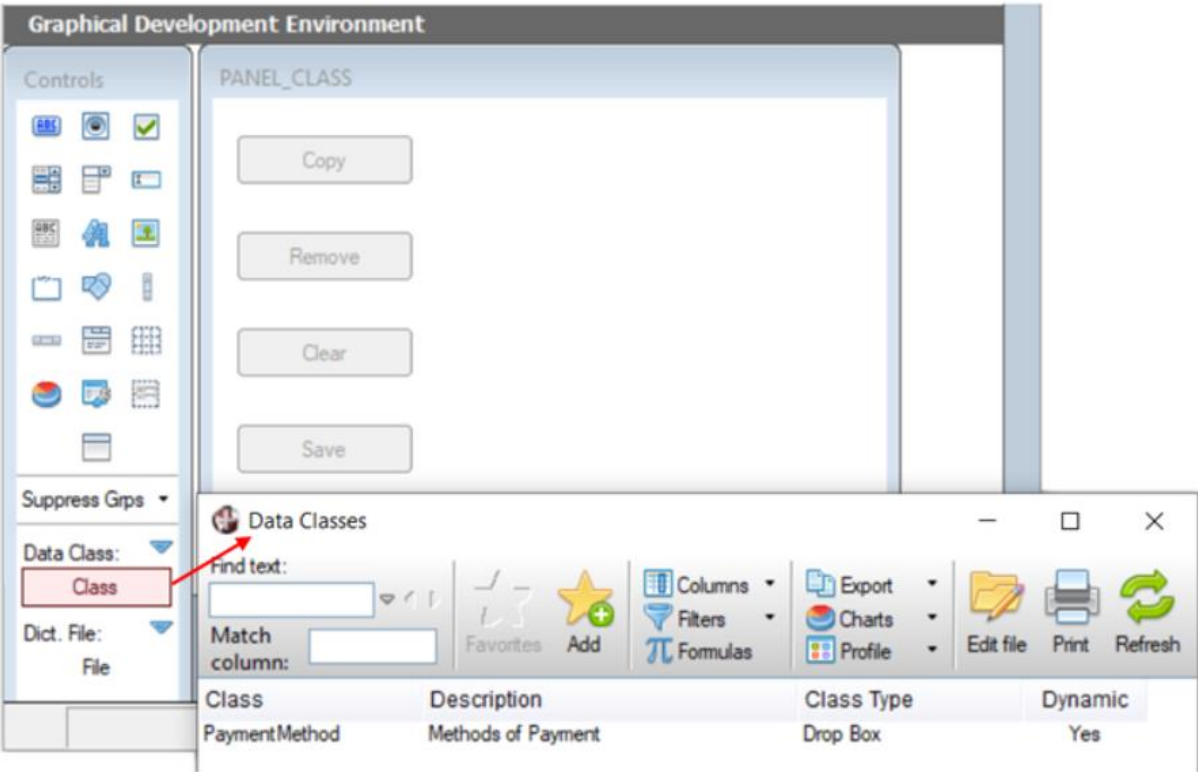


Exercise: Creating a Data Class

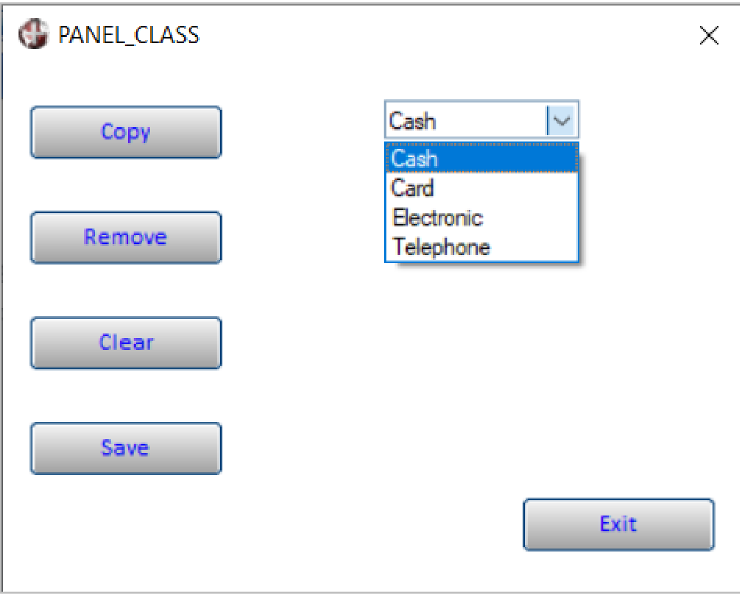
We will create a new data class called **PaymentMethod**. For [**Control Type**], we will choose **Drop_Box**. The [**Internal Data Type**] will be **String** (literal or text), and the [**Internal Size**] will be 15. The control will be 10 wide and 8 high. The [**List Values**] will be Fixed and will be Cash, Card, Electronic and Telephone. We can also assign other attributes, including a Visual Class. For now, save by clicking the [**Write**] button.

Once the data class is defined, open any panel in the NOMADS panel designer. We will open the panel created earlier called **PANEL_CLASS**, but you can select the one you want.

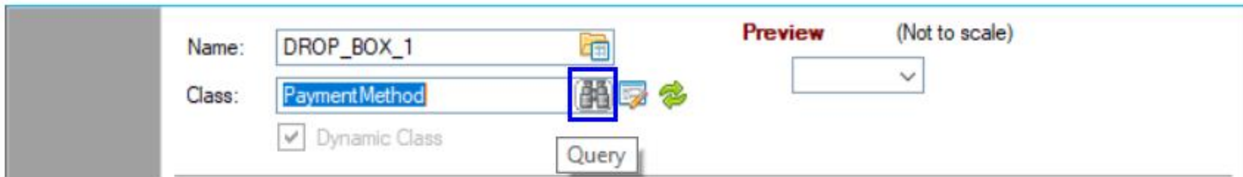
Once we have the panel open in the designer, we have several ways to use a data class. The first way is to select the [**Class**] option in the **Controls** box on the left, which opens the **Data Classes** query window similar to the one shown below:



Select the data class (PaymentMethod) and then draw a control in the part of the panel where you prefer. When the control is completed, it will automatically assume all the values of the data class. Note that you must enter the control name you want. When you save the panel, it will already have the characteristics that we defined in the data class.



Another way to create the control with a data class is to select the **DROP_BOX** control and start drawing it. When the **Drop Box Properties** window displays, select the **Class** query (binoculars icon) and select the data class (as shown below):



If a **DROP_BOX** control is drawn and the binoculars are selected to view the data classes, only the data classes defined for that type of control will be shown.

If we define a data class associated with a **MULTI_LINE** or a **GRID** control, it is possible to use a feature called **Extended Validation** that will have a table associated with it and allow for more complete validations using keys.

Refer to **Data Classes** in the PxPlus Help documentation.

In addition, refer to **Data Class Controls** for the steps to create a control from a data class when using the NOMADS panel designer.

User-Defined Controls (CTLs)

We know that, internally, NOMADS assigns unique values to each control, which allows us to identify them with a variable (usually the name of the control plus the **.CTL** extension). These numbers are unique and usually not available to the user (in the sense of freely assigning them).

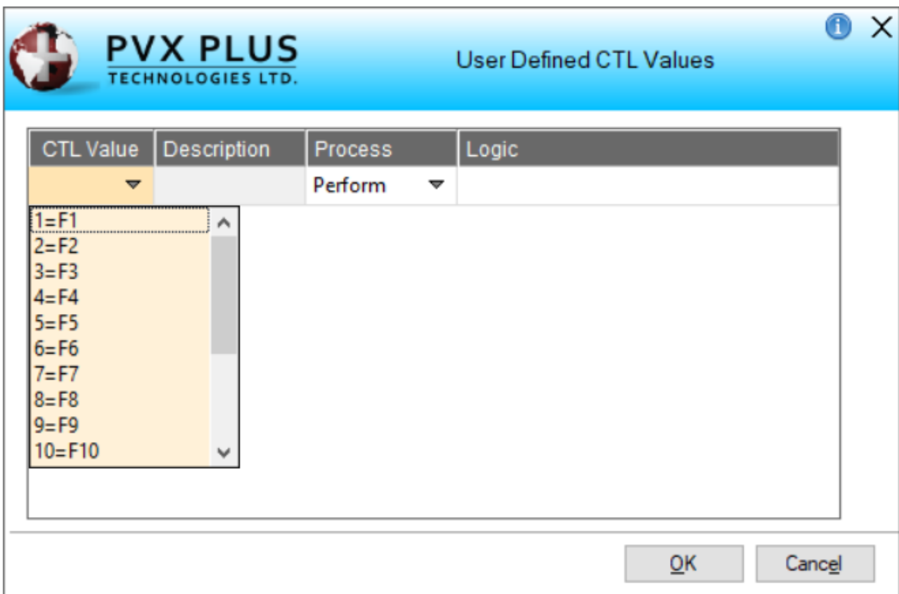
However, we can create events associated with some ConTroL (**CTL**) values. **Example:** We can designate a function key to generate a **CTL** value.

Note: By default, PxPlus assigns the **CTL** values 1, 2, 3 and 4 to the keys [**F1**], [**F2**], [**F3**] and [**F4**]. [**Enter**] generates the **CTL**=0.

Other events and special situations generate other CTL values, usually negative values:

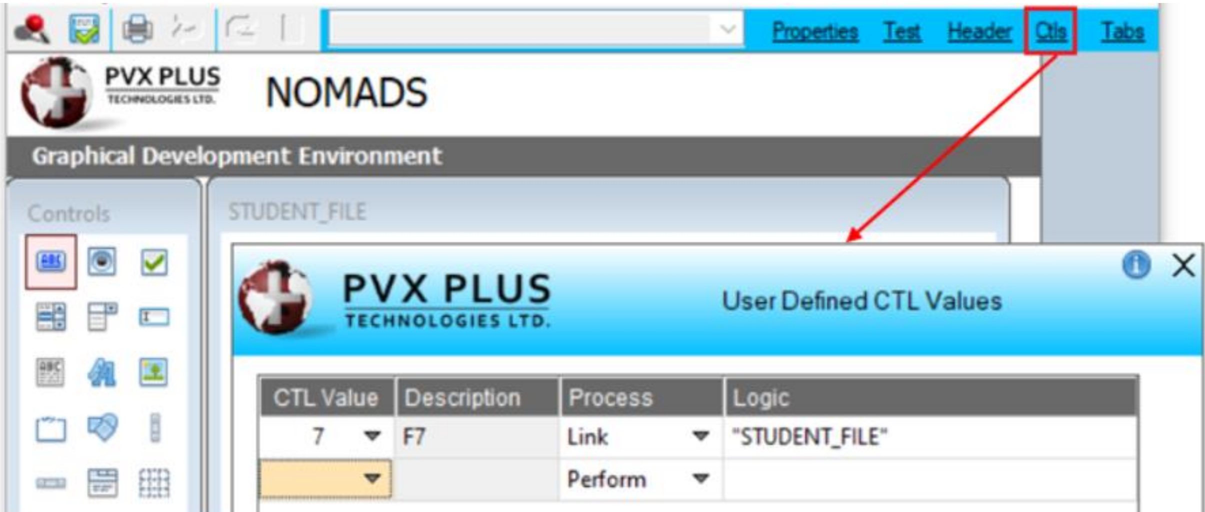
> Values between -1 and -999 are used to enable hot keys, and processing will cause PxPlus to run a special program called ***CONTROL**. This program will return control to the program upon completion of its execution.
>
> Values between -1000 and -1999 are used to indicate editing functions (keys) such as [**Home**], [**End**], [**Left Arrow**], [**Right Arrow**], etc. Many of these events are handled internally by the language.

We can define an event associated with a key (the [**F7**] key in this example) that we want to be called by a panel called **STUDENT_FILE.** To do this, from the NOMADS graphical designer window, we will execute the [**Utilities**] -> [**User CTLS**] option from the top menu bar, and it will show us the **User Defined CTL Values** window:



On the left side are the CTL values, some of which are preassigned, such as function keys and some edit keys.

We are going to program the [**F7**] key to perform the desired action. We can also use the [**Ctls**] option in the top menu to call the **User Defined CTL Values** window:



Accept the CTL definition and then save our panel. We can now call the STUDENT_FILE panel using the [**F7**] key.

You can enter your own CTL value in the **CTL Value** box, not necessarily use the default ones.

If you need to define CTL values for the entire system, you use **%NOMADS_FKEY_HANDLER$** variable with the name of a program (which you will do), and in this program, it will handle the CTL events.

Example:

An example of the program would be:

```
! PRG_HANDLING_CTL.PXP: Program to manage global CTLs IF
CTL=6 MSGBOX "The F6 key was pressed"
EXIT
```

… and fill the variable with the name of the program:

```
%NOMADS_FKEY_HANDLER$="PRG_HANDLING_CTL.PXP"
```

Refer to **User-Defined CTLs** in the PxPlus Help documentation.

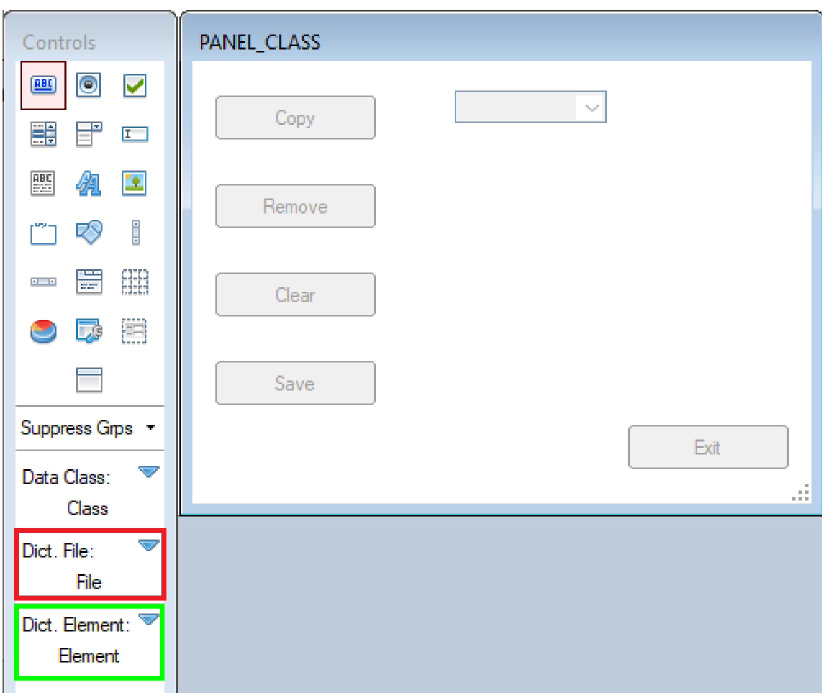
Many events, such as clicking on different parts of the panel (**Example:** the message area), detecting the mouse wheel, double clicking, etc. are handled through negative CTL events.

Refer to **Negative CTL Definitions** in the PxPlus Help documentation.

Using Data Dictionary Fields

We already know how to use Data Classes to modify the attributes of our controls. We can also do the same with the elements of Data Dictionaries; that is, we can define and embed a control associated with an element of a table. When this is done, NOMADS will define the new control with the characteristics of the selected element of the table, making it a container for that field.

In the NOMADS panel designer, the **Controls** box has a button to first select the data dictionary table (marked in red below) and then another button to select the element that we want to insert from that table (marked in green below):

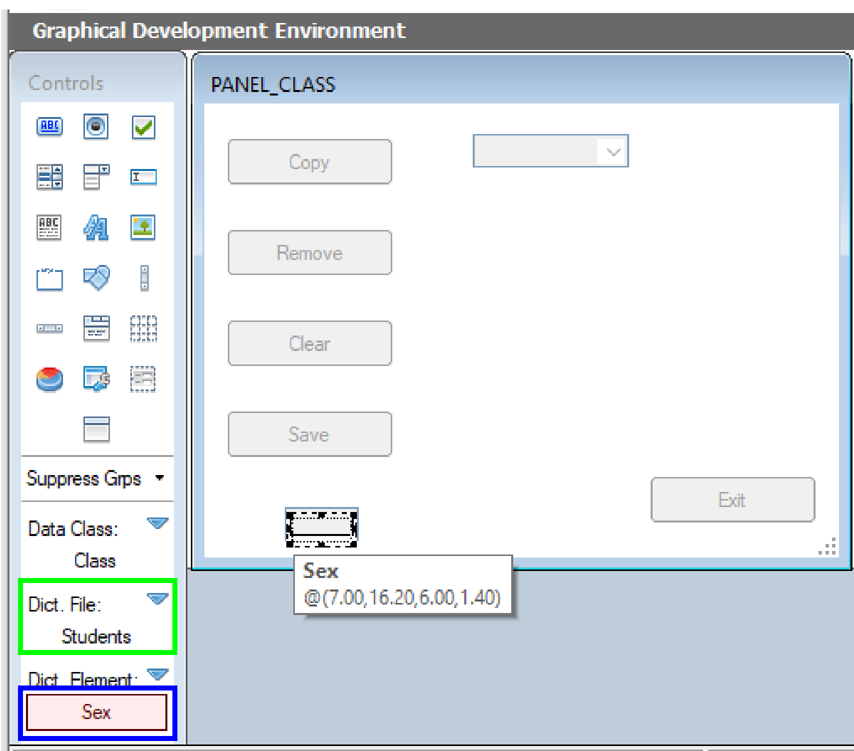


Exercise: Creating a Data Element Control

We are going to select an element from a table, and then we will select the option (marked in green below) and draw a control.

Open the panel called **PANEL_CLASS**. In the **Controls** box, click on the [**Dict. File**] triangle or click the [**File**] option below it, and select the **STUDENTS** table.

Select the Sex element from this table by clicking the [**Dict. Element**] triangle or the [**Element**] option. Then, draw the control on the panel. The new control will assume all the characteristics that the Sex element has:



All attributes, dimensions, validators, formats, etc. that have been specified in the Sex element of the **STUDENTS** table will be "inherited" by the new control, ***including the name***. If you open a table file and read it, the controls associated with the elements (fields) of the table will be updated automatically (if we have the [**Auto Refresh**] attribute active); that is, when we read the **STUDENTS** table, the content of the Sex control (and all the other controls associated with that table) will be updated:

```
channel=unt
open(channel,iol=*)"STUDENTS"
read(channel,key=Code$)
```

With these instructions, we can update the content of these fields/elements.

Refer to **Data Element Controls** in the PxPlus Help documentation for the steps to create a control from a data element when using the NOMADS panel designer.

Drag and Drop Functionality

A functionality that users really like (because of how intuitive it is) is the ability to move elements using drag and drop. This action is available among several types of controls. The controls from which we can start the "dragging" action are LIST_BOX, DROP_BOX, MULTI_LINE and GRID. The controls that are supported for "dropping" the data are BUTTON, RADIO_BUTTON, DROP_BOX, GRID and LIST_BOX.

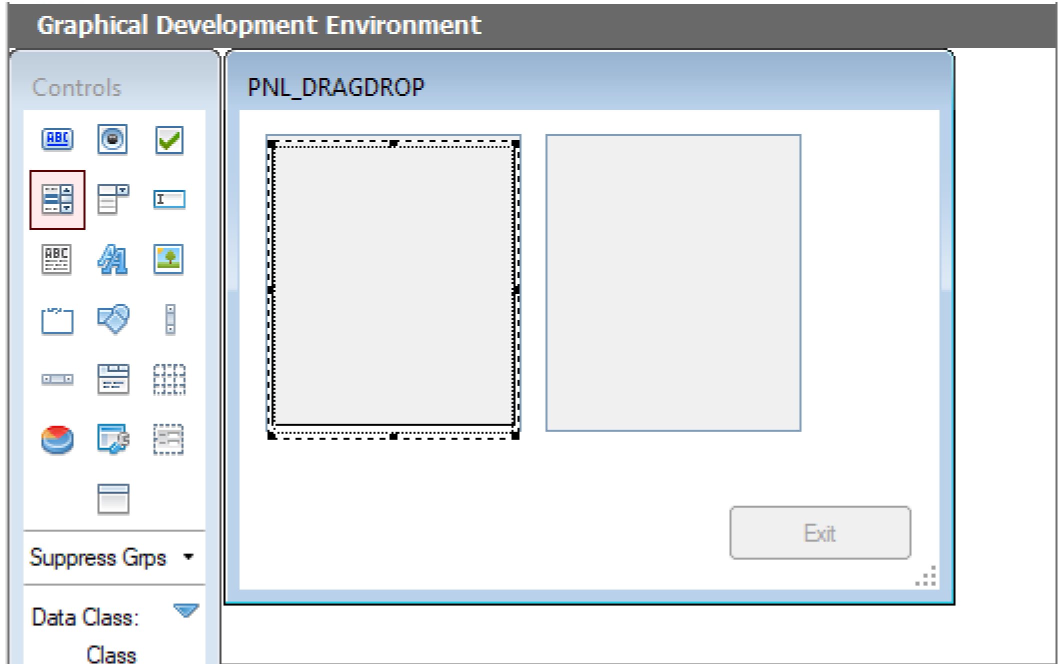
To do this, we must have two compatible controls and run the **Drag&Drop** utility, which allows us to trap these events and assign an action to them.

Exercise: Using Drag and Drop

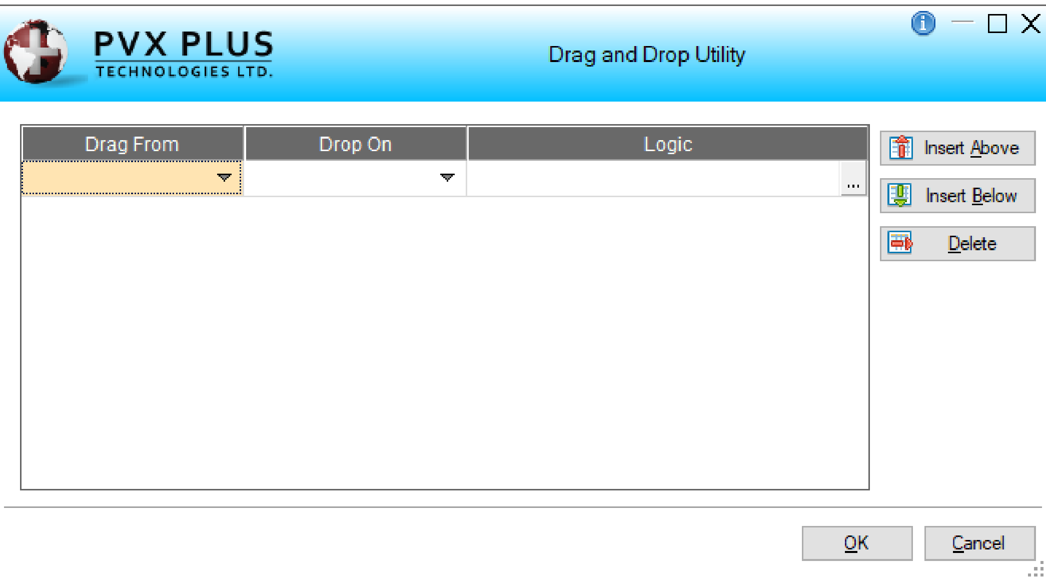
Create a panel called **PNL_DRAGDROP** with two **LIST_BOX** controls, one called **LB_START** and the other called **LB_DESTINATION**.

For the **LB_START** control, enter the following values (on the **Display** tab): Buenos Aires, Medellin, Asuncion, Bogota.

Remember to activate the [**Auto Refresh**] attribute for the panel, and create an **Exit** button with the associated **End** logic.



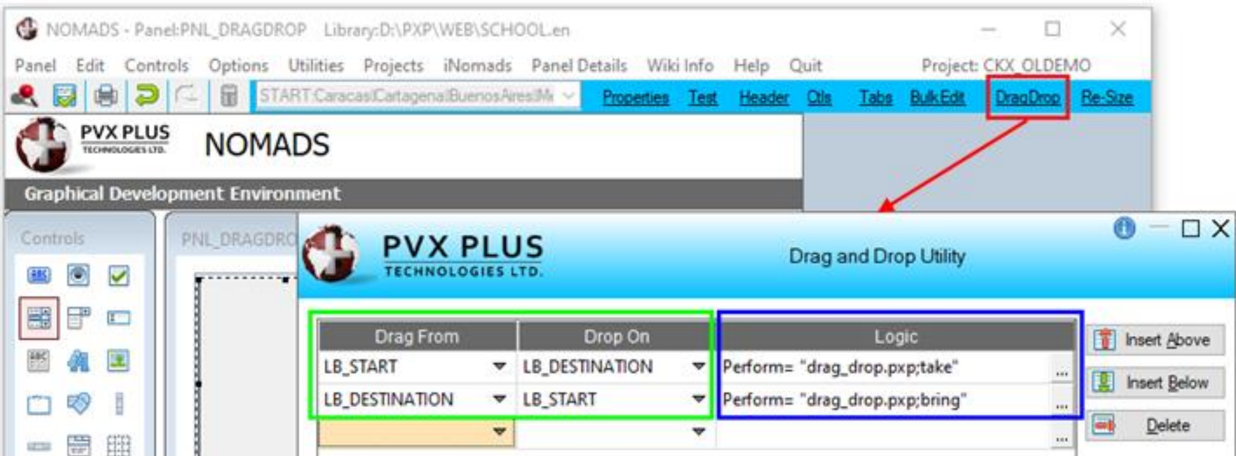
Once the panel with the two **LIST_BOX** controls has been created, click the [**DragDrop**] option (at the top of the NOMADS panel designer) to open the **Drag and Drop Utility**.



Notice that we can basically capture an event "when dragged from one control and dropped on another". This is done by specifying the control to **Drag From** and then specifying the control to **Drop On**.

To finish, we must specify an action, which, in our case, involves a program routine **drag_drop.pxp**.

Each event must be handled individually, so we are capturing both events: when we drag from the control on the left to the one on the right, we put the routine TAKE, and when we bring it back by dragging from the control on the right to the one on the left, we put the routine BRING.



Now, all that remains is to do both routines, which, as you can imagine, are very similar, only alternating the names of the LB_START and LB_DESTINATION controls:

```
!
TAKE: ! DRAG FROM
! Read the selected item
    list_box read LB_START.CTL,SE$,err=NO_HANDLE
! Copy the selected item into the drop control
    list_box load LB_DESTINATION.CTL,0,SE$
! Store the index of the item at the origin
! We need this to erase it
SEL=LB_START.CTL'CURRENTITEM
! Erase the item from the origin control
! LOADing an "*" erases the record/item
    list_box load LB_START.CTL,SEL,*
    return
!
BRING: ! DROP TO
! Read the selected item
    list_box read LB_DESTINATION.CTL,SE$,err=NO_HANDLE
! Copy the selected item into the drop control
    list_box load LB_START.CTL,0,SE$
! Store the index of the item at the origin
! We need this to erase it
SEL=LB_DESTINATION.CTL'CURRENTITEM
! Erase the item from the origin control
! LOADing an "*" erases the record/item
    list_box load LB_DESTINATION.CTL,SEL,*
    return
!
! In case no selected item
NO_HANDLE:
    msgbox "No item selected "
    exit
```

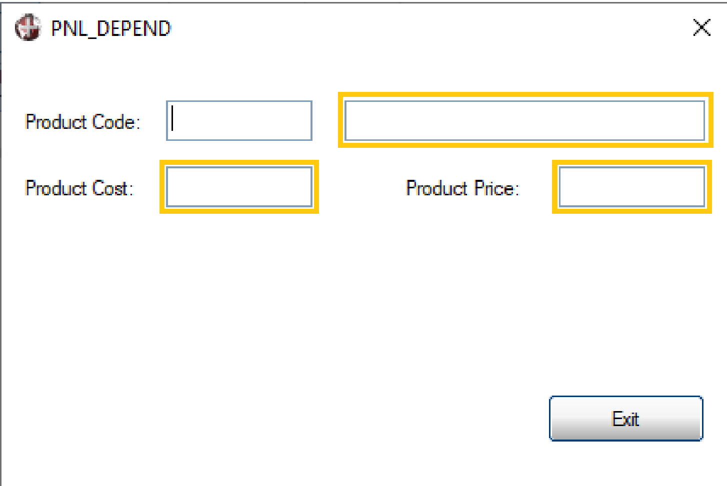
Refer to **Drag and Drop Utility** in the PxPlus Help documentation.

Dependency Definitions

A dependency table, also called a ***dependency definition***, is basically a way to condition the state of some controls. Through these tables, we can enable, disable, show, hide, lock or unlock one or more controls as long as a condition exists.

A classic example (which will be the one we will study) is to disable some controls while the "main" control has no information.

Let's create a panel called **PNL_DEPEND** with some MULTI_LINE controls and field titles, as well as an **Exit** button (with the **End** action):

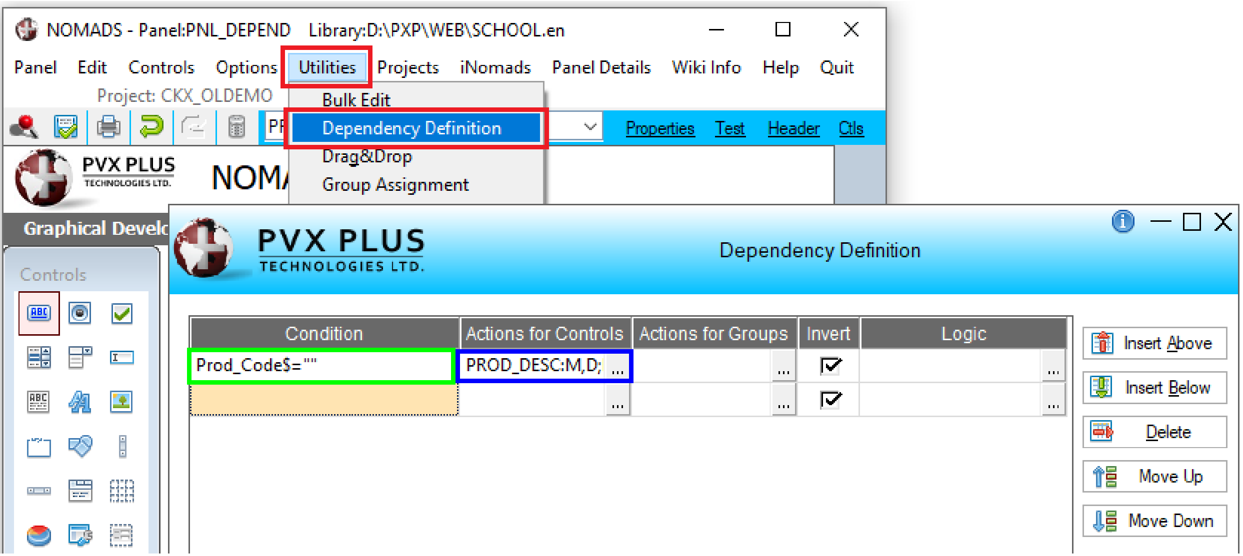


Example:

We will define a dependency for the MULTI_LINE controls corresponding to the description, cost and price of the product (marked in yellow). The condition will be that the controls will be disabled while the Product Code MULTI_LINE control has no information.

Once the panel is defined, we are going to select the [**Utilities**] -> [**Dependency Definition**] option (marked in red) in the top menu to open the **Dependency Definition** window.

Basically, it has three parts: a condition (marked in green), an action for some controls (marked in blue) and another action for controls grouped in a logical group. The operation for these two cases is the same, so we will take the example with some individual controls:

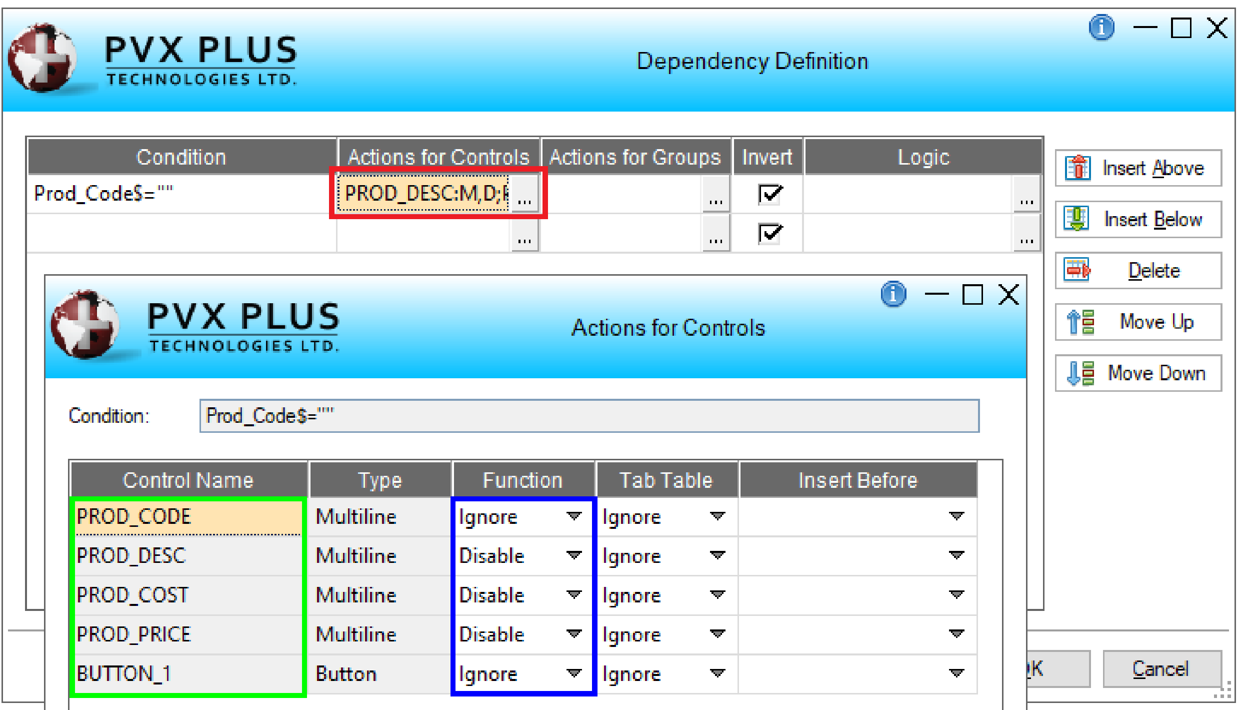


Note that we can have several dependency conditions. The conditions will be evaluated sequentially once the **Post Display** event is processed.

Apart from the actions associated with each condition (*Ignore*, *Enable*, *Disable*, *Show*, *Hide*, *Lock*, *Unlock*), it is also possible to associate a logical action through a program or routine. This is done in the [**Logic**] box at the end of each conditional line.

Once the condition **PROD_CODE$=""** is entered, we proceed to associate the logic for the individual controls. Click on the three-dotted button associated with the [**Actions for Controls**] box (marked in red).

This opens the **Actions for Controls** definition window where we can specify what action will be performed for each control:

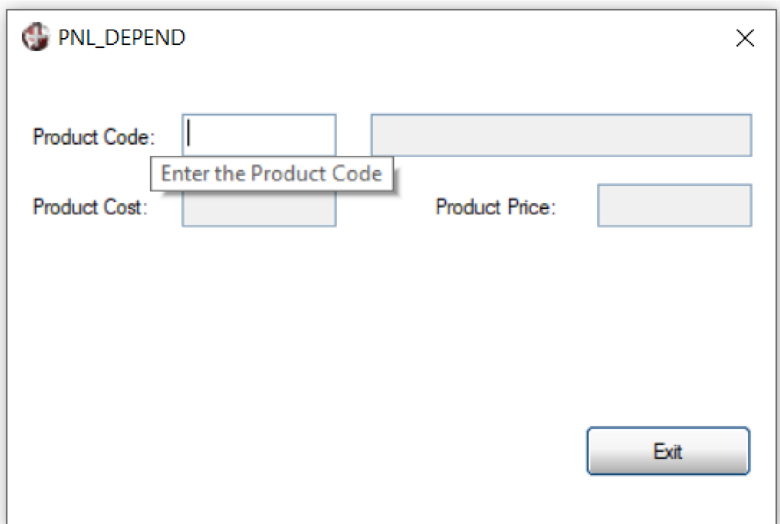


Refer to **Dependency Definitions** in the PxPlus Help documentation.

Floating User Tips (Floating Help)

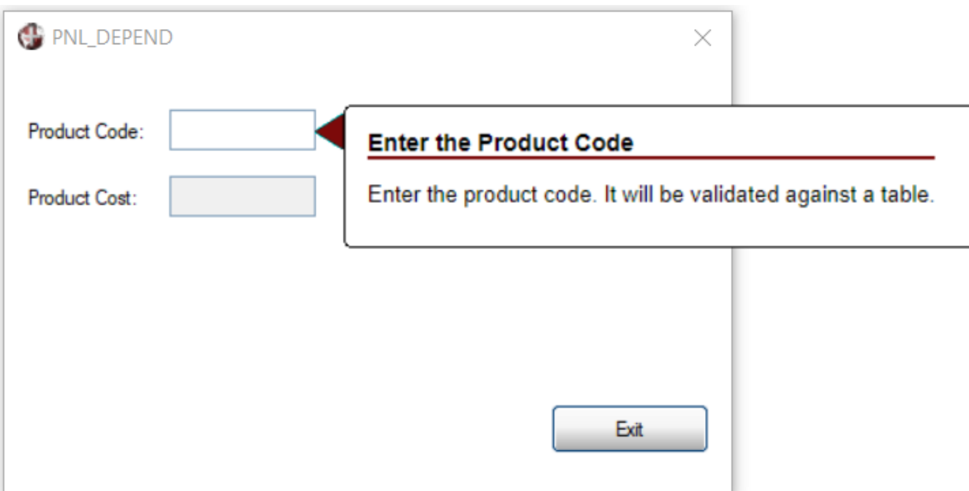
This function is used to offer the user a hint or floating help on certain controls. It only requires that the user hover the mouse pointer over the control for a couple of seconds so that a box with information about the control will appear.

Example:



PxPlus offers several types of floating messages, from the one we just saw, which is internal to Windows, to a much more personalized one that accepts pages in HTML format to be shown to the user.

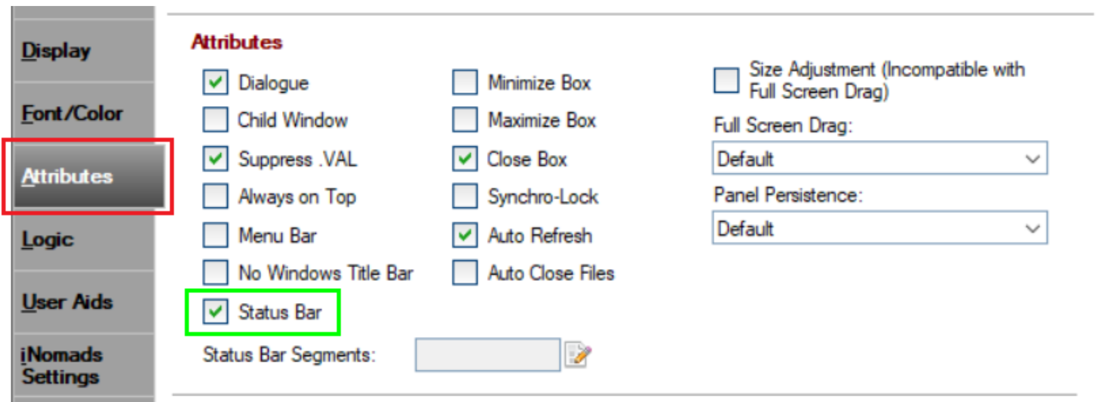
Example:



This box, its colors, appearance, etc. are user modifiable, so you can use the floating message (tooltip) of the operating system (shown in the first example) or a customizable message, which is the previous case.

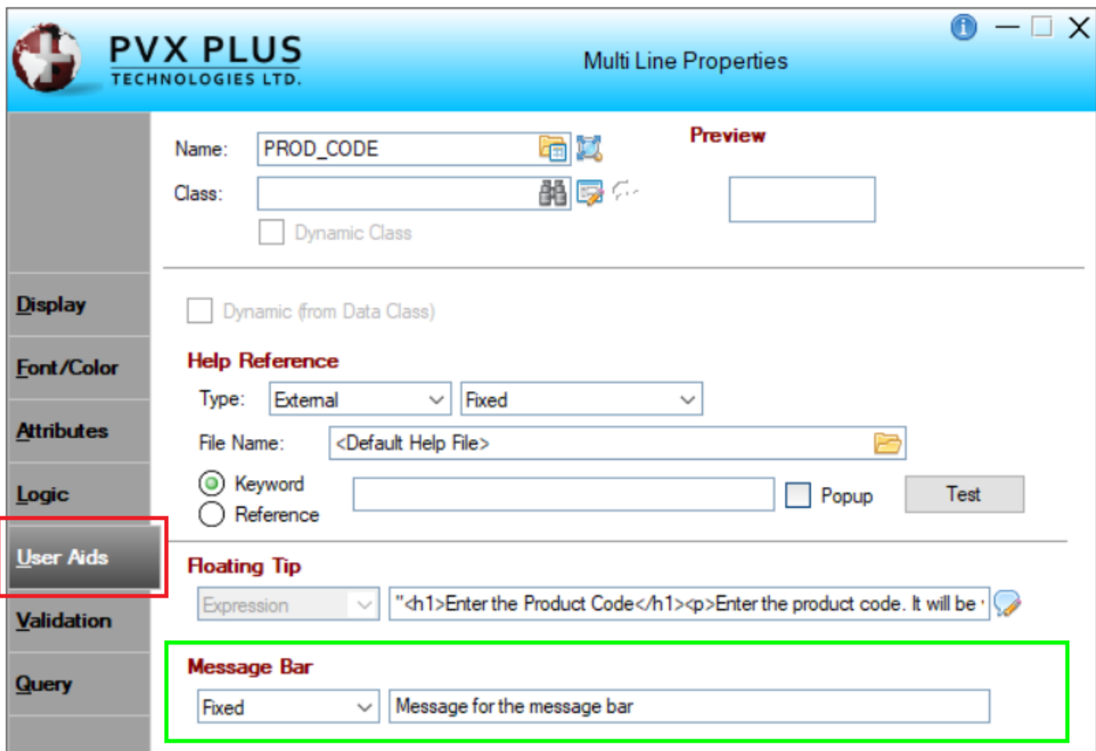
It is also possible to specify a message that will appear in the panel's message line (if it has one defined) with information regarding an associated field. The [**Help Reference**], [**Floating Tip**] and [**Message Bar**] are defined in the **User Aids** tab in the properties window of each control.

The Message Bar displays at the bottom of a panel, and by default, it is not active. To activate it, go to the panel properties by selecting the [**Header**] option in the top menu of the panel designer. On the **Attributes** tab, select the [**Status Bar**] check box.



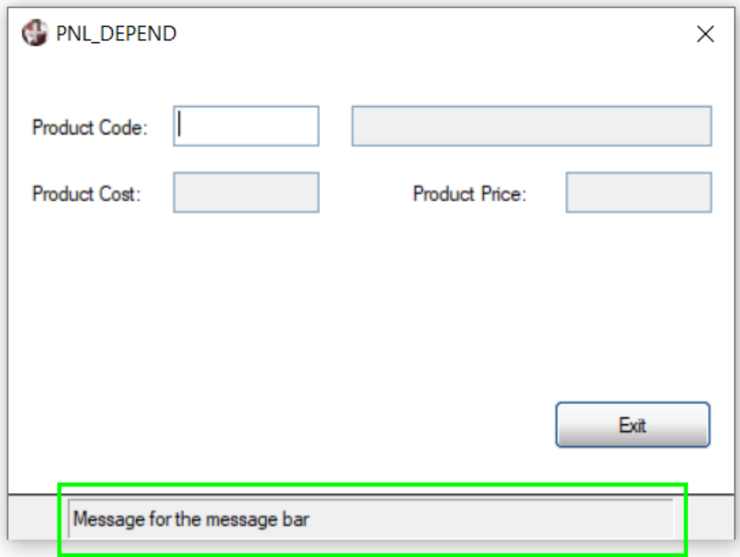
Once the [**Status Bar**] attribute is activated, we can add a message for it in the [**Message Bar**] text entry box in the **User Aids** tab of the control's properties.

We will do this for the **PROD_CODE** control and enter the following text: "Message for the message bar".



Save the changes and test the panel.

Notice that at the bottom of the panel, we now see the message when we go to this control:



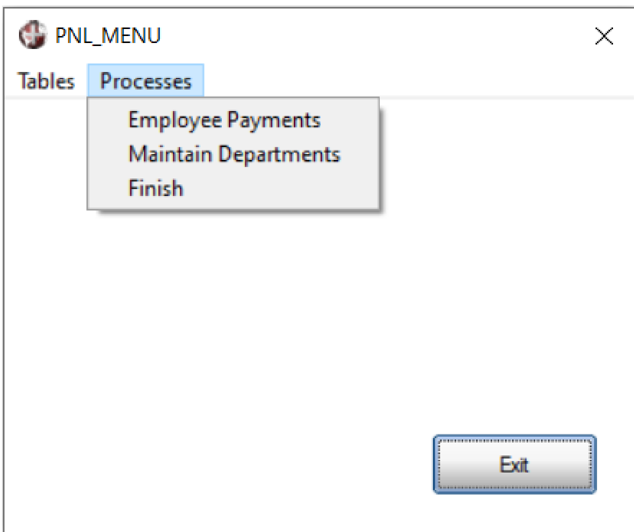
You don't need to prefix the messages. You could have them in a global variable and put the name as an expression in that control's user message.

Note: Global variables maintain their values throughout the entire PxPlus session and their names must begin with **%** (*percent sign*); for example, **%MSJ_TIP$.**

For information on creating a Floating Tip and a Message Bar for a MULTI_LINE control (which we used in this example), refer to **Multi-Line Properties** (**User Aids** tab) in the PxPlus Help documentation. (Similar information can also be found for other control types.)

Defining a MENU_BAR and POPUP MENU

PxPlus, through the NOMADS panel designer, allows you to design a menu bar in a panel, which is basically a series of options at the top.



In the case of menus for the NOMADS panel designer, these are basically composed of **Menu Items**, **Menu Groups** and **Menu Links**.

A ***menu Item*** is a menu option, usually associated with an action (execute a command, call another panel, execute a routine, etc.).

A ***menu group*** is an option that contains other options inside (other groups or items).

A ***menu link*** is the concatenating or linking of two or more menus in a transparent way for the user.

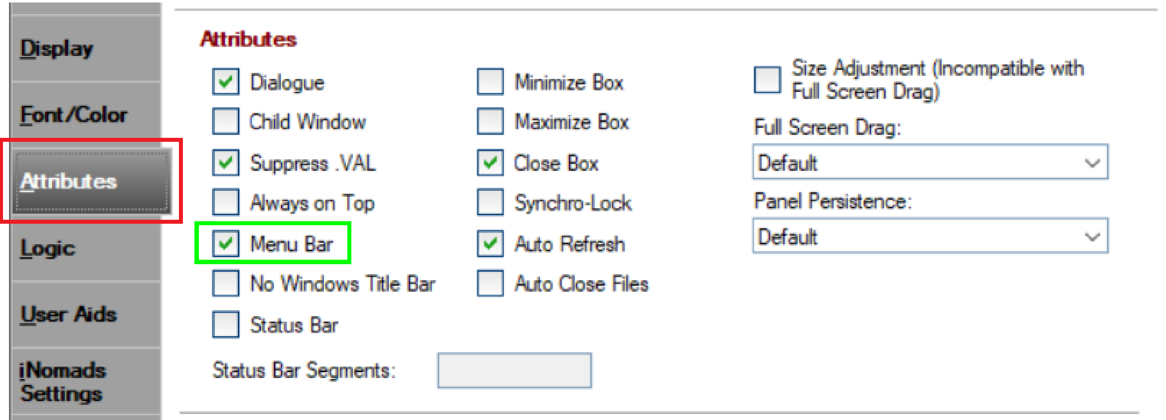
In PxPlus, there are two types of menu: one that is associated with a panel (which is defined from the panel itself) through the [**Panel**] -> [**Menu**] option, and the other that could be considered independent and can be defined from the PxPlus IDE main menu by opening the **Graphical Application Builder (NOMADS)** category and selecting **Menu Bar Definition**. The operation of both is the same.

For now, we are going to study how to define a menu.

Exercise: Defining a MENU_BAR

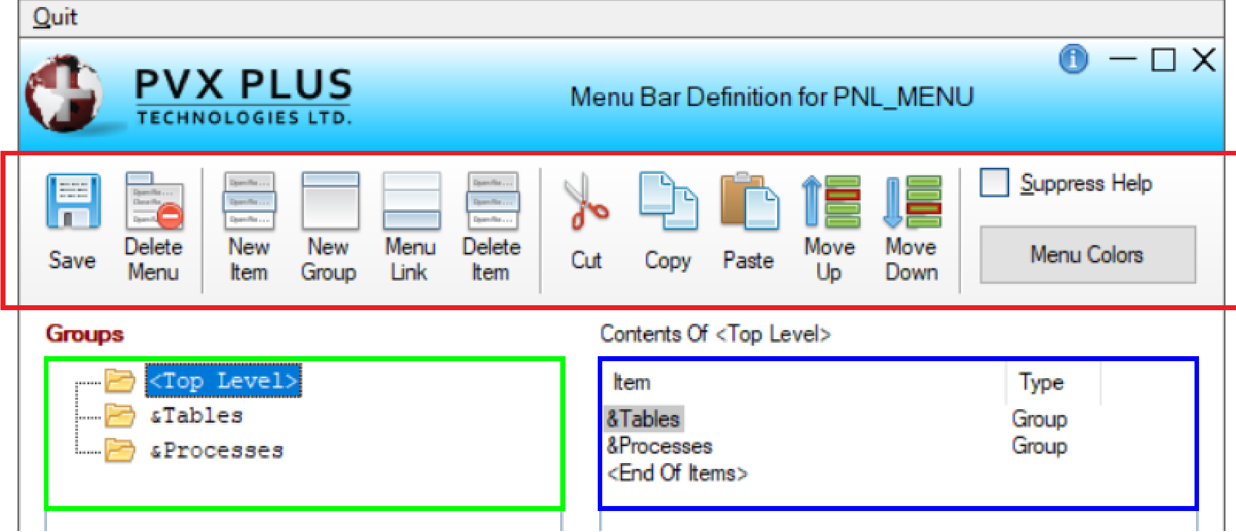
Let's begin by creating a new panel called **PNL_MENU** and adding an **Exit** button (with the **End** action).

We will be adding a menu bar to the panel; therefore, the [**Menu Bar**] attribute must be activated. Go to the panel properties by selecting the [**Header**] option in the top menu of the panel designer. On the **Attributes** tab, select the [**Menu Bar**] check box.



To define a menu bar for this panel, select the [**Panel**] -> [**Menus**] option from the top menu bar. This opens the **Menu Bar Definition** window.

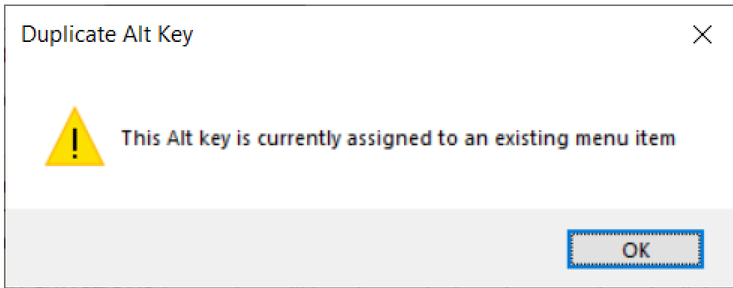
The **Menu Bar Definition** window has three basic parts: the action buttons at the top (marked in red), the menu groups on the left (marked in green), and the contents or details section (marked in blue). We will use the [**New Item**], [**New Group**] and [**Menu Link**] buttons to define the menu, and the other buttons will help us to edit or maintain it.



As we already mentioned, an **Item** is a menu option that has an action associated with it, while a **Group** is a menu option that contains other entries.

We are going to define a menu with two groups, Tables and Processes.

Note: It is possible that, when defining menu items, we may be asked to assign an accelerator key, which is a character preceded by the **&** (*ampersand*) symbol. It is not possible to have two menu items with the same accelerator key, and any attempt to do so will display a message:



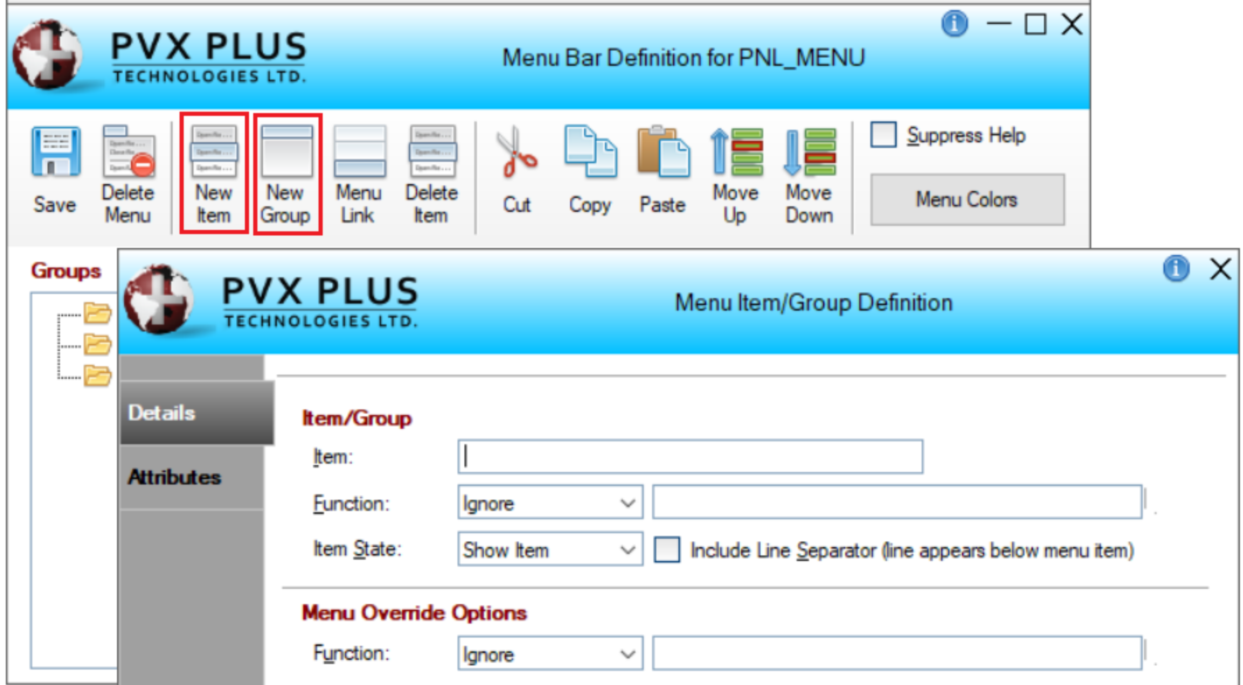
The "Tables" group will have two items: "Employees" and "Departments". The "Processes" group will have three items: "Employee Payments", "Maintain Departments" and "Finish". The menu structure would look like this:

Tables	Processes
Employees	Employee Payments
Departments	Maintain Departments
	Finish

The **Menu Bar Definition** window is quite simple. We suggest that you experiment a little with it.

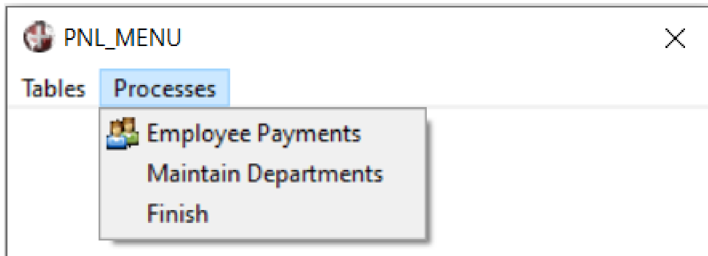
First, create the menu groups. Then, select a group and define the menu items for that group. Remember that you must first select the group on the left side of the screen where it says **Groups**.

Once you select the [**New Item**] or [**New Group**] buttons, the **Menu Item/Group Definition** window opens. In this window, you must enter the item name, the function and its argument (if you have one).



In the **Attributes** tab, you can specify an image for this menu option. For the purposes of this exercise, we have associated an image with the Employee Payments option, and we have associated the **End** action with the **Finish** option.

Once we have finished creating the menu and saved our work, we should have a menu similar to the one shown below:



Defining a **Menu Link** is very simple. We can use a menu option to connect with another existing menu. When we define the Menu Link, we indicate which menu will be the one we are going to link.

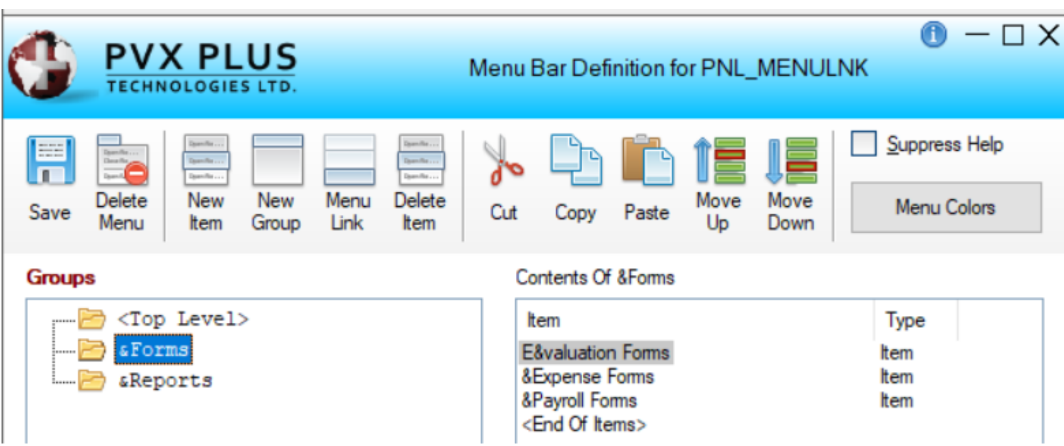
Exercise: Defining a Menu Link

In this exercise, we will define a Menu Link. Since we have no other menus, we first need to create a panel and define a menu, which we will use to link.

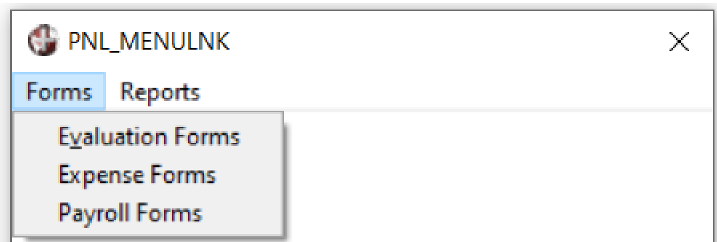
Let's begin by creating a new panel called **PNL_MENULNK** and adding an **Exit** button (with the **End** action). Select the [**Menu Bar**] attribute in the **Attributes** tab of the panel properties.

To define a menu bar for this panel, select the [**Panel**] -> [**Menus**] option from the top menu bar. In the **Menu Bar Definition** window, define two groups, **Forms** and **Reports**. The **Forms** group will have three items: Evaluation Forms, Expense Forms and Payroll Forms. The **Reports** group will have two items: Reports Without Balance and Historical Reports.

When done, save the menu bar definition.



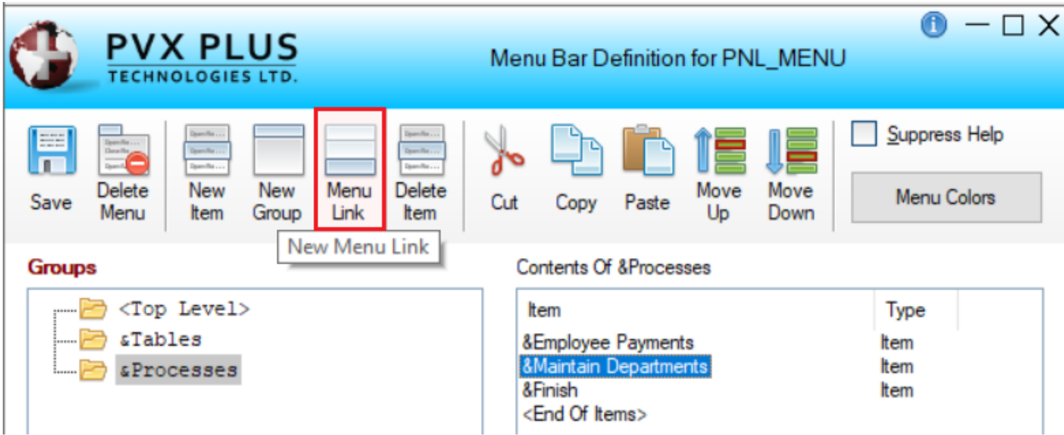
Save and test the panel:



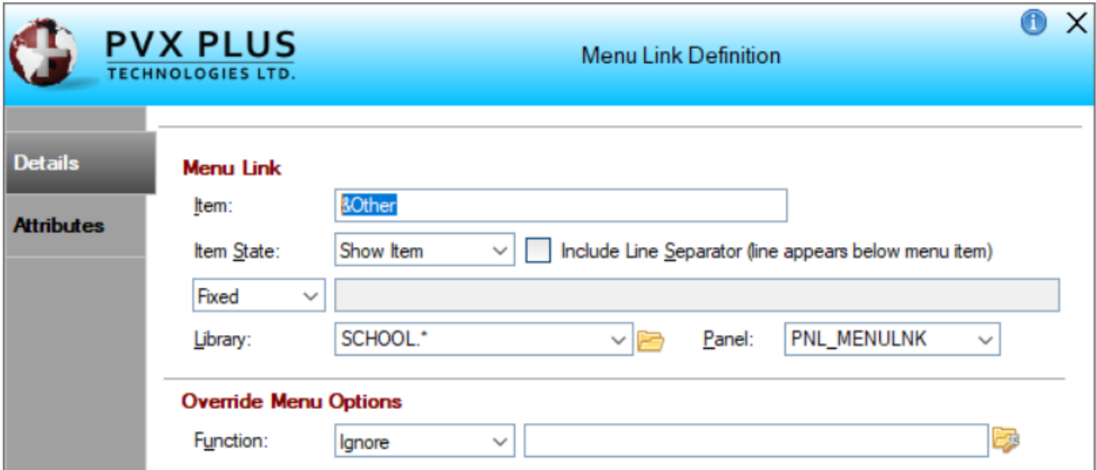
Now open the panel **PNL_MENU**. We are now ready to define a Menu Link. Select the [**Panel**] -> [**Menus**] option to open the **Menu Bar Definition**.

We will add a new menu item to the Processes group.

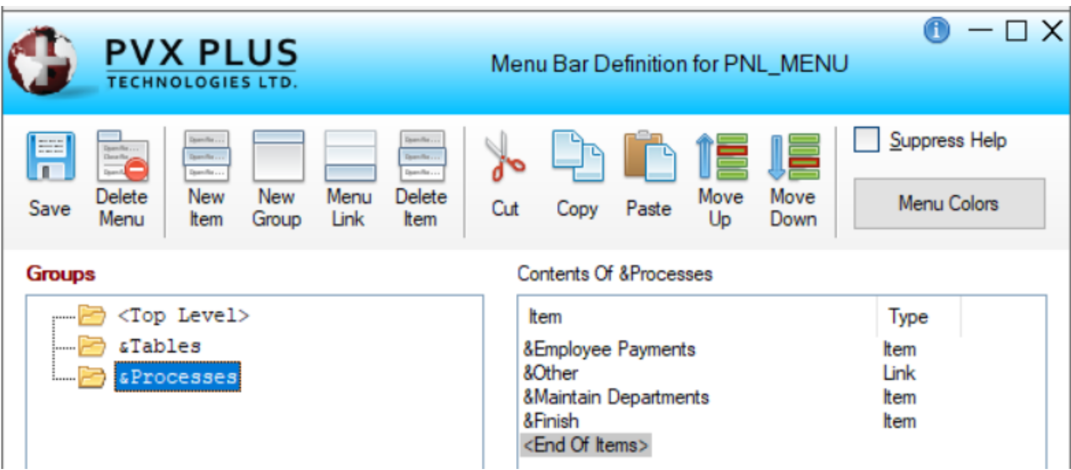
Select the Processes group in the [**Groups**] box on the left. We want to add the linked menu item under the Employee Payments item, so click once on the Maintain Departments item to highlight it. Click the [**Menu Link**] button.



In **Menu Link Definition**, enter Other as the Item name. The menu we want to link is associated with the **PNL_MENULNK** panel, so select this panel from the [**Panel**] drop-down list.

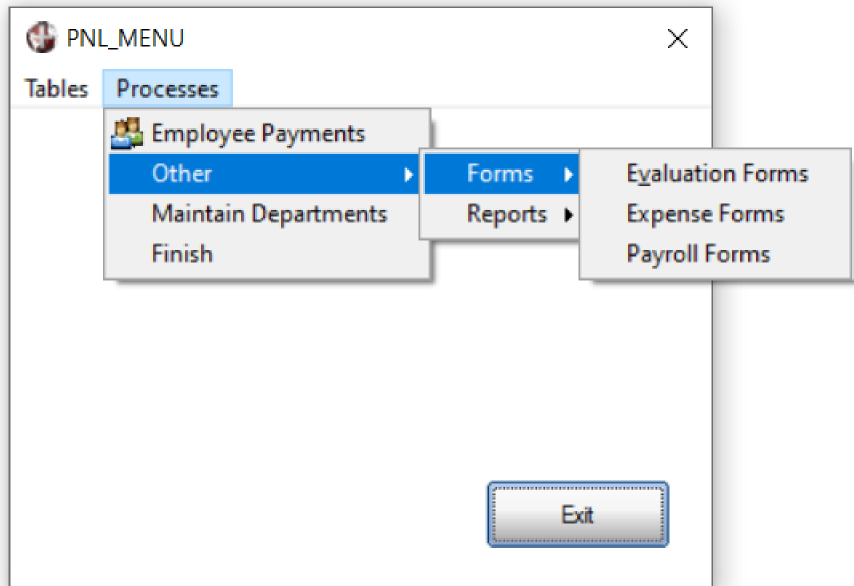


The Processes group now has the following items:



We save our work. Save and test the panel.

We see that we have a new option under the Processes group called Other, which, in turn, links to other options:



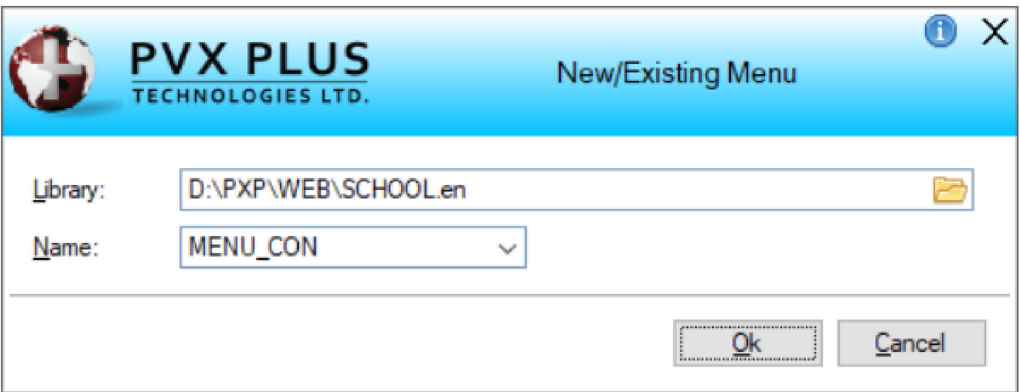
As you can see, defining the menus is a very simple process. Basically, fill in the spaces according to your criteria (Item or Group), and then assign an action to each item.

You can also play with the appearance by adding line separators, colors, item colors, images, etc.

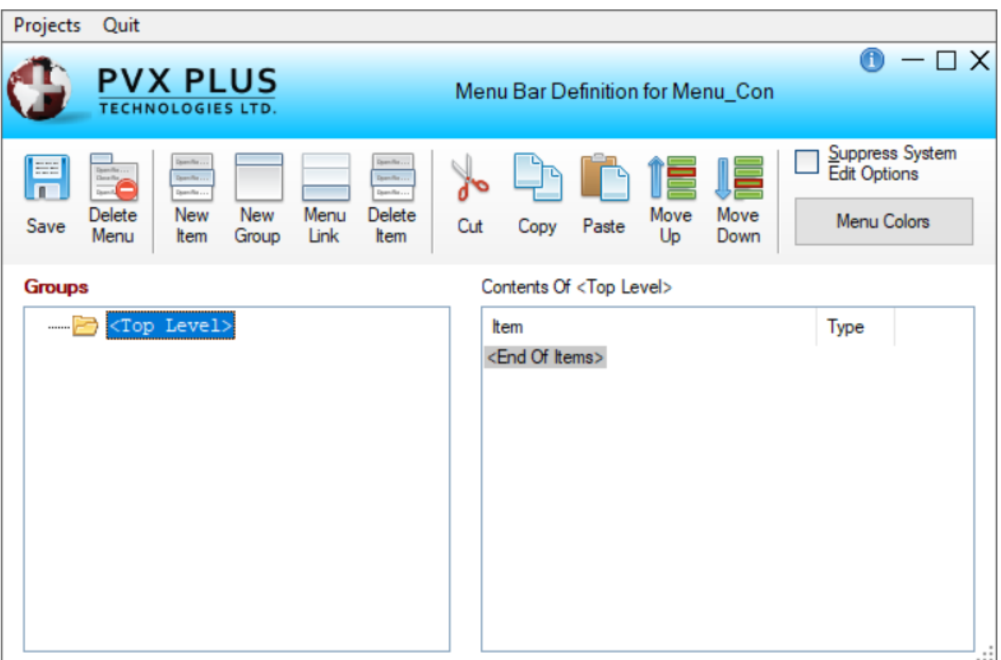
Now, we are going to return to the individual menu definition screen. This is run from the PxPlus IDE, not from the NOMADS panel designer. When we run it, notice that it does not ask for the name of a panel. It only asks for the name of a Library and a menu Name.

Enter the name of the Library (**SCHOOL.EN**) or select it by clicking the Query button on the right (yellow folder).

Enter the name of the panel as **MENU_COM**. Click the [**OK**] button.



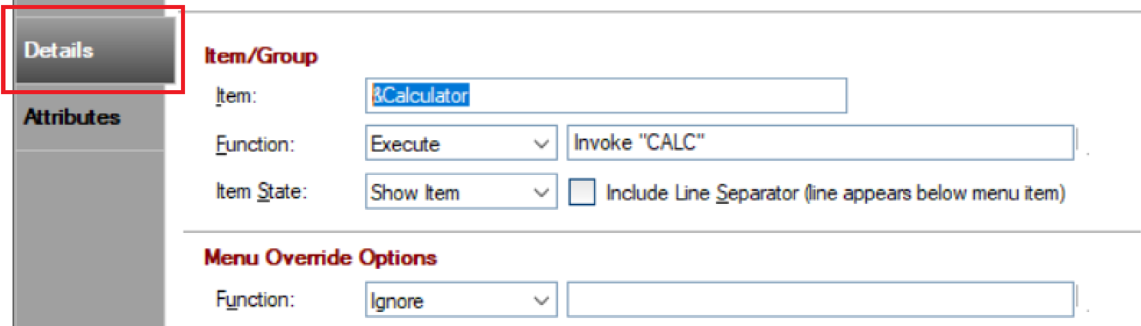
You see what looks to be the same **Menu Bar Definition** window that was used to create the other menus. But, unlike the other menus, this type of menu is not associated with any particular panel but can be called independently. This menu is referred to as an independent or contextual menu.



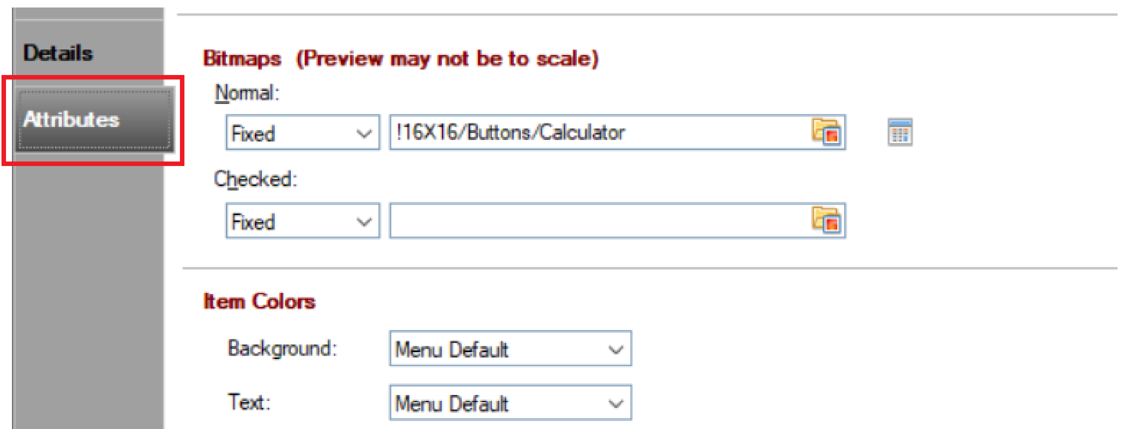
We are going to simplify things and define only two menu items for this new menu, and they will have associated logic:

Calculator -> Execute: INVOKE "CALC"
Excellent -> Execute: MSGBOX "You are an excellent person", "HELLO!"

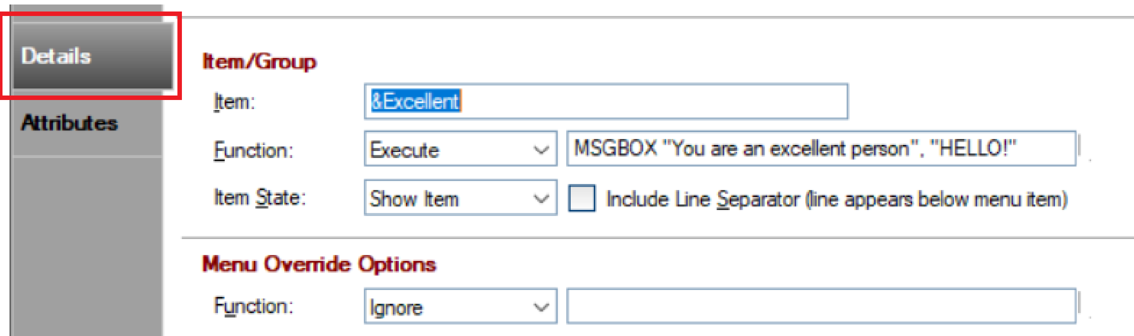
The **Details** tab for the "Calculator" menu item definition looks like this.

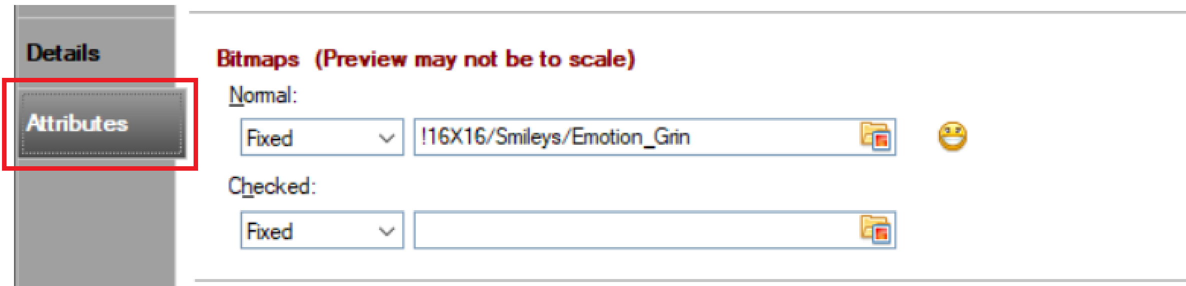


We can also add an image on the **Attributes** tab:

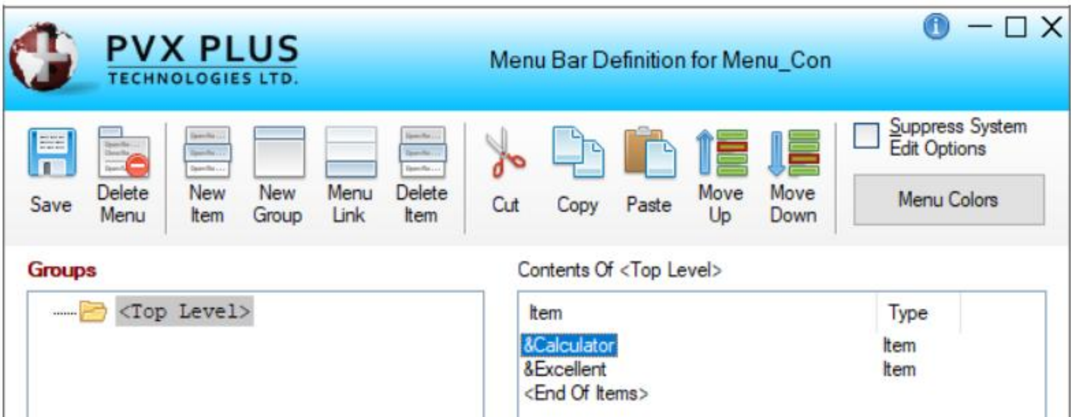


Now, let's look at the **Details** and **Attributes** tabs for the "Excellent" menu item definition:





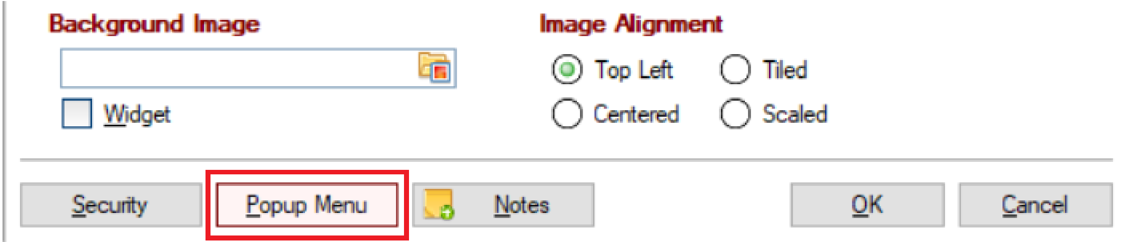
After both menu items have been defined, we save our work. The contents of the menu **MENU_CON** looks like this:



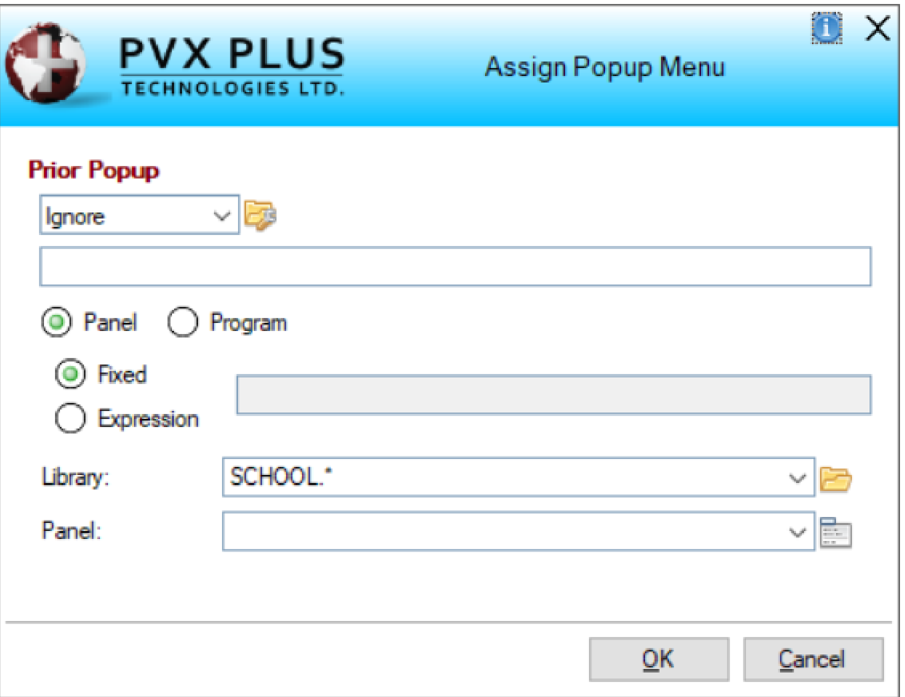
Now, we will return to the NOMADS panel designer and open the panel **PNL_MENU**. We are going to assign the new menu we just created to this panel.

Exercise: Assigning a Popup Menu

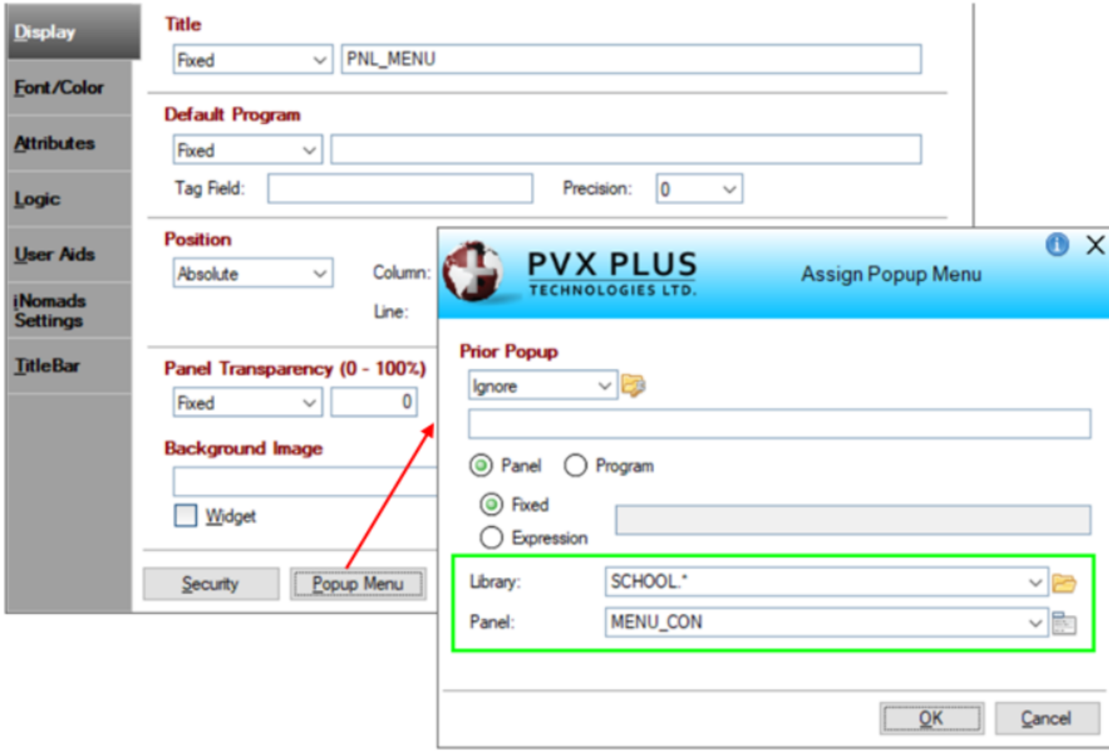
Go to the panel properties by selecting the [**Header**] option in the top menu of the panel designer. At the bottom of the panel properties, we see the [**Popup Menu**] button.



Clicking this button opens the window for assigning a **POPUP MENU** to our panel.

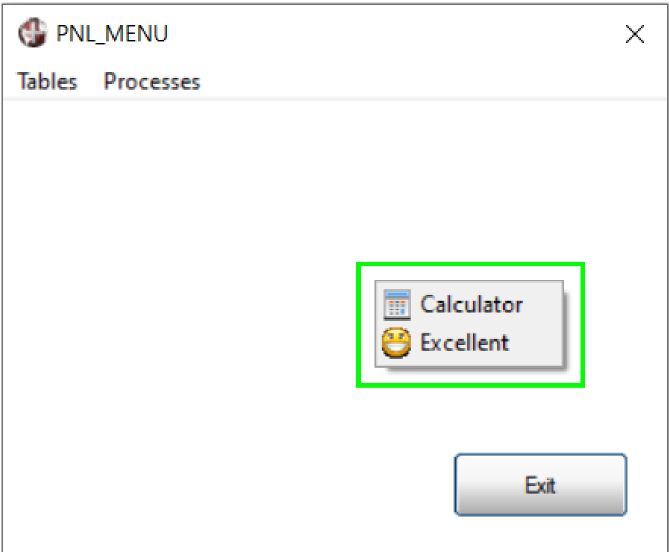


The name of the current **Library** is shown, which is where our **Popup Menu** is located. We need to select the **Panel**, which is actually ***the name of the menu***. Click the **Panel** drop-down arrow and select **MENU_CON**. The **Assign Popup Menu** window looks like this:

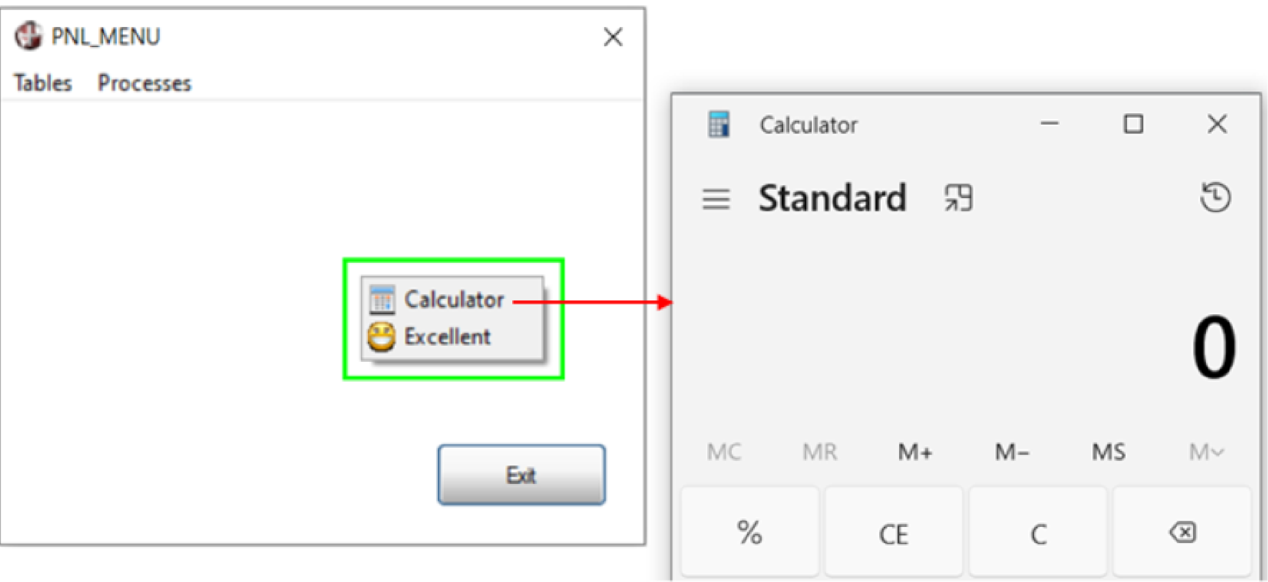


We accept and save our work.

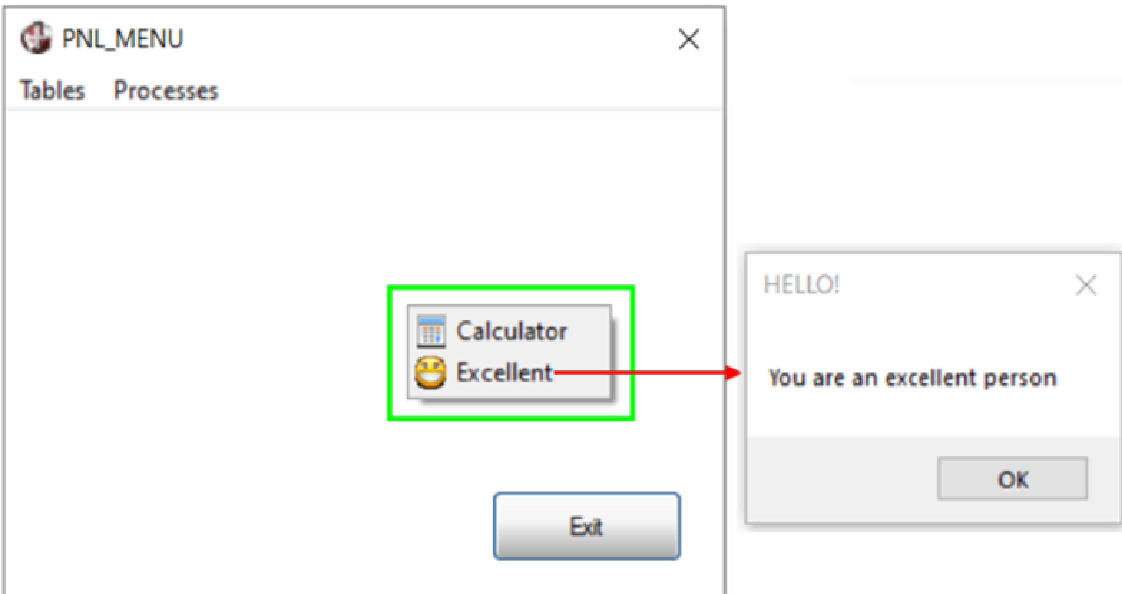
Now, how does it work? Well, we will simply test the panel and right click on any part of the panel that is not with a control.



Clicking on the "Calculator" option opens the Windows calculator application:



Clicking on the "Excellent" option shows a message:



We can associate a **POPUP MENU** with any control. We just open the control's properties window, select the [**Popup Menu**] button, select the menu and that's it!!

As we can see, both menu systems, the horizontal or bar menu and the context menu or POPUP, are very simple to define and even easier to use. Both share the same design interface, the same philosophy and a simplicity that allows you to start being productive in a very short time!

Refer to **Menu Bar Definition**, **Popup Menu** and **Applying a Popup Menu** in the PxPlus Help documentation.

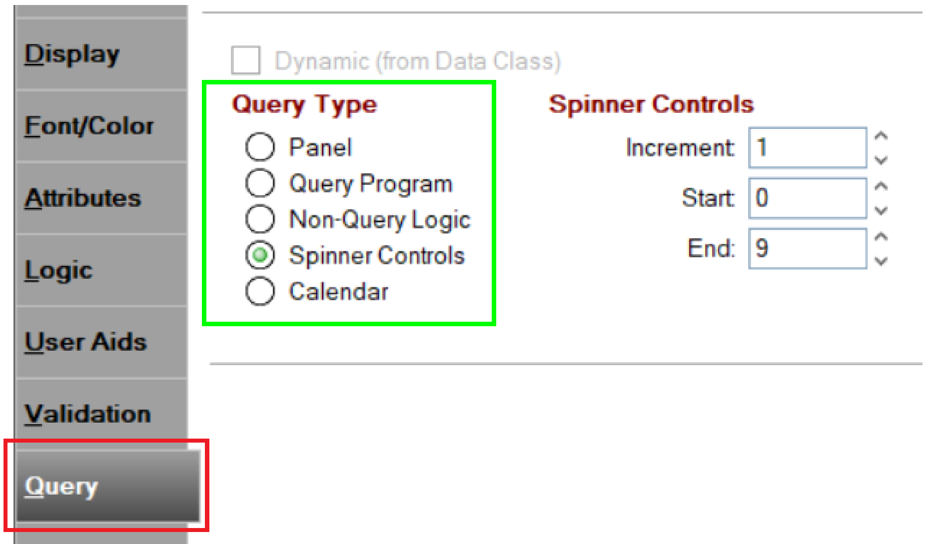
MULTI_LINE Control: Calendar and Spinner Control Query Types

The **MULTI_LINE** control allows you to associate queries of different types. There are two query types that some programmers do not know about and are very simple to implement. The first is the **Spinner Control**, which simulates a box where a value can be increased or decreased using small arrows on the side of the control. The other is a **Calendar**, which allows a calendar to be displayed so that the user can select the corresponding date.

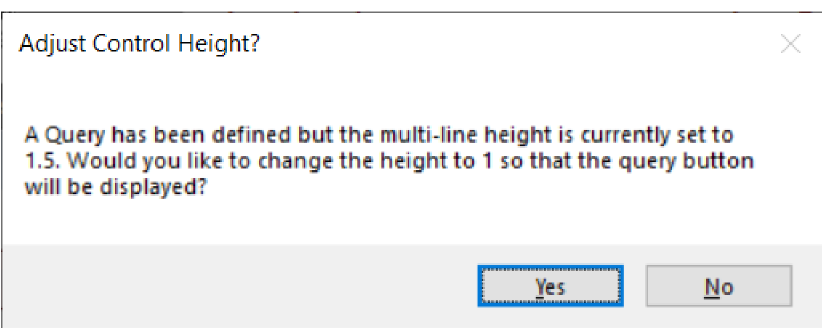
Let's begin with the **Spinner Control**. It is a variant of the **MULTI_LINE** control that serves to graphically increase a quantity of a control. It also allows direct entry of text. Visually, it looks like this:

The little lateral arrows appear when the control has focus.

Defining a **MULTI_LINE** with a **Spinner Control** is very simple. Let's begin by creating a new panel called **PNL_MULTIQRY** and adding an **Exit** button (with the **End** action). Draw the **MULTI_LINE** control. When the **Multi-Line Properties** window displays, select the **Query** tab and then choose [**Spinner Controls**] for the [**Query Type**]:

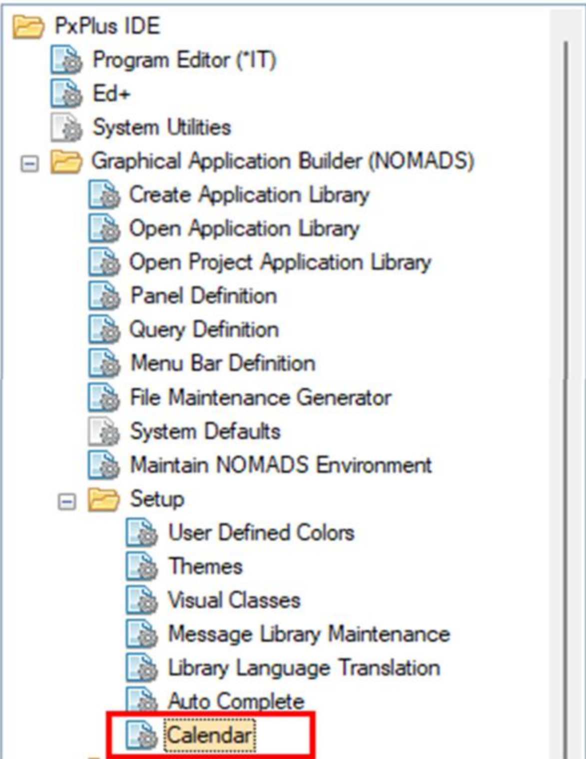


Note: The **MULTI_LINE** control ***must have a height of exactly 1*** for the Query button to display; otherwise, NOMADS will display a message if this is not the case and ask to make the adjustment:



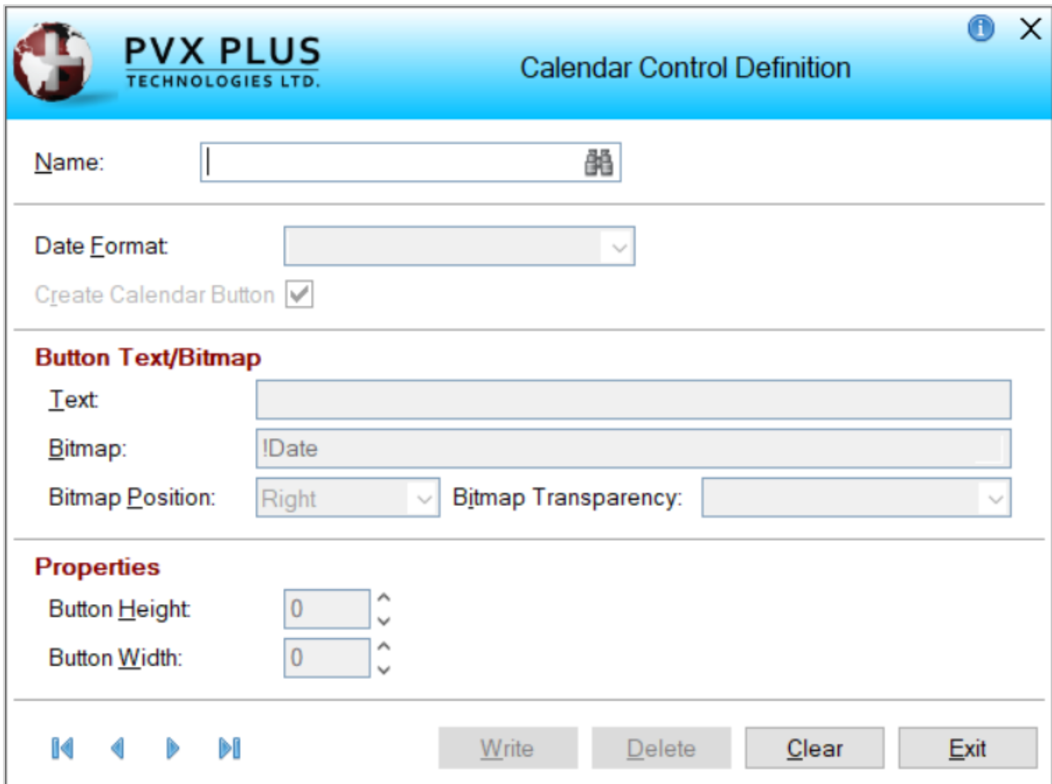
You must enter the [**Increment**] value (usually 1), the [**Start**] and the [**End**] values. Although the control will not go beyond the **Start** and **End** limits, it is possible to enter values that exceed these limits; however, the programmer must validate them separately.

The other type of query requires the prior creation of a Calendar. To do this, go to the PxPlus IDE main menu, open the **Graphical Application Builder (NOMADS)** category, expand the **Setup** category and select **Calendar**.



When you run it for the first time, it may ask for permission to create the required file. Answer **Yes**.

The **Calendar Control Definition** window displays.



You must provide a calendar [**Name**] and select a [**Date Format**].

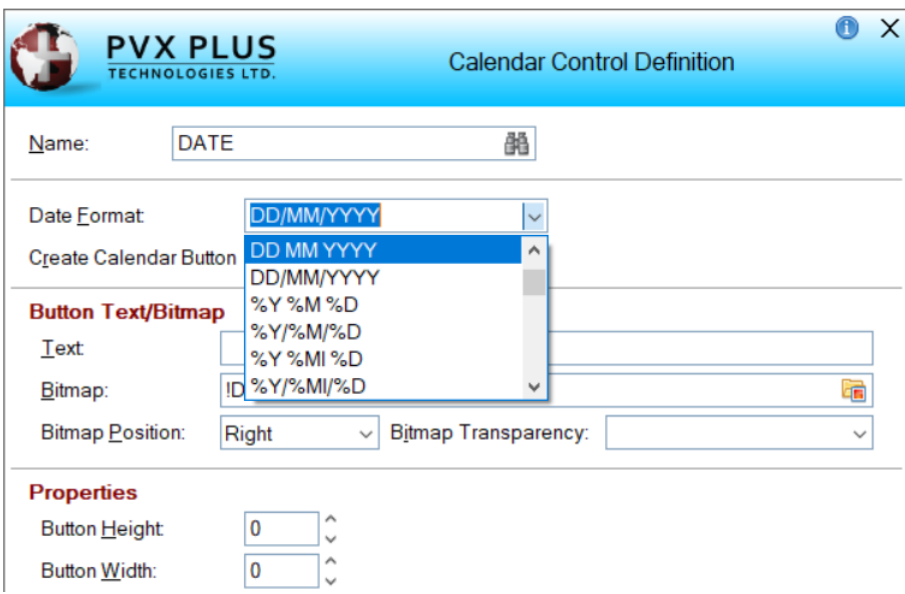
You can also select an image to display for the query and specify whether it will display on the right or left side.

Specify the [**Button Height**] and [**Button Width**]. If left at 0 (default), then it will take the dimensions of the associated **MULTI_LINE** control.

Exercise: Defining a Calendar Control

In this exercise, we will define a calendar control.

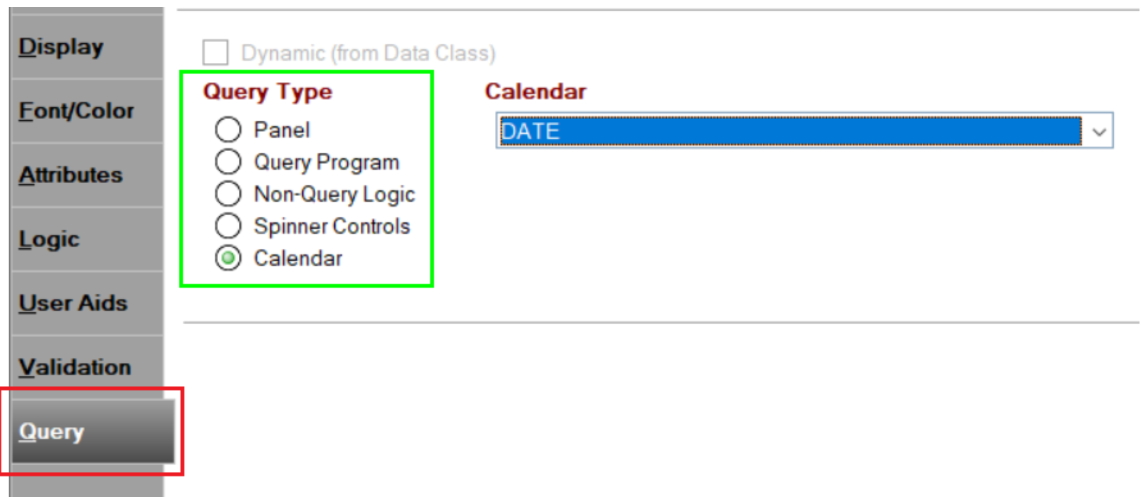
For the [**Name**], enter **DATE**. For the [**Date Format**], select DD/MM/YYYY from the drop-down list. To finish, click the [**Write**] button at the bottom.



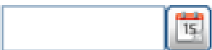
Once the calendar is defined, we can associate it with any control that accepts queries of that type (**LIST_BOX**, **DROP_BOX**, **MULTI_LINE**, **GRID**).

We are going to continue our example by opening a panel, creating a **MULTI_LINE** control and associating a calendar query type to it. This procedure will be similar to the previous one where we created a **Spinner Control**.

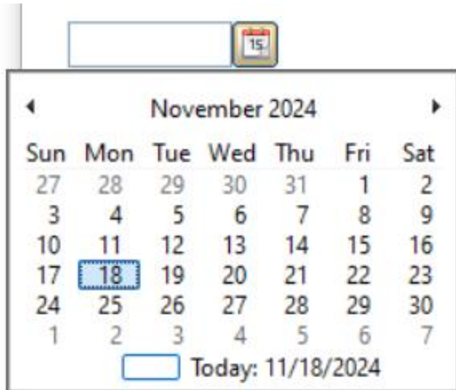
Open the **MULTIQRY** panel and draw a **MULTI_LINE** control. In the properties window, select the **Query** tab and choose [**Calendar**] for the [**Query Type**]. From the [**Calendar**] drop-down box on the right, select the name of the calendar we just created, **DATE**:



Save and test the panel. We see that the control has a small icon to its right:



Clicking that icon opens the calendar:



Associating a Spinner Control and a Calendar to a control is very effective and can be easily implemented in our applications.

MULTI_LINE Control: Other Query Types

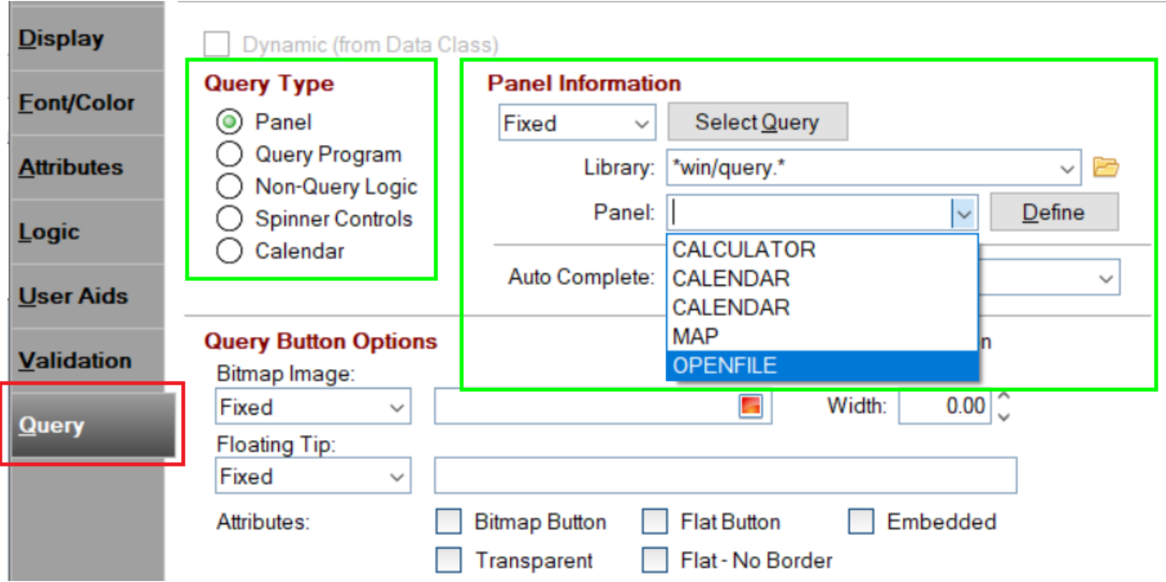
We have seen how easy it is to associate a Spinner Control and a Calendar to a **MULTI_LINE** control. Well, we can also associate other types of queries, such as opening a file selection box, opening a Web page, invoking a calculator, sending an email or consulting a map. All of these functionalities follow the same guidelines as other PxPlus tools: very simple implementation, easy to use and fully integrated with our programs!

OPENFILE Query

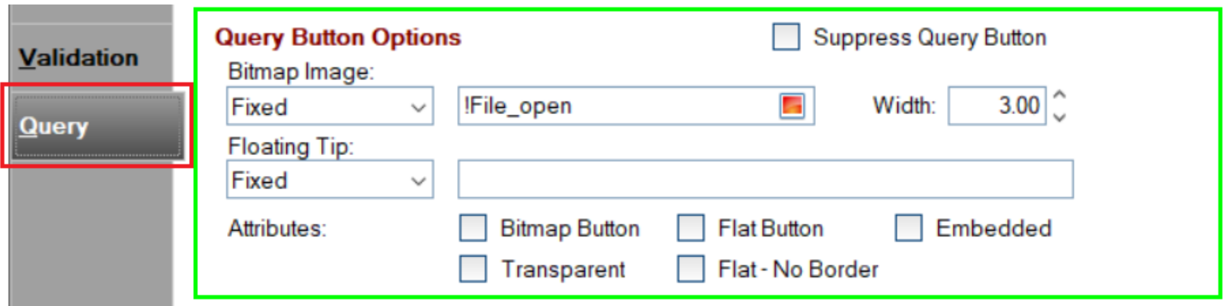
Let's see how to make a query that opens a file selection box.

Once our **MULTI_LINE** control is created, we associate a query by selecting the **Query** tab in the control's properties window. For the **Query Type**, select **Panel**, and then enter the following under **Panel Information**:

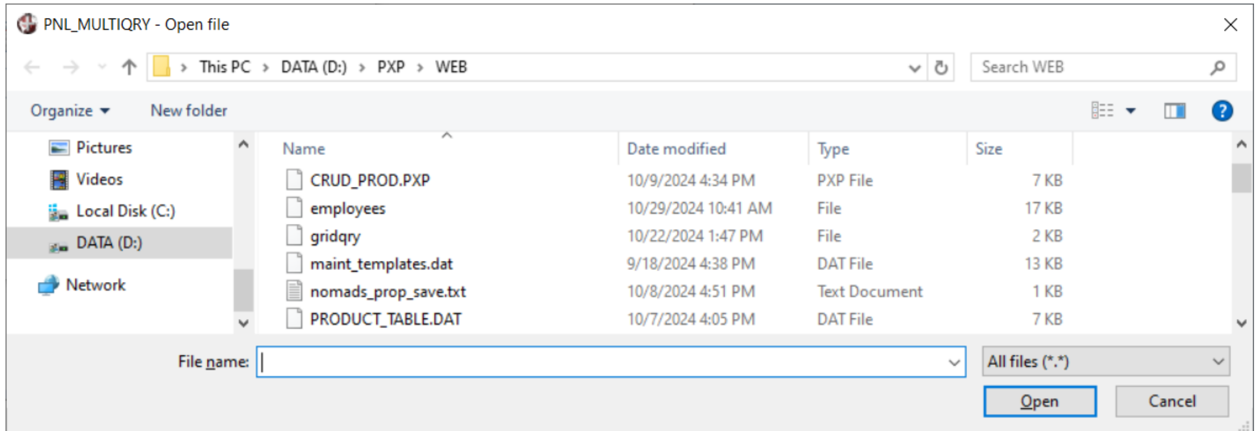
For **Library**, enter ***win/query.***
From the **Panel** drop-down list, select **OPENFILE**.



Under **Query Button Options**, define the appearance for the query button. Select a [**Bitmap Image**] and enter the [**Width**] (in columns) for the query button or use the spinner control. (For this example, we are selecting the !File_open image and entering 3.00 for the width.)



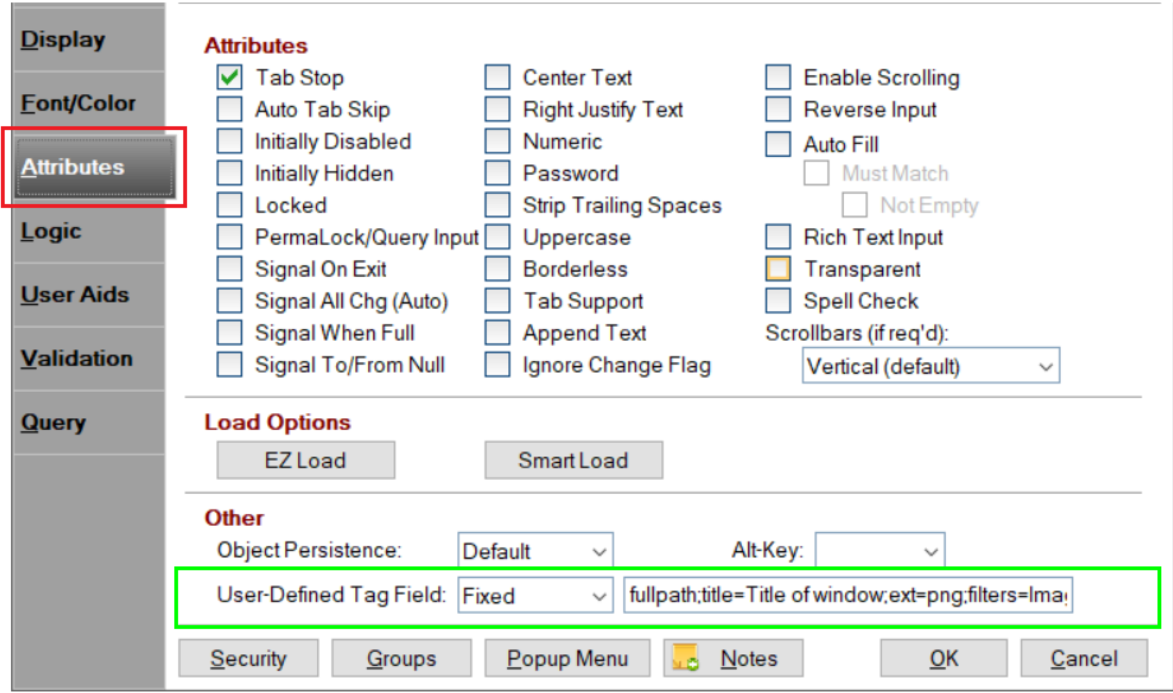
Save and test the panel. We see that the control has a small icon to its right. Clicking on it should display a file selection window similar to the following for selecting or entering a file name to open:



Note: The **MULTI_LINE** control ***must have a height of exactly 1*** for the Query button to display; otherwise, NOMADS will display a message if this is not the case and ask to make the adjustment.

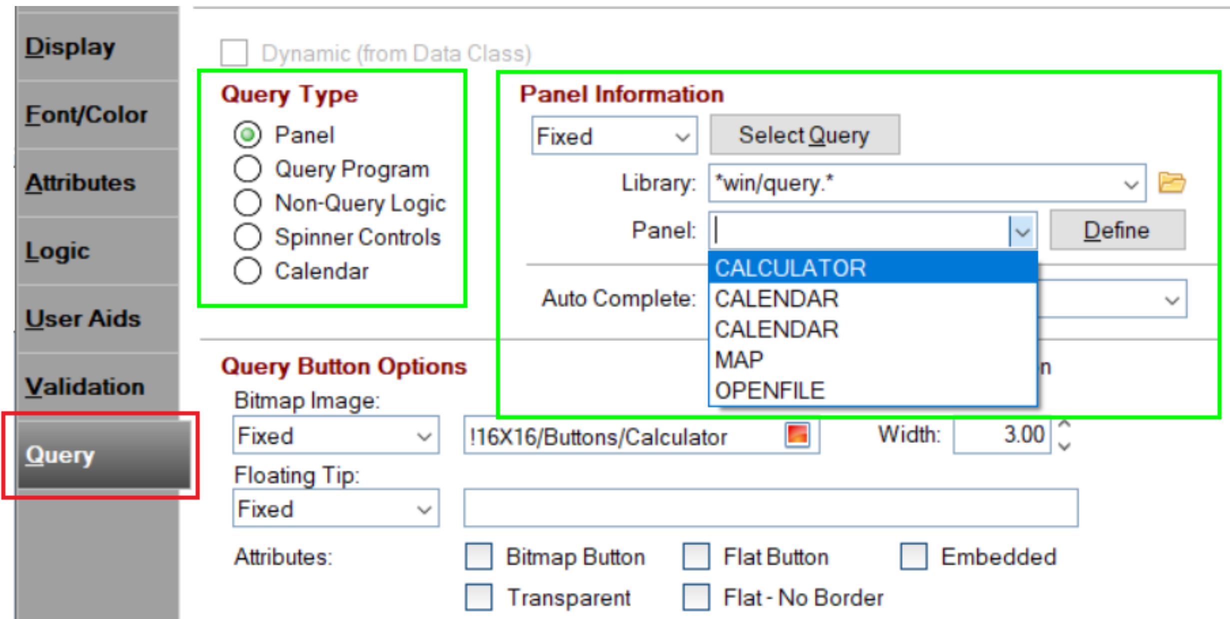
We can also specify the title and filter the type of files we want to open. To do this, we must go to the control's properties window and enter something similar to the following in the [**User-Defined Tag Field**] option in the **Attributes** tab:

fullpath;title=Title of window;ext=png;filters=Images|*.png\;*.bmp\;*.jpg\;*.jpeg,Text|*.txt\;*.log,All Files|*.*,

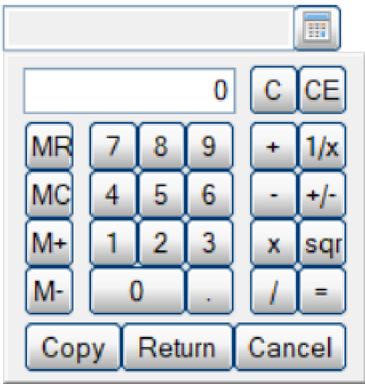


CALCULATOR Query

The Calculator type control is very simple. Go to the **Query** tab of the **MULTI_LINE** control properties. From the [**Panel**] drop-down list, select **CALCULATOR**. Under **Query Button Options**, specify a different image for the calculator.

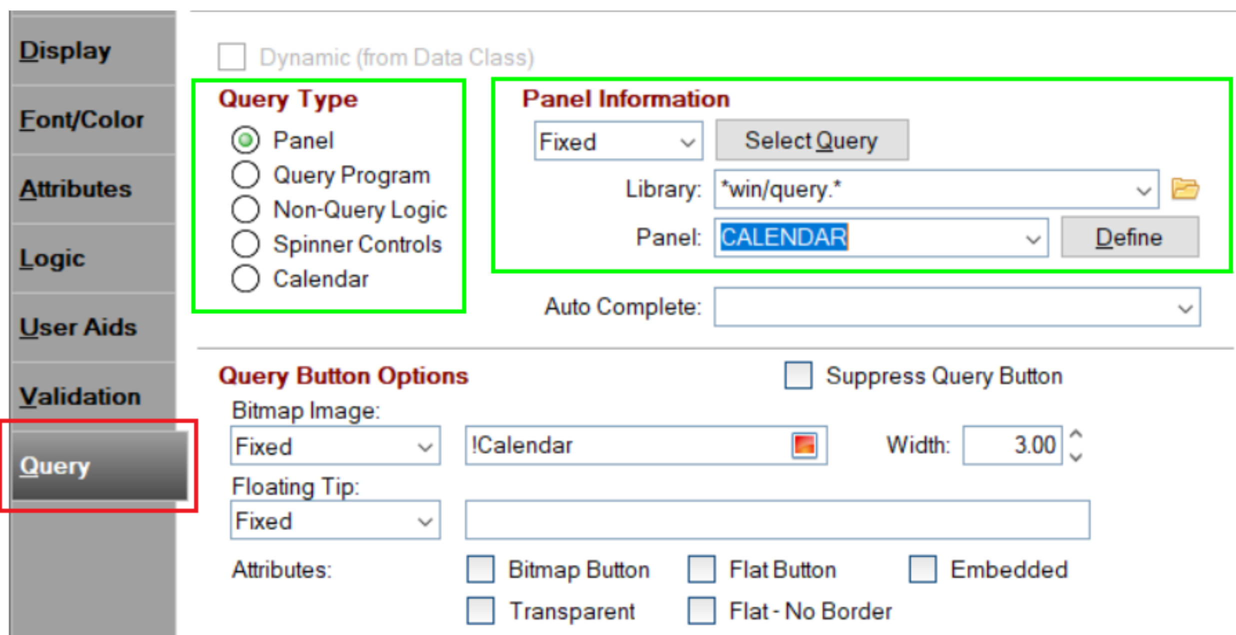


Save and test the panel. Clicking the icon to the right of the control should bring up a calculator:

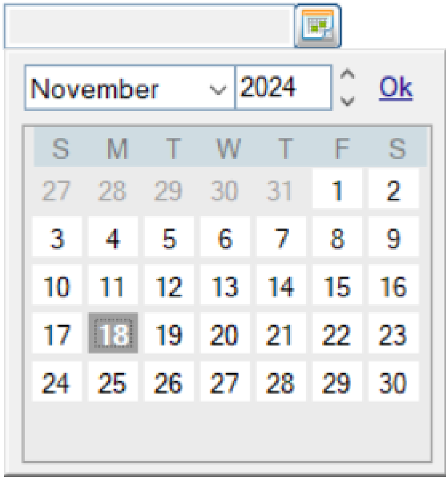


CALENDAR Query

We also have another type of calendar. In the **Query** tab of the **Multi-Line Properties** window, select **CALENDAR** from the [**Panel**] drop-down list. Specify a different image for the calendar.



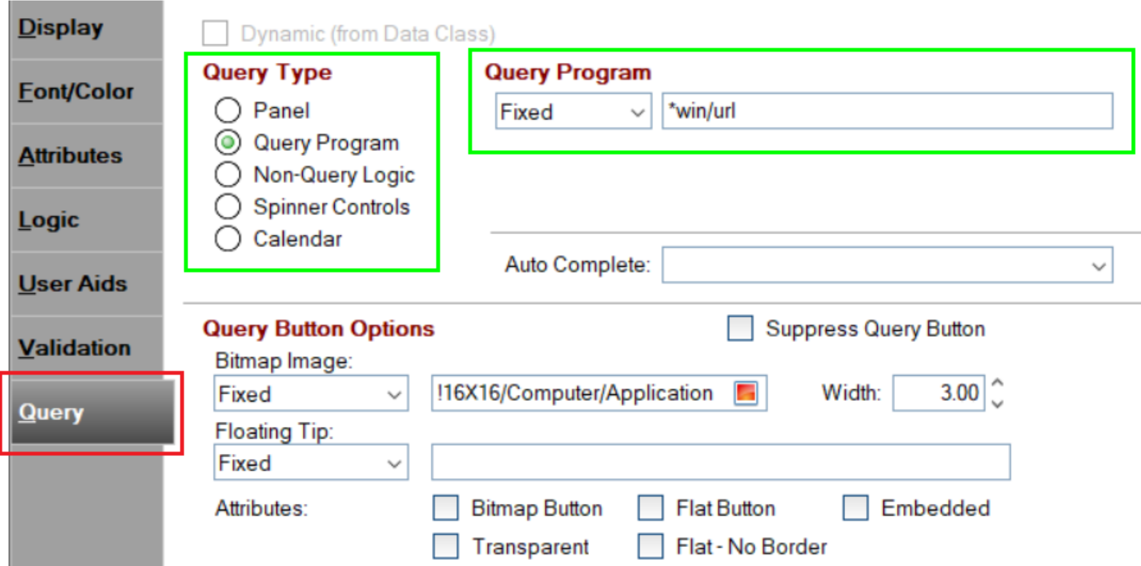
Save and test the panel. Clicking the icon beside the control displays the following calendar:



Web Query

The procedure to query a Web page or URL address is very similar.

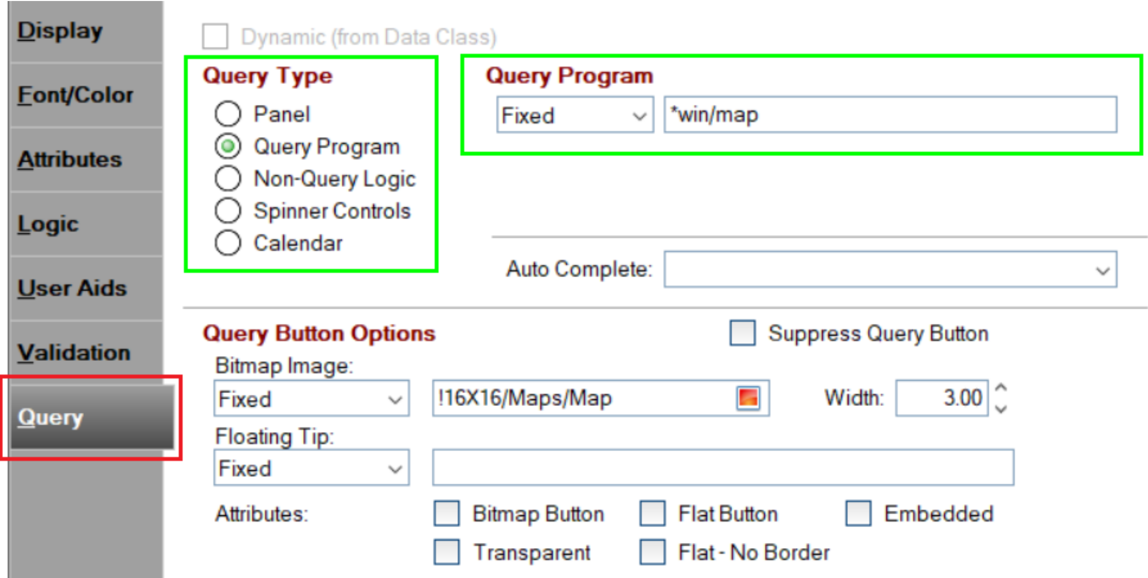
Open the properties for a **MULTI_LINE** control and select the **Query** tab. For the [**Query Type**],select **Query Program**, and then in the input box to the right under **Query Program**, enter the following program: ***win/url**.



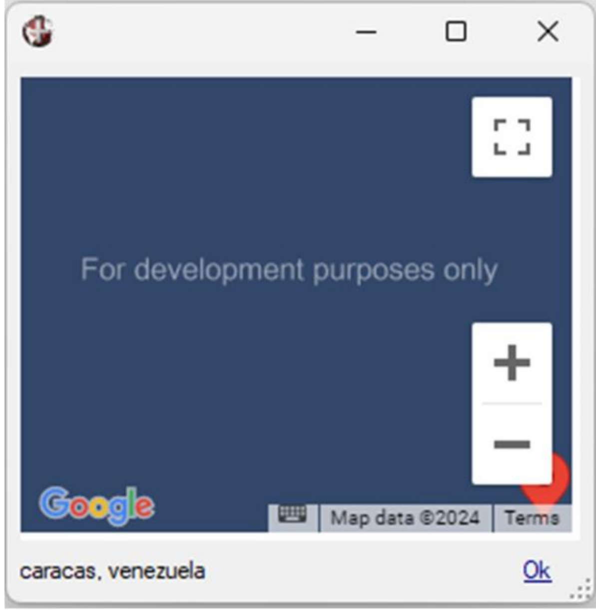
After saving the panel and running it, we can enter a URL address (such as **www.pvxplus.com**) in the control and then click the button associated with the query. The address will appear in a browser window by default.

Map Query

To open a Map query type, the procedure is similar, but for the [**Query Type**],select **Query Program**, and then in the input box to the right under **Query Program**, enter the following program: ***win/map.**



Note: If you do not have a Google API for using maps, the map may appear with a watermark.



For more information on these Query types, refer to **Service Queries for Web, Email, Map and Open File** in the PxPlus Help documentation.

What are Link Files?

One of the least known features of PxPlus, although it is one of the most brilliant solutions, is the **Link File**. In short, a ***Link File*** is a file that allows you to create a different name for a file or device and optionally run a program. Let's look at this in more detail.

A Link File is like an alias. This alias allows you to replace a Print command such as "**lp -s -d print_management 2>/dev/null**" and instead refer to it as "**LP**". Or perhaps, we need a log-type record to start when some activity is carried out with a particular file. It would also serve to hide some important file from the eyes of programmers. Or, depending on who is going to print; send it to one printer or another, automatically, transparently and independently of the operating system and even the printers themselves!

Yes, we can do all that and much more with Link Files, such as ease the opening of a Database, which asks for validations, connecting to a server, passing credentials, etc. We can define a Link File that makes all this work easier.

How does a Link File work?

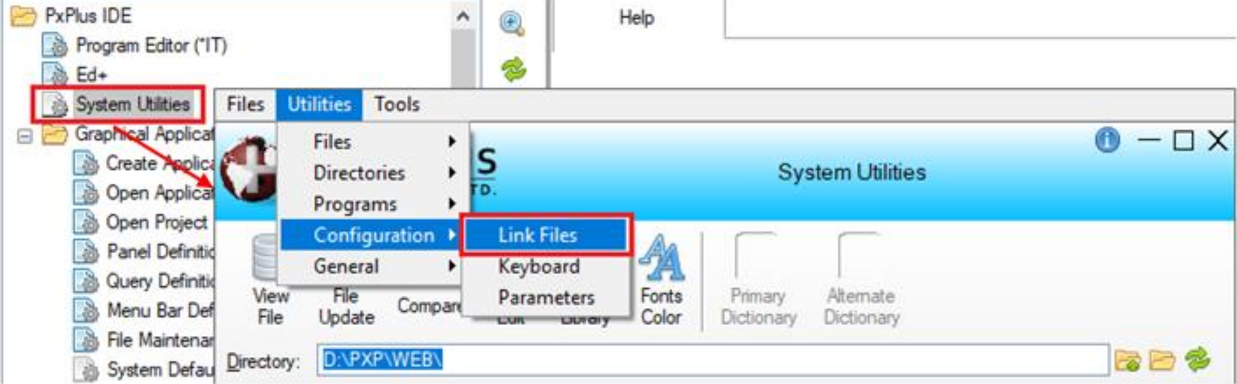
Basically, a Link File has three parts: a name, a real or alternate name and a program (normally known as a "driver"). Let's take a look at an example.

Note: There are different types of Link Files: **File**, **Printer** and **Device**. The difference is that the File type does not have an associated program (driver). The Printer type, in addition to executing the program, will send a special sequence of commands. The Device type only executes the program.

If a Link File is used, the PxPlus detects that it is a link, closes the Link File, opens the real name, and if it has a program associated with it, executes it. With this, it is possible to "hide" a file name, since when the programmer sees the name used in the programs, it will be the name of the Link File; but when executed and PxPlus determines that it is a Link File, it will close the file and open the alternate or real name.

To define a **Link File**, go to the PxPlus IDE main menu and select [**System Utilities**].

When the **System Utilities** window displays, in the top menu bar, select [**Utilities**] -> [**Configuration**] -> [**Link Files**].



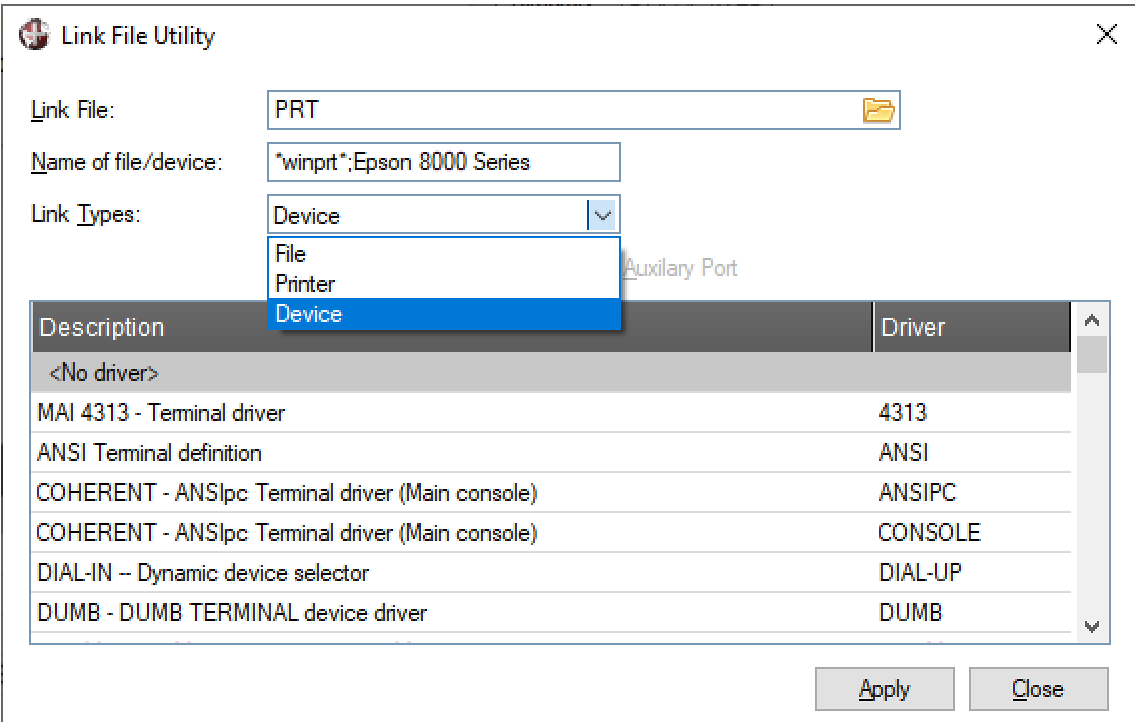
This opens the **Link File Utility**. Let's create an example of a Link File.

Exercise: Creating a Link File

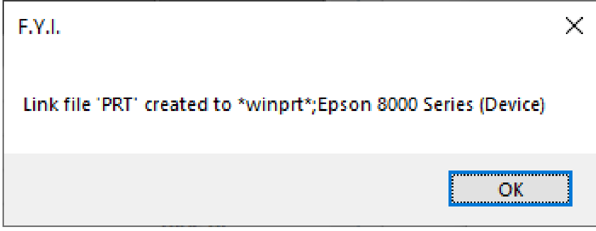
To print on the department printer, we must do it through the logical device "***winprt*; Epson 8000 Series**", but the installed system only accepts two or three characters as a printer identifier.

For the name of the Link File, enter **PRT**. For the name of the file/device, enter ***winprt*;Epson 8000 Series**.

As it is a printer, we can select the Link File type as **Device** or **Printer** to be able to associate a driver or program. For a program to be considered a "driver", it must be in the ***dev** directory of PxPlus. Later, we will see some examples.



Click the **Apply** button. A message displays to confirm that the Link File was created.



By simply providing a Link File name, a file or device name and (optionally) selecting a driver, we have a Link File.

When we associate a program to a Link File, what we know as a "driver" is a program to condition the

printer to configure certain mnemonics and make them compatible with that printer model, just as this printer driver does.

Epson printers (included in PxPlus):

```
0010  ! EPSON generic printer - 10 cpi
0020  DEFPRT (LFO)132.66
0030  MNEMONIC (LFO)'FF'=$0C$ ! <formfeed>
0040  MNEMONIC (LFO)'CR'=$0D$ ! <cr>
0050  MNEMONIC (LFO)'LF'=$0D0A$ ! <cr><lf>
0060  MNEMONIC (LFO)'RM'=$1214$:132.66 ! Reset condensed/expanded
0070  MNEMONIC (LFO)'EP'=$0E$:80.66 ! Expanded print
0080  MNEMONIC (LFO)'CP'=$0F$:212.66 ! Condensed print
0090  MNEMONIC (LFO)'SP'=$1214$:132.66
0100  LET X$=MNM('PS',0); IF X$<>"" THEN MNEMONIC (LFO)'PS'=X$ ! Start Slave
0110  LET X$=MNM('PE',0); IF X$<>"" THEN MNEMONIC (LFO)'PE'=X$ ! End Slave
0120  LET X$=FIB(LFO); IF X$(19,1)="S" THEN LOCK (LFO,ERR=*NEXT)
0130  PRINT (LFO)'RM',
0140  END
```

Notice the use of the **LFO** (Last File Opened) variable. This variable contains the number of the last channel that was opened, which is the one we use to open the Link File. A series of mnemonics are defined to send the commands necessary for the printer to execute a series of functions.

But in reality, **Link Files** go much further than that. We can make a Link File so that, depending on the user, it opens a particular file.

Example:

```
10 ! Change file depending on user
20 ch=LFO
30 close (channel)
40 file$=""
50 if who="Miguel" then file$="/usr/PAYROLL"
60 if who="Andres" then file$="/usr/BANKS"
70 if file$="" then file$="/usr/COUNTABLE"
80 open (channel,iol=*)file$
90 exit
```

This "driver" closes the open device in LFO (the Link File that the user opened), initializes a file$ variable and, using the WHO system variable (which contains the user name), will assign a different file to several users, despite that, in the program listing, they will all have the same name (the name of the Link File).

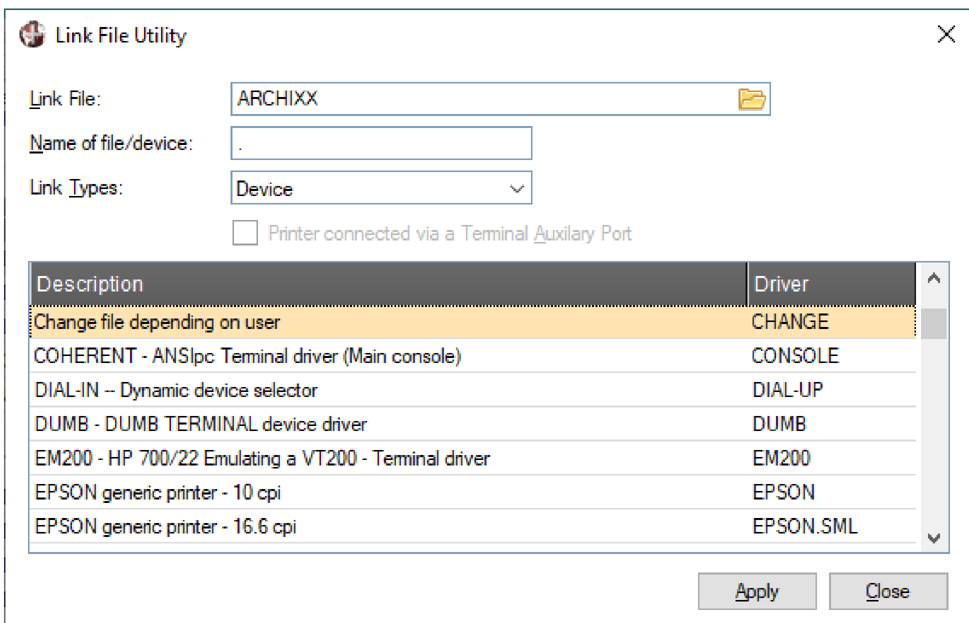
How is this done?

The first thing we must do is create and save the above program with the name "CHANGE" in the ***dev** folder of PxPlus.

Then, we must run the **Link File Utility** to define the Link File:

Link File Name: ARCHIXX
File/Device Name: **.** (*a period*)
For the Driver, select the CHANGE driver

The **Link File Utility** window looks like this:



Click the [**Apply**] button, and it will be recorded and ready for our **Link File** to work. In case of an adjustment, we may need to edit the ***dev/CHANGE** program. There will be no need to recreate the Link File.

Note: Why use "**.**" (a period) as the file/device name? For PxPlus, "**.**" (a period) refers to the current directory (where we are) and, as soon as the "driver" (in our case, CHANGE) is executed, will choose to open the file that corresponds to us as the user; so, we use that as a wild card.

We can enter any program name for the "driver". The requirement is that it be located in the ***dev** directory.

Link Files are used for many other things, such as making it easier to open and use a database, perform some checks before using a device, and much more.

Embedded Procedures (Inside Tables)

One of the advantages of databases over tables is the possibility of embedding (storing) a procedure within the same database known as a **Stored Procedure**, which is a program that is inside the database and is executed from it many times without user intervention.

Another very powerful database tool is **Triggers**, which are procedures that are executed automatically when a certain condition is met.

In PxPlus, there is a tool that offers the same functionality transparently and is very easy to implement without having to alter your programs to do so.

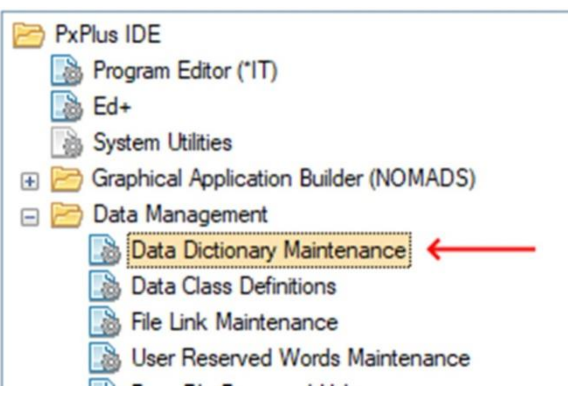
This is known as **Embedded IO** or embedded program. It is a program associated with a table, and it will be executed automatically when accessing it.

How does it work?

When a table is opened, PxPlus checks its header. If it has an embedded procedure, it executes a **PERFORM** command from it. Optionally, it is possible to place several labels to "catch" different actions on the table.

PxPlus allows, instead of a program, to declare an object to handle the different events, but each of them will be a method defined inside the object.

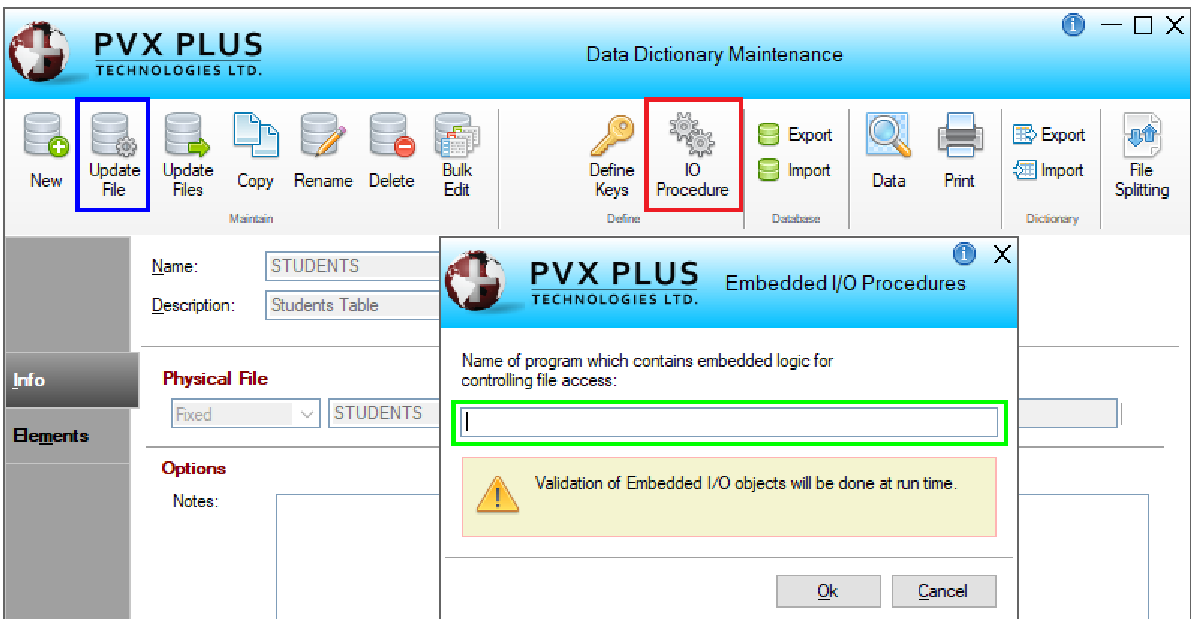
To define or embed a program or object within a table, we must go to **Data Dictionary Maintenance**. From the PxPlus IDE main menu, open the **Data Management** category and select **Data Dictionary Maintenance**:



In **Data Dictionary Maintenance**, select (or create) the table to which we need to embed the procedure, and then select the [**IO Procedure**] option (marked in red).

The **Embedded I/O Procedures** window displays. Enter the name of the PxPlus program or object in the box (marked in green) and then click the [**OK**] button.

Then, click the [**Update File**] button (marked in blue) so that it registers the changes.



Note: The specified procedure (program or PxPlus object) ***must exist***, both to be able to embed it and to use the table.

Important Note: It is necessary that the program to be embedded exists and can be executed without problems; otherwise, its table file could become unusable.

When you write a program or an object to be used as an embedded procedure, ***you must be very careful*** with the manipulation of the list of variables, so it is suggested that you declare the variables as **LOCAL**.

There are a number of possible events that can be captured in an embedded procedure (**Example: PRE_CLOSE**, **PRE_READ**, **POST_READ**, and many others). The operation is very simple. When the table is opened, PxPlus does a **PERFORM** of the embedded procedure and will execute the instructions in the regular order.

But, you could create special routines to capture, for example, a **READ**, if, in your program, you define a tag **PRE_READ** before executing a **READ** operation on the table. PxPlus will execute (if it exists, of course) the routine labeled **PRE_READ**. You can do the same for many other actions that can be executed with that table.

Example: Suppose we have a table, MYTABLE, that has an embedded procedure called MYTABLE_AUTO. Let's see what happens with some examples.

Let's assume the content of the embedded procedure is like this:

```
! MYTABLE_AUTO: Automation for Table MYTABLE
!
! When the OPEN() command is done, the start of the program is executed.
! procedure, no label necessary
!
! File access time is recordedchannel=unt

open(channel,iol=*)"review.log"
write(channel)dte(0:"%Dz:%Mz:%Yl %Hz:%mz:%sz"),who," Opened table: MYTABLE"
close(channel)
exit
!
! Before close the table, write the time
PRE_CLOSE:
channel=unt
open(channel,iol=*)"review.log"
write(channel)dte(0:"%Dz:%Mz:%Yl %Hz:%mz:%sz"),who," Closed table: MYTABLE"
close(channel)
exit
!
! BEFORE write a record
!
PRE_WRITE:
ENTER MODE,KEY$,INDEX,VALUE$
channel=unt
open(channel,iol=*)"review.log"
write(channel)dte(0:"%Dz:%Mz:%Yl %Hz:%mz:%sz"),who," Modified table: MYTABLE"
write(channel)"Previous Value: "+value$
close(channel)
exit
! AFTER write a record
!
POST_WRITE:
ENTER MODE,KEY$,INDEX,VALUE$
channel=unt
open(channel,iol=*)"review.log"
write(channel)dte(0:"%Dz:%Mz:%Yl %Hz:%mz:%sz"),who," Modified table: MYTABLE"
write(channel)"New Value: "+value$
close(channel)
exit
```

This program is quite trivial but is enough to understand how the embedded procedures work. This embedded procedure records four events of the handling of the table MYTABLE as follows: when it is opened, when it is closed, before and after writing a record.

Note: ***You do not need to do anything else!*** The embedded procedure will be executed automatically every time someone, anywhere, does any operation with that table!

Important Note: ***Don't delete the embedded procedure if you don't need it anymore!*** First, go to **Data Dictionary Maintenance** and unlink (delete) it within the table. Remember to click the [**Update File**] button afterwards. Then, you can delete the program or object.

The main events you can handle in an embedded procedure are:

COMMANDS Events	**FUNCTIONS Events**
PRE_READ	PRE_KEY
POST_READ	PRE_KEF
PRE_EXTRACT	PRE_KEL
POST_EXTRACT	PRE_KEP
PRE_WRITE	PRE_KEC
POST_WRITE	PRE_KEN
PRE_REMOVE	PRE_IND
POST_REMOVE	PRE_RNO
PRE_PURGE	
POST_PURGE	
PRE_LOCK	
POST_LOCK	
PRE_UNLOCK	
POST_UNLOCK	
PRE_CLOSE	
POST_PASSWORD	

START_UP: Automatic Execution Program

In PxPlus, there are several ways to tell it to run a program: at start up, from the Command line, as an access parameter, etc. It is possible to do it automatically in all instances. Each time PxPlus is executed, a series of instructions can be specified for its execution (**Example:** to prepare the work environment, modify search paths (PREFIX), change or prepare parameters, open global files, define global functions, instantiate objects, etc.).

The way to do this is by creating (in the PxPlus ***Start directory***) a program called **START_UP**. This is called internally by a PxPlus program (which runs first) called ***start.up**. It is possible to change the default name of the START_UP program by using the **PVXSTART** environment variable. (**Note:** This is an ***operating system*** environment variable; it is ***not*** a PxPlus variable!)

You can also use this variable to define an error handling program or change the terminal identification, as well as settings for colors, date format, decimal and thousand separators, etc.

PxPlus Error Handling

No matter how much we take care and validate user input, there will always be the possibility that an error will be generated. An error is not always synonymous with a malfunction. It can simply be an operation in the wrong order, incorrectly entered data, or a device that is turned off or offline. Regardless of what caused it, it is not appropriate to let errors (for whatever reason) prevent the correct functioning of the system.

An error can simply be a warning that a step was not completed or that the expected information is not in the proper format. In PxPlus, errors are in the form of numeric codes from error 0 to 255. Errors greater than that number usually indicate an error generated outside of PxPlus, an operating system error. In order not to confuse them with language errors, 256 is added to the original code of the operating system (**Example:** Error 260 is an operating system error 4.)

Refer to **Error Codes and Messages** in the PxPlus Help documentation.

Regarding the generation of errors, as we already said, it is unpredictable, and what we must do is minimize their appearance and handle them in the best possible way. A fundamental cause of errors is the misuse of data. An entry that is well validated or that is filtered will be much less prone to generating errors than an open entry. Validations are a very efficient way to minimize erroneous entries.

PxPlus offers the capture of errors at three levels: at the instruction level, at the program level and at the general level. The capture or handling of errors at the instruction level is normally in the form of the **ERR=label** clause. This can be interpreted as: if an error occurs when applying/calculating the function, take the flow to the "label" routine.

Examples of this are:

```
! Calculate x LOGarithm, if error go to ROUTINE_ERROR
A=LOG(X,ERR=ROUTINE_ERROR)
! Square root of VALUE, if error go to ROUTINE_ERROR
ROOT=SQR(VALUE,ERR=ROUTINE_ERROR)
! Open "FILE", if error go to ROUTINE_ERROR
OPEN (9,ERR=ROUTINE_ERROR)"FILE"
```

There are specialized functions for handling some errors, which are used as clauses in some input and output instructions, such as the **,DOM=** and **,END=** clauses. Sometimes, it is preferable to use these clauses to handle these special conditions and not leave them in the hands of the more generic **,ERR=** because the error could be something else and we may not notice it.

It is also possible to use several of these clauses in the same instruction:

```
READ (channel,err=routine_error,end=end_file)
```

At a second level, we can define a "capture" of the errors that occur in a program using the **SETERR** command:

```
40 ! Error handling routine
50 SETERR ROUTINE_ERROR:
60 !
70 ! Continue the program execution
900 ROUTINE_ERROR:
910 ! Routine to trap the error
```

Note: It is perfectly valid (and actually used quite a bit) to have both levels of error handling; that is, some errors handled at the instruction/function level and also have an error handling routine in the program. Errors generated in the instructions will be routed to the "specific" routines, and if an error occurs in another part of the program, it will be handled by the general routine.

The detail of having a routine per program is that, ideally, it should be in all the programs. This would imply modifying all the programs to incorporate it, and if there is any modification in the error routine, it would be copied as many times as programs exist. This is far from an "ideal" solution.

To avoid the previous case, PxPlus offers a general error handler through the instruction **ERROR_HANDLER**. The syntax would be:

ERROR_HANDLER "PROGRAM"

You could also define just one routine in a larger program:

```
ERROR_HANDLER "PROGRAM;ROUTINE_ERROR"
```

If we need to know what the error handling program is called, we can do:

```
ERROR_HANDLER READ PROGMAN$
IF PROGMAN$="" THEN MSGBOX "No error handler active!"; ESCAPE
```

The **ESCAPE** instruction is a program break, preserving all open channels and all memory structures so that the programmer can review what happened with his/her program.

Once an error handling program has been defined, the execution priority would be from lowest to highest; that is, if the command where the error occurs has an error clause, it branches towards it. If it does not exist and there is an error routine at the program level, it is sent there. If neither of the two exist and there is an error handling program, it would be activated.

There are several variables and functions that allow us to obtain more information about errors. The first system variable is **ERR** that tells us the numeric value of the last error that occurred. There is also a variable called **ERS** that stores the line number where the error occurred (if the program does not have a number of lines, it will use its physical lines).

Note: There are some commands (such as **START**, **BEGIN**, **CLEAR**, **END**, among others) that will clear the error code, as well as other information related to it.

The **ERR()** function allows us to have more information about the error. It has many parameters.

Example: ERR("LINE") to know the line where the error occurred, ERR("PROGRAM") to know the name or ERR("LASTPATH") to know the name of the last file that was accessed.

Note: PxPlus offers a "template" error handling program called ***ERROR**, which could be used as an example to develop your own.

To learn more about error handling, refer to the section **Fixing Errors, Debugging and Testing Programs** within this book.

NOMADS: Using Embedded Panels

The NOMADS **Embedded Panel** feature allows the content of a different panel to be placed in a location within the current panel. A source panel object typically contains a common set of controls intended to be reused to be embedded in other panels throughout the application. **Example:** You might have a common format for a set of First, Previous, Next, and Last navigation buttons. Instead of recreating these controls in multiple panels, you can create them once in one panel and then embed the content of that panel in all the panels that need them.

The main advantage of this functionality is that you only need to edit the source panel so that the changes are reflected in all the panels where those controls are embedded.

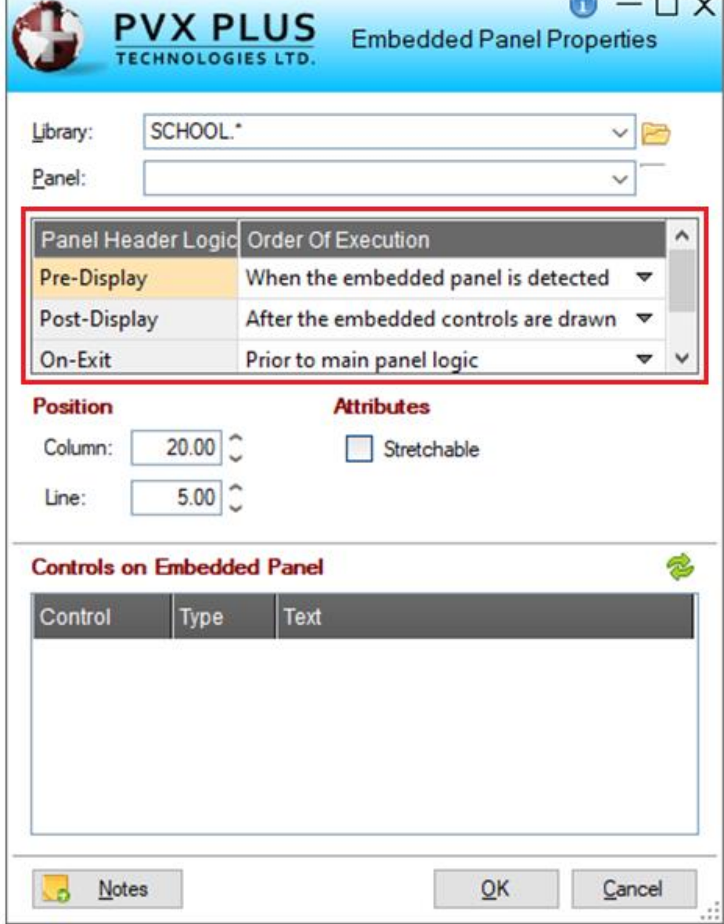
General Concepts of Embedded Panels

At run time, all controls in the source panel will appear in the current panel. The embedded panel can contain any of the control types available in the **NOMADS Panel Designer** (buttons, multi-lines, drop-down menus, etc.).

Embedded panels can be moved within the container panel but cannot be resized. Multiple embedded panels can be assigned to one panel.

The header **Logic** events (**Pre-Display**, **Post-Display**, **On-Exit**), along with the **Default Program** assignment in the embedded panel, can be inherited in the container panel. The **Default Program** in the container panel is used only if it does not exist in the embedded panel.

The execution order for the embedded panel is defined in the **Embedded Panel Properties**:



Note: It is convenient (although not mandatory) to assign "special" names to the panel controls to be embedded to avoid possible duplicate names (**Example:** by prefixing each name with the prefix "IN_" to indicate that it is an embedded panel.) Therefore, when the panel is integrated with the container panel, there will be no problems with duplicate names.

Exercise: Creating an Embedded Panel

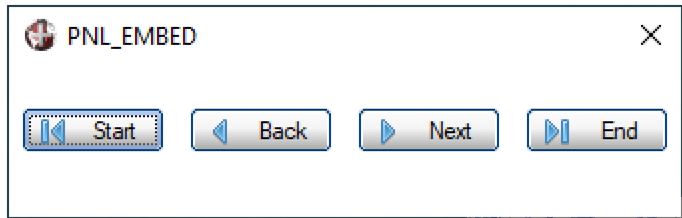
Let's look at a small example. We are going to return to the NOMADS panel designer and create a panel that will contain some action buttons (like the ones we would use to navigate a table).

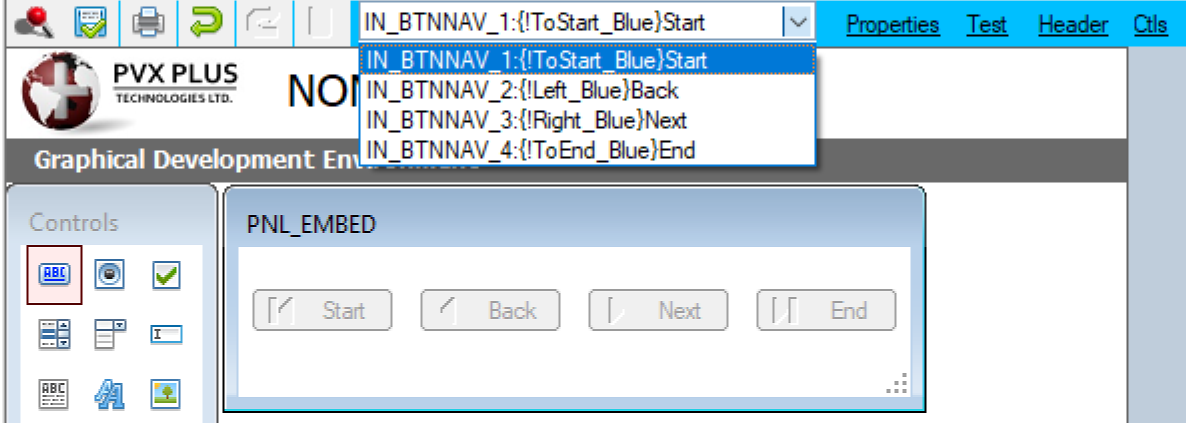
The panel will be called **PNL_EMBED** and will be defined with a size of 48 columns wide x 5 lines high. The buttons will have the following names and have images:

IN_BTNNAV_1: Start
IN_BTNNAV_2: Back
IN_BTNNAV_3: Next
IN_BTNNAV_4: End

Note: For the purposes of this exercise, ***the buttons will not have any action associated with them***. We will only see the process of embedding the panels.

The panel will look like the one shown below and have these characteristics:

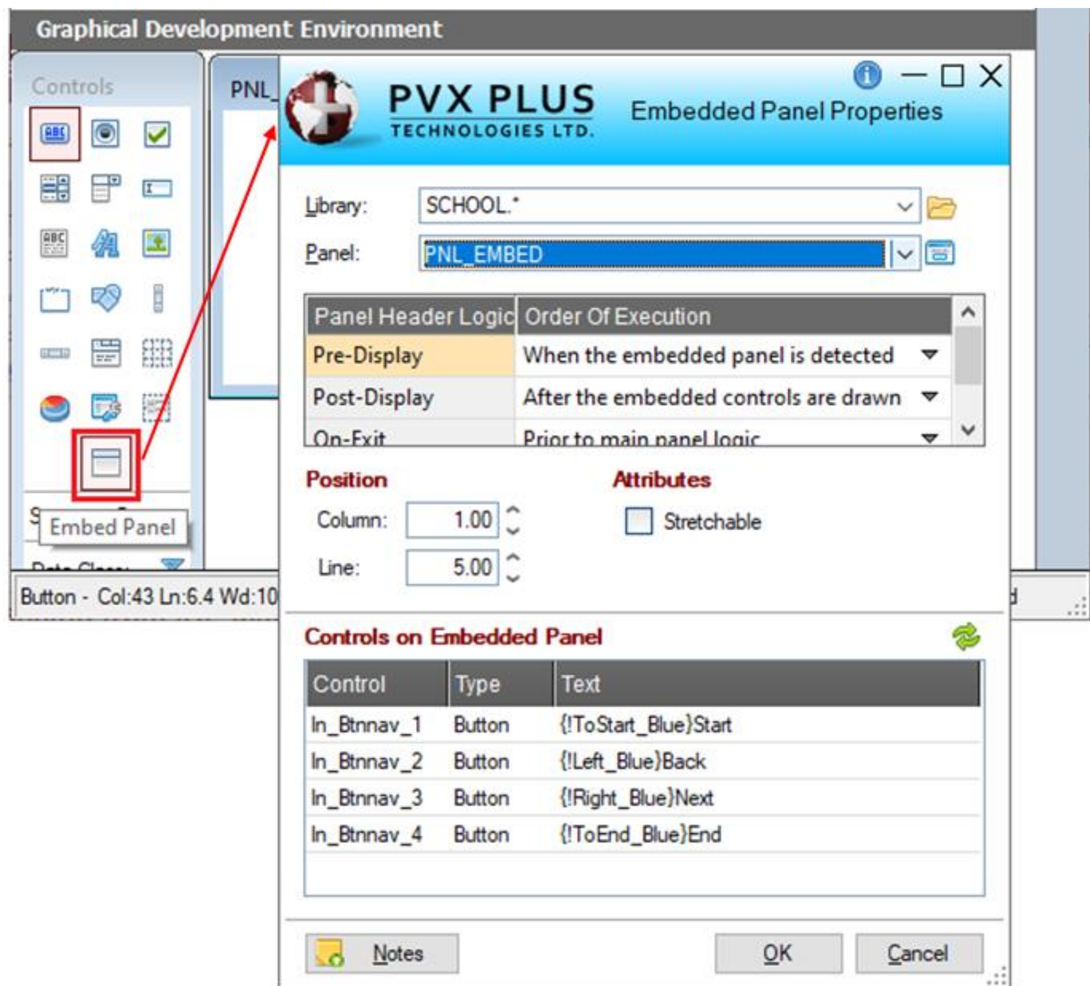




Note: Always remember to activate the [**Auto Refresh**] attribute so that the controls update their values in case they change.

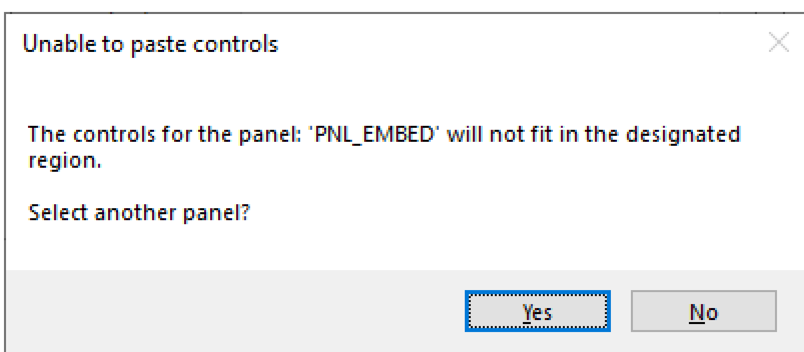
Once the **PNL_EMBED** panel is defined, we are going to create another panel so that we can embed the **PNL_EMBED** panel within it. The dimensions of the container panel must be large enough to contain the other panel without problems.

Define the new panel as **PNL_RECEIPT**, activate the [**Auto Refresh**] attribute and add an **Exit** button. Click the [**Embed Panel**] button in the **Controls** box on the left. This button allows us to embed a panel. We must draw an area large enough so that it can contain the panel controls to be embedded. Remember that the outer frame will not be embedded.

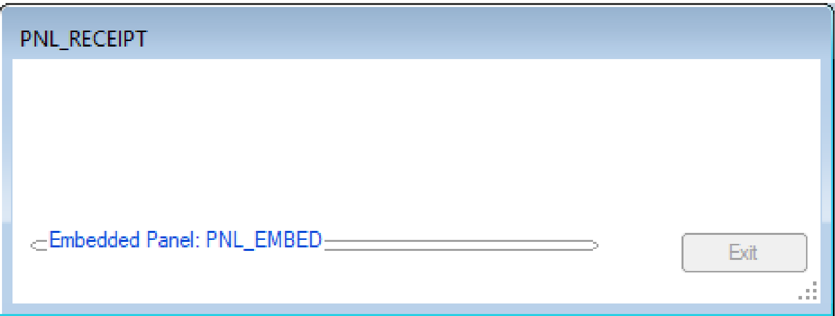


At the top of the **Embedded Panel Properties**, you can select the execution priority of actions associated with embedded panel events. If you have actions that are executed before and/or after displaying the embedded panel, you can choose what priority they will have in relation to the new container panel.

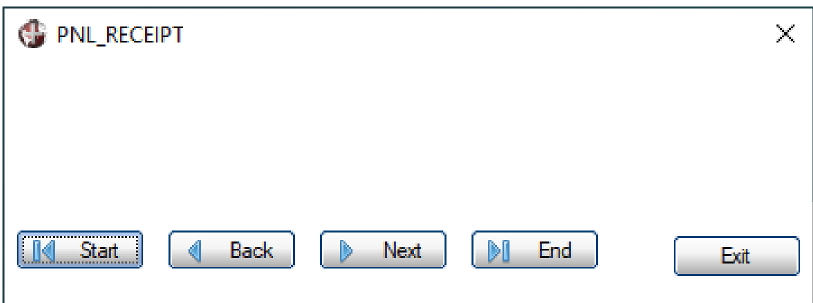
If the panel to be embedded does not fit within the specified dimensions, the system will give you a warning similar to the following and allow you to choose another panel:



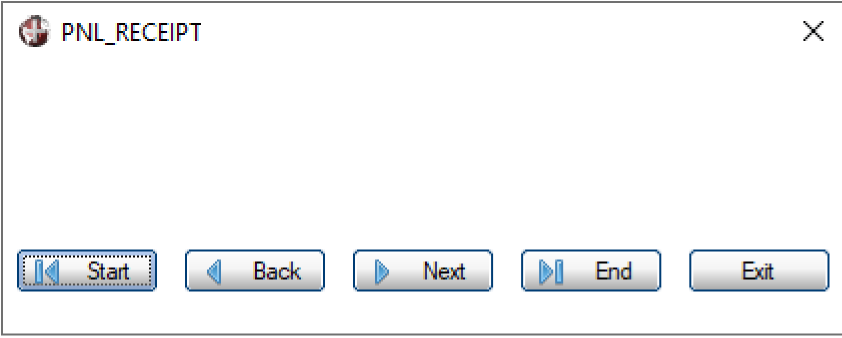
Once you have embedded the new panel, the container panel **PNL_RECEIPT** looks similar to the one shown below:



When you save and test this panel, it looks similar to the one shown below:



As you can see, we have a small misalignment with regards to the positioning of the embedded panel and the **Exit** button. To fix this, we simply relocate and resize the **Exit** button:



Now, our panel has a more pleasing appearance.

Refer to **Embedded Panels** in the PxPlus Help documentation.

The most important thing about this is that we are taking advantage of work done previously, and as we have previously mentioned, one of the advantages of PxPlus is that it allows us to do something once and reuse it many times.

Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming model or paradigm that is based on the concept of "objects", which can contain data in the form of attributes and functions also called methods. The four main laws of OOP are:

1. **Abstraction:** Allows real-world entities to be modeled as objects within the code. An object does not require anything else to function; it is "autonomous".
2. **Encapsulation:** The object will be an entity that groups data and methods that operate on external data in the same unit being able to limit access to certain components from the outside.
3. **Inheritance:** Allows you to define new classes based on previously defined classes, inheriting their attributes and methods. An object that is "born" from another object will inherit the functions and methods of the "parent".
4. **Polymorphism:** Allows objects of different classes that share a certain hierarchy to be treated uniformly, responding differently to the same message. The functions of the objects will be functional in different types of scenarios.

Very good! We have already completed the formal educational part. Now, what good is that for us? How do we take advantage of it?

Let's simply imagine that, in PxPlus, an object is a program that allows us to extend the operation of PxPlus or our application. It could be something as simple as giving us the time or the currency exchange rate or the number of days between two dates ... and it can be as complicated as MS Excel!

We are going to see the basic concepts to program an object, how to create properties and methods, and then how to use them. We will see that there is a lot of theory and that, with a little practice, you can begin to take advantage of it.

Creating an object in PxPlus is actually creating a program to define a "class" to which we can add properties and functions or methods. We can define an object (actually a "class" of object) that initially allows us to know the exchange of one currency for another; if we place a value in currency 1, it will return the exchange rate in currency 2.

We could figure out that solution with a simple calculation! But, as we have done with almost all the chapters of this book, we will go one step at a time! Then, we will see if there are things that would be very difficult to do without OOP.

The creation of an object starts with the creation of a "class", and this is simply a PxPlus program with a special construction:

```
DEF CLASS "class"
PROPERTY property
FUNCTION method(parameter)
END DEF
```

Once we have the "object" created (actually the "class"), to "execute" the object, we must also do it in a special way, since in reality, what we are doing is "instantiating" or creating a replica of the created class. That is to say:

For us, it is:	In OOP, it is:
Make a program	Define a class
Run a program	Instantiate a class/object

It seems as if they wanted to complicate things!! Let's leave the "classes" thing out and think that defining a class and programming are the same!

Exercise: Object-Oriented Programming

Let's start with a simple example: an object that provides an argument and has a method (function) that will return the exchange rate from the dollar to our currency. Suppose that the exchange rate is 36.5 and that the object does Amount*36.5 and returns that value to us.

```
DEF CLASS "DOLLAR"
FUNCTION CHANGE(AMOUNT)
ENTER AMOUNT
LET AMOUNT=AMOUNT*36.5
RETURN AMOUNT
END DEF
```

The first and last commands define the class (the object):

```
DEF CLASS "DOLLAR"
END DEF
```

Whatever is inside that should be a function (or method) or a property. Let's look at a function called CHANGE, which requires an argument called AMOUNT:

```
FUNCTION CHANGE(AMOUNT)
```

Next, as objects are self-contained entities, only values that are expressly declared so may be shared. For this, we use the command:

```
ENTER AMOUNT
```

This tells the DOLLAR class object that it will enter a value called AMOUNT, which is numeric. Then, you have to do the calculation and then use the RETURN directive or command to return this value to the program that invoked the object:

```
RETURN AMOUNT
```

We must save this program with the name "**DOLLAR.PVC**". This must be the extension, and (preferably) the name.

To "use" our object, what we must do is:

```
Object_identifier=new("CLASS")
```

Where:

CLASS will be the class (object) defined.

In this case, DOLLAR and the object are the same variable that we will use to refer to the object while

we are "using" it:

```
A=new("dollar")
```

To see what properties or methods that object has defined, we can do:

```
PRINT a'*
CHANGE(),
```

Note: We can also use the **OBJ** command to see the characteristics of object "A":

```
OBJ A'*
```

We have a new function in our environment called **A'CHANGE().** To use it:

```
PRINT a'change(10)
365
```

The idea of an object is that when it is "used" (actually instantiated), our program has those functions (methods) and those properties, which can be as trivial as this. But, we can see something more productive: an object called DATE that tells us the number of days elapsed from 01-01-2024 to the date we set. It tells us how many days the year lasts. It will do this through a function called NUMDAYS. It will also have a property that will say what day of the week is the date we entered.

Let's see how it works first:

```
OD=new("DATE")
PRINT OD'NUMDAYS("09052024")
129
PRINT OD'WEEKDAY$
Thursday
```

Note: Remember that you can use ***any unused numeric variable*** as the object identifier. We use OD for "Object Date". You can use whatever variable you want!

We are going to show the complete list, and then we are going to comment on the lines:

```
DEF CLASS "DATE"
FUNCTION NUMDAYS(DATE$)CALCULATE_NUM_DAYS
PROPERTY WEEKDAY$
END DEF

! NUMDAYS Function routine
CALCULATE_NUM_DAYS:
ENTER DATE$
IF LEN(DATE$)<>8 THEN GOTO DATE_ERROR
GOSUB PREPARE_DATE
INIDATE=JUL(2024,1,1)
NUM_DAYS=JULDATE-INIDATE
RETURN NUM_DAYS

! Date initial preparation
```

```
PREPARE_DATE:
YEAR=NUM(DATE$(5,4))
MONTH=NUM(DATE$(3,2))
DAY=NUM(DATE$(1,2))
JULDATE=JUL(YEAR,MONTH,DAY,ERR=DATE_ERROR)
WEEKDAY$=DTE(JULDATE:"%Dl")
RETURN

! Wrong date, return -1
DATE_ERROR:
NUM_DAYS=-1
RETURN NUM_DAYS
```

Let's analyze this, line by line:

```
! Define the class "DATE"
DEF CLASS "DATE"

! Define the NUMDAYS function with an argument (DATE$)
! The associated routine is CALCULATE_NUM_DAYS
FUNCTION NUMDAYS(DATE$) CALCULATE_NUM_DAYS

! Define a property WEEKDAY$
PROPERTY WEEKDAY$

! End the class DATE definition
END DEF

! Routine associated to function NUMDAYS
CALCULATE_NUM_DAYS:

! Enter the argument (with date)
ENTER DATE$

! If the length of date isn’t 8 digits, refuse it
IF LEN(DATE$)<>8 THEN GOTO DATE_ERROR

! Let's go to routine PREPARE_DATE
GOSUB PREPARE_DATE

! Calculate the JULian date for 01-01-2024
INIDATE=JUL(2024,1,1)

! Subtract both JULian dates
NUM_DAYS=JULDATE-INIDATE

! Return the result
RETURN NUM_DAYS
```

```
! Initial date preparation
PREPARE_DATE:

! Separate year and make it numeric
YEAR=NUM(DATE$(5,4))

! Separate month and make it numeric
MONTH=NUM(DATE$(3,2))

! Separate day and make it numeric
DAY=NUM(DATE$(1,2))

! Let's calculate the JULian date, if error go to DATE_ERROR
JULDATE=JUL(YEAR,MONTH,DAY,ERR=DATE_ERROR)

! Calculate the day of the week
WEEKDAY$=DTE(JULDATE:"%Dl")

! Return
RETURN

! Wrong date, return -1
DATE_ERROR:

! Assign -1 to NUM_DAYS
NUM_DAYS=-1

! Return the value
RETURN NUM_DAYS
```

Note: A JULIan date is the number of days elapsed from the given date to an origin date. In the case of PxPlus, that date is 01-01-1970, but you can change that date by using the **'BY'** parameter.

Important Note: Do you have questions about the management of dates in PxPlus? Refer to the commands **DAY_FORMAT** and **DEF DTE**, the **DAY** system variable, and the **JUL()** and **DTE()** functions in the PxPlus Help documentation.

A popular feature of objects is that it is possible to define methods or functions that use different numbers of arguments. In other words, the object can be defined by allowing the user to enter different numbers of arguments.

Example: If we have an object that handles dates and has a function or method to calculate the number of days between two dates, we could do this:

```
DateObj=new("date")
PRINT DateObj'date_dif(date1$, date2$)
```

In that example, the object would return the number of days between date1 and date2. But, we can also do:

```
DateObj=new("date")
PRINT DateObj'date_dif(date1$)
```

The object would return the number of days elapsed between date1 and the current date, for example.

We are going to include the complete list of the object, which includes some important aspects, including the one we just mentioned. Remember that an object must be a program that has the extension **".pvc"** and that begins with the class definition **DEF CLASS** and end with **END DEF**:

```
! DATE.PVC: OOP Demo, Date object
! ------------------------------------------------------------------------------------------------------------
! John Smith
! Dom 19/05/2024 - 08:48AM
!
DEF CLASS "DATE"
! Object properties
PROPERTY DATE$
PROPERTY WEEKDAYS$
PROPERTY WEEKDAYL$
! Methods or functions
!
! Number of days between 2 dates
FUNCTION DATE_DIF(DATE_1$,DATE_2$)
ENTER DATE_1$,DATE_2$
DATE_AUX$=DATE_1$; GOSUB VALID_DATE
!
JULDATE1=JUL(YEAR,MONTH,DAY,ERR=DATE_ERROR)
!
DATE_AUX$=DATE_2$; GOSUB VALID_DATE
!
JULDATE2=JUL(YEAR,MONTH,DAY,ERR=DATE_ERROR)
!
DIFFERENCE=JULDATE2-JULDATE1
RETURN DIFFERENCE
!
! Number of days between a given date and system's date
!
FUNCTION DATE_DIF(DATE_1$)
ENTER DATE_1$
DATE_AUX$=DATE_1$; GOSUB VALID_DATE
JULDATE0=JUL(0,0,0)
JULDATE3=JUL(YEAR,MONTH,DAY,ERR=DATE_ERROR)
DIFFERENCE=JULDATE0-JULDATE3
RETURN DIFFERENCE
!
END DEF
! Execute when the object is created
!
ON_CREATE:
DATE$=DTE(0:"%Dz/%Mz/%Yz")
```

```
WEEKDAYS$=DTE(0:"%Ds")
WEEKDAYL$=DTE(0:"%Dl")
RETURN
!
! Date validation
VALID_DATE:
IF LEN(DATE_AUX$)=4 THEN
     DATE_AUX$="0"+DATE_AUX$(1,1)+"0"+DATE_AUX$(2,1)+DATE_AUX$(3)
IF LEN(DATE_AUX$)=5 THEN
     DATE_AUX$="0"+DATE_AUX$(1,4)+"20"+DATE_AUX$(4)
IF LEN(DATE_AUX$)=6 THEN
     DATE_AUX$=DATE_AUX$(1,4)+"20"+DATE_AUX$(5)
IF LEN(DATE_AUX$)<>8 THEN GOTO DATE_ERROR
YEAR=NUM(DATE_AUX$(5,4),ERR=DATE_ERROR)
MONTH=NUM(DATE_AUX$(3,2),ERR=DATE_ERROR)
DAY=NUM(DATE_AUX$(1,2),ERR=DATE_ERROR)
RETURN
! Wrong date
DATE_ERROR:
RETURN -1
!
```

This object has a property DATE$, WEEKDAYS$ and WEEKDAYL$ that will have the current date and the day of the week in short (Sun) or long (Sunday) format, and has a method or function, DATE_DIF(), that accepts one or two arguments.

If it has one argument, the difference between the supplied date and the current date (system date) will be calculated.

In the case of two arguments, the difference in days between both dates will be calculated.

Each case has its own particular routine, and it seems that it is the same function declared twice (in reality, it is).

There is a date validation routine that accepts dates in 4, 5, 6 and 8 digit format (DMY, DDMY, DDMMYY and DDMMYYYY with D being the day, M being the month and Y being the year).

The automatic function of the object is used to have a routine that is executed automatically when creating or instantiating the object, **ON_CREATE**, which will be where we update the values of the properties that we are using.

Using the LIKE Command to Inherit Other Classes

Let's suppose we have a previously defined class or object where we have some functions and properties.

```
OBJ_OLD=new("colors")
print OBJ_OLD'*
letter_color$, background_color$, RGB_combination$, mix$()
```

Now, we want to create a new object (class) that incorporates that functionality, plus other things that we are going to add to it. To avoid duplicating the object, we can use the **LIKE** command, which makes the object we are defining "inherit" the properties and functions of the class in question, in addition to those we will define. We do it like this:

```
DEF CLASS "decoration"
LIKE "colors"
PROPERTY reason$
PROPERTY letter$
…
```

In the previous example, the "decoration" class will inherit all the properties and functions of the "colors" class: letter_color$, background_color$, RGB_combination$ and the mix$() method, in addition to the properties and functions that are defined at the time of creating the "decoration" class.

It is possible to specify more than one class:

```
LIKE "colors", "dates"
```

Note: We have talked about the automatic routine **ON_CREATE** that is executed automatically when a class is instantiated. In addition to that routine, we have the **ON_DELETE** routine that will also be executed automatically when closing the object (using the command **DROP OBJECT**).

When we start working with classes (objects), we will quickly learn that there are certain rules that it is better to follow to perform better and make the most of this powerful technology. **Example:** Use local variables inside the object and only return the value that is exchanged, and also the need to establish a link with the object (instantiate it) in order to use it. When we test objects, we must be aware that even if we make changes to the program (the "**.pvc**"), it is necessary to close the object and open it again so that the changes are recorded. By default, PxPlus will read and maintain an object and its properties in memory, and the changes made to the program (which defines its class, the "**.pvc**") are not incorporated automatically.

Note: Just as we can use objects developed in other programming environments, the objects that we make in PxPlus can also be used in other languages (**Example:** to make a C++ application read a PxPlus table or file). To do this, we must install and configure the PxPlus OLE Server (included for the Windows version). This will actually create an instance of a program (**pxpcom.exe**) that will be registered as a Windows service, registering it in the "Registry" of the operating system. Once you have this service registered, you can access PxPlus objects from other applications. You will also be able to make PxPlus as a scripting language, whose applications will be executed from the Command line or from the Windows console (such as PowerShell). For that, you must put the extension "**.pvs**".

For this, PxPlus has a library called **pxpscript.dll**.

This will allow us to make PxPlus programs that can be used in any application that is registered in the **MS Scripting** service (including MS Office applications, some browsers, and many other applications that make use of the **IactiveScript** and **IactiveScriptParse**).

Note: You can internally reference the object where it is located. Making use of the special variable **_OBJ**; that is, **_OBJ'FUNCTION$()**, will refer to a function that is found in the object (class) that you are modifying.

When we have an object that requires "exclusive" access from some device or that you do not want multiple instances running simultaneously, you can add the word "UNIQUE" to indicate that there can only be one active instance of that class or object at a time (per session and user, of course):

```
DEF CLASS "date" unique
```

In some circumstances (such as when an object is part of an "inheritance" within another object), the **ON_CREATE** and **ON_DELETE** routines will not be executed.

Example:

```
DEF CLASS "date"
LIKE "Calendar"
```

The **ON_CREATE** and **ON_DELETE** routines of the "Calendar" class will not be executed because they are part of a larger object. (**Note:** These routines will be executed when the "Calendar" class is instantiated by itself.)

To ensure that the **ON_CREATE** and **ON_DELETE** routines are executed under any circumstances, we must add the condition:

```
DEF CLASS "date" on_create required
```

The properties (variables) of objects have two possible events or conditions: when they will be read or when they will be written. We can force a routine for each of these events:

```
PROPERTY type$ GET routine_when_read
PROPERTY type$ SET routine_when_written
```

In the program that defines the class, you must have the routines "routine_when_it_is_read" and "routine_when_it_is_written", although it is also possible to specify a routine that is in an external program:

```
PROPERTY type$ GET "program;routine"
```

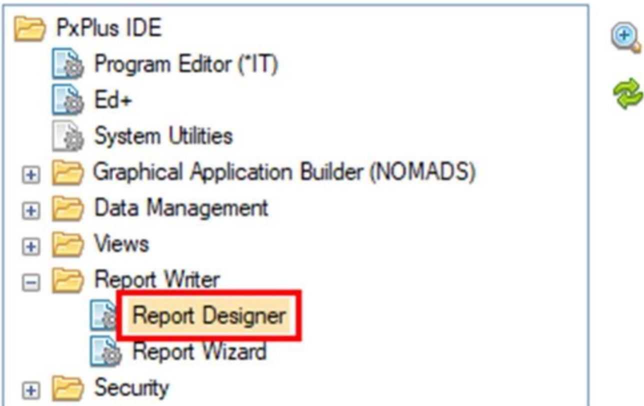
Object-oriented programming is a very powerful feature of PxPlus. Although simple objects are very simple to implement, you may encounter problems when trying to define objects embedded in objects or objects with dynamic properties or implementing any of the advanced functions of OOP. Our suggestion is that you analyze, review and practice the basics first. Go one step at a time, and then you can add more functionality to your classes. As with many other aspects of life, it will be practice and experience that will help you master this excellent tool in depth.

The Report Designer: An In-Depth View

We already saw that the Report Writer offers two ways to prepare or define a report. In a previous chapter, we learned about the Report Wizard. Now, we are going to see the Report Writer as a more complete tool.

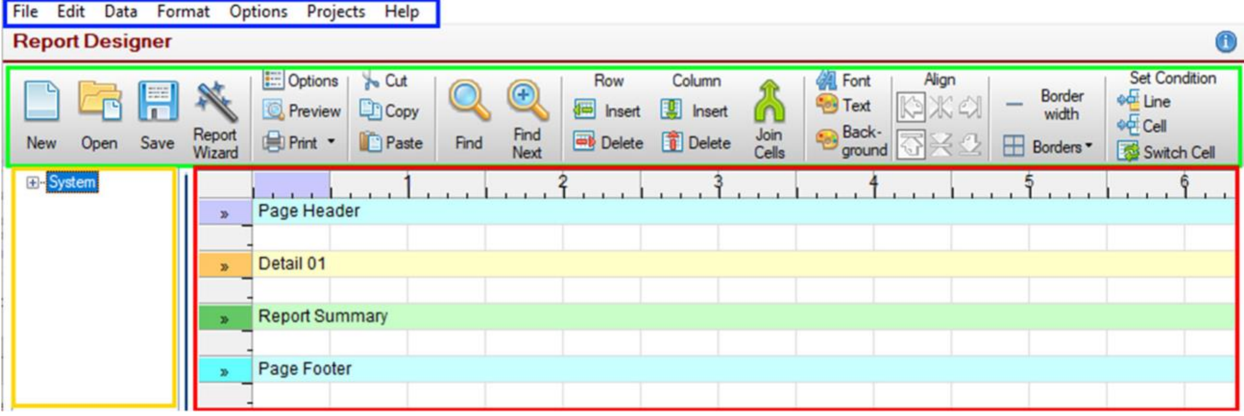
When we run the Report Writer, what we will do is define a structure or format and a data source or origin so that the user can then execute ("generate") a report based on that format and with the data extracted from the table (or any other selected data source).

The Report Designer can be run from the PxPlus IDE main menu. Open the **Report Writer** category and then select **Report Designer**:



It is also possible to run it from the NOMADS Session Manager (enter **nom** from the PxPlus Command line) by selecting [**Utilities**] -> [**Report Writer**] from the top menu bar. You can also run it by entering **RW** from the Command line.

When we execute it, the **Report Designer** window displays.

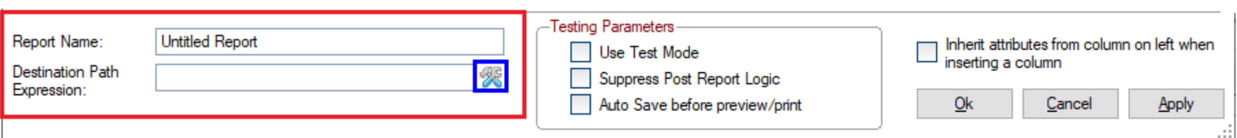


It contains four large areas: a top menu bar (marked in blue), a toolbar (marked in green), an area of variables or elements (marked in yellow) and the report layout area (marked in red).

The basic operation of the report is that you select some field or element from the elements panel on the left side (marked in yellow) and drag it to the report layout on the right side.

We can see that the report layout itself has four areas: a Page Header, a detail line (Detail 01), a summary area (Report Summary) and a Page Footer.

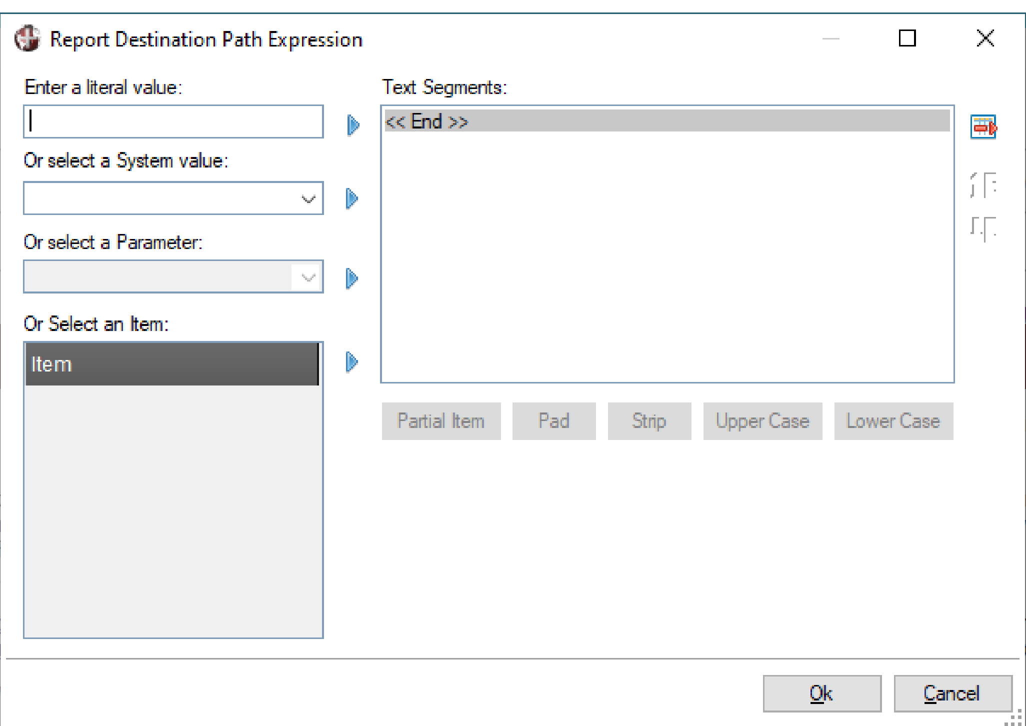
Additionally, at the bottom (marked in red below), there is an area for defining the name of the report and the destination path or folder. The name of the report will be basically that - an identifier for the report. The destination folder or path can be anything from a file name (in case, for example, we want to take the report to a .PDF file) to a dynamic structure that will change its name depending on the values provided.



We could say that the output folder will depend on the selected company and the date, such as "02APRIL" to indicate that the April reports from company "02" will be stored in that folder.

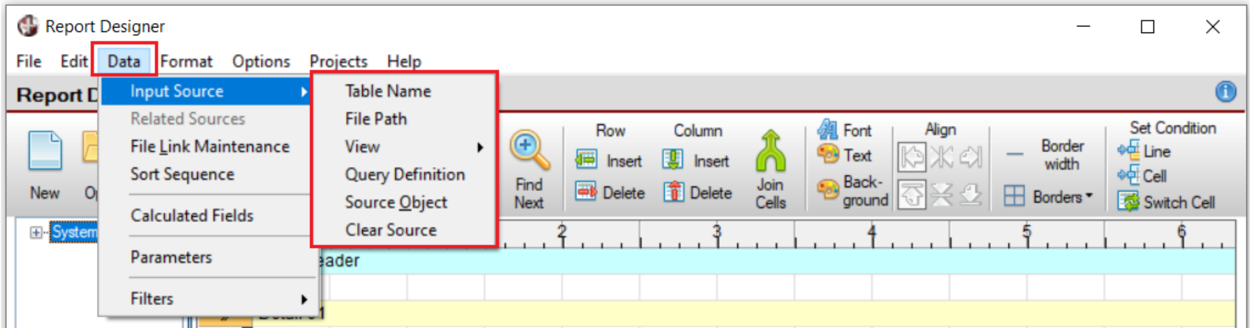
To complement the definition of the destination path, you can access the **Report Destination Path Expression** window by clicking the icon with the tools to the right of the [**Destination Path Expression**] field. This opens the window (shown below) where we can combine preset values with system variables, report elements and/or parameters entered by the user at the time of the report execution to give it the greatest versatility in defining the destination.

We can also simply enter the name of a file or folder.

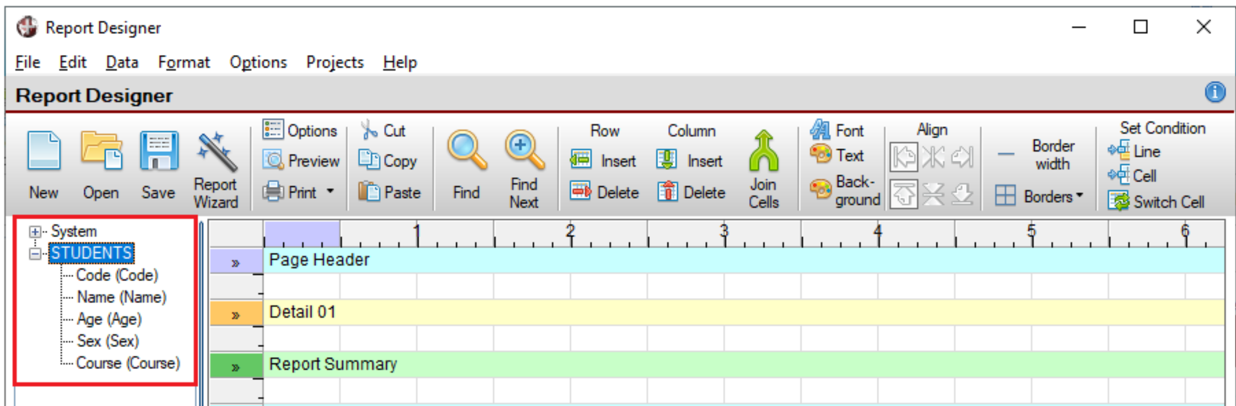


Note: The **Report Writer** allows reports to be designed grouped into one. **Example:** Suppose we have three sellers. We can produce a consolidated report and three individual reports, one for each seller.

The first part of defining our report is defining the origin of the data (where the report will be fed from to show the information). This is done by selecting the [**Data**] -> [**Input Source**] option from the top menu, and then, selecting the type of input, which can be a table or a file, a view, a query definition, a database, a program or an object:



Once the data source has been selected (in our case, we will use the **STUDENTS** table that we defined previously or you can use whatever you want), the elements or fields that make up that data source will appear on the left side. The idea is that you position the field that you want to incorporate into the report by selecting and dragging it to the position where you want it to appear. You can also select any of the predefined system parameters.

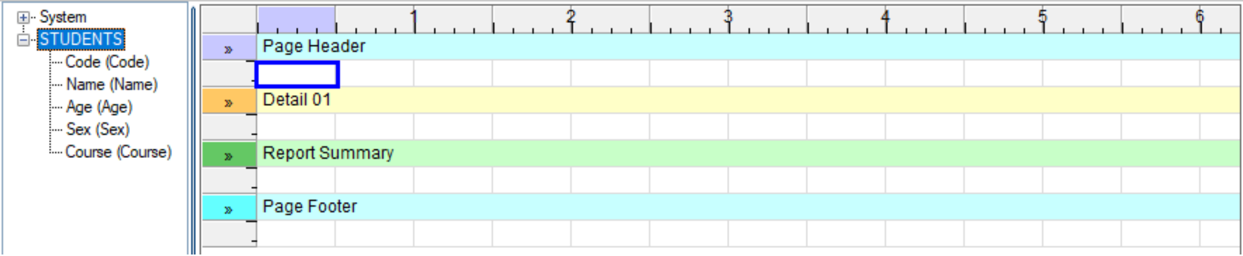


The elements of the System are basically date and time (in different formats), as well as the user and network node from which the report is executed.

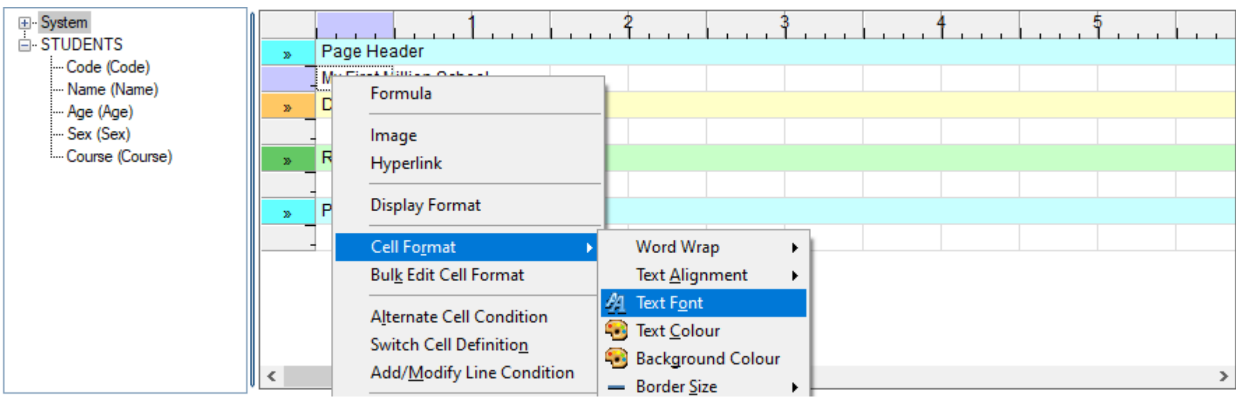
If the data source has other data sources associated with it (**Related Data Sources**), you can select it and then select which elements of the new source will be part of the elements to be displayed in the report. For now, we will take it one step at a time.

Exercise: Creating a Report with Report Designer

With a defined data source, we can now create our report. Locate the first column of the **Page Header** (marked in blue), click on it and enter the text **My First Million School**.

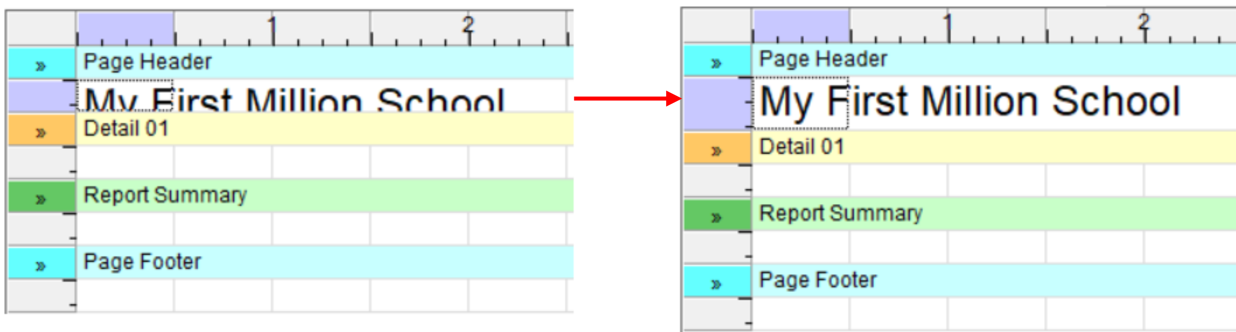


Change the font by right clicking on the cell where the text is located. Select the [**Cell Format**] option and then the [**Text Font**] option.

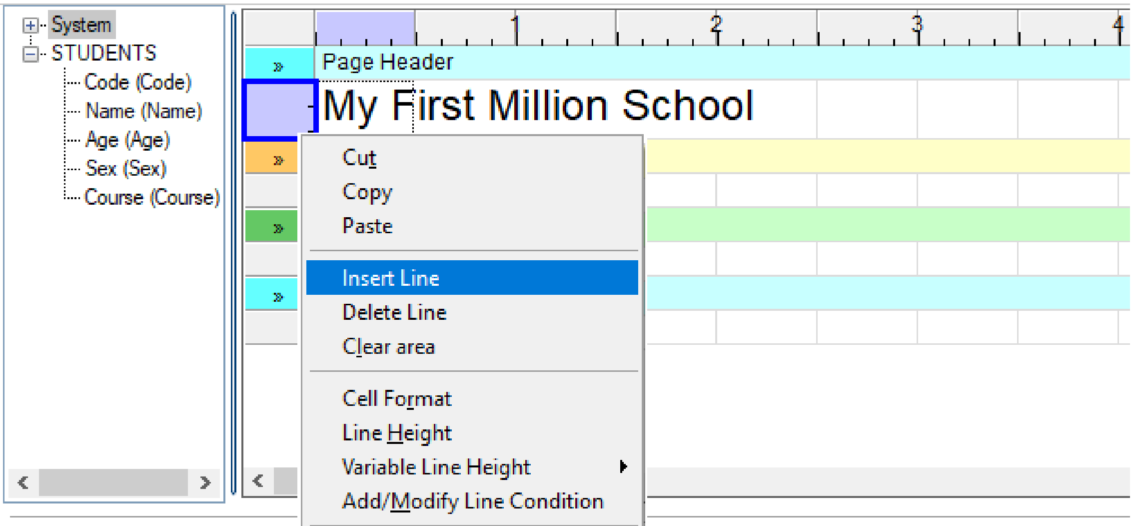


Select the Arial font in 16 point size.

After changing the font size, the title now appears to be cut off. We need to adjust the height of the row in the report layout so that the title is fully visible. To do this, position the mouse pointer on either the top or bottom border of the row until the pointer changes to a double-headed arrow. Then, click and drag the mouse to adjust the row height.

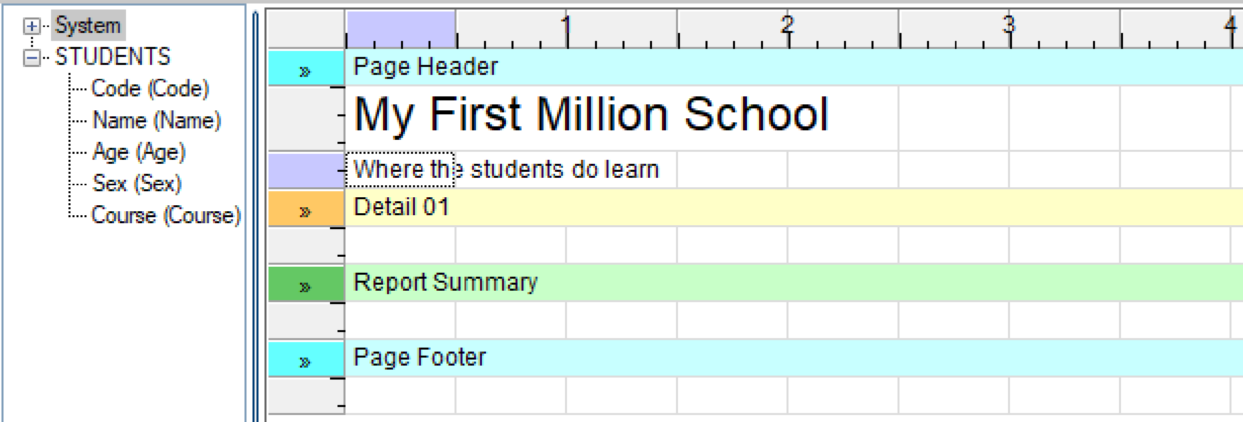


We need to add another line below the title. Right click on the cell highlighted in light purple (marked in blue), and in the context menu, select the [**Insert Line**] option.



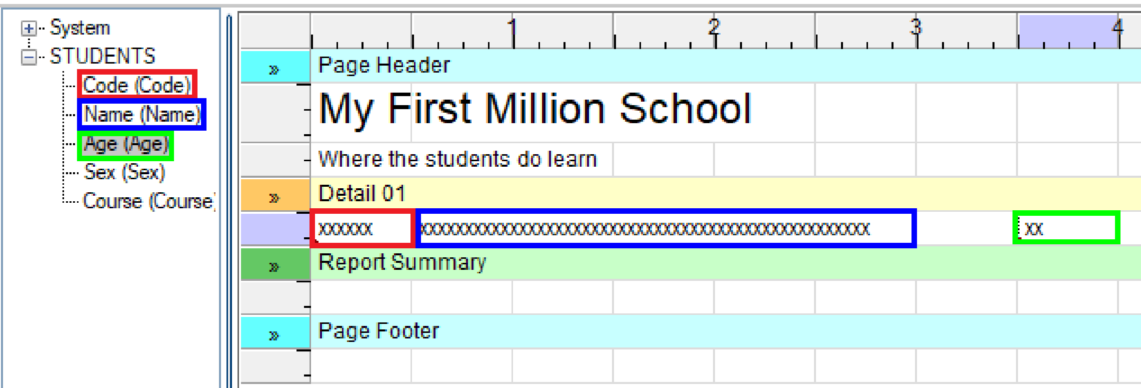
We now have a second header line where we enter the text: **Where the students do learn**.

Our layout should be similar to the one shown below:



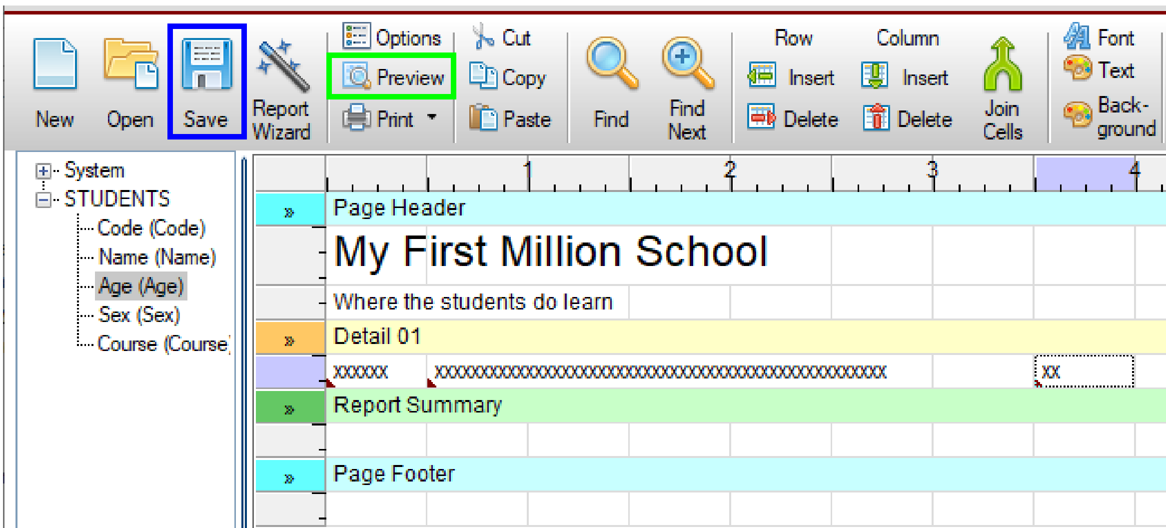
We will now add some of the elements from the **STUDENTS** table to the report layout. In the elements panel on the left, select the Code field and drag it to the first column of the line under the **Detail 01** line.

Drag and drop the Name and Age fields to the same line, leaving a space.



In the first cell, we now have the student's Code, in the second cell, the student's Name, and a little further to the right, the student's Age.

We can save and test our report. The options to [**Preview**] and [**Save**] the report are located in the toolbar at the top (marked in green and blue respectively). To save the report, you must provide a valid report name. We will save our report as **Million School.pvr** (**.pvr** is the default extension for a PxPlus report definition).



Let's preview the report. The report will appear on the screen in a separate window:

My First Million School

Where the students do learn

000001	John Hendersen	40
000002	Ana Victoria	19
000003	Irma Rose	45
000010	Amy Lucia	30
000011	Michael Ross	34
000012	Diana Martin	37
000013	Louis Briton	25
000020	Janet Perry	40
000021	Veronica Diaz	37
000022	Joseph Diego	52
000030	Michael Martin	60
000100	Brian Peters	41
000101	Simon Soler	27
000102	Diana Ruiz	36
000201	Fiona Lopez	47
000202	Jordan James	28

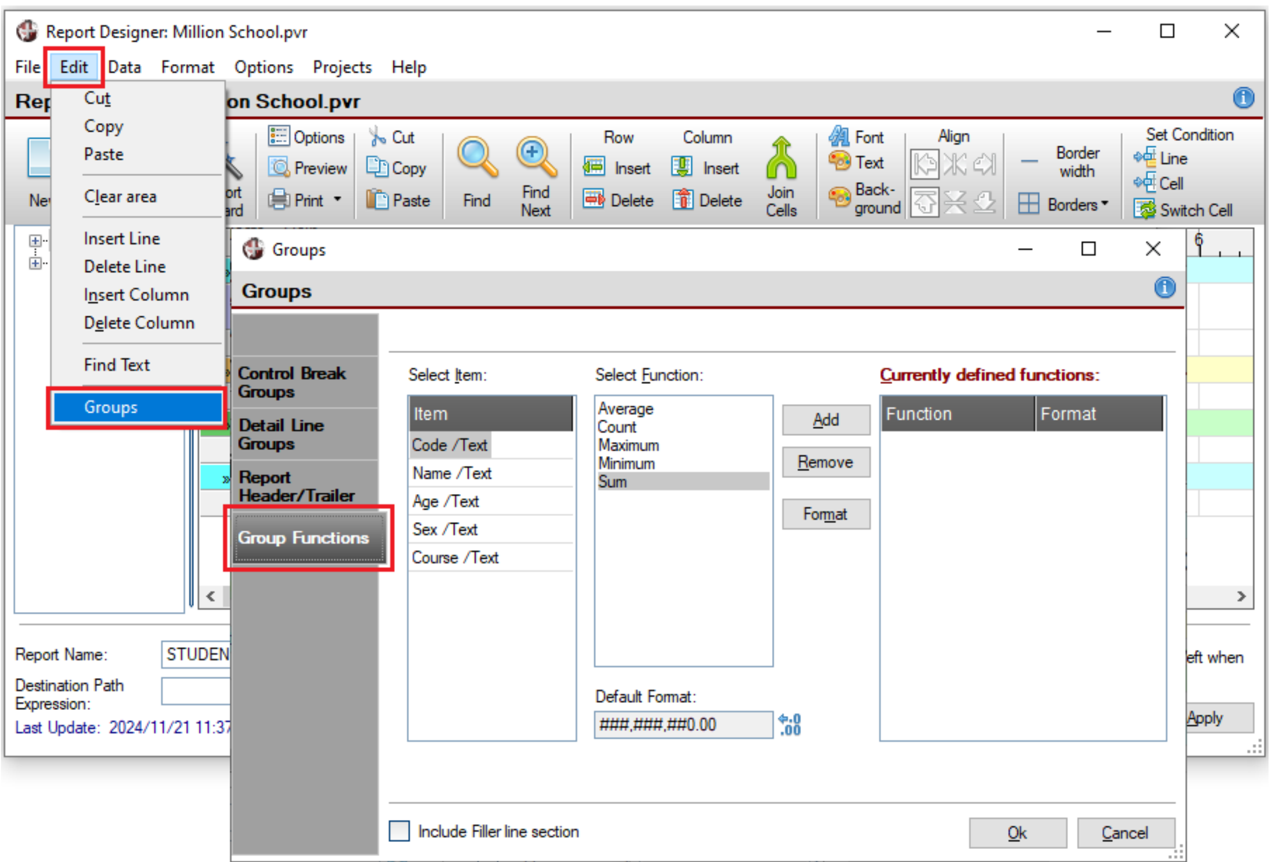
We have our first report!

Exercise: Defining a Group Function

We will now add a function to calculate the average age of the students, and to do this, we will use the [**Groups**] option.

Data groups allow you to define elements that will have common characteristics and therefore share similar values. The [**Group**] option provides functions such as Average, Count, Maximum, Minimum and Sum.

To define groups and the functions associated with them, in the top menu bar, select [**Edit**] -> [**Groups**], and in the **Groups** window, select the **Group Functions** tab.

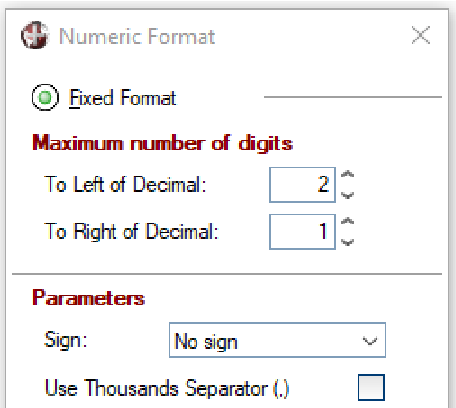


To define a **Group** function, select an element and the function, and define the output Format.

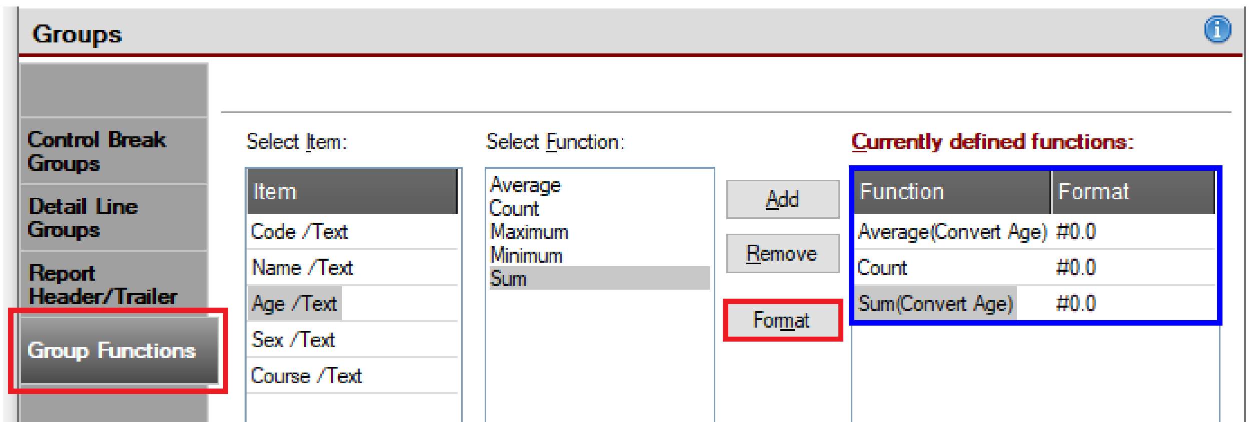
For our report, we want to define the **Average** function. Select Age from the list on the left. Now, select the **Average** function, and then click the **Add** button. A warning message displays because, in this case, we are trying to define a numeric function (**Average**) for a text element. Click **Yes** to continue. The **Report Writer** is "smart" enough to perform the conversion of the element and subsequently apply the calculation.

Next, change the output format for this function by clicking the [**Format**] button.

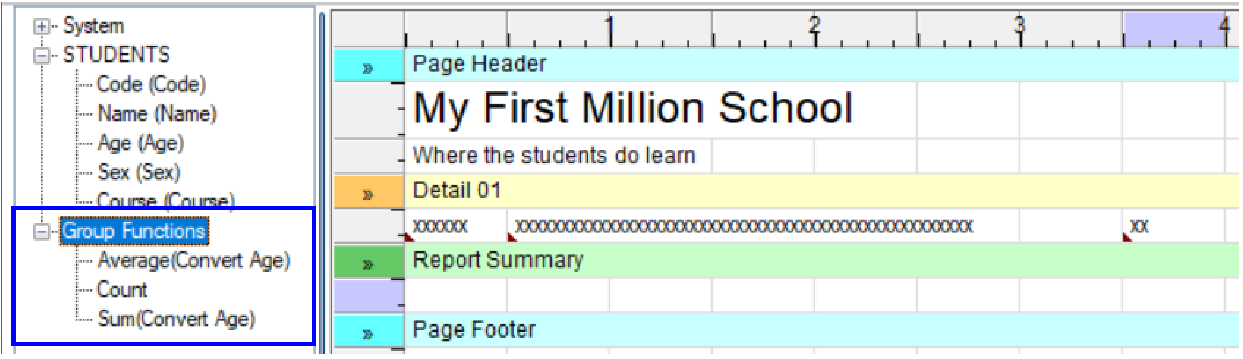
This opens the **Numeric Format** box. Change the number of digits to the left of the decimal to 2 and the number of digits to the right of the decimal to 1. Uncheck the [**Use Thousands Separator (,)**] check box.



For practice, we will add two more functions, **Count** and **Sum**. Remember to change the format as we did for the **Average** function.

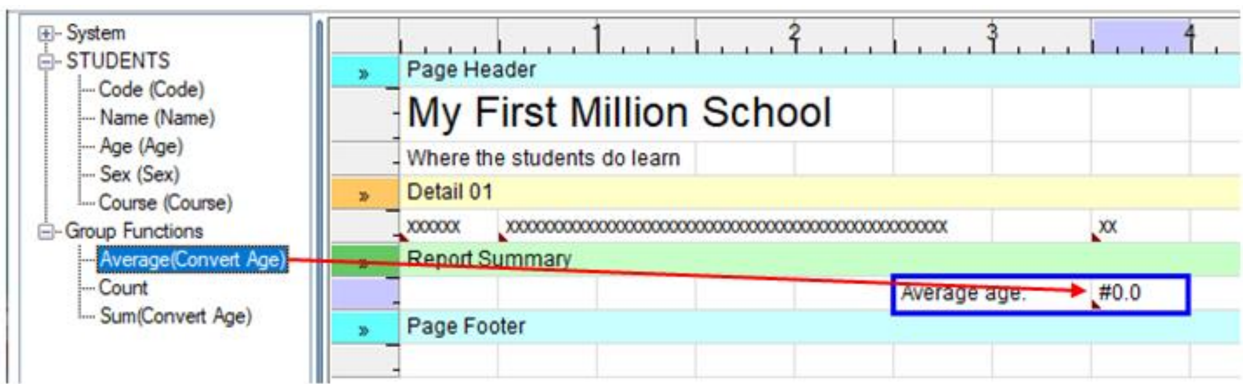


Once the functions have been defined, they will appear in the panel to the left of the report layout, and then, we can place it where we need it.



In the **Report Summary** section of the report layout, we will place the new calculation, along with explanatory text. Click on any cell of the summary and enter the text "Average age:" (followed by a

colon). Leave a space and in the next cell, select the Average function from the **Group Functions** list on the left and drag it to the summary.



Let's save the report and test it using the [**Preview**] option:

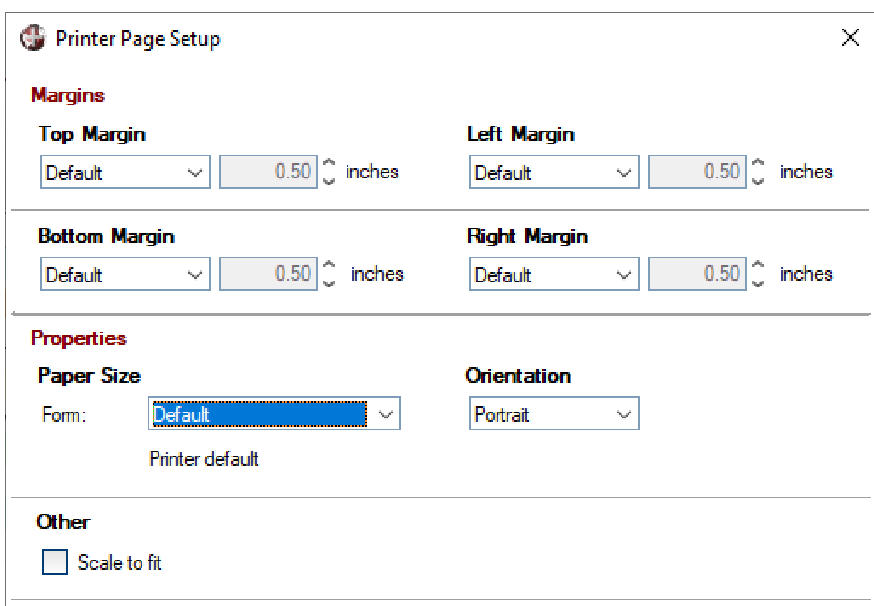
My First Million School

Where the students do learn

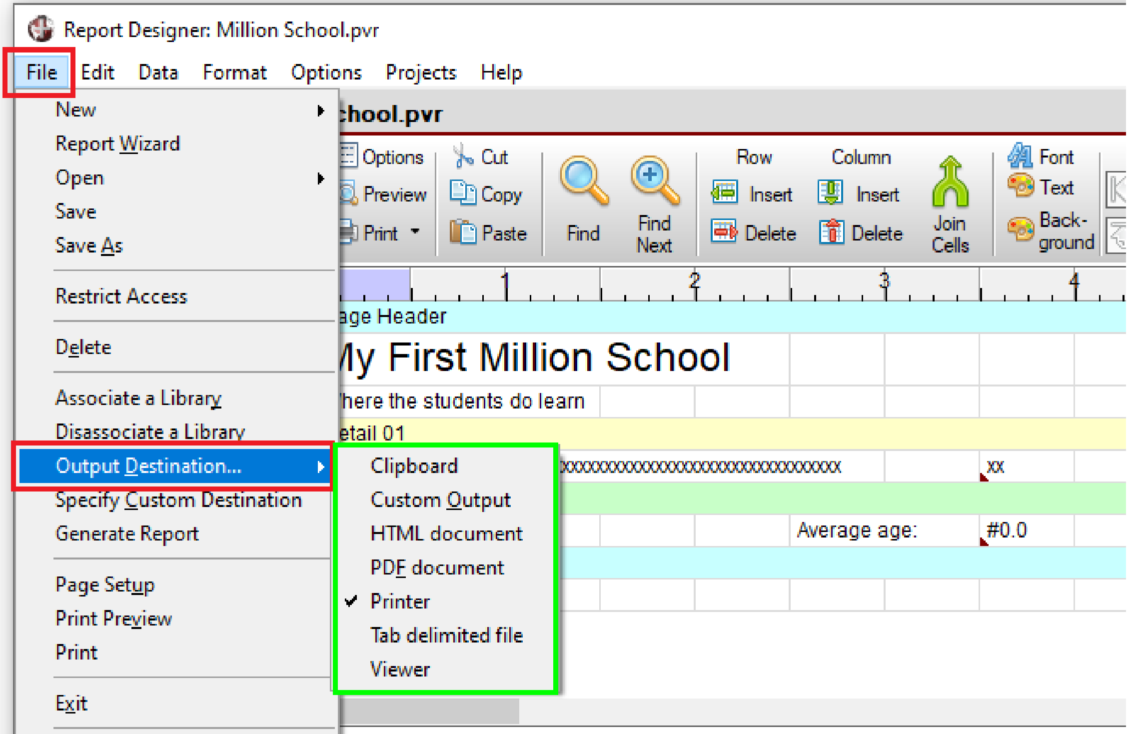
000001	John Hendersen	40
000002	Ana Victoria	19
000003	Irma Rose	45
000010	Amy Lucia	30
000011	Michael Ross	34
000012	Diana Martin	37
000013	Louis Briton	25
000020	Janet Perry	40
000021	Veronica Diaz	37
000022	Joseph Diego	52
000030	Michael Martin	60
000100	Brian Peters	41
000101	Simon Soler	27
000102	Diana Ruiz	36
000201	Fiona Lopez	47
000202	Jordan James	28
	Average age:	37.4

We already have our first report with some "intelligence" and without writing a single line of code!!

Some basic aspects, such as margins, paper size and orientation (if the report output applies), can be defined by selecting [**File**] -> [**Page Setup**] from the top menu bar. The first four settings are the margins: **Top Margin**, **Bottom Margin**, **Left Margin** and **Right Margin**. We can also specify **Paper Size** and **Orientation**, which indicates whether the report will be output in vertical format (Portrait) or in horizontal format (Landscape).

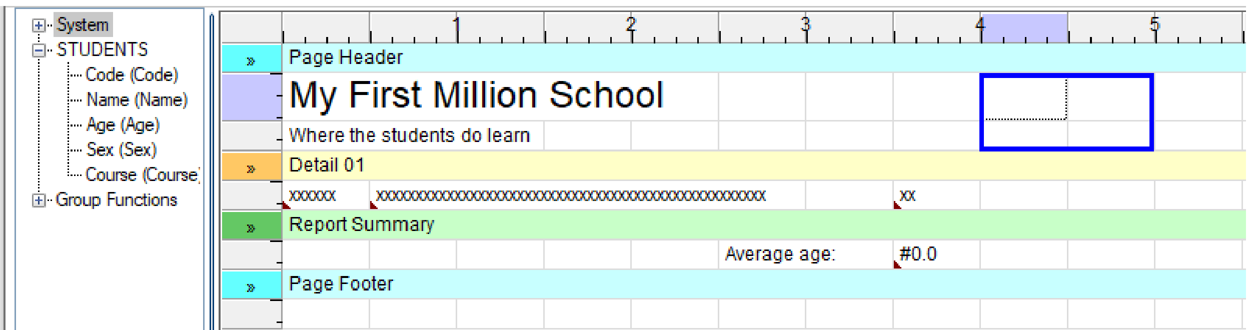


You can also specify a type of output for our report. From the top menu bar, select [**File**] -> [**Output Destination**]. The options (marked in green) are Clipboard, Custom Output, HTML document, PDF document, Printer, Tab delimited file and Viewer.

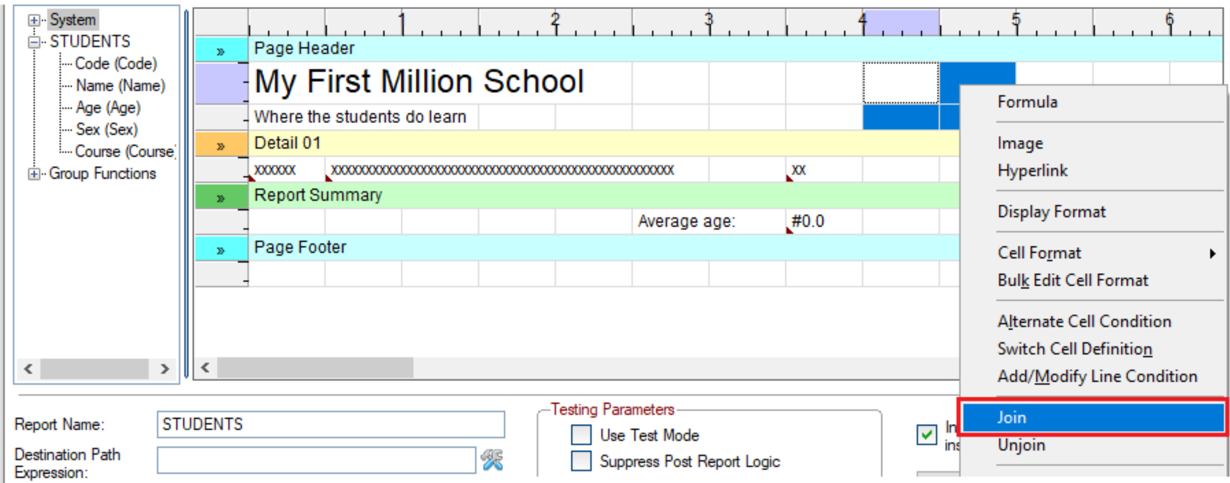


These "default" settings can be changed programmatically through properties of the **pvxreport** object.

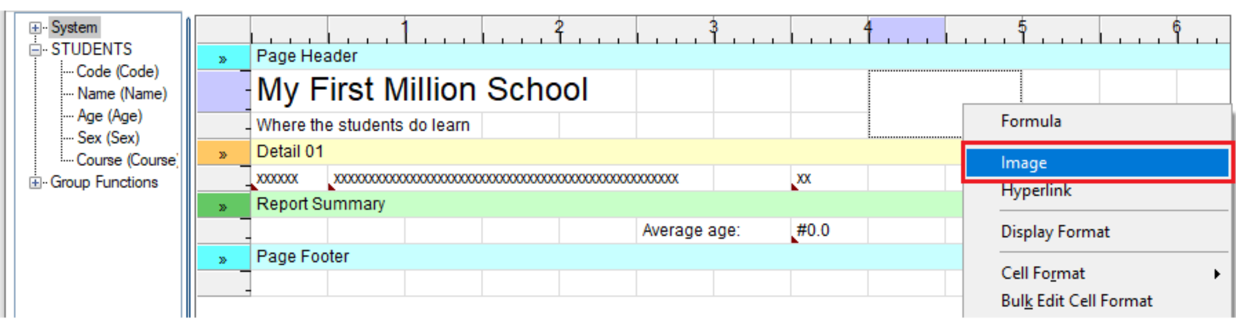
Let's create an image for our report. Find an appropriate logo for the school, and insert it in the upper right area. For that, we are going to join a group of cells (marked in blue):



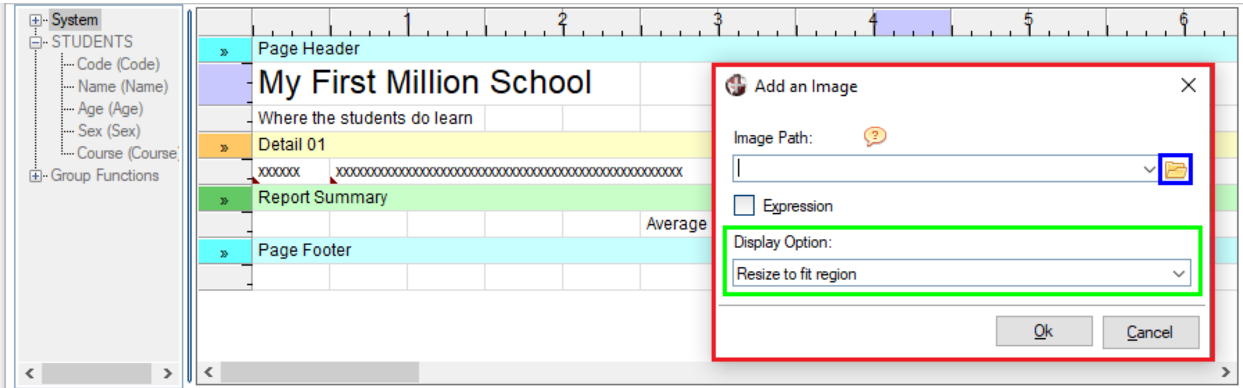
The way to do this is to mark the cells. Click on the first one and then use the [**Shift Click**] combination to select the other three. When all four are selected, right click and select the [**Join**] option from the context menu:



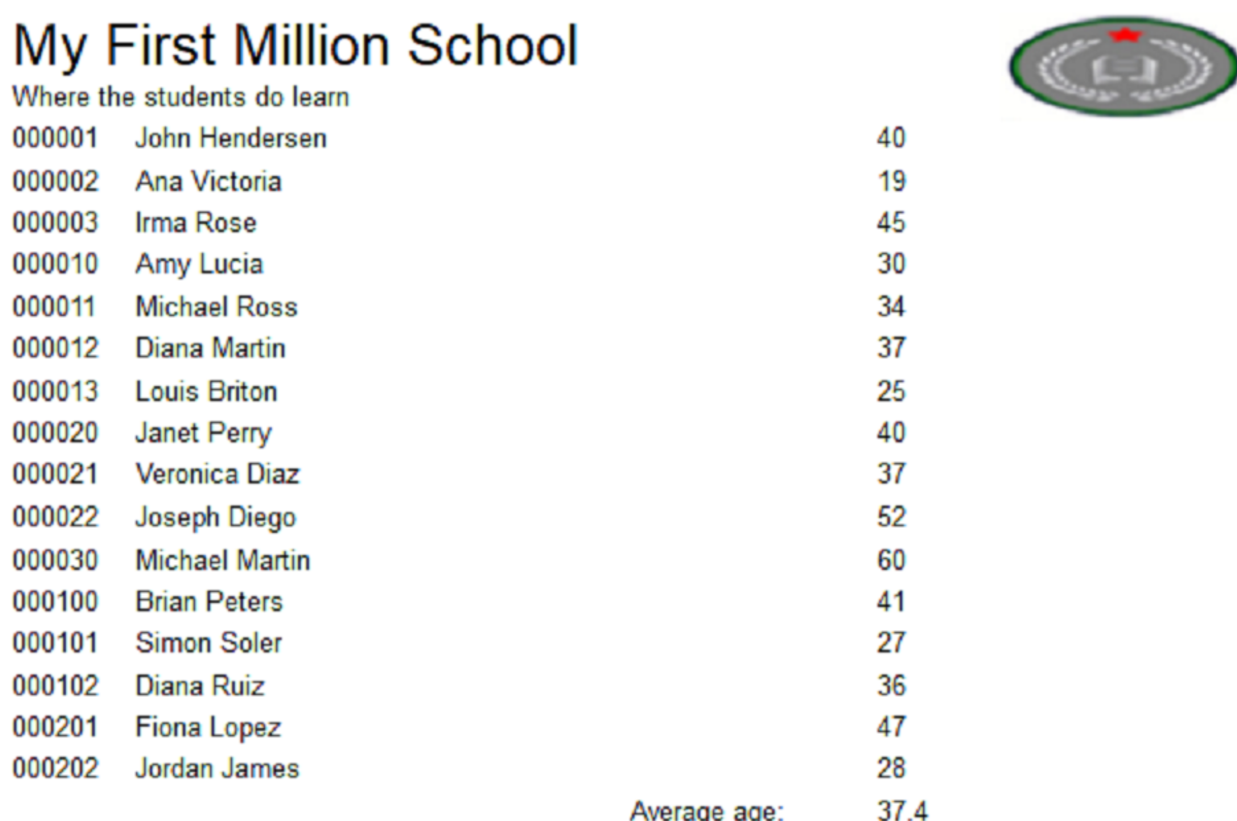
Right click again on the joined cells and select the **Image** option:



When the **Add an Image** window displays, click the yellow folder icon (marked in blue) to the right of the **Image Path** field and select the file with the school logo. Verify that the [**Display Option**] (marked in green) is set to **Resize to fit region**:



Save the report, and when you preview it, it should look something like the one shown below:



My First Million School

Where the students do learn

000001	John Hendersen	40
000002	Ana Victoria	19
000003	Irma Rose	45
000010	Amy Lucia	30
000011	Michael Ross	34
000012	Diana Martin	37
000013	Louis Briton	25
000020	Janet Perry	40
000021	Veronica Diaz	37
000022	Joseph Diego	52
000030	Michael Martin	60
000100	Brian Peters	41
000101	Simon Soler	27
000102	Diana Ruiz	36
000201	Fiona Lopez	47
000202	Jordan James	28
	Average age:	37.4

Good! Let's continue and see how to place the value or format of a cell conditionally.

We want to add a condition to the Sex element. If the value is "F" (Female), it will display in the "normal" Black color; otherwise, if it is "M" (Male), it will be in Blue. This requires adding the Sex element to the report layout, specifying the condition (which is, Sex="M"), and finally, creating an alternate cell definition for which we will associate the Blue color.

If everything goes as planned, our report will look similar to the one shown below:

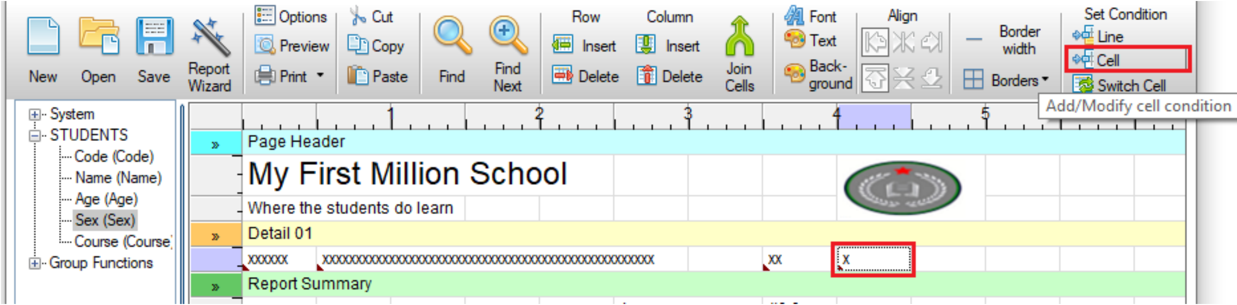
My First Million School

Where the students do learn

000001	John Hendersen	40	M
000002	Ana Victoria	19	F
000003	Irma Rose	45	F
000010	Amy Lucia	30	F
000011	Michael Ross	34	M
000012	Diana Martin	37	F
000013	Louis Briton	25	M
000020	Janet Perry	40	F
000021	Veronica Diaz	37	F
000022	Joseph Diego	52	M
000030	Michael Martin	60	M
000100	Brian Peters	41	M
000101	Simon Soler	27	M
000102	Diana Ruiz	36	F
000201	Fiona Lopez	47	F
000202	Jordan James	28	M
	Average age:	37.4	

Let's begin by dragging and dropping the Sex element from the left side panel to an empty cell to the right of the Age element in the **Detail 01** section.

Next, define the condition for this cell. In the top toolbar, under the **Set Condition** grouping, click the [**Cell**] option:



This opens the **Define Filters** window for adding an alternate cell condition:

Define Filters

Define Filters

Set #1 of 1 | Use the alternate cell definition if all conditions are met

Element	Condition	Case Sensitive	Value 1	Value 2
Code (Code)	None	Yes		
Name (Name)	None	Yes		
Age (Age)	None	Yes		
Sex (Sex)	None	Yes		
Course (Course)	None	Yes		
AVERAGE(Convert Age)	None			
COUNT	None			
SUM(Convert Age)	None			

Filter Set

Note: Use double quotes ("") to indicate a null value.

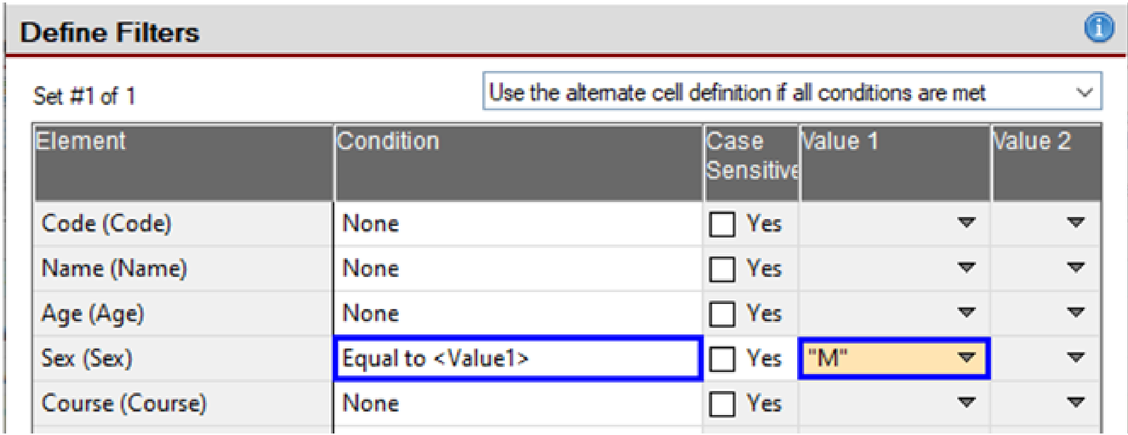
Ok Cancel

An alternate definition for individual cells can be specified within a report. The alternate cell definition has a condition associated with it to determine which definition will be used for display when the report is generated.

Once the alternate definition has been set, the [**Switch Cell Definition**] option can be selected to switch between the two definitions and "toggle" the way the data is presented. You can also have two images, a woman and a man, and alternate them depending on the content of the selected filter (in this case, the Sex element).

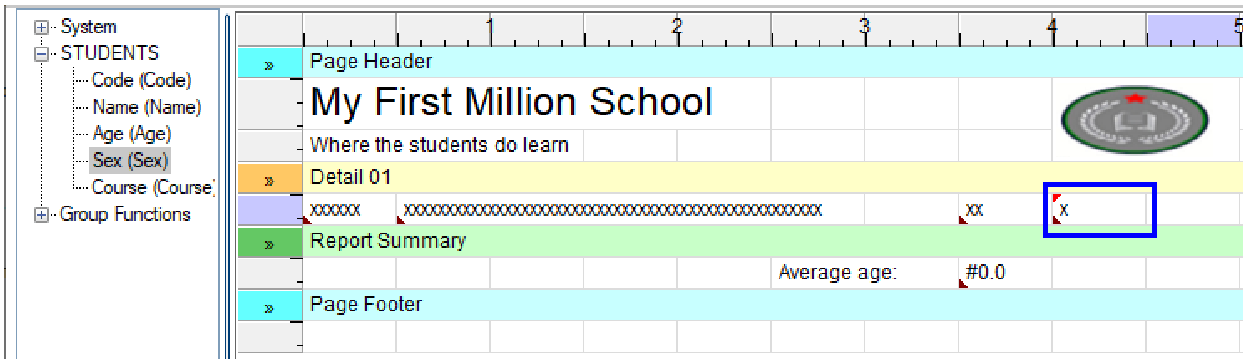
For this report, we will set an alternate cell condition for the Sex element.

For the Sex element row, click inside the [**Condition**] cell and select **Equal to <Value1>** from the drop-down list. Click inside the [**Value 1**] cell and enter "M".



Click the [**OK**] button to accept this filter definition.

Notice that a small red tick mark displays in the top left corner of the cell to signal the alternate cell definition.



Now that we have an alternate condition for this cell, we want to apply the Blue color. Click the [**Text**] option in the top toolbar, and select Blue.

Preview the report. It should look similar to the one shown below:

My First Million School

Where the students do learn

000001	John Hendersen	40	M
000002	Ana Victoria	19	F
000003	Irma Rose	45	F
000010	Amy Lucia	30	F
000011	Michael Ross	34	M
000012	Diana Martin	37	F
000013	Louis Briton	25	M
000020	Janet Perry	40	F
000021	Veronica Diaz	37	F
000022	Joseph Diego	52	M
000030	Michael Martin	60	M
000100	Brian Peters	41	M
000101	Simon Soler	27	M
000102	Diana Ruiz	36	F
000201	Fiona Lopez	47	F
000202	Jordan James	28	M
	Average age:	37.4	

Note: Data Filters can be set up to filter the data to appear on the report. Two different kinds of data filters can be set up, Static and Dynamic. **Static Filters** are set filters that are automatically applied to the report whenever it is run. **Dynamic Filters** are filters that are defined by the end user at run time.

Apart from the options that we have seen, you can change the presentation of each element (cell) of the report, such as lines, boxes, colors, fonts and letter sizes, etc. Now that you are beginning to see the potential of the Report Writer, it is a good time to evaluate the options available when defining data sources or origins, establishing links and file relationships, filters, etc.

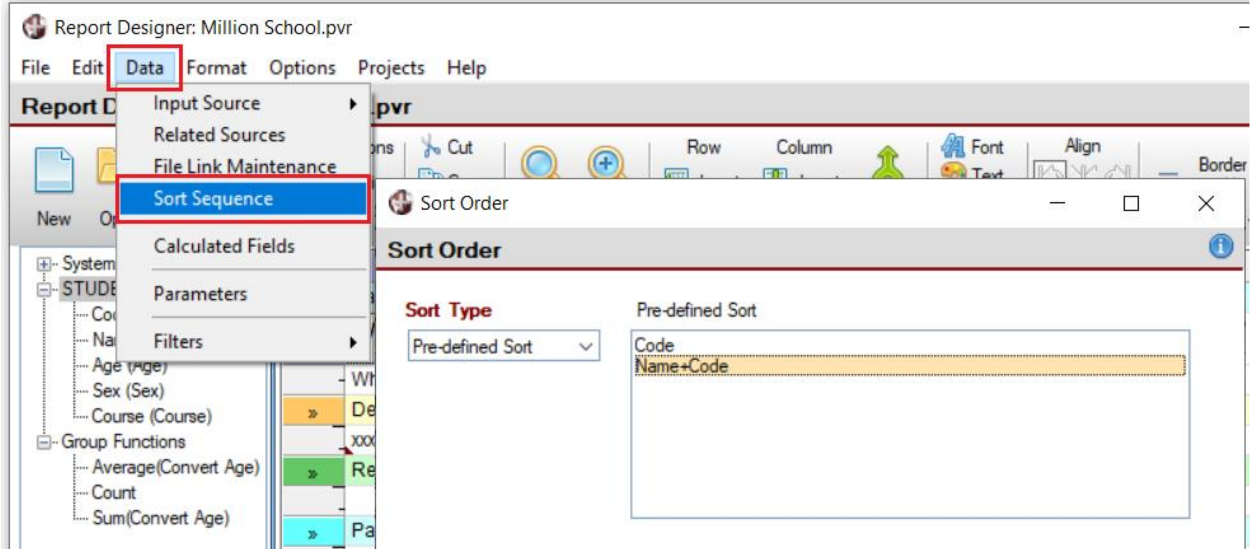
Example: Let's take a look at another example of other options we can use to change the list.

We will organize the list alphabetically (the examples so far have been ordered by Student Code). We will add a filter so that only students with names that begin with the letter "A" through to "I" will display. We will also change the list to show the data (lines) with alternate colors so that each line is easier to read.

To specify the sorting method, select the [**Data**] -> [**Sort Sequence**] option in the top menu bar.

This opens the **Sort Order** window for selecting the [**Sort Type**], either a **Pre-defined Sort** (a key) or a **Custom Sort**.

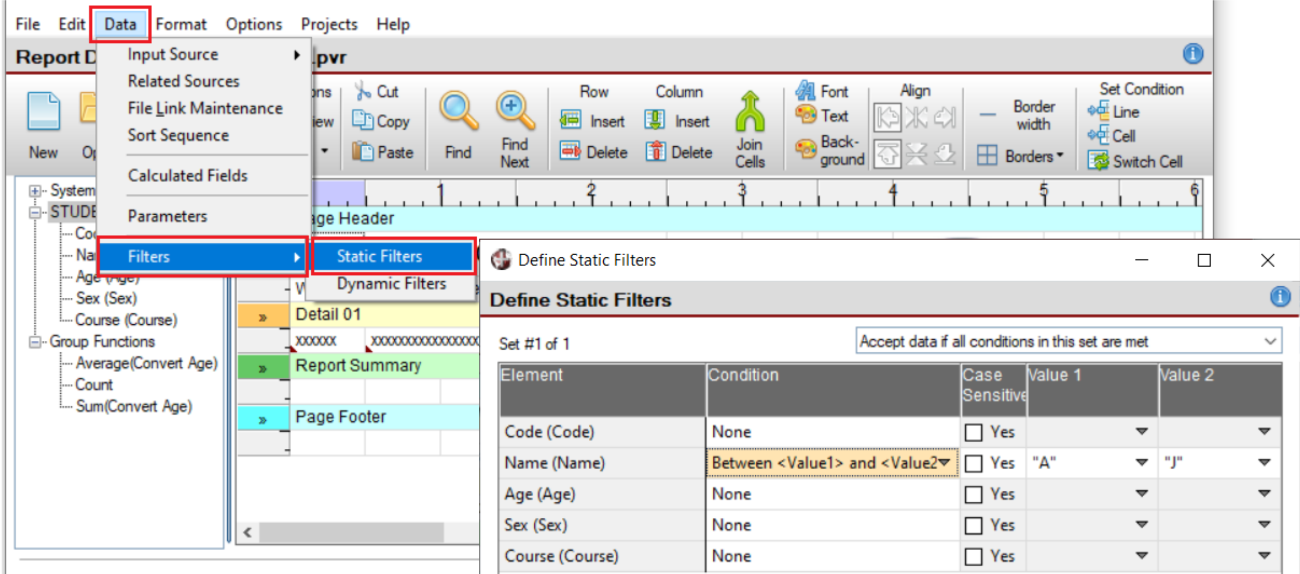
Select the Pre-defined Sort, Name+Code.



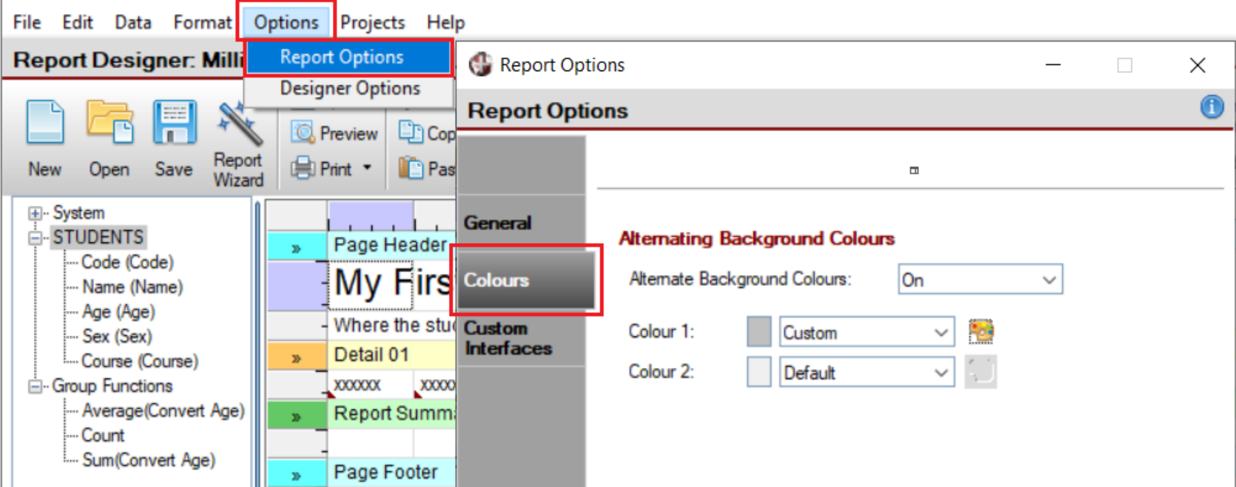
Note: Remember that when you are defining data sources, you can also perform selections and filtering actions.

Next, we will define the records to include. In the top menu bar, select the [**Data**] -> [**Filters**] -> [**Static Filters**] option. In the **Define Static Filters** window, enter the selection criteria to filter the data for the report.

As shown in the following example, specify a condition for the Name element, and enter the range from "A" to "J". But, only names that contain "J" would be accepted, not "JA" or anything similar, and since there are no names with only one "J", it would filter down to the previous letter, which would be any name with the letter "I". You can also define "Izz", which covers any name that begins with "I" and has up to "zz" in positions 2 and 3. Do several tests to be sure that the established filters are working properly.



With the filter defined, we will apply the alternate colors to the lines. In the top menu bar, select [**Options**] -> [**Report Options**]. In the **Report Options** window, select the **Colours** tab. Set the [**Alternate Background Colours**] option to On and then choose the colors you want.



Save and preview the report. It should look similar to the one shown below:

My First Million School
Where the students do learn

000010	Amy Lucia	30	F
000002	Ana Victoria	19	F
000100	Brian Peters	41	M
000012	Diana Martin	37	F
000102	Diana Ruiz	36	F
000201	Fiona Lopez	47	F
000003	Irma Rose	45	F
	Average age:	36.4	

Now, we have arranged the names alphabetically and filtered the list so it now shows fewer students. It lists only those whose name begins with "A" through "I", so the student "Janet Perry", for example, does not appear. In addition, we see that there is a colored stripe on the even lines, and it is easier to read the report.

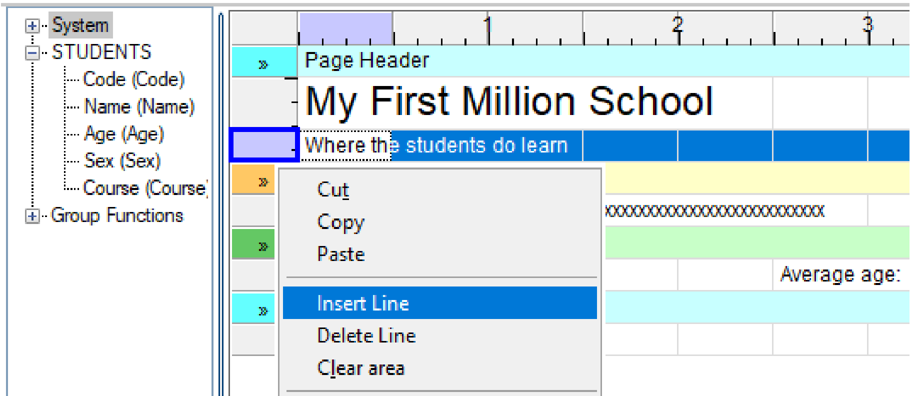
The aesthetic part has many aspects, and it will be practice and experimentation that will guide you in making the best use of the benefits of the **Report Writer** where you can manipulate the report.

Example: You can add column headings and format them, as shown below:

My First Million School
Where the students do learn

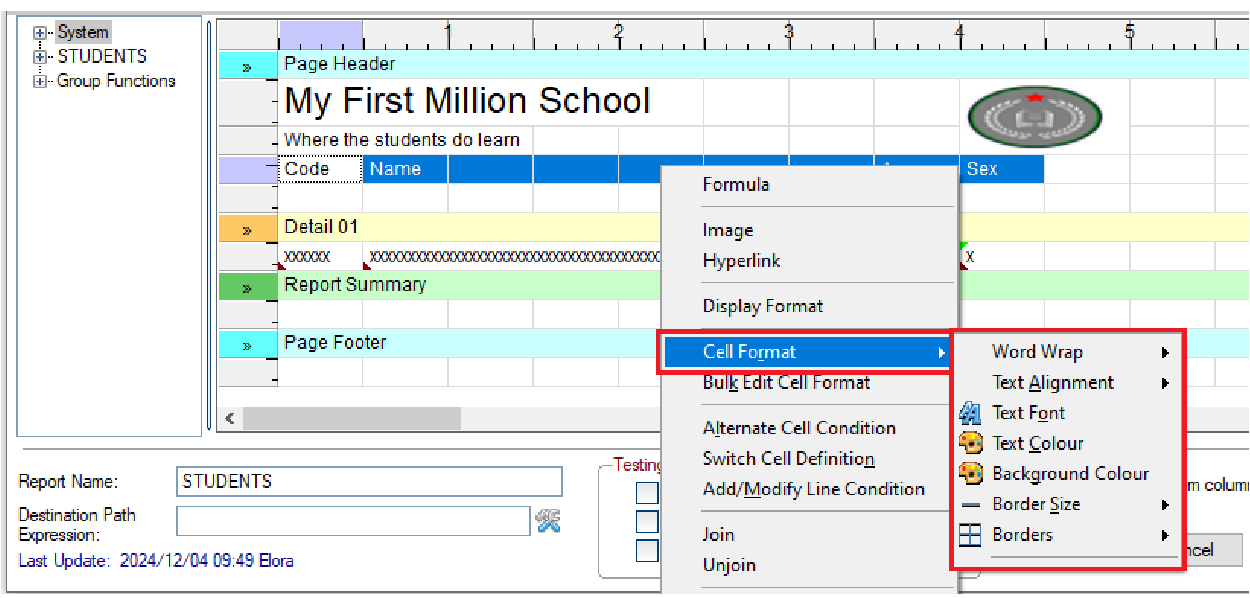
Code	Name	Age	Sex
000010	Amy Lucia	30	F
000002	Ana Victoria	19	F
000100	Brian Peters	41	M
000012	Diana Martin	37	F
000102	Diana Ruiz	36	F
000201	Fiona Lopez	47	F
000003	Irma Rose	45	F
	Average age:	36.4	

To do this, add two lines below the second header **Where the students do learn**. Right click on the cell highlighted in light purple (marked in blue below), and in the context menu, select the [**Insert Line**] option.



In the first inserted line, enter the text for the column headings (Code, Name, Age, Sex), placing each heading above its corresponding element.

We now want to format these headings to apply bolding and an outside border. Select these cells, and then, right click and select the [**Cell Format**] option:



Select the [**Text Font**] option, and then, for **Style**, select **Bold**.

Next, we will add a border. With these cells still highlighted, right click again, and select **Cell Format**. Select the [**Borders**] option, and then choose **Edges Only**.

When you preview the report, the column headings are now bolded and show an outside border:

My First Million School

Where the students do learn

Code	**Name**	**Age**	**Sex**
000010	Amy Lucia	30	F
000002	Ana Victoria	19	F
000100	Brian Peters	41	M
000012	Diana Martin	37	F
000102	Diana Ruiz	36	F
000201	Fiona Lopez	47	F
000003	Irma Rose	45	F
	Average age:	36.4	

With more complete data, you can make slightly more sophisticated reports where there are breaks, subtotals for each group, colors, etc.

Example: This shows a summarized Sales report, grouped by Branch, with sub-totals:

Big Money C.A.

Summary of Sales by Branch

Printed: **04/12/24**

Branch 01

Code	Salesrep	Sales
001	Miguel Perez	$250
002	Diana Rodriguez	$725
003	Cristina Lopez	$1,500
005	Angel Luciano	$140
021	Leonor Paz	$8,700
030	Maria Montanez	$2,100
	Total:	$13,415

Branch 02

Code	Salesrep	Sales
008	Luna Limongi	$2,300
009	Pedro Ponce	$8,900
010	Lorena Diaz	$1,680
016	Zaida Leon	$6,745
025	Lucero Perez	$3,450
038	Ana Marquez	$8,310
053	Ramon Mendez	$7,300
	Total:	$38,685

Branch 03

Code	Salesrep	Sales
004	Corina Casado	$6,320
011	Orlando Siso	$400
012	Salvador Roa	$10,400
017	Maria Montes	$3,800
034	Tania Ponce	$7,900
040	Antonio Leon	$1,860
	Total:	$30,680

Total Sales: **$82,780**

Exercise: Creating a Report (from the beginning)

This would be a good time to get more familiar with using the **Report Writer** and also practice what you have previously learned. Take some time to create this report from the beginning.

Here are some guidelines to help you work through the steps:

1. In **Data Dictionary Maintenance**, create a Salesreps table (you can call it whatever you want) with the minimum necessary data: Branch number, Salesrep Code, Salesrep Name and Sales Amount. Define a primary key (sorted by Code) and an alternate key (sorted by Branch+Code).

   Remember to click the [**Update File**] button to create/update the file.

2. Use the **File Maintenance Generator** to create a panel for the Salesreps table.

3. Enter the following Salesreps data into the generated file maintenance panel:

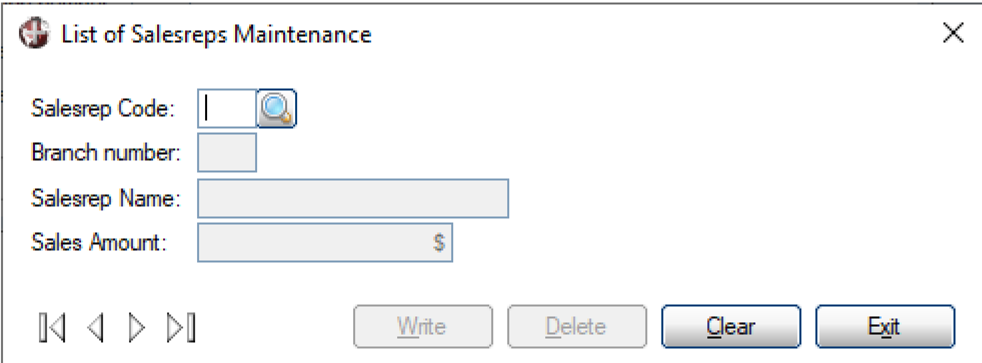
Branch number	Salesrep Code	Salesrep Name	Sales Amount
01	001	Miguel Perez	250
01	002	Diana Rodriguez	725
01	003	Cristina Lopez	1500
03	004	Corina Casado	6320
01	005	Angel Luciano	140
02	008	Luna Limongi	2300
02	009	Pedro Ponce	8900
02	010	Lorena Diaz	1680
03	011	Orlando Siso	400
03	012	Salvador Roa	10400
02	016	Zaida Leon	6745
03	017	Maria Montes	3800
01	021	Leonor Paz	8700
02	025	Lucero Perez	3450
01	030	Maria Montanez	2100
03	034	Tania Ponce	7900
02	038	Ana Marquez	8310
03	040	Antonio Leon	1860
02	053	Ramon Mendez	7300

4. Define a new **Query** called QRY_SALES for the table based on the Salesrep Code.

5. In the properties window for the Salesrep Code control, click the **Query** tab and specify the Query panel to use (QRY_SALES).

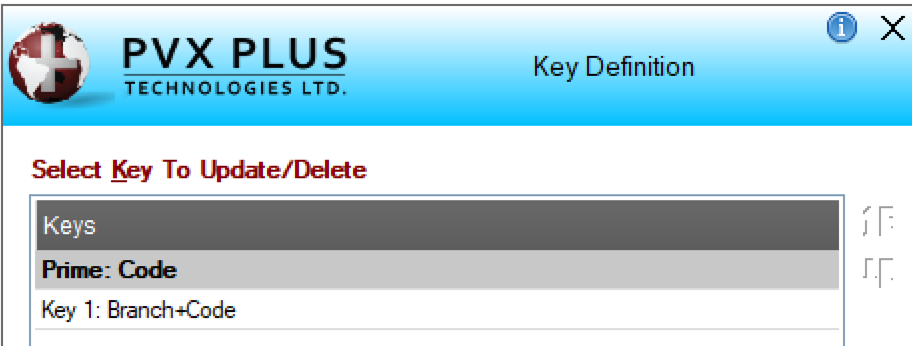
   Specify a bitmap image and width for the query.

   Your panel may look something like the one shown below:

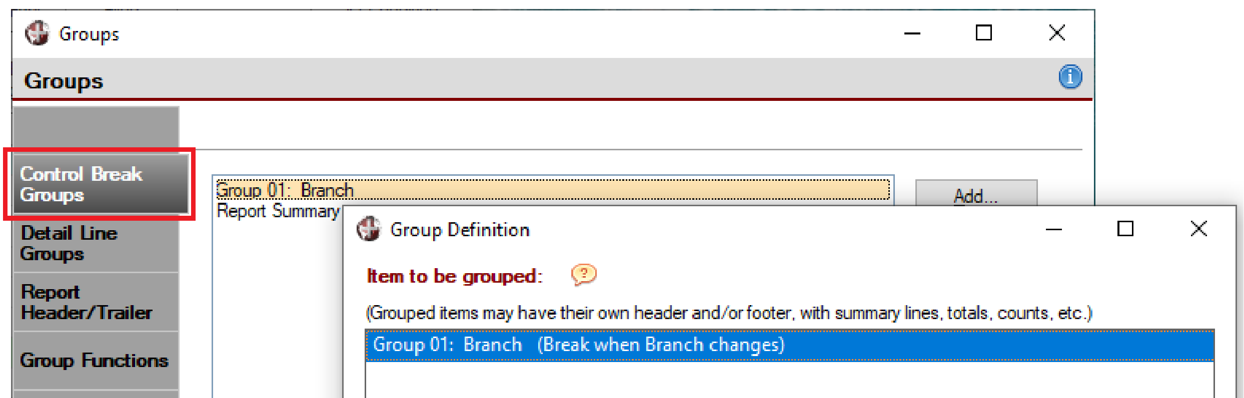


6. Create the new report (as shown above) in the **Report Designer**. Refer to the steps we used previously to create the My First Million School report. Some additional help is provided below:

   To create the groups, you must have an alternate key define, Branch+Code, in **Data Dictionary Maintenance**:

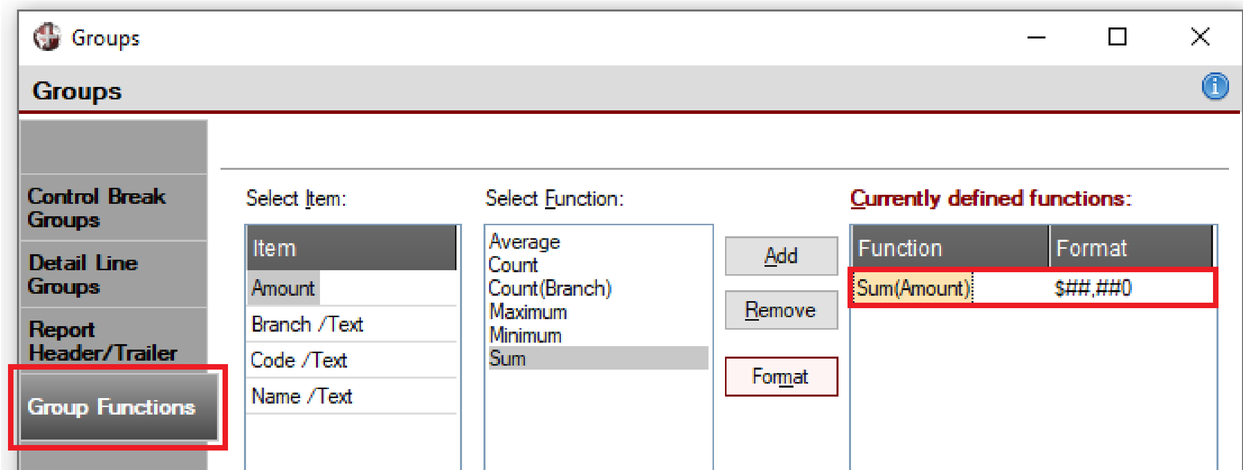


   To define a Control Break Group, select [**Edit**] > [**Groups**] in the top menu of the **Report Designer**:



   To insert the **Printed** date in the **Page Header** section of the report layout, expand the **System** option on the left side and then drag/drop a Date format to the desired location on the report layout.

To define a **Sum** Group Function for the Footer and Report Summary sections of the report layout, select [**Edit**] > [**Groups**] in the top menu of the **Report Designer**:



These Group Functions will depend heavily on previous data source definitions, linked files and the type of dependencies they have, if any.

7. To add formatting to a cell, such as text font, text color and borders, right click on a cell and select the [**Cell Format**] option.

8. Save and preview the report.

Since it is very simple to modify the reports, we suggest that you do several tests, including some with linked files.

Once you have designed the report, you can generate or "execute" it in several ways.

One way is from the Command line (or from your program):

```
call "*rpt/runreport;Run_Call","/pxp/rpt_client.pvr"
```

You can also do it through the **pvxreport** object. Once instantiated (first line), you will have the **RunReport()** method or function available where you can place the name of the report as an argument:

```
rpt=new("*rpt/pvxreport")
RunReport("/pxp/rpt_client.pvr")
```

Example: Let's look at a slightly more complete example where the output of the report will be a PDF file that we have defined in a channel:

```
! Report output to a PDF file
!
! Create an rpt report type object
rpt=new("*rpt/pvxreport")
!
! Find the free channel
channel_rpt=unt
! Define the PDF name
pdf_file_name$="myreport.pdf"
! Open the pdf device
open(channel_rpt,opt="File="+pdf_file_name$)"*PDF*"
!
! Execute the report, specify the PDF channel as output
rpt'runreport("myreport.pvr",channel_rpt)
! Close channel
close (channel_rpt)
!
exit:
! Drop object
drop object rpt
exit
```

The **Report Writer** is a powerful tool that allows us to design simple or more complete reports, and we can manipulate it directly from our application.

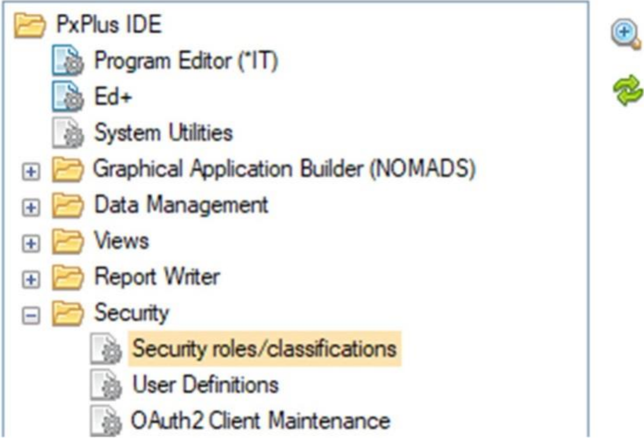
NOMADS Security System

NOMADS incorporates an optional security subsystem that allows or restricts access through the setup of security classes and the users associated with each of the classes.

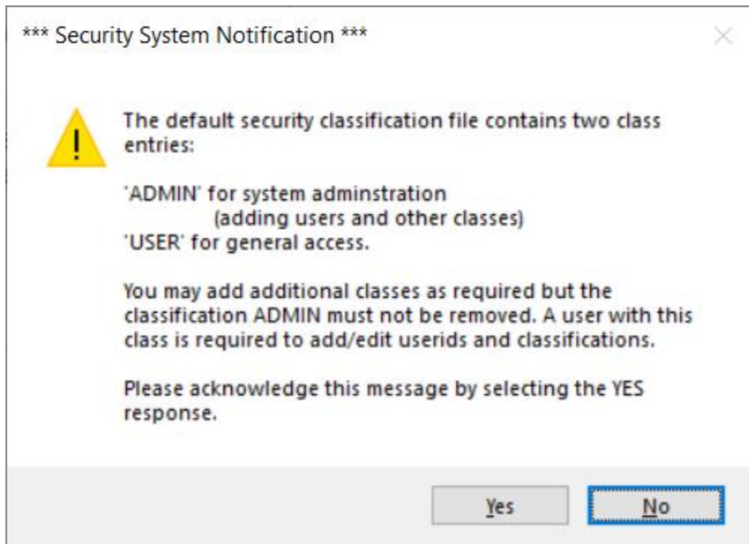
When we run NOMADS, the panels and their components do not have any security classes associated with them; therefore, everything is available to all users.

If a security class is assigned, then only users with explicit access will be able to view/run the panel or component. So, the approach of the NOMADS security system is that there are several users, which must belong to a class, and each class will have permissions or not to access different components. The restriction or accessibility is actually class-based, and only users who belong to the class that has access will be able to execute the control or panel.

To access (and implement) the security system, go to the PxPlus IDE main menu, open the **Security** category and select **Security roles/classifications**.

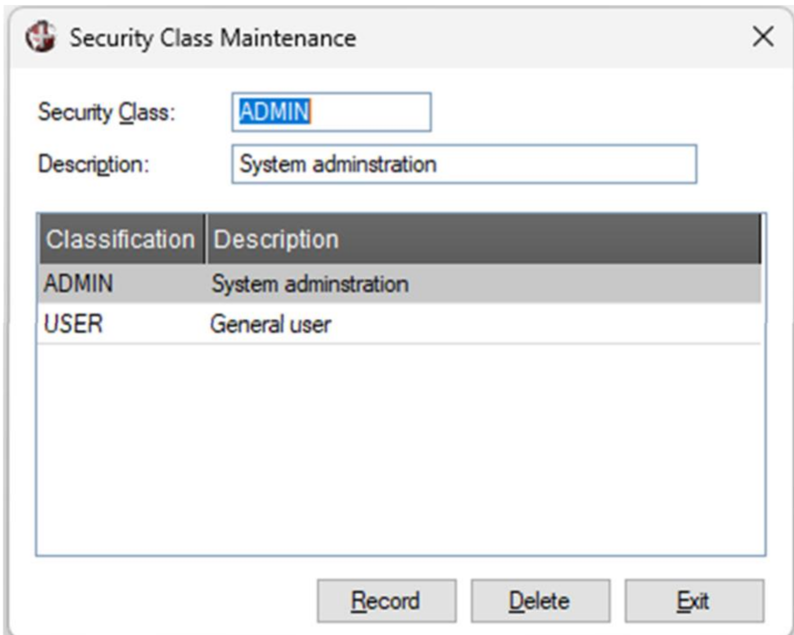


When setting up the security system for the first time, the system asks for permission to create the security system control tables. Click Yes. The following message will display:



At this point, you can either click No to halt the process or click Yes to continue. Click Yes. The **Security Class Maintenance** window will display.

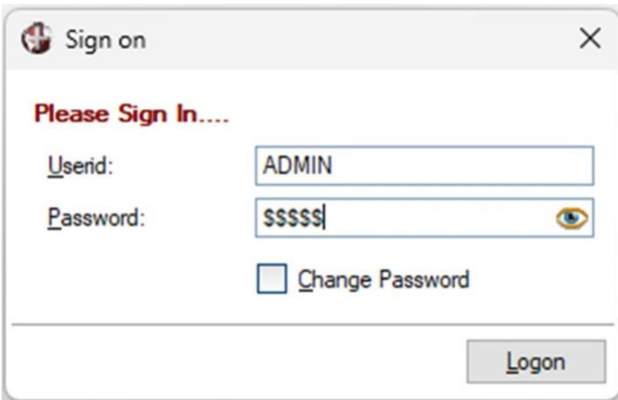
By default, two classes are provided: an **ADMIN** class, which is for the NOMADS system administrator (for adding users and other security classes), and a generic **USER** class for regular users:



You can define any classes you want, and a user can belong to multiple security classes. **Example:** You could have module classes: SALES, BANKS, PAYROLL, INVENTORY, etc. The correct procedure is to create the security classes first and then the users, since, when creating the users, they must be assigned to the class(es) to which they belong.

Refer to **Security Class Maintenance** in the PxPlus Help documentation.

Once you have access to the security system, your identification (Sign In) will be necessary in the system. By default, at the beginning, there is only the user **ADMIN** with the password **ADMIN**.

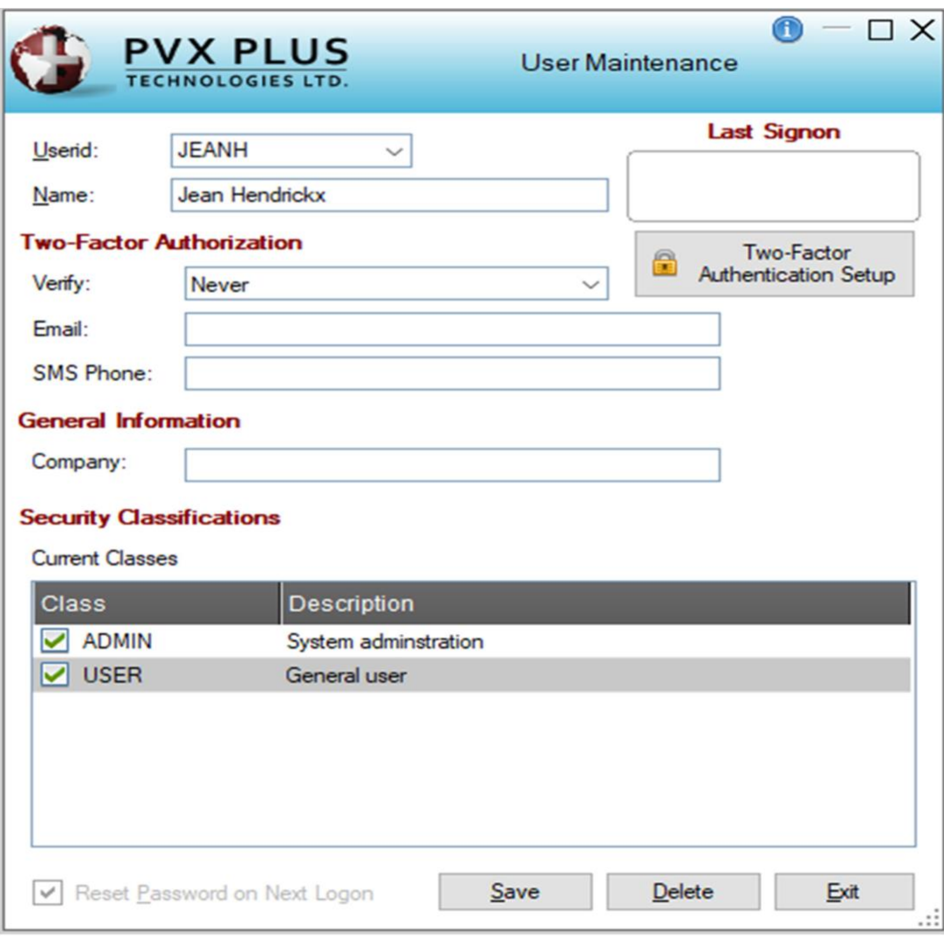


Important Note: We strongly recommend that you immediately change the **ADMIN** user password to

ensure more efficient security control.

The other component of the security system is called **User Maintenance**, where you define a user (USERID) and specify the user's real Name, mostly for internal control and references. You assign the class(es) to which the user belongs, and you can optionally implement **Two-Factor Authentication** (by other external means, such as email or a text message to the cell phone/mobile).

To access the **User Maintenance** window, go to the PxPlus IDE main menu, open the **Security** category, and select **User Definitions**. The **User Maintenance** window will display:



Specify the User ID, which must be unique, as there could be two users named José Perez, but they would have to have different User IDs, such as JOSEPEREZ and JPEREZ.

Then, enter the real name (for reference, more than anything else) and assign the security class to which each user belongs.

In the above example, the user "JEANH" belongs to the classes "ADMIN" and "USER", so this user will have the privileges and permissions that those two classes have. The user identification (USERID) can have a maximum length of 12 characters.

The **Two-Factor Authentication** option requires additional configuration (which will not be discussed here) and consists of sending a code via email or text message to the cell phone that must be entered as proof that it is actually the user in question and not only someone who knows the login credentials (User ID and Password).

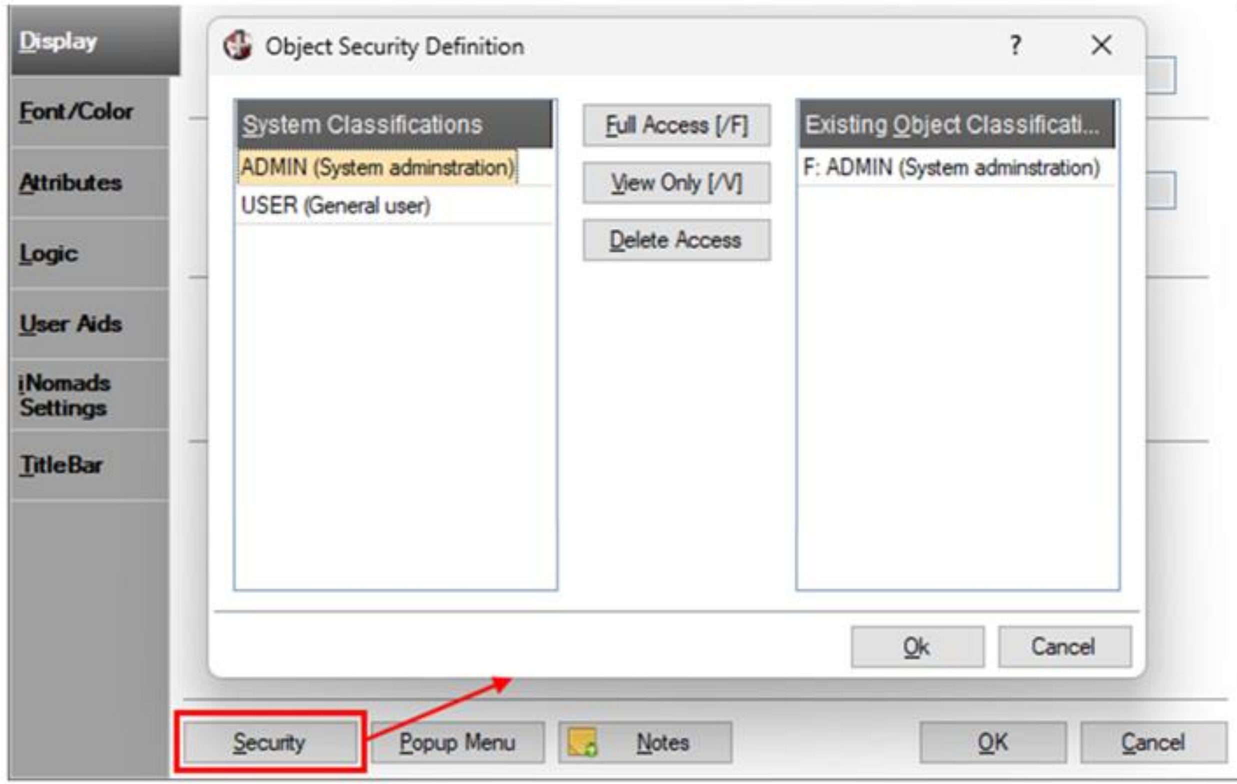
Refer to **User Maintenance** and **Two-Factor Authentication Setup** in the PxPlus Help documentation.

Once the classes and users are defined (and the users are assigned to the created classes), you can test the security system.

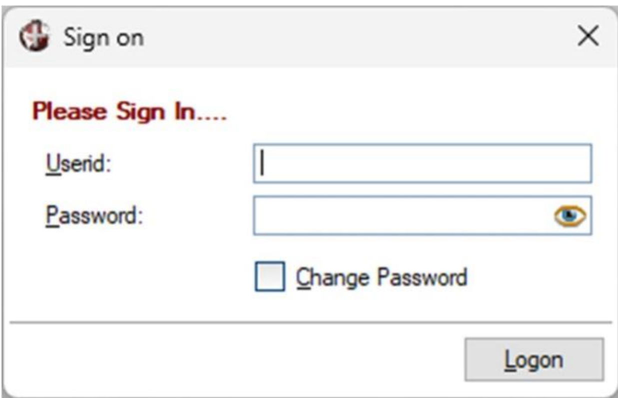
Exercise: Using NOMADS Security

Open a NOMADS library, and open any panel. Change the properties of the panel itself by using the [**Header**] option at the top of the panel designer. At the bottom is the [**Security**] button, and when it is selected, the **Object Security Definition** window displays.

On the left, we see the existing security classes, and to each one, we can assign **Full Access [/F]**, **View Only [/V]** or **Delete Access**. For now, set Full Access permission to the ADMIN class by selecting the ADMIN class and then clicking the [**Full Access [/F]**] button. The security definition displays on the right.



Save the panel, and run it. The security **Sign on** window displays:

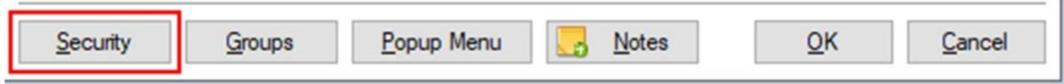


Depending on the user and the security applied to each panel or control, this window will prompt users to enter their User ID and password when they select an object that has access restrictions.

You can see that there is a **Change Password** check box. When it is selected, a **Password Change** window displays to allow users to change their password once they enter the system.

Note: When you create a new user, the password will initially be the same as the User ID. The new user will be prompted to change the password when logging on for the first time.

The NOMADS security system can be applied at the panel level, to different panel controls (Buttons, Images, List Boxes, etc.), data dictionaries, data sources, views, queries, reports, etc. In most cases, the **Security** button is located at the bottom of the object's properties window.



How does the security allocation work?

Remember that the security classification can be Full Access, View Only and No Access.

Let's look at a few examples to see how the panels and controls behave:

Example 1: Full access to the panel without access to the controls.
Result: Access to the panel is allowed. Controls are hidden/not displayed.

Example 2: View only access to the panel.
Result: Access to the panel is allowed. Controls are disabled.

Example 3: Full access to the panel with access to some controls.
Result: Access to the panel is allowed. Only the allowed controls are shown; the others are hidden.

Refer to **Security Manager** in the PxPlus Help documentation.

Using Passwords to Protect Programs and Tables

There are several places in PxPlus to assign passwords for restricting access to programs and data files at the language level and in the development environment. We can assign a password to a program or table so that trying to use it directly is not possible if the access key (password) is not known.

We can assign a password to our program. It can be loaded into memory and executed, but its content cannot be examined or modified without first providing the password. Once the password is known, it can be deleted, leaving the program unprotected again.

We can also protect our files and tables - from protecting your information so that it is not available to unauthorized people to key locking and encryption so that the information inside the file or table will not be recorded in a readable format until it is supplied the correct password.

To perform password protection of a program or file, we use the **PASSWORD** command.

Refer to the **PASSWORD** directive in the PxPlus Help documentation.

Example: Let's look at an example in a program:

```
10 ! Line 10
20 input "Name: ",name$
30 end

save "MYPROG"
! Program MYPROG saved, no password, after load you can see it:

Load "MYPROG"
LIST

10 ! Line 10
20 input "Name: ",name$
30 end
```

Let's protect it (we assume the program is in memory):

```
password "123"
save "MYPROG"
```

Reload the program, and now try to list it:

```
Load "MYPROG"
LIST

Error #52: Program is password protected
password "123"

LIST

10 ! Line 10
20 input "Name: ",name$
30 end
```

Let’s remove the password:

```
password
save "MYPROG"
```

You can use the "*" clause to indicate that all programs saved in that work session must be protected by the last specified key.

To protect a file or table, you must open the file and specify one of the following commands:

1. **PASSWORD** (channel) key$ REQUIRED FOR OPEN
2. **PASSWORD** (channel) key$ REQUIRED FOR WRITE
3. **PASSWORD** (channel) key$ REQUIRED FOR OPEN AND ON DATA
4. **PASSWORD** (channel) key$ REQUIRED FOR WRITE AND ON DATA

This will tell PxPlus to protect the file/table in four different ways:

1. Correct password is required to open/use the file/table.
2. Correct password is required to write; can be used/opened/read without password.
3. Correct password is required to open/use the file/table; data is encrypted.
4. Correct password required to write; can be used/opened/read without password; data is encrypted.

Example: Let's see an example to block the file "DATASHEETS.DAT", and it is necessary to provide the password "ABC" to access it:

```
! Check the next free channel -> CHANNEL
channel=unt
! Open the file only for us, nobody else can access it:
open lock(channel)"DATASHEETS.DAT"
! Let's activate the password access:
password (channel) "ABC" required for open
close(channel)
```

To use the file in question, it will ask us for a password. Try it directly from the console:

```
Open(99)"DATASHEETS.DAT"
```

The password assignment and/or access restriction operation requires the exclusive use of the file or table; therefore, the file or table must be opened with the command:

```
Open lock(channel)"file"
```

To open the file with a password, you must enter:

Open(99,key="ABC")"DATOSHIJOS.DAT"

Remember that, previously, we used the **KEY=** clause to read and/or write files or tables, not to open them. This is a new use of this clause.

This command indicates that the password must be supplied to open, read or write the file:

PASSWORD (channel) pswd$ REQUIRED FOR OPEN

This command indicates that the password must be supplied to write the file:

PASSWORD (channel) pswd$ REQUIRED FOR WRITE

This command indicates that the password must be supplied to open, read or write the file; the information saved in the file is encrypted:

PASSWORD (channel) pswd$ REQUIRED FOR OPEN AND ON DATA

This command indicates that the password must be supplied to write the file; the information saved in the file is encrypted:

PASSWORD (channel) pswd$ REQUIRED FOR WRITE AND ON DATA

Note: PxPlus uses the SHA256 security and cryptography standards to encrypt the information in files and tables. If you wish to use a different level of encryption, it is possible through the **'EA'** parameter.

To remove the password from a file or table, we must have access to the password, have exclusive access to the file or table and it ***should have no information***, and the following command is used:

PASSWORD (channel)"ABC" REMOVE

Note: This protection method is independent of the NOMADS security subsystem; it is possible to combine them.

Note: Keeping files password protected and perhaps encrypted will make it more difficult to use file management tools, since some of them will try to read the information directly (unprocessed), and this will not be possible if it is encrypted or protected.

14. The Final Steps

Publishing Our System

It is commonly accepted that a "system" is a collection of programs and tables that fulfill a specific function: control a warehouse, manage a payroll, maintain the income and expenses of a clinic, etc. This "system" can be something as simple as a single program and as complex as thousands of programs, tables, databases, objects, etc. distributed on a network throughout the country or across several continents.

In the case of PxPlus, the integrated development environment (IDE) allows panels and tables to be managed separately, allowing the tables to be accessed from different panels. The same happens with panels. There is no dependency relationship (by default) between the application windows and the tables; the user interface is separate from the storage layer.

However, through projects, you can group all these elements: NOMADS panels, programs, reports, data files and tables, data dictionaries, HTML pages and settings, configuration, colors, location, fonts, etc. When working with projects, it is necessary to manage the NOMADS panels and the creation of all these elements after opening and/or creating a project.

Note: It is not necessary to have a project. If you are clear about the elements that your system contains (NOMADS library, tables, data dictionaries, etc.), you can simply copy it to a particular folder or directory and take it to your client or production site.

Note: It is a great idea to have a certain order when storing the elements; for example, perhaps one folder for the programs, another for the files and another for the NOMADS libraries and panels.

PxPlus is an interpreted language, which means that by simply placing a program as its argument, it will be executed. Now, PxPlus can run independently on Windows and Linux platforms and through a Thin-Client on local or remote networks, or over the Web.

It is not the intention of this book to dig into the technical aspects of installing PxPlus on the different platforms. We will just do a brief review of what we need to do to put our system to work.

When we install our system, it is normal that there is a startup program, which prepares the entire work environment, adjusts directories and prefixes, changes parameters, colors, variables, etc.

To run our startup program, we just need to do something like:

```
/path/pxplus FIRSTPROGRAM.PXP
```

In addition, create a shortcut, link or whatever allows us to run the operating system for it.

/path/ will be the folder or directory where PxPlus was installed. We assume our START program is in the same folder. Remember that there is the possibility to run a program automatically when running PxPlus through the START_UP program, which will be run automatically when the PxPlus language is started.

If PxPlus runs independently and perhaps for a single user, say on an MS Windows machine, many times, by modifying the PxPlus shortcut to incorporate an argument or name of the program to be executed, we have something like this:

"E:\PXP\pxplus.exe" E:\CKX\UTIL\FIRSTPROGRAM.PXP"

This will run the PxPlus environment (which was installed on the E: drive, PXP folder) and then pass as an argument the program **FIRSTPROGRAM.PXP**, which is in the **E:\CKX\UTIL** folder. In the case of Linux, it is something similar if we are going to run PxPlus independently or single user, as it is also known.

So, the basic step is to copy the folder(s) where our system is located from the development computer to the computer where it will be run. There will be some possible additional steps in case you have to manage administrator permissions, routes, etc.

But, the only step required to run our system or program on a PxPlus already operational on a machine is to copy it and "link" or pass the name of the program to the PxPlus as an argument, as we have already seen. There are a series of additional arguments and parameters to condition the execution of PxPlus.

PxPlus Working in Web Mode

In the event that PxPlus is running together with a Web server and that the programs are served through a browser (such as Opera, Chrome, Edge, Safari, etc.), it involves first copying and linking the programs and files with our server and then configuring a series of parameters for our Web server (Apache, IIS, PxPlus EZWeb) and perhaps an additional module (**Webster+** or **iNOMADS**).

Note: PxPlus offers its own Web server called **EZWeb**, which offers the functionality required to meet the needs of PxPlus products that require Web access: **PxPlus Web Services**, **PxPlus Dashboard**, **PxPlus IDE**, etc. Yes, there is a version of the PxPlus development environment that runs on the Web, do you remember it?

PxPlus offers Web functionality or "communication" in three different instances:

- Directly through Web services (**PxPlus Web Services**), some of which are accessible through the ***plus/web/request** utility, where, basically, its application or program can interrogate websites for travel tickets, quotes, etc. using the native syntax and using the standard Web communication protocols, the **POST** method to send information to a server and **GET** to request it from the server.

- With the **Webster+** product, which allows the integration of NOMADS panels (file maintenance, query, among others) so that they are compatible with any Web browser without the client needing to install anything; that is, **Webster+** is an application that is installed between the Web server and your browser (it is installed on the server side, of course) to "translate" or facilitate the execution of information panels, maintenance panels and queries from your browser. **Webster+**, unlike **iNOMADS**, does this by "translating" the panels into a standard HTML5 format on the market. **Webster+** has a series of short codes to integrate between our PxPlus environment and Web-compatible programming. In addition, several of the outputs of the NOMADS graphical designer have been modified to directly generate panels compatible with the Web, generating HTML pages. **Webster+** offers a complete set of tools to facilitate the integration of our PxPlus programs and tables with the Web environment, such as access system, menus, utilities, security, etc.
- The other form of Web communication is through **iNOMADS**, which is a product that is installed in the form of an extension to the browser to "translate" NOMADS panels and queries so that they are executed within a browser. We could think of **iNOMADS** as a "converter" from the NOMADS panels for execution and that in a browser. Note that, although we have talked about "installing" in the browser, in reality, this process is done transparently and no installer or similar process is required, effectively being able to access your system from a café, a hotel or airport.

Refer to these pages in the PxPlus Help documentation: **EZWeb Server**, **PxPlus Web Services**, **Webster+** and **iNOMADS**.

To execute from our browser, we need a symbolic link, hyperlink or URL address, like this:

https://ourweb.com/?tx$=*program_initial

You must coordinate with the system administrator (server) to obtain more information about the address, syntax and/or way to access the system that is hosted on the Web server.

PxPlus Working in Network Mode

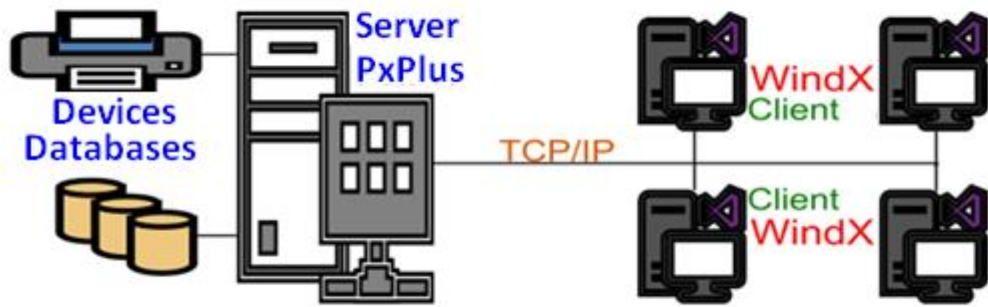
Network environments are traditionally known as "client-server" networks because there are one or more computers that offer their services or resources and are called "servers", and other computers (or nodes) that require and use the services or resources, and these are called "clients". The scope or size of the network can range from a couple of machines connected by a cable to hundreds of computers distributed throughout the world.

PxPlus offers a "client" called **WindX**, which is responsible for managing the communication between the "client" machine and the "server" machine.

In this arrangement, the server will be in charge of "storing", "managing" and "controlling" all access to the system, effectively residing the programs, files and tables (as well as the other components, such as data dictionaries, panels, objects, etc.) on the server's hard drive centrally. Each client will "point" to the same server, which will be in charge of providing the resources according to the access, security and permissions structure.

By default, this connection between WindX (the client) and PxPlus (the server) is made through the TCP/IP protocol commonly used between network machines (local or remote). We will not go deep into the use of TCP/IP. We will only touch on some terms that we will try to explain in due course.

Visually, it could look like something like this:



The "server" for us is simply a computer running PxPlus as a server, which will share "some resource" with other machines that are running another program called "WindX" and that request permission and access to some of the resources that the server has.

PxPlus offers several client-server connection methods: **Simple Client Server (CS), SSH,** and **NTHOST/NTSLAVE.** Alternatively, there is also the legacy Application Server. To facilitate this, PxPlus offers (via WindX) a connection tool called **Connection Manager**.

The three schemes work very well and can be chosen depending on the needs of each installation.

It is important to note that if we select a connection protocol, both the client and the server must be configured with the same protocol.

The server must run a non-interactive program that runs in the background as a service. This is a program that will be accepting requests automatically, continuously and transparently. It will be in charge of sending to the other party (the client) what it requests.

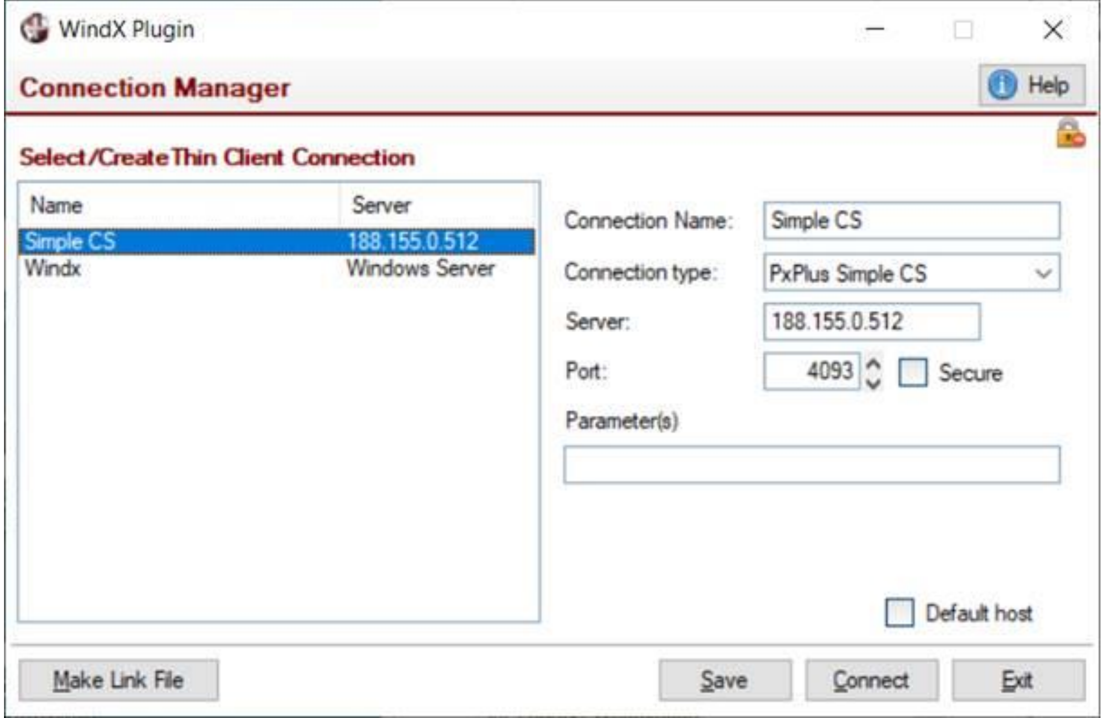
Although it sounds very complicated, in reality, it is telling PxPlus that it must execute a program and specify a clause that will indicate that this task is run in the background, something similar to this line:

```
./pxplus/pxplus -bkg "*plus/cs/host" -arg "4093"
```

In the case of Linux/Unix, as in the case of Windows, it is desirable that this action be carried out automatically every time the computer is initialized. There are several mechanisms for this.

The "client" part will be managed in all cases with WindX, a separate application (part of the PxPlus family, which is integrated into several products but is also available separately), and it will be our "tool" to establish the connection with the server.

We can use the **Connection Manager**, a powerful and flexible tool to configure the connection between the client and the server. The screen is similar to this:



Once the type of connection is established, we will need an IP address, a unique network identifier, required in TCP/IP (something like this: 192.168.1.100), and possibly a network port, socket or service number, which is an identifier, within that server, of the requested service.

Let's use a company as an analogy: 192.168.1.100 would be the company's phone number and the network port would be the extension number: Sales, Administration, Support, etc.

Example:

Connection Type: PxPlus Simple CS
Server: 192.168.1.100
Port: 20000

Note: Some communication "protocols", such as **Simple Client/Server**, come with a "default" port number, which, in their case, is 4093.

In many cases, it is possible to create a direct link from the Windows desktop for WindX to establish a direct connection to the server and launch an application:

```
c:\pxplus\pxplus.exe *plus\cs\client -arg 192.168.1.100;20000 "/usr/HOME.PXP"
```

Once the client (or clients) communication is set up, we should be able to make connections and take advantage of the best of both worlds.

Note: It is possible to have Windows clients running against a Linux server. PxPlus will take advantage of the processing power and storage of Linux and the graphics and versatility of the Windows environment for the user interface.

How does the Client-Server protocol work in PxPlus?

Once client-server communication has been established between the client machine (machine A, Windows) and the server (machine B, Linux), all file operations, processing logic and input/output operations will be executed against the server. If, on the client machine, we do:

```
Open(99)"TABLE"
```

WindX on machine A will ask machine B to locate the table "TABLE" and use channel 99 to establish a connection with it.

Likewise, if we do:

```
Call "/usr/pgms/payroll; routine",arg$
```

The procedure will be the same as the previous one. Machine B (server) must search in the path "/usr/pgms/" for a program called "payroll" and execute a routine called "routine", sending it an argument called "arg$". That execution ***will be performed on the server***, even though the request came from client A.

Note: Although machine A has the Windows operating system where the paths are something like "c:\path", in the client's WindX, it is ordered to execute "/usr/pgms/", a directory with Linux nomenclature, which is the server operating system.

Now, if we find an instruction:

```
Process "panel","library.en"
```

…which has graphical controls, such as buttons, images, list boxes, icons, etc., those instructions are executed locally on machine A. PxPlus detects that they are "presentation" commands and causes them to be executed locally by the WindX client.

Example:

If we use a printer:

```
Can_prt=unt
open(can_prt)"my_printer"
```

… this printer may possibly be a printer on the server, or it will be connected as a network to the server, but it will not be a local printer. It will not be connected via USB, for example, to the machine from which the request was made.

The "graphic" type instructions, such as graphic controls, multi_media, images, etc., are derived for execution locally, while all other instructions and commands will be executed on the server.

For many purposes, it is as if the operator of machine A is running a local copy of PxPlus and has a connection to the server to read/write files and run programs. This is a simplified view, but initially, we can think of it like this.

The same thing happens with parameters and configuration. If we have a problem that requires the activation of a parameter, it must be activated on the server, but in some cases, it must be activated on ***both instances***:

```
set_param 'WK'=1
if %WINDX then execute "[WDX]SET_PARAM 'WK'=1"
```

In this case, we are seeing that we have a variable that allows us to detect if it is running on a WindX client, and if so, the same command is executed to change the parameter locally.

We can specify if we need to open a resource that is located locally (that is, located on the client machine) using the **[WDX]** tag. If we prefix it to the name of the resource, PxPlus will open ***locally*** (on the client machine) the same.

Example:

```
channel=unt
! Open the "FILE" file on the client machine, not the server
open(channel)"[WDX]FILE"
! Open the Windows printing system of the client machine
open(99)"[WDX]*winprt*"
```

If we specify the file by prefixing the [WDX] tag, PxPlus will look for the file on the client machine, not on the server, so we can pass information from a local file to one on the server or vice versa:

```
channel_loc=unt
open(channel_loc)"[wdx]arch_local"
channel_rem=unt
open(channel_rem)"remote_arch"
read_loop:
read(channel_loc,end=end_cycle)record$
write(channel_rem)record$
goto read_loop
end_cycle:
msgbox "Operation completed","Notice"
close(channel_loc),(channel_rem)
exit
```

In this way, you can also specify the name of a printer, including conditioning it, so that if it is executed in a client-server manner, it is searched on the local machine (client):

```
Open(channel)local_tag$+printer_name$
```

The variable LOCAL _TAG$ could contain [WDX] (or be empty), and the variable printer_name$ the name of the printer.

A (more flexible) variant of the [WDX] tag is the [LCL] (local) tag, which is an improvement in case of portability.

Example:

```
Call "[WDX]program;routine"
```

That program works if it is run on a WindX station (on a client machine), but if you are working on a standalone machine (i.e. not in a client-server system), the above instruction will give an error, because it is not running on WindX and a WindX location was specified; however, this instruction will work properly both on a client-server machine and on an independent machine:

```
Call "[LCL]program;routine"
```

Using the client-server connection system with WindX as a Thin-Client, we will need to perform some operations locally on the client machine (**Example:** executing a command locally, not having it executed on the server). Likewise, we can define a local object, etc.

Some of the commands that can be executed natively are:

```
CALL "[LCL]..." ! Call a public program
KEYED "[LCL]..." ! Create a key file
DEF OBJECT id,"[LCL] ! Define an object
NEW ( "[LCL]..." ) ! Create a new instance of a class/object
DIRECT "[LCL]..." ! Create a direct file
OPEN (..) "[LCL]...." ! Open a file or device
DIRECTORY "[LCL]..." ! Create a directory
PROGRAM "[LCL]..." ! Define a program
ERASE "[LCL]..." ! Delete a file
REFILE "[LCL]..." ! Reset a file
EXECUTE "[LCL]..." ! Run a command
SERIAL "[LCL]..." ! Create a flat/text file
INDEXED "[LCL]…" ! Create an indexed file
INVOKE "[LCL]..." ! Run an operating system command
```

There is also a utility, ***windx.utl**, to help us manage some situations that arise in client-server connections.

Example:

Let's take a look at some examples:

```
! Change to the specified directory on the local machine
CALL "[WDX]*WindX.utl;CWDIR",dir$
! Get the IP address of the local machine
CALL "[WDX]*WindX.utl;GET_ADDR",X$
! Open a file selection/input box on the local machine
CALL "[WDX]*WindX.utl;GET_FILE_BOX",path$,dir$,title$,ex_list$,def_ex
! Create a new WindX session (duplicating the window and being able to work in two or more
programs simultaneously
CALL "*WindX.utl;SPAWN",X$,I$,F$,HideClient,HideServer
```

Note: In the latter case (to create a new instance of WindX), there is ***no*** [WDX] tag at the beginning of the program).

For more detailed information on this ***WindX.utl** utility and the commands that can be executed locally, refer to ***WindX.utl** in the PxPlus Help documentation.

There are some considerations, which are beyond the scope of this book, about the best way to program in client-server mode, especially in cases where the client is located far (physically) from the server. This is not so much because of the physical distance, but rather because of the connection speed that may possibly be lower than the local connection; for example, if you have panels that show thousands of data, may require a progressive load, or some other method that allows us to speed up the operation at remote clients.

Handling Programs and Files

In addition to the already known commands, PxPlus has a series of utility programs to help us manipulate programs.

This table lists some of them:

Command	Description
LOAD	Load a program from disk to memory.
EDIT	Edit a program line in memory.
SAVE	Save a program from memory to disk.
LIST *or* **/**	List a program in memory by screen.
PASSWORD	Protect a program with a password.
it	Run the program editor.
START	Delete a program from memory and start from scratch.
clip	Copy part of a program to the Clipboard.
f	Search the program in memory.
gui	Run the utility program's menu.
help	Shows the online Help system.
kill	Kill active PxPlus system processes.
ls	Shows the contents of the directory.
lv	Shows program variables.
nom	Run the NOMADS graphical designer.
pastte	Paste program lines from the Clipboard to memory.
pkg	Displays PxPlus activation information.
rw	Run the Report Writer.
sa	Runs a PxPlus system scan in case of errors.
tasks	Displays information about active PxPlus tasks.
windx	Runs the WindX Thin-Client.

Some of these commands require more detailed study, which is beyond the scope of this book.

Example: The graphical user interface command shows us a complete menu of utility programs:

We can see and manipulate files, programs and directories, configure part of the system, rebuild files, show available images and fonts, among many other things.

Remember that you have program libraries, which are used to group multiple programs in a single file, using the **[LIB]** tag.

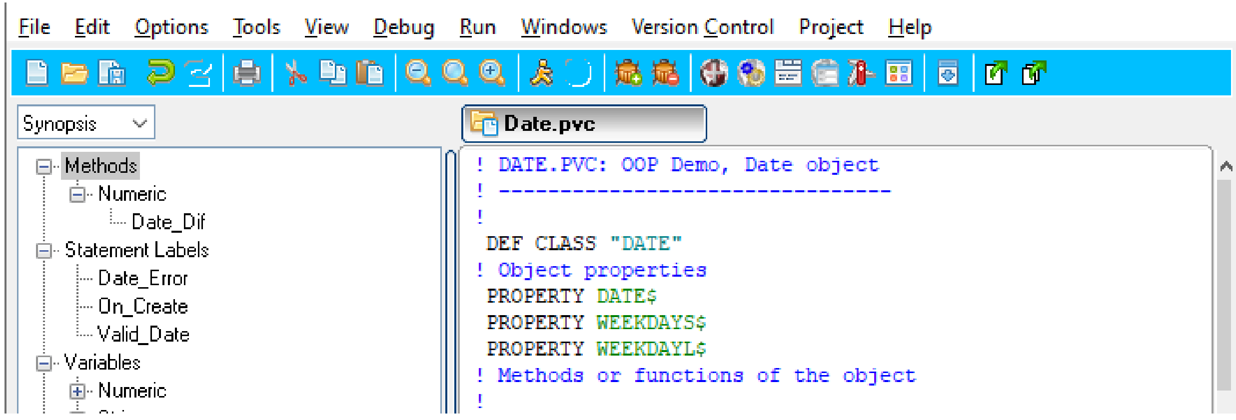
Fixing Errors, Debugging and Testing Programs

The most basic form of an error occurs on the console Command line where we can encounter a situation like this:

```
Introduction to PxPlus
Program  Utilities  Edit  Debug  Help

PxPlus-2024 Web (Ver:21.00/MS-WINDOWS) Serno:XXXX-XXX-XXXXXXX
(c) Copyright 2005-2024 PVX Plus Technologies Ltd. (All rights reserved)
  Website: http://www.pvxplus.com
->pirnt "A line with error"
Error #20: Syntax error ..."A line wi...
->
```

In that case, by pressing the **Up Arrow** keyboard button, we can call the line again and correct "PIRNT" to "PRINT".

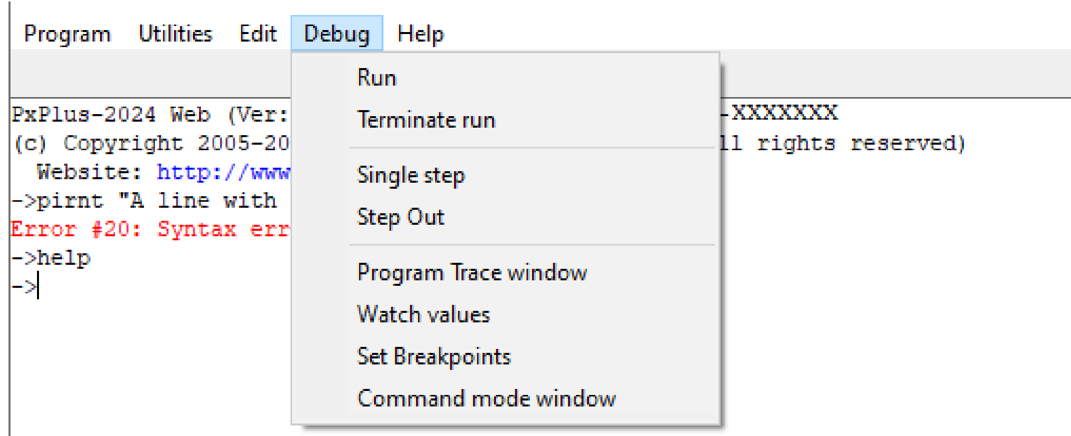
In the case of the graphical program editor (Integrated Toolkit - *IT), it has, in addition to the program synopsis (shown below), the ability to request help online, to directly insert a reserved word, and to provide an immediate assistance function where it tells us the syntax and elements of the function or command that we are entering:




```
DTE( ) Convert Date
1. Convert Numeric Date:  DTE(jul_date[,time][:fmt$][,ERR=stmtref])
2. Convert Date String:  DTE(date$[:fmt$][,ERR=stmtref])
a=dte (
```

However, these aids allow you to filter or debug the program for syntax or writing errors, but the logical part, the debugging part, sometimes requires other tools; for example, step-by-step execution, which is used to execute a program or routine one instruction at a time.

You can do this from console mode by entering a **.** (*period*) and pressing [**Enter**] or through the [**Debug**] option of the console:



The step-by-step execution is complemented by these command variations:

Enter	Result
.	Execute a program line and return to console mode
.n	Execute **n** lines of program and return to console mode
..	Executes the full cycle (FOR-NEXT, GOSUB-RETURN or CALL-PERFORM-EXIT)
...	End the entire public program and return (CALL-PERFORM..EXIT)
;	Execute step-by-step in combined lines
;n	Execute **n** steps in combined lines

Note: You can press [**F2**] to repeat the last execution command step-by-step.

In addition to this step-by-step execution, PxPlus incorporates a plotted output where the program lines that are executed are shown on the screen as they are executed. This is done through the **SETTRACE** and **ENDTRACE** commands. There are a number of modifiers for these commands.

Refer to the **SETTRACE** and **ENDTRACE** directives in the PxPlus Help documentation.

As a complement to the debugging aids, there are four additional options in the **Debug** menu: **Program Trace window**, **Watch values**, **Set Breakpoints** and **Command mode window**.

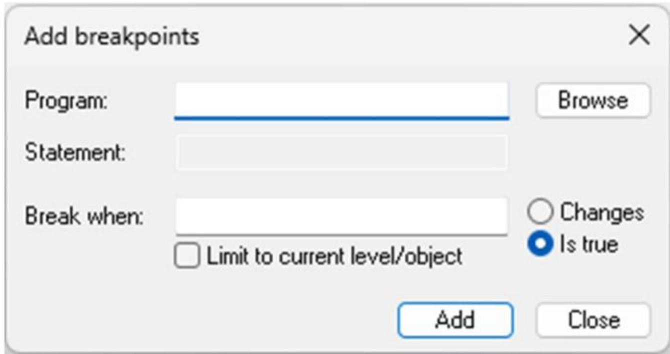
The **Program Trace window** records the program instructions that are executed in a separate window. This window has a series of options to log different events, such as errors, object property changes, file opens, program loads, SQL commands, I/O operations, etc. It also allows you to set a maximum size, search within the trace, save or copy it, etc.

The **Watch values** window allows you to specify variables and structures to monitor their changes. It also has some options like the previous window, although given its characteristics, these are minor.

The third option, **Set breakpoints**, allows you to define places where the program interrupts its execution (perhaps to review conditions or values. This can also be done through the **ESCAPE** command, which interrupts the execution of a program where this command is found. The difference

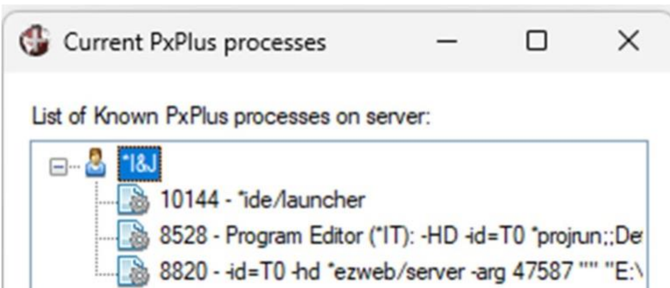
is that these interruptions do not include modification of the program.

Apart from being able to specify in which program and instruction the execution will be interrupted, a condition can be placed (if SALARY<0, the program is interrupted or if a variable changes its value).



The last option is the **Command mode window** where a replica window of the program we run will be created where we can perform certain commands and checks without disrupting the regular operation of the program.

In addition, the developer has other powerful tools. The **info** command shows expanded information about a program or file. The **users** command provides information about the user's processes:



The powerful and versatile **dbg** command allows us to connect to another user session and remotely obtain extended information about another user's work session:

```
Program  Utilities  Edit  Debug  Help

Commands are:

Connect procid            : Connect to process
Disconnect                : Disconnect from process
Tasks                     : List known processes (Tasks * includes program)
Halt                      : Halt/Suspend the process
Go                        : Resume the process
Execute xxxx              : Execute command within process
List [from [to]]          : List statement (option from/to)
Kill                      : Terminate/kill the process
Print xxx                 : Evaluate and show xxxx from process
Files                     : Show files
Stack                     : Show Call Stack
Where                     : Where is the process
. xxx                     : Step through program where xxx is:
                          :  Program - Till program change
                          :  Around  - Around next gosub/call/etc
```

As you can see, PxPlus offers a full set of tools, which, together with version control, allows developers to detect and eliminate any condition or situation that prevents the normal execution of their programs and to guarantee their clients and users a high-quality product.

15. Appendix

Error Codes in PxPlus

Whether you are entering a command at the PxPlus prompt or entering program code using a full-screen editor, PxPlus may detect various errors due to problems with program logic, invalid data or status conditions such as a duplicate record.

All errors detected by PxPlus have a numeric code associated with them. The value of this code represents the type of error. Error codes ranging from 0 to 255 represent PxPlus specific messages. Error codes starting from 256 are operating system errors.

When an error is detected, the error code and its associated message text is displayed.

Example:

```
print hello world
Error #20: Syntax error ...world
print "hello world"
hello world
```

The resulting Error #20 indicates a syntax error.

For a complete list, refer to **Error Codes and Messages** in the PxPlus Help documentation.

Examples of PxPlus Programs and Useful Routines

Program (game) to guess a number between 1 and 99:

```
! Counter for the TRYs
TIMES=0
! SECRET Number
SECRET=INT(RND*100)
! ASK Label
ASK:
PRINT "Guess a number between 1 and 99"
! Increase the TIMES counter
TIMES++
PRINT TIMES," chance? ",
INPUT GUESS
! Check the GUESS is a positive, integer number
IF GUESS<>INT(ABS(GUESS)) THEN PRINT "Enter a positive, integer number"; GOTO
ASK
! Check the GUESS is between 1 and 99
IF GUESS<1 OR GUESS>99 THEN PRINT "Enter a number between 1 and 99"; GOTO
ASK
! If the GUESS = SECRET we have a WINNER!
IF GUESS=SECRET THEN GOTO WINNER
! Provide some clues
IF GUESS<SECRET THEN PRINT "You're lower" ELSE PRINT "You're higher"
IF TIMES>=5 THEN GOTO LOSER
PRINT 5-TIMES," chances left"
! Back to keep the loop
GOTO ASK
! WINNING Routine
WINNER:
PRINT "You got it!!"
END
! LOSING Routine
LOSER:
PRINT "You wasted your 5 opportunities!"
PRINT "The secret number was: ",SECRET
END
```

Data Manipulation Routines

Convert from numeric to literal:

```
number=1900
text$=str(number)
```

Convert from literal to numeric:

```
text$="2450"
number=num(text$)
```

Add a validation routine in case of error (if it cannot do the conversion), branch to that routine:

```
text$="I like coding"
number=num(text$,err=cannot_convert)
...
cannot_convert:
number=0
```

Set only two decimals in a given number:

```
number=123.3
! Convert the possible decimals to two whole digits
number=int(number*100)
! Restore the number to the original amount
! Add an output mask/format
PRINT number/100:"##,###,##0.00"
    123.30

! Can also use ? to print
? number/100:"##,###,##0.00"
    123.30
```

Add a bonus of $0 if the person has no sons, $100 if have one son and $200 if have two or more sons (do not use the IF directive):

```
bonus=(sons<1)*0+(sons=1)*100+(sons>1)*200
```

If the condition sons<1 is true, becomes 1; else will be 0. In addition, when evaluated, if sons=1 (that is, the person has one kid), then sons=1 will be true, so becomes 1. If sons is different than 1, the expression will become 0; and at last, when sons>1 is true (the person has more than one kid), the expression itself becomes 1.

Example:

```
sons=0
bonus=(sons<1)*0 + (sons=1)*100 + (sons>1)*200
          1     *0 +    0     *100 +    0     *200 = 0+0+0
```

It is possible to have only one of the expressions (conditions) as TRUE at run time; the others must be zero.

```
sons=1
bonus=(sons<1)*0 + (sons=1)*100 + (sons>1)*200
          0     *0 +    1     *100 +    0     *200 = 0+100+0
```

The second condition is the true one (sons=1) and multiplies by 1x100; first and third conditions will be 0 (false).

```
sons=2
bonus=(sons<1)*0 + (sons=1)*100 + (sons>1)*200
          0     *0 +    0     *100 +    1     *200 = 0+0+200
```

The third condition is the true one (sons>1) and multiplies by 1x200; the first and second conditions will be 0 (false).

Strip the negative sign and decimal digits to a number:

```
number=-2.3
number=abs(int(number))
print number
2

number=0.5
number=abs(int(number))
print number
0

number=-100
number=abs(int(number))
print number
100
```

Find out how many digits in a number (whole):

```
number=100
number=abs(int(number))
print ept(number)
3

number=56873
number=abs(int(number))
print ept(number)
5
```

Find out how many characters in a text (literal):

```
text$="A test"
print len(text$)
6

text$="The base system bundles the PxPlus engine with a suite of highly productive
development tools and application-level utilities. It offers a robust native database, convenient
access to many external technologies, and true platform-independence--no matter where you
develop PxPlus applications, they can be implemented on any other supported platform."
print len(text$)
349
```

Convert text (literal) text$="This IS A Test" **to lowercase, uppercase:**

Using **CVS()** function:	print cvs(text$,4) **THIS IS A TEST**
Using **UCS()** function:	print ucs(text$) **THIS IS A TEST**
Using **CVS()** function:	print cvs(text$,8) **this is a test**
Using **LCS()** function:	print lcs(text$) **this is a test**

Invert text (begin from the last character until put the first character as first on the new literal):

```
TEXT$="This is a test"
FINAL_TEXT$=""
! Find out how many characters the text has
CHARS=LEN(TEXT$)
! Let's do a loop from the number of chars descending to 1
FOR I=CHARS TO 1 STEP -1
    ! Will be adding from the last character until find the first to the new literal
    ! TEXT_FINAL$
    FINAL_TEXT$+=TEXT$(I,1)
! End of loop
NEXT I
PRINT FINAL_TEXT$

tset a si sihT
```

Add "*" if the text is shorter than 10 characters:

```
print pad("Prueba",10,"*")
Prueba****
print pad("Test",10,"*")
Test******
print pad("A",10,"*")
A*********
```

Drop/erase some characters from the text:

```
! Erase the letter "a" from the text:
print stp("This is a test",3,"a")
This is  test

! Erase the letter "e" from the text:
print stp("This is a test",3,"e")
This is a tst

! Notice that E and e are DIFFERENT characters for PxPlus
! Erase all the characters after the "*", in this case, the vowels
print stp("This is a test",3,*"aeiou")
Ths s  tst

! You can use variables too: Erase the letter "a" from text$ variable:
text$="This is a test"
text$=stp(text$,3,"a")
print text$
This is  test
```

Erase characters from text or literal:

```
text$="***Px**Plus*****"
! Erase the "*" at the beginning of the variable
print stp(text$,0,"*")
Px**Plus*****

! Erase the "*" at the end of the variable
print stp(text$,1,"*")
***Px**Plus

! Erase the "*" at the beginning and end of the variable
print stp(text$,2,"*")
Px**Plus

! Erase all the "*" from the variable
print stp(text$,3,"*")
PxPlus
```

Erase the formatting/mask from a number. If we have, for example, a phone number such as (XXXXX) XXX-XX and want to extract only the number (the X's), we can do:

```
! Phone number
tel$="(58412) 228-8791"
! Strip the formatting
tel$=stp(tel$:"(00000) 000-0000")
print tel$
584122288791
```

Find out if one character (from a list) exists in a text/literal. Find out if the text$ variable contains at least one of these characters: "xyz":

```
! This is the content/value of the text$ variable
text$="This is a test using the POS function"
! Find the POSition of ANY of the characters "xyz" (lowercase only)
num_xyz=pos("xyz":text$)
! Let's check if any of them exists
print num_xyz
0
! No occurrences of xyz
! Let's change the text to add a z
text$="Thiz is a test using the POS function"
! Let's check the text$ variable again
num_xyz=pos("xyz":text$)
! Let's check if found something
print num_xyz
4
! Yes, at position 4, there is one of the characters in the list
```

Convert a text, given an equivalent or translation table. Let's suppose we want to make a simple conversion where each digit 1 - 5 has an equivalent letter A – E (1=A, 2=B, 3=C, 4=D, 5=E):

```
print tbl("212","12345","ABCDE")
BAB

print tbl("543","12345","ABCDE")
EDC

print tbl("123","12345","ABCDE")
ABC
```

We can define a translation table using a position where every character of a list will correspond to a different text:

```
freq$="D"
print
tbl(pos(freq$="DWMA"),"Err","Daily","Weekly","Monthly","Annually")
Daily

freq$="M"
print
tbl(pos(freq$="DWMA"),"Err","Daily","Weekly","Monthly","Annually")
Monthly

freq$="W"
print
tbl(pos(freq$="DWMA"),"Err","Daily","Weekly","Monthly","Annually")
Weekly

freq$="X"
print
tbl(pos(freq$="DWMA"),"Err","Daily","Weekly","Monthly","Annually")
Err
```

Date validation routine:

```
INPUT "Day: ",D
INPUT "Month: ",M
INPUT "Year: ",Y
FEC_JUL=JUL(Y,M,D,ERR=BAD_DATE)
END

BAD_DATE:
MSGBOX "Invalid Date"
```

A Complete Routine for Table/File Handling

As we must remember, in PxPlus, a file or table is a "storage" of data, normally organized under a key. Although there are variations of it, we are going to study the example at hand without going into many alternative details.

When we create a table, we can do it in several ways: from **Data Dictionary Maintenance** to console mode. A table basically has a physical name, a key or sorting key, and a possible list of fields.

Key 0	Records →			
Field1	Field2	Field3	Field4	Record #1
Field1	Field2	Field3	Field4	Record #2
Field1	Field2	Field3	Field4	Record #3
Field1	Field2	Field3	Field4	Record #4

The function of the key is to have the records organized (ordered). **Example:** With a catalog of magazines, we could define the year and month of publication as an ordering method to have easy access to them, even if we have many magazines.

In the case of commercial and administrative applications, it is normal to maintain several simultaneous sorting methods. PxPlus allows the ordered reading of information using several sorting criteria; for example, sorting journals by publication date, which can also be sorted by type of article, author, subject or section.

Exercise: Sorting Data in a Table

This exercise seeks to clarify this scenario through an example. We will use a table for a pet store where the information will be saved with a unique registration code for each animal, the name of the pet, the type of animal (dog, cat, fish or bird), the age of the animal, and finally, the owner's name. It will be something similar to this:

Field	Length
CODE	4
NAME	20
TYPE	20
AGE	2
OWNER	32

The sort criteria will be the code since it is unique, shorter and easier to use; however, it is also required that pets can be sorted by name, facilitating the work of system users. The vet wants to see the data by the type of animal. To facilitate contact, it is desirable to issue a list ordered by the name of the animal's owner.

As we have previously seen in this book, when there are possibilities of duplication (such as in names), it is suggested to sort first by name and then by code. This means if there are several identical names, the ordering of those names will be according to the code.

Example:

John Smith, 0010
John Smith, 0054
John Smith, 0215

So far, we have four sorting criteria:

CODE	(Key 0 primary)
NAME+CODE	(Key alternate 1)
TYPE+NAME+CODE	(Key alternate 2)
OWNER+CODE	(Key alternate 3)

With these keys defined, we tell PxPlus which sorting method we want to use at the time of a reading (we will see some examples later).

To facilitate the example, we will create a table called **PETS.DAT**. This table will contain five fields (CODE, NAME, TYPE, AGE and OWNER) and four keys (sorting criteria), according to the information we have just seen.

We need to add the following information to the table:

Code	Name	Type	Age	Owner
0001	Fido	Dog	3	Miguel Rios
0002	Mickey	Cat	8	Ana Rojas
0003	Fifi	Dog	5	Lucia Gonzalez
0004	Coco	Dog	8	Pedro Perez
0005	Mosquito	Cat	5	Lucia Beltran
0010	Ralph	Bird	2	Ana Rojas
0011	Cone	Bird	5	Luis Medina
0015	Cruzado	Dog	7	Angel Moya
0017	Felix	Cat	4	Lucia Tenia
0020	Andrea	Cat	7	Pablo Marmol
0021	Tornillo	Fish	2	Angela Lopez
0024	Nunca	Cat	4	Miguel Martinez
0025	Torombolo	Fish	1	Monica Rosales
0026	Minerva	Dog	11	Andrea Rojas
0028	Lulu	Bird	3	Berta Bartolino
0030	Gladiolo	Bird	4	Carlos Gonzalez
0031	Rapaz	Fish	3	Maria Rodriguez
0033	Hercules	Fish	6	Maria Marino
0034	Lobezno	Cat	8	Jose Tadeo

To process/read our table, we can use several methods, both at the programmatic level and at the command level. We are going to use two examples simultaneously.

Note: Both examples have the same functionality. We just want to take advantage of the opportunity to show two ways to reach the same result when manipulating files/tables.

Example 1:

```
OPEN (1,IOL=*)"PETS.DAT"
LOOP:
READ (1,END=END_LOOP)
PRINT CODE$,@(10),NAME$,@(20),TYPE$,@(28),AGE$,@(34),OWNER$
GOTO LOOP
END_LOOP:
CLOSE (1)
```

Example 2:

```
SELECT * FROM "PETS.DAT"
PRINT CODE$,@(10),NAME$,@(20),TYPE$,@(28),AGE$,@(34),OWNER$
NEXT RECORD
```

Both routines do the same thing. They read the **PETS.DAT** table and display the information in a columnar form.

Since we have not specified any sorting criteria, the primary key is assumed (sorted by code).

Both of these examples simply read a table and display the information according to the default key (primary key or Key 0).

If we want the criterion to be different, for example, so that the ordering is by the name of the pet owner (alternate key number 3), we must modify the reading program as:

Example 1:

```
OPEN (1,IOL=*)"PETS.DAT"
LOOP:
READ (1,END=END_LOOP,KNO=3)
PRINT CODE$,@(10),NAME$,@(20),TYPE$,@(28),AGE$,@(34),OWNER$
GOTO LOOP
END_LOOP:
CLOSE (1)
```

Note: Only the **READ** line was modified to add the clause, KNO=3. That is, we go from this:

```
READ (1,END=END_LOOP)
```

To this:

```
READ (1,END=END_LOOP,KNO=3)
```

The only thing we are doing is indicating to PxPlus that the reading criterion (or access key) is 3. We can interpret the KNO clause as "Key Number". If we enter 0, it will use the primary key. If we enter 1, we will use the first alternate key, and so on.

The result of the program will be this (note that it is sorted by the last column, column 5):

```
0002        Mickey      Cat      8     Ana Rojas
0010        Ralph       Bird     2     Ana Rojas
0026        Minerva     Dog     11     Andrea Rojas
0015        Cruzado     Dog      7     Angel Moya
0021        Tornillo    Fish     2     Angela LoFish
0028        Lulu        Bird     3     Berta Bartolino
0030        Gladiolo    Bird     4     Carlos Gonzalez
0034        Lobezno     Cat      8     Jose Tadeo
0005        Mosquito    Cat      5     Lucia Beltran
0003        Fifi        Dog      5     Lucia Gonzalez
0017        Felix       Cat      4     Lucia Tenia
0011        Cone        Bird     5     Luis Medina
0033        Hercules    Fish     6     Maria Marino
0031        Rapaz       Fish     3     Maria Rodriguez
0024        Nunca       Cat      4     Miguel Martinez
0001        Fido        Dog      3     Miguel Rios
0025        Torombolo   Fish     1     Monica Rosales
0020        Andrea      Cat      7     Pablo Marmol
0004        Coco        Dog      8     Pedro Perez
```

Let's see the same modification using the second example program:

Example 2:

```
SELECT * FROM "PETS.DAT",KNO=3
PRINT CODE$,@(10),NAME$,@(20),TYPE$,@(28),AGE$,@(34),OWNER$
NEXT RECORD
```

Note: Remember that both programs ***are equivalent***. We are only showing two ways to reach the same result.

However, now we can make use of an additional function of the **SELECT** structure that does not exist directly with the first program. For example, if we want to filter the type of pet so that it shows only dogs, we must modify the program like this:

Example 1:

```
OPEN (1,IOL=*)"PETS.DAT"
LOOP:
READ (1,END=END_LOOP,KNO=3)
! The next condition checks that TYPE$ only has "Dog", and if it doesn't, then branch back to
the label LOOP. If the content is "Dog", then will be shown. If TYPE$<>"Dog", then goto LOOP
PRINT CODE$,@(10),NAME$,@(20),TYPE$,@(28),AGE$,@(34),OWNER$
GOTO LOOP
END_LOOP:
CLOSE (1)
```

The result will be similar to this:

```
0026    Minerva   Dog   11   Andrea Rojas
0015    Cruzado   Dog   7    Angel Moya
0003    Fifi      Dog   5    Lucia Gonzalez
0001    Fido      Dog   3    Miguel Rios
0004    Coco      Dog   8    Pedro Perez
```

Now, using the **SELECT** directive is easier:

Example 2:

```
SELECT * FROM "PETS.DAT",KNO=3 WHERE TYPE$="Dog"
PRINT CODE$,@(10),NAME$,@(20),TYPE$,@(28),AGE$,@(34),OWNER$
NEXT RECORD
```

Note: Both programs are equivalent and should give the same result.

Internal vs. External Keys

When we show the visual representation of a table, we show the following drawing:

Key 0	Records →			
Field1	Field2	Field3	Field4	Record #1
Field1	Field2	Field3	Field4	Record #2
Field1	Field2	Field3	Field4	Record #3
Field1	Field2	Field3	Field4	Record #4

But, it can also be represented like this:

Key 0	Records →				
	Field1	Field2	Field3	Field4	Record #1
	Field1	Field2	Field3	Field4	Record #2
	Field1	Field2	Field3	Field4	Record #3
	Field1	Field2	Field3	Field4	Record #4

Or, it can be in this form, too (there are many other ways):

Key 0	Records →	Key 1		
Field1	Field2	Field3	Field4	Record #1
Field1	Field2	Field3	Field4	Record #2
Field1	Field2	Field3	Field4	Record #3
Field1	Field2	Field3	Field4	Record #4

In the first example, we have one internal key. In the second example, we have one external key, and in the third example, we have two internal keys.

What does this actually mean? Nothing special; in reality, an ***internal key*** indicates that it is made up of one of the IOLIST fields, which is one of these variables:

    CODE$, NAME$, TYPE$, AGE$, OWNER$

In the case of a key/foreign key, it is not part of the data and is stored in another part of the file. It could be information that is not found in the file, such as the identification (DNI or SSN) of the OWNER.

File creation differs slightly.

Create file with internal key:

    Keyed "FILE", 10,0,100

Create file with foreign key:

    Keyed "FILE", [1:1:10],0,100

Note: None of this is significant to us. You will see that it is possible to have files with external and internal keys simultaneously. The purpose of this comment is to know that both cases exist.

How to Read a Table from Last to First Record (Reverse Order)

(without having a key defined for that)

Taking advantage of the example table and using the functions **KEL** (locate last key of a table) and **KEP** (previous key), we will see a way to read a file from back to front, which is a REVERSE ordering to that of the key.

Basically, we locate the last key and read the previous records until we reach the end of the table (which, in this case, the end will be the first record of the table).

Note: PxPlus allows you to incorporate keys dynamically through the **ADD INDEX** command or directly from the Data Dictionary definition tool. We want to show you how to do it if you want to do it manually.

```
! Open the table
OPEN (1,IOL=*)"PETS.DAT"
! Find the last key and store it on K$
LET K$=KEL(1)
! Let's go to the reading label to use those lines (and not duplicate them)
GOTO READING

! Define a LOOP label
LOOP:
! Read the previous key and store it on K$
LET K$=KEP(1,END=END_LOOP)
! Define a READING label
READING:
! Read the record's info
READ (1,KEY=K$)
! Show date at screen
PRINT CODE$,@(10),NAME$,@(20),TYPE$,@(28),AGE$,@(34),OWNER$
! Go to read the next table's record
GOTO LOOP

! Define an END_LOOP label
END_LOOP:
! Close table's channel
CLOSE (1)
```

16. Conclusion

Congratulations! You have now completed the initial introduction to PxPlus!

You have worked hard to get through this initial training and have now gained a better understanding of this exceptionally powerful and versatile development environment. You have been introduced to many of the features within PxPlus, including the PxPlus IDE, NOMADS, Data Dictionary, Queries, Report Writer, and Webster+ … just to name a few.

Now is the time to take advantage of the information you have been given. It is essential to review what you have learned and experiment with the different functions and components to build your confidence with using PxPlus. It is recommended that you go through this process one step at a time, starting with the basics first, and then adding on more functionality. With practice and experience, you will be able to enjoy all the benefits that PxPlus has to offer.

Our goal is to continue to develop and provide additional training material on the topics presented in this book.

We value your opinion and would love to hear your feedback on the topics presented in this book or on topics you would like to see in the future to help us ensure that we provide the information you need.

This is only the beginning. Stay tuned!

Send us an email with your comments and suggestions to **info@pvxplus.com**.

ISBN: 978-980-18-4638-3