ProvideX

Version 8.30

User's Guide

sage

The capabilities, system requirements and / or compatibility with third-party products described herein are subject to change without notice. Refer to the Sage ProvideX website www.pvx.com for current information.

Publication Release: V8.30
May 27, 2009

Contents

6. Graphical User Interfaces

7. Printing

8. Client-Server

9. External Components

10. Data Integration

11. Object-Oriented ProvideX

A. Appendix

Index

Preface

The ***ProvideX User's guide*** contains general information on the ProvideX system, its use, and the basic concepts required to develop ProvideX applications. Although this volume is intended for application programmers and analysts, it does not try to explain how to design or create applications – instead, it lays out the fundamental building blocks necessary to create programs in ProvideX.

Refer to the ProvideX *Language Reference* for descriptive lists of ProvideX keywords and concepts: directives, functions, system variables, mnemonics, parameters, specialty files, reserved words and system limitations. Other ProvideX products (i.e., *NOMADS*, *WindX*, *JavX*, the *ODBC Driver*, the *WebServer*, and the *Application Server*) are fully documented in separate publications. Rather than reproduce existing material, references to these publications are supplied where applicable.

Using this Documentation

This documentation is designed for both viewing and printing via **Acrobat® Reader**. Click **Help > Reader Guide** on the menu bar to learn how to display, copy, search, and print PDF documentation. While there are several ways to navigate the contents of a PDF-based document, the following methods are highly effective, and are consistent with other documentation distributed by ProvideX:

Bookmarks

The list of bookmarks, displayed on the *left side* of the Acrobat window, serves as a hyperlinked *table of contents*. Bookmarks are displayed in a hierarchy where subordinate headings appear indented below main headings. When subordinates are hidden or *collapsed*, a plus sign (in Windows) or triangle (in Mac OS) will appear next to the main heading. Simply click on the *plus sign* or *triangle* to display all collapsed headings.

Cross-References

Blue hyperlinks appear throughout this document wherever one section cross-references another. They also appear in the form of hyperlists; such as the *Table of Contents*, the *Index*, and the linked tables placed at the beginning of some chapters.

The mouse pointer looks like an index finger when it is positioned over a linked cross-reference — simply click to activate the link. For example, "**Using this Documentation**" is hyperlinked back to the beginning of this section.

PDF Navigation Tips: The *chapter name* at the top left corner of the page head can be used as a hyperlink to the beginning of the chapter. Use the page-up / down / back / forward buttons ▼ ▲ ◀ ▶ to move one page at a time. Use the *Back* button ◀◀ to jump to the *previous view.*

Conventions

The following syntax items are used in this documentation to illustrate the format of program statements in ProvideX.

...	Dots indicate the continuation of a list of elements.
[ ]	Square brackets enclose optional elements in the format. For example, in **ABS**(*num*[,**ERR=***stmtref*]) you can omit the **ERR=***stmtref* portion of the statement as in **ABS**(**X-Y**). (Exceptions are noted for individual commands where the brackets are "real"; i.e., part of the syntax.)
{ }	Curly brackets enclose a list of elements in syntax formats where it is mandatory to select one item. For example, with {**YES** | **NO**}, you must select either **YES** or **NO**. In descriptions in this manual, they denote {bitmap / icon} buttons. (Exceptions are noted for individual commands where the brackets are "real"; i.e., part of the syntax.)
|	Vertical bars (pipes) separate a choices; e.g., {**YES** | **NO**}.
chan	Channel or logical file number. It must be an integer between 0 and 127. This identifies the channel to which your directive applies; e.g., `CLOSE (14)`. • Channel zero (0) is the console. If you omit the channel, the system defaults to 0 (the console). • Channels 1 to 63 are commonly used for local files. • Channels 64 to 127 are used for global files. • ***Exception***: In extended file mode (**'XF'** system parameter) the channels range from 0-32767 for local files, and 32768-65000 for global files.
col,ln, wth,ht	Position / coordinates. Numeric expressions. Column and line coordinates for top left corner, width in number of columns, and height in number of lines.
ctrlopt, fileopt	Optional syntax elements — three-character codes followed by an equals sign and argument (`DOM=3250`).

stmtref Statement reference. This can be either the line label or line number of a statement in the current program. Line numbers must be in the range of 0 to 64999.

If your given line number does not exist, ProvideX goes to the statement with the next higher line number. For example, if line 1000 doesn't exist and 1010 is the next line number, then for `GOTO 1000` ProvideX will go to 1010 and proceed with execution from there.

Exception: ProvideX verifies the existence of an IOList and *stmtref* for **IOL=***stmtref*. It does not proceed to the next higher statement number.

varlist List of comma-separated variables. Typically, a mix of string and/or numeric variables is acceptable; e.g., `DEPT,ITEM,DESC$...` (See individual directives for restrictions.)

Numeric Expressions and Variables

When a syntax format in this manual includes a ***numeric*** variable like *chan, index* or *num* (lowercase), you can normally substitute a ***numeric expression*** consisting of variables, literals, functions, and operators. For instance, your value could be something like `HFN` or `4` or `NUM(A1$)*3-2`. (`NUM` in upper case is the function.) When numeric variables are used in numeric expressions, subscripts are allowed; e.g., `COST[4]`.

Example:

To apply the format **FOR** *var***=***first* **TO** *last***[STEP** *val***] ...**

```
FOR I=1 TO 10 or
FOR LEAPS=-10 TO XYZ STEP ABC*.1
```

Note: Exceptions and valid values are stated when there are restrictions on the use of numeric or string expressions in a format (e.g., where only variable names are allowed).

String Expressions and Variables

When a syntax format in this manual includes a ***string*** variable like *prog_name$* or *title$*, you can normally substitute a ***string expression*** consisting of variables, literals, functions, and operators; e.g., `PRINT "Printing "+REPORT$`. When string variables are used in string expressions, subscripts and substrings are allowed; e.g., `CUSTOMER$(15,4)`.

Example:

For the **CHECK_BOX READ [*]***ctl_id*,*state$***[**,*mode$***][,ERR=***stmtref***]** format, you need string variables to receive the current ***state*** and optionally, the ***mode*** of selection.

```
CHECK_BOX READ 14000,ON_OFF$,KEYSTROKE$
```

Chapter Outlines

1
Getting Started

Welcome to ProvideX ... a powerful, versatile, intuitive programming language and integrated development environment for building sophisticated business applications.

Topics

About ProvideX

The ProvideX software development environment is used around the world for designing and building business applications. It includes a programming language, database/file system, SQL-compliant ODBC drivers, graphical development toolkit, web server, COM and thin-client interfaces, and much more – all designed to run on a wide variety of computer systems (Windows, Linux, UNIX, and MAC-OS).

Unique Implementation

At the most elementary level, ProvideX is a system that interprets and executes computer instructions written in the ProvideX language. Instructions may be entered one statement at a time for immediate execution, or contained in a software program. But, ProvideX programs can only be executed on systems where ProvideX is running.

ProvideX serves as both the development and runtime environment – it comprises all the facilities needed to design, create, and run an application. This unique implementation presents several distinct advantages (in program size, cohesion, speed, maintainability, platform independence, and so on).

Complete Development Solution

All ProvideX components are highly optimized to work together. The application language is closely tied to the database technology allowing for high-speed data access. The graphical development environment is optimized to work with thin-client technologies for universal access. When used together, the different

components take advantage of a common architecture to provide maximum speed and flexibility. Not only does this reduce the risk of incompatibilities between various system components, but it also means that your product support comes from a single source instead different (*and indifferent*) vendors.

ProvideX also adheres to the most widely-accepted software standards and protocols (data, communications, security, etc.) in the industry. This allows for a high level of extensibility and promotes the full integration of ProvideX applications into non-ProvideX applications and vice versa.

Product Options

ProvideX can be configured for multiple uses (depending on the license and the platform), but every installation begins with the ***base system*** that includes:

- *ProvideX*. Language interpreter and application development environment.
- *NOMADS*. Toolset for developing GUI-based applications (*MS Windows*).
- *COM/OCX/ActiveX, DDE, DLL, ZLIB, SSL, PDF, etc.* Built-in support for a number of industry-standard technologies.

Extend the functionality of the ProvideX base system with a set of tightly integrated application development and deployment solutions:

- *WindX, JavX*, and *UltraFX*. Thin-clients for displaying and interacting with GUI-based ProvideX applications running from a server.
- *Application Server*. Secure configurable hosting facility for connecting thin-client implementations via MS Windows, UNIX / Linux, and MAC OS X.
- *Local and Client/Server ODBC*. Open DataBase Connectivity (ODBC) for external access to ProvideX databases.
- *Web Server*. Interface for producing ProvideX-coded websites that allow browser access to ProvideX and ODBC data sources.
- *Internet Toolkit*. Utilities for developing e-mail / web-enabled applications.
- *RPC*. *Remote Processing Capability* for distributed processing of ProvideX.
- *XML Support*. Implementation for parsing and serializing XML documents.
- *OCI, DB2, MySQL, ODBC*. Native external database support.
- *Smart Controls*. Auto-load capability for list boxes / grids.
- *Customizer*. Utility for customizing panels dynamically without changing source.
- *Multiple Image Support*. Extended support for a variety of image file types.
- *Report Writer*. Powerful interface for designing and generating reports (runtime module included with base system).
- *Views*. End-user "viewing" of application data for simplified extraction and reporting (runtime module included with base system).

• *Charting Control*. Control object for creating advanced chart illustrations.

• *OLE Server*. Interface allowing external applications to access ProvideX objects directly.

Note: These products may be sold as stand-alone add-on packages or as part of a ***Professional*** or ***eCommerce*** bundle. Contact your local ProvideX dealer / distributor or visit www.pvx.com for complete product information and licensing.

Online Resources, Documentation, and Training

Sage provides a variety of opportunities for you to learn more about ProvideX. Check our website (www.pvx.com) for the latest product announcements, documentation, and training. Another valuable resource is the ProvideX ***mailing list*** – an online forum for developers to share techniques, solve problems, and learn more about ProvideX. Members of the ProvideX *Technical Team* monitor and participate in the discussions.

Installation and Setup

ProvideX can be obtained from your dealer / distributor or downloaded for direct installation from the ProvideX website www.pvx.com. Browse the Downloads page for a list of all available products, then click on the appropriate link to download and save the installation file for your particular operating system. Refer to the ProvideX *Installation and Configuration* guide for complete details.

Because ProvideX is an interpreted language, it is important to remember that programs written in ProvideX can only be run on a system where ProvideX has been installed. This is further explained in *Chapter 2.* **Language Elements,** *p.19*.

Note: For the purposes of this documentation we assume that ProvideX has been fully installed for use on an MS Windows or UNIX / Linux system.

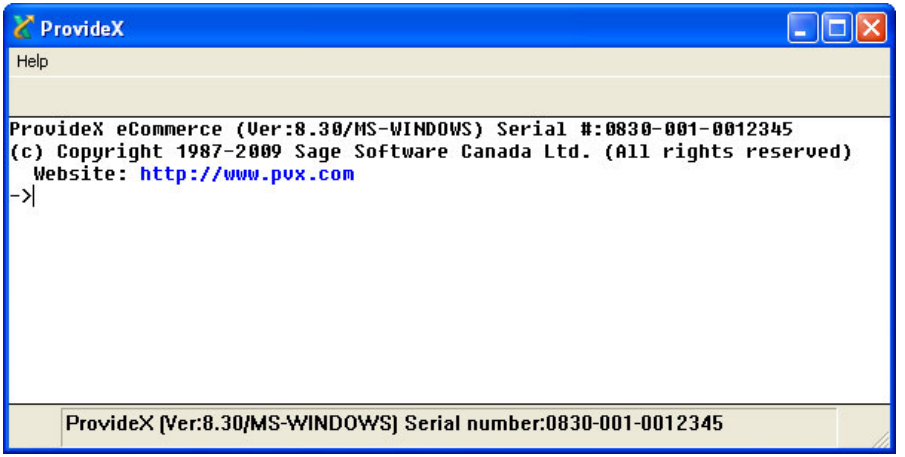
ProvideX Environment

ProvideX has capabilities and features that make the process of program development easier for both novice and expert users. When you are ready to develop a program in ProvideX, you begin by starting a ProvideX ***session***. This serves as the environment where programming instructions are input, understood and executed by the system. It is also where you create, load, modify, and execute applications. Once the session is started, you can create a new application or load an existing application for modification.

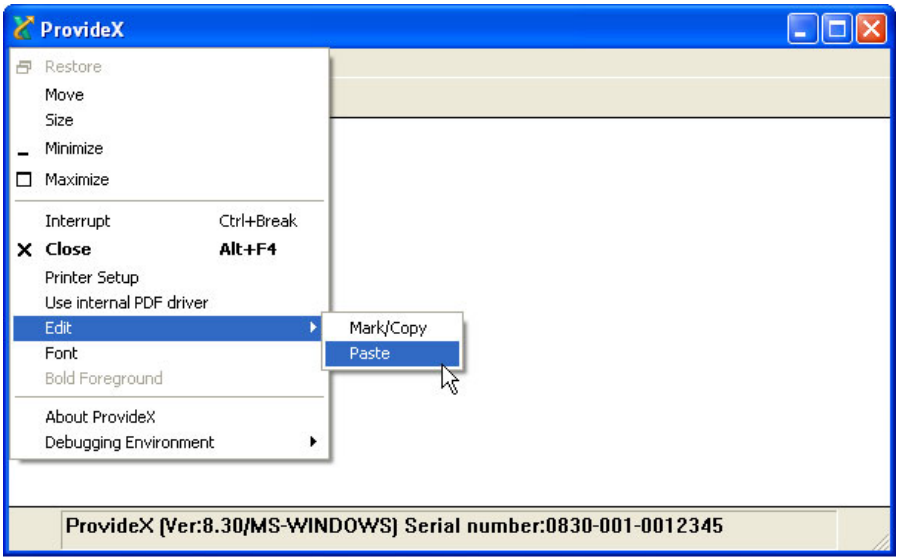
Tools for modifying and debugging programs are introduced in *Chapter 4.* **Programming Constructs,** *p.69*; however, the basic start up and operation of the ProvideX programming environment begins with the procedures described in the sections that follow.

Starting ProvideX in MS Windows

In MS Windows, you can invoke a session by clicking the ProvideX shortcut under the Start > Programs > Sage Software Canada Ltd menu. This will start the ProvideX session within a new interface window (console), as indicated below:



Click on the PVX icon at the top left corner of the window for a drop-down menu that enables you to manipulate console properties and session-related actions; e.g., changing the font size changes the size of the console work area.



These and other settings are saved in the `pvx.ini` file each time ProvideX is terminated.

By default, the **Help** menu below the PVX icon contains links that will not function properly until they are configured for use in an application.

Starting ProvideX in UNIX/Linux

On a UNIX/Linux system (non-graphical environment), ProvideX can be started by entering the command path (e.g., *path*/sage/pvx). The session opens with a serial number and *Sage Software Canada Ltd* copyright notice, and is followed by the ProvideX prompt, **->** (or **-}** under WindX). At this point, ProvideX interposes itself between you and the operating system and controls all work.

The options for starting an application are fully explained under **Customizing ProvideX** in the *ProvideX Installation and Configuration* guide.

Modes of Operation

A ProvideX session is either in **Command Mode** or **Execution Mode**. The session initializes in Command mode, as indicated by the standard ProvideX prompt **->.**

This prompt represents a "pointer" to the session input area known as the ***command line***. It comprises two characters that may be altered by the system, depending on the current input mode or the "state" of a currently loaded program. The standard prompt, **->** or **-}**, changes to **-:** when a program is being modified. It reverts back once the program is saved. When a program has been interrupted due to an error, the dash in the prompt changes to a number that indicates the current program level.

Command Mode

This is the state in which the system is "waiting for instructions" – in ProvideX, these instructions are known as ***directives***. When a valid directive is entered at the command line, the system proceeds directly to execution; there are no intermediate stages of compilation, assembly, or linkage. When execution is complete, the system displays the result (if any) and awaits the next directive. Error messages pertaining to the syntax of the input are displayed as soon as they occur.

If a directive has a leading ***line number***, the system does not proceed to execution, but uses the directive in the construction of a program. The ProvideX prompt changes from **->** to **–:** to indicate that the program is being ***edited*** (and not yet saved). There may be other input modes during a ProvideX session (e.g., using an editor, running an application); however, command mode is where the system returns when these other tasks are completed.

Note: Programs may be created without using line numbers. However, numberless programs must be edited and saved via **Full-Screen Editors**, *p.51*. For more information, see **Numberless Programs**, *p.47*.

Execution Mode

Execution mode begins whenever a **RUN** or a **CALL** directive is used to execute a program during a ProvideX session. Some programs may be *interactive* and have a *user interface* that awaits input from the user while being executed – programs remain in Execution mode until completed via **STOP** or **END**, halted due to an error, or suspended via Ctrl - Break or an **ESCAPE** instruction.

Some ProvideX applications can be designed to start and run directly from an operating system prompt/icon. These are actually launched using an ***initialization procedure*** that "hides" the ProvideX session running underneath.

Refer to *Chapter 2.* **Language Elements,** *p.19* for more on loading, executing, and terminating programs in ProvideX.

Command Line Editing

The command line allows some basic editing functionality (but you cannot move from line to line, page up/down). When entering text on the command line, most ***edit keys*** function as per the normal operation of a standard keyboard: BACKSPACE deletes the preceding character, SPACEBAR inserts a space, cursor (arrow) keys move through the input, etc. In addition to the basic keyboard actions, the following key combinations can be used for further editing and cursor functionality:

Ctrl - End clears input from the current cursor position to the end of the line.

Tab / Shift - Tab advances/backspaces 10 spaces over input.

Ctrl -RIGHT-ARROW advances to next word.

Ctrl -LEFT-ARROW moves to the previous word.

Note: More advanced editing features of the ProvideX development environment are described in *Chapter 3.* **Development Tools,** *p.47.*

Command Shortcuts

ProvideX allows the following shortcuts for frequently-used language and console commands:

Key	*Function/Purpose*
?	Question mark substitutes the **PRINT** directive.
***/ or ***	Either forward or back slash substitutes the **LIST** directive.
`	Back apostrophe substitutes the **EDIT** directive.
!	Exclamation mark substitutes the **REM** directive for a comment.
.	Period steps through a program. See **Stepping Operations**, *p.62*
;	Semi-colon steps through compound statements.
"	Double quote opens a shell to the operating system from ProvideX.

Note: The **LET** directive is assumed when ProvideX encounters a statement that begins with a variable. Also, comma delimiters can be used to represent multiple **LET** directives in a statement; e.g., `Z4=A+4.5,Z5=Z4*.85`.

Typos and Alternate Spellings

The following alternate spellings are accepted without error (and corrected) when a statement is processed:

- ProvideX automatically appends quotes where closing quotes are missing.
- **LIST** directive can be entered as **LSIT** without causing an error.
- **LOAD** directive can be entered as **LAOD** without causing an error.
- **END_IF** directive can also be entered as **FI**.

The above spellings / shortcuts are built-in. Use the **USER_LEX** directive to change exisiting directive names in ProvideX.

Command Line Recall

A history of command line input is maintained by the system throughout a ProvideX session. When a line is entered, the input is appended to the end of a buffer – the number of lines preserved in the buffer is configurable using the **'SL'=** system parameter. When the buffer fills to capacity, the earliest lines are deleted to make room for new material.

Use the **UP-ARROW** (or **DOWN-ARROW**) key to select and recall lines to the ProvideX prompt **->.** Each recalled line can be edited and resubmitted by pressing Enter, which has the same effect as typing and entering a new line at the prompt. You can reuse only one logical line at a time.

Using the Mouse (Windows)

When running a ProvideX session in MS Windows, the ***mouse*** can be used to move the cursor along the command line, copy and paste text, or activate the and **Help** menu options.

Note: While the initial console offers limited mouse support, the NOMADS toolset takes full advantage of the functionality available when working in a graphical environment. For more information, see *Chapter 6.* **Graphical User Interfaces,** *p.129*.

Copy and Paste

Activate the Edit option from the drop-down menu to copy or paste text using the mouse. An example of the console drop-down menu appears under the section **Starting ProvideX in MS Windows**, *p.12*.

To copy (mark) text from the current session window into the Windows clipboard:

1. Select > Edit > Mark/Copy.
2. Drag the mouse pointer over the block of text you wish to copy.
3. Press Enter.

To paste text from the Windows clipboard into the current session window:

1. Place the cursor at a location on the command line.

2. Select > Edit > Paste.

Note: The drop-down menu can also be accessed directly (without using the mouse) by pressing the Alt - SPACEBAR key combination.

Exiting a ProvideX Session

A ProvideX session can be terminated using one of three directives: **BYE**, **QUIT**, or **RELEASE**. When invoked from the command line, or used in an application, these directives close all opened files, return all memory in use to the operating system, and terminate the ProvideX session.

On a Windows system, you can also click the *(close)* button on the top right side of the ProvideX console to terminate the session and close the window.

If **STOP** or **END** are used in an application that was invoked directly by an operating system command (bypassing Command mode), they will automatically terminate the session and return you to the operating system.

System Utilities

ProvideX comes equipped with several utilities that are not, strictly speaking, part of the language, but provide various design, configuration, maintenance, and diagnostic services within the development environment. These are actually separate programs that reside as auxiliary files in the PVX system sub-directory `lib`. All of the utilities are installed with the ProvideX base system, but some were originally designed for use in a character-based environment, some apply to a graphical (Windows) system only, and some require a thorough understanding of ProvideX internal code and may not be available for general use.

The majority of system utilities are intended for the creation, debugging, and management of ProvideX programs, files, and directories. Editing and debugging tools are covered in *Chapter 3.* **Development Tools,** *p.47*.

Command Line Utilities

The ***classic*** utilities subsystem comprises a set of system utilities that may be called directly from the command line. These are all accessed using a leading *asterisk* * (e.g., the character-based editor is invoked by entering `CALL "*e"`). You can also access these utilities via the F5 function key, or by issuing a `CALL "***"`, which displays a set of menu-driven commands at the top of the console:

```
Selection:
Calculator Files Phone Utilities Edit Quit
```

This character-based menu can be navigated using cursor keys or by moving the mouse pointer (if available). Press Enter (or left-click the mouse) to select one of the options. The `Utilities` option (`*u`) directs you to a subsequent menu of character-based utilities, grouped under the following major categories:

```
System Utilities:
Files Directories Programs Configuration General Quit
```

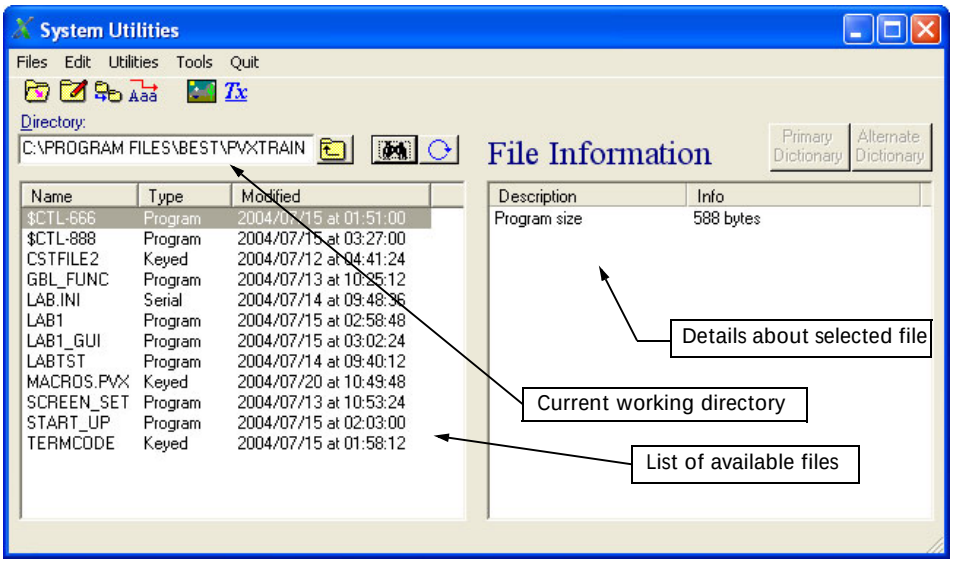
Where:

`Files`	Options to create, delete, rename, view, update or otherwise manipulate data files. (`*uf`)
`Directories`	Options to create, delete, rename, or view directories. (`*ud`)
`Programs`	Options to create, delete, rename, edit, list or otherwise manipulate program files. (`*up`)
`Configuration`	Alters the configuration of the ProvideX environment. (`*uc`)
`General`	Accesses some general utilities. (`*ug`)

Select `Quit` or press the Esc or F4 keys to exit an opened utility and to move back through the main menu.

Graphical Utilities

Graphical utilities are available for systems running ProvideX for Windows or via WindX. They have all the capabilities of the classic character-based utilities plus access to a few features that are specific to a GUI-based development environment; i.e., the **GUI-based Program Editor (*IT)**. Type `gui` at the command line to access the System Utilities interface:



The initial panel displays your current working directory and allows you to select a specific file for use in the utilities. Use the *browse* and *folder* buttons to change the list of available files.

In the *left pane*, the file name, type, and time stamp is displayed for every file in the current directory. The *right pane* provides further details about the file that is currently highlighted in the list. This information is dependent on the type of file: program, data file, link reference, or device. If the selected file is a ProvideX data file containing a primary / alternate **Embedded Data Dictionary** then the two buttons above the right pane will be enabled so that details of the file layout can be viewed.

Menu Bar

Various utilities can be invoked via the menu bar or by clicking the toolbar icons for quick access to commonly-used selections:

Where:

Files	Displays a list of ***open*** files that can be sorted by file number, type, record size, key length, path name, and Keyed / Indexed information.
Edit	Opens the currently-selected file for editing in the GUI ***Program Editor.***
Utilities	Accesses a drop-down menu for system utilities, grouped under by major categories: Files, Directories, Programs, Configuration, and General.
Tools	Provides direct access to the Windows ***Calendar***, ***Calculator***, and ***Notepad***.
	Shortcut to the File View utility.
	Shortcut to the File Update utility.
	Shortcut to the Program Compare utility.
	Shortcut to the Program Bulk Edit utility.
	Shortcut for viewing available bitmaps.
	Shortcut for viewing font and colour selections.

NOMADS

The Non-procedural Object Module Application Development System (NOMADS) is an integrated suite of utilities that simplify the development of GUI-based applications using ProvideX. NOMADS allows you to separate data access, logic, and graphical controls into reusable segments, localize changes, design a graphical front-end for character-based components, and create event-driven applications. The toolset also includes a common data manager for consistent access to a variety of data files as well as a generic error-handling interface. For more information on GUI development in ProvideX, see *Chapter 6.* **Graphical User Interfaces,** *p.129*.

2
Language Elements

This chapter provides an overview of the basic concepts and elements used for building ProvideX applications.

Topics

Background

While the language itself is easy to learn, the method of execution may seem a little unusual for novices, particularly if they are used to ***compiled*** languages. ProvideX is ***interpreted***, which means it is executed directly from its source form:

- When a valid instruction is entered at the ProvideX console, it is evaluated for syntax, converted to tokenized code, and then executed ***immediately***.
- When several instructions are entered together in the form of a program, they are evaluated for syntax, converted to tokenized code, and then stored in memory.
- A program can be executed directly from memory, or saved to a file for later use. All program files are loaded and executed within the ProvideX session itself.

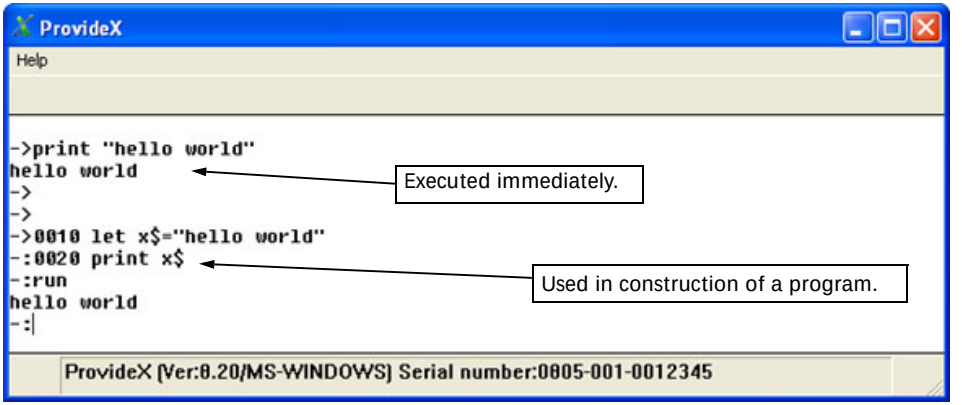
ProvideX is ideal for development, debugging and testing. By comparison, most ***compiled*** languages require separate procedures for entering source and for generating object code, which must be saved to a different file outside of the development environment to be executed. ***Interpreted*** code is quicker, because it requires fewer steps; i.e., *enter* **+** *interpret* instead of *enter* **+** *compile* **+** *run*.

When ProvideX tokenizes source statements, each multi-character reserved word and value is compressed into 1-byte tokens (ignoring white spaces). Maintaining tokenized source uses far less memory and is much faster to interpret and execute than working in the original format. When further editing is required, the tokenized code is simply reconstructed back into readable text when displayed in ProvideX.

Important: If you are new to ProvideX, remember that applications written in ProvideX can only be executed on systems that run a copy of ProvideX.

Directives, Statements, and Programs

All processing in ProvideX is controlled by ***directives***, keywords that "direct" the system to perform certain tasks. Directives can be entered in Command mode to be ***executed immediately*** (as described at the beginning of *Chapter 1*). They also represent the core ingredient in the ***statements*** that constitute a ProvideX ***program***.



In the session above, the **PRINT** directive is executed as soon as it is entered on the command line. However, the ***line number*** inserted before the **LET** directive prevents that input from being executed – when ***numbered statements*** are entered:

- the input is not executed, but added to a program in memory.
- the prompt changes from **->** to **-:** to indicate that a program is being ***edited***
- the series of numbered statements currently in memory will only be executed when a **RUN** or **CALL** directive is issued.

Each statement can be made up of one or more directives. A ProvideX program consists of a series of statements that are organized in a logical manner and stored together in memory. When a program is executed, all the statements (and the directives inside each statement) are evaluated and processed in the order in which they are read by the system. Once the directives in one statement have been executed, ProvideX proceeds to the next statement (next line of code) in the sequence, and so on.

Note: Some statements can alter the normal (top to bottom) flow of execution by transferring control to other points in the program. This is explained in *Chapter 4*.

In theory, each program statement begins and terminates on one line – a carriage return marks the end of a line of code, and ProvideX executes it as though it were entered directly on the command line. However, as described later in this chapter, there are exceptions to this rule; e.g., if the **LIST** **EDIT** directive is used to *format* program code for readability, lengthy statements may be listed over several lines – the line number representing the entire statement appears on the first line only.

Statement Syntax

A valid program statement requires at least one directive. Also, depending on the directives used, various arguments and syntax elements may be needed to complete the statement. The example below illustrates the format of a typical program statement:

```
0010 Total: PRINT a,b,c+d ! Totals
```

Where:

`0010` ***Line Number*** (optional). Any integer between 1 and 64999. Each line number must be unique – the best practice is to assign line numbers in increments of 5 or 10 to allow for easy insertion of additional statements. When a line number precedes a statement the statement will be included in the construction of a program. See **Line Numbers**, *p.72*.

`Total:` ***Line Label*** (optional). Descriptive name up to 127 characters long, followed by a colon. Alphanumeric line labels can be used in place of line numbers to identify statements within the program.

`PRINT` ***Directive.*** Keyword identifying the task to be performed.

`a,b,c+d` ***Arguments*** (optional). Data to be processed or system keywords that further define the directive. A space is required to separate the directive from its arguments. See **Arguments**, below.

`! Totals` ***Comment*** (optional). Descriptive text beginning with an exclamation point **!** or the keyword **REM**. Text following an exclamation point is ignored by the system; i.e., the exclamation point terminates a statement and passes execution to the next line of code.

Note: Comments are very important for the readability and maintainability of a program. However, this text is included in the final compiled program, so be aware that excessive comments may increase memory requirements and degrade overall system performance.

Arguments

While some directives are executed without arguments, most accept / require data or keywords to perform their tasks. Argument components may be optional or mandatory – this depends on the directive and the task involved.

Most arguments supply some form of data for processing – a text or numeric value delivered in literal or variable form (see **Data Types, Literals, and Variables**, *p.34*). Other types of arguments can redefine or augment the functionality of the directive itself – system parameters, mnemonics, options, and built-in keywords. For more information, see **Primary Syntax Elements**, *p.24*.

Compound Statements

If more than one directive appears in a statement, it is considered a ***compound*** statement. Semicolons are required to separate the different directives (and their arguments) until the end of the line; e.g.,

```
0020 let x=a+b; print "Answer=",x; goto 1000
```

Each of the directives in a compound statement (between the semicolons) would function the same if they were entered in sequence on separate lines.

A transfer of control into a compound statement starts processing from the ***first*** directive only. In the above example (statement 0020) it would be impossible to start processing at the **PRINT** or **GOTO** directive. However, it is possible to transfer control out of (and then back into) a compound statement if the directive is designed to return back to the current location (**GOSUB** / **RETURN** and **FOR..NEXT**); e.g.,

```
0030 x=10;print x; gosub xyz; goto 1000
```

Directives that transfer control without returning (e.g., **GOTO**, **EXITTO**, **RETURN**) must appear at the end of a compound statement. The following is not valid because **RETURN** transfers control and the last statement is unreachable code:

```
print "hello"; return; if name$<>"" print "-", name$
```

Some directives cannot be included in a compound statement. Restricted directives are identified as they are documented in the *Language Reference*.

Saving, Loading, and Executing a Program

A ProvideX program is a collection of statements that are stored in memory (or saved to a file) for consecutive execution. In Command mode, the ***current program*** represents all the statements in memory that are immediately available for editing and / or execution. In Execution mode, the current program is also referred to as the ***main-line*** program (also known as *execution level 1*).

The current program can be **RUN** without being saved to a file. Once saved, a program can also be **RUN** directly from a file. By default, program files are saved in a semi-processed state that allows them to be executed as though they are already loaded in memory. These pre-compiled tokenized program files are not readable in a standard text editor and must be loaded into a ProvideX session for editing. For information on working with text-based program files, see **Text (ASCII) Format**, *p.23*.

Whether you create a new program or **LOAD** an existing program from a file, all changes you make to numbered statements at the command line will apply to the program currently loaded in memory. The different methods for creating and saving ProvideX programs are described under **Writing and Modifying Program Code**, *p.47*.

Saving a Program to a File

During a ProvideX session (in Command mode), the current program will cease to exist if a **START** or **DELETE** is issued, or another program is loaded. All changes to the current program will be lost.

To retain your changes for later use, use the **SAVE** directive to copy the current program to a file on disk; e.g.,

```
SAVE "Myprogram"
```

If the file name specified by `Myprogram` does not exist, a new program file is created and the program is written to it. If a program exists with the same name, it will be overwritten by the current program. Programs can also be saved in program libraries (see **[LIB] Program Library**, in the *Language Reference, p.777*).

Retrieving a Program File

As mentioned earlier, programs that are saved to disk, may be recalled using the **LOAD** directive. This clears the current program (if any) and replaces it with the program specified; e.g.,

```
LOAD "Myprogram"
```

Executing a Program

The **RUN** directive can be used to execute the current program. If the **RUN** command includes a program name, the system will automatically load the specified program from disk before executing it; e.g.,

```
RUN "Myprogram"
```

Text (ASCII) Format

It is possible to **RUN** / **CALL** / **PERFORM** / **LOAD** and **SAVE** ProvideX programs in text (ASCII) file format. If the target file for a **SAVE** is not a program file, ProvideX will automatically write the program out in ASCII format; e.g.,

```
->DELETE
->0010 FOR I = 1 TO 20
->0020 PRINT I
->0030 NEXT
->SERIAL "PROG999"
->SAVE "PROG999"
```

This would output the program to the text file `PROG999`. If desired, you can insert the keyword **EDIT** following the **SAVE** directive and the system will save the program in formatted form. Text-based ProvideX programs do have some advantages.

- Use of more advanced editing / formatting functionality that is available in other types of editors (spell checking, multiple undo, etc.).
- Version control and configuration management systems.
- Accessing multiple files via bulk editing tools.

However, when running a text-based program from a ProvideX session, each statement must be evaluated for syntax and converted to tokenized code prior to execution. These extra steps may have an impact on system performance. One way to avoid a performance hit is to **LOAD** a text-based program prior to executing it. At this point, it can be **RUN** as the current program in memory.

Primary Syntax Elements

The ProvideX language uses standard set of keywords and symbols to deliver instructions in a statement as well as to create / execute programs. Some of these are required to make statements executable, while others are used to further define the tasks and the data in the operation being performed. ProvideX keywords are reserved for internal use by the system – they should never be used as variable names in applications. The primary elements in the ProvideX language are grouped as follows:

Topics

Refer to the *Language Reference* for complete syntax, along with detailed descriptions, of all of the language elements recognized by ProvideX.

The language also reserves several keywords and symbols to be used as parameters, qualifiers in a statement. See **Other Syntax Elements**, *p.32*. How these language elements may be used to build programs in ProvideX is discussed in *Chapter 4.* **Programming Constructs**. For a discussion on defining content to be manipulated in a statement, see **Data Types, Literals, and Variables**, *p.34*.

Directives

Commands that instruct the system to perform a task. A directive is the first keyword in a valid program statement. There may be more than one directive in a statement. The list below represents some of the more commonly-used directives:

BEGIN	Re-initialize current session; close files, clear all non-Global variables, release memory to the system (see also **CLEAR**, **RESET**).
START	Reset files and variables (including global files and variables).
BYE	Terminate session; close all opened files, return memory to the operating system and terminate ProvideX; see also **QUIT**, **RELEASE**.
PRINT	Send information to display or a file (if file number specified).
LET	Assign value to a variable.
DUMP	Print all variables in use (including system variable **ERR**), program name, line number, and the **FOR..NEXT** / **GOSUB** stack.
LOAD	Read a program into memory for execution, listing, or modification.
LIST	Convert program statements to readable format and output to display or file.
DELETE	Remove lines from program or if no lines specified, unload program.
RUN	Start or continue execution of a program.

END Terminate execution of program (functionally identical to **STOP**).
EDIT Change an existing statement.
SAVE Copy current program to file specified.
INPUT Issue prompt and process response.
CWDIR Change working directory.
GOSUB Transfer control to line referenced.
MSGBOX Display a message window in the middle of the screen.

For the complete list of directives, see *Chapter 2* in the *Language Reference*.

Examples

The **PRINT** directive (or **?** shortcut) displays a string literal at the console (with output positioning determined via the **@** function):

```
PRINT "HELLO WORLD"
? @(10,10),"HELLO WORLD"
```

The **LET** directive assigns values to a string variable and a global string variable:

```
LET X$="HELLO"
LET %X$= "JELLO"
```

The **LOAD** and the **LIST** directive are used to view a program (options for the **LIST** directive include listing a range and displaying with indentation):

```
LOAD "**"  *1
LIST
LIST 5100,5220 *2
LIST EDIT 5100,5220 *3
```

The **DELETE** directive removes lines from a program:

```
DELETE 5100,5220 *4
```

The **DELETE** directive removes the program from memory:

```
DELETE
```

The **RUN** directive loads and runs a program (**END** halts execution):

```
RUN "**
stop execution <ctrl+break>
END
```

System Functions

Numeric or string operations that process values and yield results within a program statement. ProvideX system functions consist of a three-character name followed by arguments in parenthesis; e.g., `AND(A$,B$,ERR=0300)`. The following common system functions can be applied to any Business BASIC expression.

ASC( ) Returns the internal numeric value of a given ASCII character.
CHR( ) Returns an ASCII character for a given numeric value.

NUM( ) Converts a string containing numeric characters into a corresponding real numeric value (e.g., `NUM("1.34")` yields `1.34`).

STR( ) Converts a numeric (or string) argument into a formatted string using an optional masking argument (e.g., `STR(5*6:"000")` yields `"030"`).

DTE( ) Converts a date argument from Julian form to a formatted string.

JUL( ) Converts a date argument from year, month, day values to Julian form; this defaults to the current date if no value is supplied.

NUL( ) Tests argument and returns either `1` (*true*) or `0` (*false*) code for null string.

NOT( ) Returns the inverted value of a string or numeric (i.e., with all `ON` bits turned `OFF` and vice versa).

CHG( ) Returns only the variables within the given argument that have changed since the last **CHG( )**.

PTH( ) Returns the absolute path name for a given channel or logical file number.

STK( ) Returns the line number / program name executing at the level specified.

FFN( ) Checks all open files for a reference to specified filename and returns the file number or `-1` if the file is not open.

FID( ) Returns the file information descriptor for the file specified.

FIB( ) Returns the file information block descriptor for the file specified.

FIN( ) Returns detailed physical aspects of file specified.

TCB( ) Returns value regarding task information, activation / licensing, the OS.

For the complete list of functions, see *Chapter 3* in the *Language Reference*.

Examples

The **PRINT** directive (or **?** shortcut) displays system function results.

```
PRINT ASC ("A")
 65
PRINT CHR (65)
A
N=NUM("65");PRINT N + 1
 66
STR$="HELLO "+STR(N); PRINT STR$
HELLO 65
? DTE(0)
 11/20/08
DATE$=DTE(0:"%Dl %Ml %D/%Y" ); PRINT DATE$
Thursday November 20/2008
?JUL(DAY)
 14203
```

```
DATE$=DTE(JUL(DAY):"%Y/%M/%D"); PRINT DATE$
 2008/11/20
```

Code sample using **NUL( )** and **NOT( )**:

```
0010 INPUT "ENTER YOUR NAME: ",N$
0020 IF NUL(N$) THEN PRINT "YOUR NAME IS REQUIRED!"; GOTO 0010
0040 INPUT "ADDRESS: ",A$
0050 IF NOT(NUL(A$)) THEN GOTO DONE ELSE PRINT "ADDRESS PLEASE!"; GOTO 0040
0060 DONE: PRINT "BYE"
RUN
ENTER YOUR NAME:
YOUR NAME IS REQUIRED!
ENTER YOUR NAME: Anthony
ADDRESS:
ADDRESS PLEASE!
ADDRESS: 8920 Woodbine Ave
BYE
```

Using **CHG( )**:

```
0010 PRINT CHG("X,Y,A$,B$")
0020 LET X=666
0030 LET A$="HELLO WORLD"
0040 PRINT CHG("X,Y,A$,B$")
RUN
X,A$
```

Using **FFN( )**:

```
0010 LET X=FFN(".")! FFN returns channel number if the file is open
0020 IF X=-1 THEN LET X=HFN; OPEN (X)"."! if -1 then file is not open
0030 READ RECORD (X,END=DONE)R$
0040 PRINT R$; GOTO 0030
0050 DONE: PRINT "ALL DONE, BYE-BYE!"
0060 END
RUN
libeay32.dll
libharu.dll
license.txt
```

~*snip*~

```
pvxzlib1.dll
ssleay32.dll
windx
xerces-c_2_8.dll
ALL DONE, BYE-BYE!
```

Using **FIN( )**:

```
7210 GET_FILE_INFO:
7220 LET FILE$="."
7230 OPEN (1)FILE$; LET X$=FIN(1)
7250 LET TIME_STAMP$=X$(5,4),T=DEC($00$+TIME_STAMP$),D=INT(T/86400),
      T=T-D*86400
7260 LET SV_BY=PRM('BY'); SET_PARAM 'BY'=1970; LET TIME_STAMP$= " on
      "+DTE(D,T/3600:"%Ws, %Ms %D %Yl at %hz:%mz:%sz"); SET_PARAM
      'BY'=SV_BY
7270 PRINT FILE$+" last modified : "+TIME_STAMP$
7280 CLOSE (1)
RUN
. last modified :  on Thu, Oct 9 2008 at 08:36:36
```

System Variables

Internally-defined numeric and string variables for providing system information such as the date and time. All system variables have reserved three-character names. To avoid potential conflicts with the reserved list we strongly recommend that you do not use three-character names when defining your own program **Variables**.

The following system variables can be used wherever program variables are used (but they cannot be assigned a value):

LWD	Current working directory.
TIM	Current system time in hours past midnight; see also **TMS**.
PRM	Current parameter settings for ProvideX.
SSN	System software identifier.
WHO	Current userid; see also **UID**.
QUO	ASCII quote character ".
DLM	Operating system path delimiter.
DAY	System date in MM/DD/YY format (unless reset via **DAY_FORMAT**).
PGN	Current program path name.
HFN	Highest unused local file number; see also **UNT**.
ERR	Value of the last error detected by the system.
SEP	Providex record field separator.
CTL	Code indicating how input was terminated.
EOM	Character string that terminated last input.
MSE	Current state (details) of the mouse.
LFA	Number of last file accessed.

For a complete list, see *Chapter 4* in the *Language Reference*.

Examples

The **PRINT** directive (or **?** shortcut) and system variables can be used to display current system information.

```
? LWD
C:\Program Files\Sage Software\ ProvideX V8.30
? TIM
 11.055314
? PRM
-'3D',-'AD',-'AH','AI'=10,-'AP',-'AW',-'B0','BF'=10,'BL'=0,-'BT',-'BX','BY'=1970
,-'CD','CH'=12,'CI','CO'=4,'CS','CT'=0,'CU'=36,-'D0','DB'=31,-'DC','DF'=0,'DL'=0
,'DP'=46,'DT'=0,'DW'=0,-'EG','EL'=1,-'EO',-'ES',-'EX',-'F4','FB'=5,-'FC','FF'=0,
-'FI','FO'=0,-'FU',-'FL','FP','FS'=138,-'FT',-'FX',-'F',',-'I0',-'I2','IC',-'IM',
'IR','IS'=5,'IW',-'IZ','KF'=0,-'KR','LB'=4,-'LC',-'LD',-'LE',-'LM','LS'=1,-'LU',
-'LW',-'LZ','MB'=0,-'MC','MF'=50,-'MP',-'MX',-'NE',-'NI',-'NK',-'NL',-'NN',-'NR'
,-'OC','OL'=25,'OM',-'OP','OR','OW'=0,'PC'=0,'PD'=2,-'PE','PL'=10,-'PO','PP','PQ
'=100,-'PU','PW'=36,-'PZ','QD'=4,'QF'=1,'Q_'=2,'Q^'=2,'QK',-'QS',-'QT',-'RI','RN
'=1,'RP',-'RR',-'RS',-'SC',-'SD',-'SF',-'SK','SL'=32,-'SP',-'SR',-'SS','SV'=1,'S
W'=1,'SZ'=32000,'TA'=0,-'TB','TC'=0,'TH'=44,-'TL',-'TN',-'TT',-'TU',-'TX','UL',-
'VC','VP'=48,'VR'=0,'VW'=0,'WB','WD'=100,-'WF','WH'=0,'WI'=1000,-'WK','WT'=2,'WZ
'=512,-'XC',-'XF',-'XI',-'XT',-'ZP',-'DD','!B'=3,'!P'=0,'!U'=0,-'1U'
? SSN
 0829-001-0012345
? WHO
 akhodge
```

The **QUO** variable can be used to include quotes within a string:

```
PRINT "These are "+QUO+"QUOTES"+QUO
These are "QUOTES"
```

Applying **DLM** and **QUO** to a pathname:

```
SAVE "C:"+DLM+QUO+PVXTRAIN$+QUO+DLM+"TEST"
```

Using the **DAY** variable to get system date and the **DAY_FORMAT** directive to change the display:

```
? DAY
11/20/08
DAY_FORMAT "YYYY/MM/DD"
? DAY
2008/11/20
```

Using **HFN** to open a file to the highest available channel:

```
LET MY_FN=HFN
OPEN (MY_FN)"."
```

Code sample using **ERR** to retrieve the number of the last error:

```
10 LET MY_FN=HFN
20 OPEN (MY_FN, ERR=OOPS)"my_file"; goto DONE
30 OOPS: PRINT "YOU MADE BOOBOO NUMBER "+STR(ERR)
40 DONE: END
```

Displaying the default separator (**SEP**):

```
PRINT SEP ! PVX executes the separator (carriage return & line feed).
PRINT HTA(SEP) ! HTA( ) function converts unprintable hex to ASCII.
PRINT "hello "+SEP+"world"
```

Retrieving the screen & mouse-position information using the **MSE** variable:

```
LET CUR_STAT$= MID(MSE,1,1) ! MID string function gives substring
PRINT HTA(CUR_STAT$)
```

System variables are generally read-only; however, certain ones (**CTL**, **ERR**, **LFO**, **LFA**, **EOM**, **REC**) can have their contents modified via **DEF sysvar=** syntax; e.g.,

```
->PRINT ERR
0
->DEF ERR=12
->PRINT ERR
12
```

Mnemonics

Special control sequences for output to a display device or printer. Mnemonic instructions are inserted within a **PRINT** or **INPUT** statement, and are specific to the channel on which they are defined. The language provides an extensive list of pre-defined mnemonics, but additional 2-character mnemonics may be created via the **MNEMONIC** directive. Commonly-used mnemonics include:

'CS'	Clear screen.
'CH'	Position cursor at home; i.e., `@(0,0)`.
'RED' & '_RED'	Input/output will be in red foreground or background. Refer to the *Language Reference* for other colours.
'CE'	Clear from cursor to end of screen.
'CL'	Clear from cursor to end of line.
'DEFAULT' or 'DF'	Set current attributes as defaults (colour, foreground/background, etc.).
'RM'	Reset to default attributes.
'BR'	Begin reverse video mode.
'LF'	Advance one line and return to column 0 (line feed).
'FF'	Advance page(issue form feed).

For the complete list of ProvideX mnemonics, see *Chapter 5* of the *Language Reference*. A listing based on mnemonic functionality is provided under **Mnemonic Categories** in the *Language Reference, p.579*.

Several examples of how mnemonics can be used for print and display output are available throughout this document, most notably, in the following sections:

Controlling Output, *p.96*
Interface Windows, *p.138*
Display Objects, *p.201*
Chapter 7. **Printing**
Device Drivers, *p.390*.

System Parameters

Internally-defined options for setting up a system's operation or behaviour under ProvideX. Most act as Boolean switches (0 or negative sign indicates off, 1 or no sign indicates on), but some require specific values in order to be set. Commonly-used parameters include the following:

'FU' / **'FL'**	Convert all file / program name references to upper / lowercase.
'LC'	**LIST** variables in lowercase.
'LD'	**LIST** directives in lowercase.
'LE'	**LIST** outputs indented program.
'NL'	Suppress **LET** in program listing.
'PC'=	Use program caching on **LOAD**.
'NE'	Report errors occurring in subprograms.
'XT'	Terminates ProvideX when program drops to Command mode.
'XI'	Allows extracted records to be read by other ProvideX processes.

For the complete list of system parameters, see *Chapter 6* in the *Language Reference*.

Control Object Properties

Properties of ProvideX **Control Objects** (button, drop box, scrollbar ...) can be referenced and modified ***dynamically*** using a control's assigned CTL value (*ctl_id*) followed by the apostrophe operator and an associated property name. Common properties include:

Col	Screen position (column of control).
Lines	Height in number of lines.
Tip$	Tip message
Msg$	Message line
Focus	Focus indicator

For a complete list, refer to *Chapter 7* in the *Language Reference*. For further discussion on this subject, see **Dynamic Control Properties**, *p.137* in this document.

Other Syntax Elements

Depending on the statement, other keywords may be required to provide data characteristics, usage information, comments. The following syntax symbols also have fixed meaning in ProvideX:

! ***Exclamation***. ProvideX accepts an exclamation mark as a substitute for the **REM** (*remark*); e.g., `! this remark.` An exclamation mark as the leading character of a string also denotes a ProvideX embedded bitmaps; e.g., `!STOP`.

" ***Quotes***. Standard quotation marks enclose string literals. A leading quotation mark can also be used as a substitute for the **INVOKE** directive; e.g., `"NOTEPAD` is the same as `INVOKE "NOTEPAD".`

$ ***Dollar sign.*** A dollar sign at the end of a variable name marks a string variable; e.g., `CUST$`. Dollar signs can also enclose hexadecimal values, for example `$8A$`.

' ***Apostrophe.*** Single quotation marks (apostrophes) enclose **Mnemonics** and **System Parameters**, for instance `'TL'` and `'CS'`. The **Apostrophe Operator**, is used to indicate a control object property.

; ***Semicolon.*** Directives and entry points are separated by semicolons in program statements. When entered as the first character of a line, ProvideX hides the line from line listings making it appear as if it did not exist. The line will execute correctly, but it cannot be interrogated.

***** ***Asterisk.*** ProvideX includes a number of auxiliary applications that are stored under the `LIB` directory. The names of these utilities and subsystems are preceded by an asterisk when accessed in ProvideX; e.g., `*UPB`, `*IT`. An asterisk may also have specific meaning in the syntax of different directives or functions; e.g., as a wildcard character.

% ***Percent Sign.*** A percent sign *before* a variable name denotes a *global* variable or function; e.g., `%DEPT`. A percent sign *following* a variable name indicates that the variable is an *integer*; e.g., `DEPT%`. A variable name having both leading and trailing percent signs denotes a global variable for integer values; e.g., `%DEPT%`

[ ]** ***Asterisk + Square Brackets. The ***search utility*** (for searching programs in console mode) is invoked by enclosing a search string within square brackets preceded by an asterisk.; e.g.,

```
->*[print]
0090 REM Printing
0100 PRINT DAY
0120 PRINT "Today's date is ",DAY
0610 IF LEN(X$)>100 THEN PRINT "TOO LONG"; GOTO 0210
```

[ ]=[ ]** Global ***search and replace can be used to make changes in programs in console mode; e.g., `*[CST$]=[CUST$]` changes all instances of `CST$` to `CUST$`.

-:
->
-} ***Prompts.*** When your ProvideX prompt is a dash with a colon, that indicates that your current program has not been saved. After you save your program, the prompt reverts to an arrow. Under WindX, the prompt is a dash and a right brace.

/ or *** ***Slashes. ProvideX accepts either slash (forward or back) as a substitute for **LIST**; e.g., `/ 30` is the same as `LIST 30`.

xxxx: ***String–trailing colon.*** Use a trailing colon to denote that your string is a line label (statement reference, branch, entry point); e.g.,

```
0110 IF UPDATE$="Y" GOSUB CUSTOMER
...
2000 CUSTOMER:
2010 INPUT 'CS',@(5,5),"Enter customer number",CST
2020 ! REST OF ROUTINE ...
2200 RETURN
```

? ***Question Mark.*** ProvideX accepts a leading question mark as a substitute for **PRINT**; e.g., `? CUST$` is the same as `PRINT CUST$`. ProvideX also places a question mark between a line number and program statement to denote a syntax error.

' ***Back Apostrophe.*** ProvideX accepts the back apostrophe as a substitute for the **EDIT** directive..

Data Types, Literals, and Variables

All content to be manipulated and processed through a ProvideX program can be categorized within one of two primary data types: **Numeric Values**, which are used in calculations (i.e., account balances, prices, and quantities), and **String Values**, which usually involve text-based information (i.e., account names, addresses, and product descriptions). Most data is processed in the form of **Variables**; however when constant values are required, numeric or string **Literals** can be written directly into the program code.

Topics

Note: There may be size limits imposed by the ProvideX activation or the operating system on different types of string and numeric data. For a list of ***system limits***, refer to the *Language Reference, p.821*.

For a discussion on how source data is accepted into a program for processing, or where resultant data is sent from a program for storage or to be displayed, see **Basic Input/Output**, *p.87*.

Variables

A variable is a named location in memory that is used for storing data temporarily during program execution. In ProvideX, the methods for creating/changing variables include ***assigning*** (using the **LET** directive) and ***inputting*** (using **INPUT** or **READ**). The data type is determined when the variable is created: either ***string*** or ***numeric*** (not both). The initial value of a numeric variable is **0** *zero*. The initial value of a string variable is a *null string*.

The two variable/data types are distinguished by the fact that string variables have names that end with a **$** *dollar sign*. In the following example, the numeric variable X is assigned the number 1234 and string variable X$ is assigned the text "START TEST, X=".

```
->LET X=1234,X$="START TEST, X="; PRINT X$,X
START TEST, X= 1234
```

The first character of a variable name must be alphabetic (**A** through **Z**), the remaining characters may contain any of the following: **A** through **Z**, **0** through **9**, **_** ***underscore***, or **.** ***period***. By default, variable names are not case-sensitive, but are listed in uppercase only. The **'MC'** and **'LC'** parameters can be used to maintain mixed or lowercase variable names in listings. Variable names cannot start with **FN**, as this denotes a user-defined function; see **DEF FN**, *p.85*.

Variables should have unique names that do not conflict with ProvideX keywords. It is also best to avoid three-character names – ProvideX reserves three-character names for **System Variables**. For a complete list of reserved words, refer to the *Language Reference, p.823*.

A **%** *percent sign* before a variable name is used to denote a *global* variable – a percent sign *following* a variable name indicates that the contents is an *integer*. Every variable in ProvideX is defined for a specific ***scope***, which indicates to what extent the variable can be accessed and used (***local*** or ***global***). Typically, a variable is only visible to subroutines within the current program that created it.

Global Variables

The scope of a variable can be extended for "global" use if it is named with a leading **%** *percent sign*; e.g., %VAR1 will be visible to ***all programs*** that are executed within a given ProvideX session.

Note: Global variables can only exist within one session of ProvideX – they cannot be shared or carried between sessions.

Remember that VAR1 and %VAR1 are two different variables. The variable named VAR1 can be deleted at any time using a **CLEAR** or **BEGIN** directive; whereas, the one named %VAR1, would remain active until the end of the user session or the execution of a **START** directive.

Local Variables

The scope of a variable can also be narrowed for "local" use if it is declared using the **LOCAL** directive. This means that an existing variable can be reassigned for the duration of a specific subroutine, subprogram, for / next loop, or user-defined function. The local declaration does not affect the original contents of the variable. If the variable name is already in use, the system preserves the current value. Once the stack entry has been removed, the system restores the variables to their original values.

In the following example, the variables X$, I, and N are declared **LOCAL** for the duration of the subroutine:

```
0130 GOSUB 1000
....
1000 REM Subroutine 1
1010 LOCAL X$, I, N
1020 READ (1, KEY=K$) X$
1030 I = POS(","=X$)
1040 IF I <> 0 THEN X$(I,1)=" "; N++; GOTO 1040
1050 PRINT "There were ",n, " commas"
1060 RETURN
```

Original values will be restored upon execution of the **RETURN** directive. The local declaration can also be placed in front of variables within a **DEF FN** definition; e.g.,

```
DEF FN%DATE$(LOCAL DT$) = DT$(1,2)+"/"+DT$(3,2)+"/"+DT$(5,2)
```

Literals

Literals are numeric, string or hexadecimal values that are written directly into the program code – they cannot be modified at runtime. While variables are meant to be changed during processing, literal values remain ***read-only***. Literals are commonly used to display messages, assign constants, or place initial values into variables.

Two formats are supported for the definition of numeric literals in ProvideX: simple numbers (signed, with / without decimal point) or floating point numbers (using scientific notation). String literals consist of a series of ASCII characters contained within **""** ***quotation marks***.

The various formats used in the definition of numeric and string values are further described in the sections that follow.

Numeric Values

The numeric data type defines values that are used primarily in mathematical operations. In ProvideX, these values can appear in the form of literals, variables, expressions, functions, and arrays from 1 to 3 dimensions. When numeric data is output, it can include formatting, such as commas (**'TH'=**), decimal points (**'DP'=**), and currency symbols (**'CU'=**). For more information, refer to the **SET_PARAM** directive and **System Parameters**, *p.651* in the *Language Reference*.

The most common format for a numeric value is as a simple number consisting of a sign, followed by series of digits and optional decimal point; e.g.,

```
7  3.1415  -13.210 .333  -934.
```

By default, ProvideX maintains a precision of 2 (digits to the right of the decimal). During calculations, numeric values are rounded to the currently set precision. The default can be reset (up to 18 digits) using the **PRECISION** directive.

The rounding of numeric values is controlled using the **ROUND** directive. If rounding is set to the default, a ***divide*** operation in ProvideX will maintain the precision of the starting value; e.g., `1014.475/100` becomes `10.14475` which gets rounded to `10.145`. The automatic rounding of ***intermediate*** results can be turned off by setting the **'NR'** parameter. Various other types of rounding can be controlled using the **'RN'=** parameter.

Use the **FLOATING POINT** directive to set numeric data to *scientific notation*. Floating point values take the following format:

{+-} *x.xxxxx***E {+-}** *nn*

where *x.xxxxx* is a number multiplied by ten (10) raised to the power of *nn*. The following numeric values are expressed in different formats:

```
3    = 3E+00     = 0.3E+01
2.78 = 0.278E+01 = 278E-02
1000 = 1E+03     = 0.1E+04
```

Numeric Variables

The initial value of any numeric variable is 0 ***zero***. Numeric variables that are named with a trailing **%** *percent sign* are restricted for use with integers only; e.g., COUNT%. If set to a fractional value, the fractional part will be truncated; e.g.,

```
-:x%=1.8
-:?x%
1
```

ProvideX includes a set of ***system variables*** that provide access to internally-defined numeric values such as time, memory size, etc. These may be referenced like any other variable but can never be altered by the program. For the complete list, see **System Variables**, *p.553* in the *Language Reference*.

Numeric Arrays

Numeric arrays provide the ability to handle numeric lists, tables, or matrices. Arrays are created using the **DIM** directive. This directive defines the array's name, number of dimensions (one, two, or three), and minimum to maximum subscript in each dimension.

The names given to arrays are completely independent of the names associated with numeric variables; e.g., a variable named X1 would have no relationship to an array with the name X1. However, the conventions regarding the naming of numeric variables apply as well to array variables. Unless specified, arrays are zero-based.

DIM X[4] yields a one-dimensional array X with 5 elements:

X[0]	X[1]	X[2]	X[3]	X[4]

DIM X[1:4] defines a one-dimensional array X with 4 elements:

X[1]	X[2]	X[3]	X[4]

DIM Y[2,5] defines a two-dimensional array Y with 18 elements:

Y[0,0]	Y[0,1]	Y[0,2]	Y[0,3]	Y[0,4]	Y[0,5]
Y[1,0]	Y[1,1]	Y[1,2]	Y[1,3]	Y[1,4]	Y[1,5]
Y[2,0]	Y[2,1]	Y[2,2]	Y[2,3]	Y[2,4]	Y[2,5]

DIM Y[1:2,1:5] defines a two-dimensional array with 10 elements:

Y[1,1]	Y[1,2]	Y[1,3]	Y[1,4]	Y[1,5]
Y[2,1]	Y[2,2]	Y[2,3]	Y[2,4]	Y[2,5]

Access to an entry within a numeric array is specified using the array name followed by array subscripts (contained within square or round parentheses). The subscripts may be specified as **Literals**, **Variables**, or **Numeric Expressions**; e.g.,

```
A(3)  CUST[6,A]  TABLE(A*5)  Z(3,A(M,N,O))
```

Attempting to use non-integer subscript results in an `Error #41: Invalid integer encountered (range error or non-integer)`. Specifying subscripts on a variable for which no array has been defined yields an error.

Use the **DIM( )** function to determine information about array dimensions; e.g.,

```
-:DIM X[1:10]
-:PRINT DIM(READ NUM(X)) ! Read total number of elements
10
-:PRINT DIM(READ MIN(X)) ! Read minimum element number
1
-:PRINT DIM(READ MAX(X)) ! Read maximum element number
10
```

To access a ***range*** of entries, specify the array name followed by subscripts ranging ***from*** : ***to*** (or `ALL`) enclosed in ***braces*** instead of parentheses; e.g.,

Assigning a value to an entire array:	`LET A{ALL}=10`
Assigning a value to a range of array elements:	`LET A{1:5}=10`
Copying the contents of one array into another:	`LET A{ALL}=B{ALL}`
Assigning a range of values from one array to another:	`LET A{1:5}=A{6:10}`

The element positions in an array can be shifted / rearranged using a combination of subscripts (literals, variables, expressions and ranges); e.g.,

Shift up by deleting at subscript `P`:
```
X{P:ElementCount}=X{P+1:ElementCount+1}
```
Shift down by inserting at subscript `P`:
```
X{ElementCount+1:P+1}=X{ElementCount:P}
X[P]=NewValue
```

Numeric Expressions

A numeric expression can consist of numeric constants, variables, functions, and / or other numeric expressions each separated by arithmetic or logical operators. Refer to the ***footnotes*** for further information on the listed numeric operations. The following operators are grouped by ***order of precedence*** (See ①):

Auto-Increment/Decrement (See ②).

++	`++var1` *pre-increments* by 1, `var1++` *post-increments* by 1.
--	`--var1` *pre-decrements* by 1, `var1--` *post-decrements* by 1.

Exponentiation (See ③).

^	A ^ B *raises* A to the *power* of B (** is equivalent to ^).

Multiplication, Division, Modulus (See ③).

*****	A * B *multiplies* A by B.
/	A / B *divides* A by B.
|	A | B *remainder* resulting when A is divided by B .

Addition, Subtraction (See ③)**.**

+	A + B *adds* A to B.
–	A - B *subtracts* B from A.

Relational (See ④)**.**

=	A = B yields 1 if A and B are equal, else yields 0 *zero*.
<	A < B yields 1 if A is less than B, else yields 0 *zero*.
>	A > B yields 1 if A is greater than B, else yields 0 *zero*.
< >	A <> B yields 1 if the A and B are not equal, else yields 0 *zero*.
< =	A <= B yields 1 if A is less than or equal to B, else yields 0 *zero*.
> =	A >= B yields 1 if A is greater than or equal to B, else yields 0 *zero*.

Logical (See ⑤)**.**

AND	A AND B yields 1 if both values are non-zero, else yields 0 *zero*.
OR	A OR B yields 1 if either values are non-zero, else yields 0 *zero*.

Footnotes:

① ***Precedence.*** If two operators of equal precedence occur, execution takes place left to right. In the following expression,

```
A + B - C * D
```

C and D are first multiplied together and the result saved; A and B are then added together; and finally, the saved value (C*D) is subtracted.

Round parentheses ( ) may be used to change the order of evaluation. The expressions within parentheses are evaluated first. Where parentheses are nested, as in the following expression,

```
(A ^ (B * (A + 2)))
```

the innermost parenthesized expression (A + 2) is evaluated first. The use of parentheses is encouraged in complex expressions in order to make these expressions more readable and easier to understand.

② ***Auto-Increment / Decrement.*** If an error occurs during execution of a directive, no increment or decrement (**+ + *or* – –**) takes place. This is true for both pre- and post- operations.

With ***pre***-increment / decrement, the value returned is the value of the variable after the increment or decrement. With ***post***-increment / decrement, the value returned is the value of the variable before the increment or decrement. After your directive is executed, your variable is incremented or decremented by 1.

Example:

```
A=0, Y$=""
READ (1,IND=A++,ERR=*NEXT)X$; Y$+=X$; GOTO *SAME
```

③ ***Assignment Operators.*** ProvideX supports an alternate notation for defining numeric expressions that result in assignments. A combination of a numeric operator (+ - * / | ^) with an = *equals sign* is used to form shorthand expressions; e.g., `A+=1` is equivalent to the expression `A=A+1`, and `B^=A` is equivalent to the expression `B=B^A`.

See **Assignment Operators**, *p.818*, in the *Language Reference*.

④ ***Relational Operators.*** **< >**, **< =**, and **> =** may be entered as **> <**, **= <**, and **= >** respectively.

⑤ ***Logical Operators.*** When ProvideX encounters either an **AND** or an **OR** logical operator, it attempts to perform a shortcut in the evaluation of the expression. If the value to the left of an **AND** operator is zero (false) then the expression / value on the right is skipped and the relationship returns 0 *zero*. If the value to the left of an **OR** is non-zero then the expression / value on the right is skipped and the relationship returns 1.

Numeric System Functions

ProvideX includes various internal functions that return numeric values based on the parameters provided. System functions can be used to evaluate or convert specific values, but many of them perform built-in mathematical (arithmetical or algebraic) calculations. These include trigonometry (**SIN( )**, **COS( )**, **TAN( )**, **ASN( )**, **ACS( )**, **ATN( )**), logical comparison (**AND( )**, **IOR( )**, **XOR( )**, **NOT( )**), and several other numeric operations (**EPT( )**, **EXP( )**, **LOG( )**, **MAX( )**, **MIN( )**, **MOD( )**, **PRC( )**, **SQR( )**).

For details, refer to **System Functions**, *p.387* in the *Language Reference*.

String Values

The string data type defines values that represent any sequence of displayable characters (letters, numbers, spaces, punctuation symbols, etc.). Strings typically contain ASCII data but may contain any sequence of 8-bit bytes of data. In ProvideX, these values can appear in the form of literals, variables, arrays, and substrings. String manipulation facilities include concatenation, comparing, scanning and conversion.

String literals must be contained within **""** *quotation marks*. Spaces within the quotation marks are considered an integral part of the string; i.e.,

`" Sage Software Ltd. "` ***is not the same as*** `"Sage Software Ltd."`

In order to include an actual quotation mark symbol within a string literal, two quotation marks must be specified back-to-back; e.g.,

`"My name is ""Joe"""` ***produces the result*** `My name is "Joe"`

Hexadecimal String Literals

Hexadecimal string literals provide the ability to define a string of data which may contain other than displayable ASCII characters. In ProvideX, a hexadecimal string is delimited by two dollar signs. A single character is defined by a pair of hexadecimal digits (**0-9**, **A-F**) with each pair representing a single byte of data. The following are examples of hexadecimal strings:

```
$414243$    =   "ABC"
$30313233$  =   "0123"
$0D0A$      =    Carriage Return /  Line feed.
```

String Variables

The name of a string variable is always terminated by a single **$ *dollar sign***; e.g.,

```
A$          Name$       CUST_ADR$
User.ID$    X1$         WEEK_DAY$
```

A null string is defined as a string that contains no data, in other words a string that is set to "" or $$ and whose length is 0 *zero*. Initially all string variables are defined as null strings. String variables can also be defined via the **DIM** directive, which allows the programmer to define the length (and contents) of a string variable.

ProvideX includes a set of ***system variables*** that provide access to internally-defined string values such as formatted date, pathname, etc. These may be referenced like any other variable but are generally read only. For the complete list, see **System Variables**, *p.553* in the *Language Reference*.

String Arrays

String arrays, like **Numeric Arrays**, provide the ability to handle lists, tables, or matrices. They are also defined and referenced in the same manner as numeric arrays; The only distinguishing feature between the two types of arrays is that string array names always end with a **$** *dollar sign*.

Arrays are created using the **DIM** directive. This directive defines the array's name, number of dimensions (one, two, or three), and minimum to maximum subscript in each dimension. For example, a one-dimensional array with the name ADR$ with eleven elements would be defined as follows:

```
DIM ADR$[10]
```

Access to an entry within a string array is specified by the array name followed by the array subscript (contained within square brackets). The subscripts may be specified as **Literals** or **Variables**.

Use the **DIM( )** function to determine information about array dimensions; e.g.,

```
-:DIM X$[1:10]
-:PRINT DIM(READ NUM(X$)) ! Read total number of elements
10
-:PRINT DIM(READ MIN(X$)) ! Read minimum element number
1
```

```
-:PRINT DIM(READ MAX(X$)) ! Read maximum element number
10
```

To access a ***range*** of entries, specify the string array name followed by subscripts ranging ***from*** : ***to*** (or ALL) using ***braces*** instead of parentheses; e.g.,

Assigning a value to an entire array:	`LET A${ALL}="ABC"`
Assigning a value to a range of array elements:	`LET A${1:5}="ABC"`
Copying the contents of one array into another:	`LET A${ALL}=B${ALL}`
Assigning a range of values from one array to another:	`LET A${1:5}=B${6:10}`

The element positions in an array can be shifted / rearranged using a combination of subscripts (literals, variables, expressions and ranges); e.g.,

Shift up by deleting at subscript P:
```
X${P:ElementCount}=X${P+1:ElementCount+1}
```
Shift down by inserting at subscript P:
```
X${ElementCount+1:P+1}=X${ElementCount:P}
X$[P]=NewValue$
```

Substrings

A substring consists of a portion of a string variable. Substrings are accessed by specifying the string variable followed by the *starting character* position within the string and (optionally) the *length* of the substring enclosed by parentheses. If no length is specified, then the substring consists of all characters from the starting character up to and including the last character within the string. For example, if string A$ contains "ABCDEFGHIJK", then the following are valid:

```
A$(1,1) = "A"
A$(3,4) = "CDEF"
A$(4) = "DEFGHIJK"
```

A substring must not exceed the current size of the string variable it references. If a variable Z$ contains "ABC", then Z$(3,2) would be invalid (resulting in Error #46: Length of string invalid). One exception to this rule would be the substring Z$(4) or Z$(4,0), which would equate to a null string. To avoid Error #46, the **MID( )** function can also be used to return a substring.

Substrings may be used wherever a string variable is used. When the value being assigned to a substring is less than the length of the substring, it will be padded with space characters. If the string is longer it will be truncated.

The following examples assume string A$ contains "ABCDEF":

```
LET A$(2,2) = "XX"     yields     A$="AXXDEF"
LET A$(2,2) = "X"      yields     A$="AX DEF"
LET A$(2,2) = "XXX"    yields     A$="AXXDEF"
LET A$(2) = "X"        yields     A$="AX    "
```

Substrings can also be used on string arrays by specifying both the array element (within square brackets) and the substring (within parentheses); e.g.,

```
ADDR$[1,2](2,3)
```

The above substring indicates characters 2 through 4 of element 1,2.

String Concatenation

The mathematical **+** *plus* operator can be used to concatenate strings. When two strings are concatenated, the resultant string consists of the contents of the string left of the **+**, followed immediately by the contents of the string to the right; e.g.,

```
->x$="hello"+"world";print x$
helloworld
->a$="Sage Software",b$="Canada Ltd."
->print a$+" "+b$
"Sage Software Canada Ltd."
```

Any number of strings may be concatenated as long as the total length of the resultant string does not exceed current **System Limits**.

String Comparison

The following relational operators can be used to compare two strings:

=	A$ = B$ yields 1 if A$ and B$ are equal, else yields 0 *zero*.
<	A$ < B$ yields 1 if A$ is less than B$, else yields 0 *zero*.
>	A$ > B$ yields 1 if A$ is greater than B$, else yields 0 *zero*.
< >	A$ <> B$ yields 1 if the A$ and B$ are not equal, else yields 0 *zero*.
< =	A$ <= B$ yields 1 if A$ is less than or equal to B$, else yields 0 *zero*.
> =	A$ >= B$ yields 1 if A$ is greater than or equal to B$, else yields 0 *zero*.

< >, **< =**, and **> =** may be entered as **> <**, **= <**, and **= >** respectively.

In string comparisons, each character in one string is compared to the corresponding character in another string, to yield a binary result. The comparison is performed on the internal binary value of each byte.

When comparing strings of unequal lengths (and the longer string matches the shorter string for the full length of the shorter string), the longer string is considered the greater of the two strings; e.g.,

`"Sage Software"` is ***always less than*** `"Sage Software Canada Ltd."`

String System Functions

ProvideX includes various internal functions that return string values based on the parameters provided. System functions can be used to evaluate, convert, and format strings, or to determine system and file information. For details refer to **System Functions**, *p.387* in the *Language Reference*.

Composite Strings

ProvideX provides the ability to define strings as a collection of data elements. These strings are called ***composite strings***. A composite string is basically a record defined by an **IOLIST** statement consisting of variables and associated formats used to create the string. Assigning data to a composite string causes the elements defined in the IOList to be loaded with their associated values. Referencing a composite string returns all the variables in their respective formats as defined by the IOList, thus it can be considered a logical record or structure.

Defining a Composite string

The ProvideX **DIM** directive is used to define a composite string. Provide the name of the string variable, a colon, then the associated IOList.

For example, to define a composite string consisting of 4 variables:

```
0100 DIM X$:IOL=1000
1000 IOLIST A$,B$,X,Y
```

or

```
0100 X_IOL$=PGM(1000); DIM X$:X_IOL$
1000 IOLIST A$,B$,X,Y
```

or

```
0100 DIM X$:CPL("IOLIST A$,B$,X,Y")
```

Unlike normal IOLists, the variables referenced in a composite string will be ***prefixed*** with the name of the composite string. Therefore, the variables in the above example referenced by X$ are not A$, B$, X, and Y but rather X.A$, X.B$, X.X, and X.Y. This is comparable to the **REC=** option found in input/output directives (see **Processing Data Files** , *p.117*).

Referencing a Composite String

When a program references a composite string, it will receive a string comprised of the variables defined in the IOList.

Example 1:

Given the following composite definition and values

```
0100 DIM X$:IOL=1000
1000 IOLIST A$,B$,X,Y
X.A$="CAT", X.B$="DOG", X.X=1,X.Y=2
```

Referencing X$ ***will yield:***

CAT-*SEP*-DOG-*SEP*-1-*SEP*-2-*SEP*-

Since the IOList specified general formatting of the data, each field was placed in ASCII in the output with a standard field separator (-*SEP*-) between them.

Example 2:

Given the following:

```
0100 DIM CST$:IOL=1000
1000 IOLIST NAME$:[CHR(20)],ADR1$:[CHR(20)]

CST.NAME$="Sage Canada"
CST.ADR1$="8920 Woodbine"
```

Referencing `CST$` ***will yield:***

```
"Sage Canada         8920 Woodbine       "
```

Since the IOList specified formatting, the output consists of the name (`NAME$`) padded to 20 characters followed by a 20 character address (`ADR1$`).

Assigning Data to a Composite String

When a program updates a composite string, the variables that make up the composite will be updated automatically.

Example 1:

Given the following composite definition and values.

```
0100 DIM X$:IOL=1000
1000 IOLIST A$,B$,X,Y

X$="SAGE"+SEP+"PROVIDEX"+SEP+"123"
```

Will result in:

```
X.A$="SAGE"
X.B$="PROVIDEX"
X.X = 123
X.Y = 0
```

Since the data assigned to the composite string `X$` only contains 3 fields, the fourth field (`X.Y`) will be set to zero.

Example 2:

Given the following:

```
0100 DIM CST$:IOL=1000
1000 IOLIST NAME$:[CHR(20)],ADR1$:[CHR(20)]

CST.NAME$="Sage Canada"
CST.ADR1$="8920 Woodbine"
LET CST$(1,20)=""
```

The above `LET` will result in the field `CST.NAME$` being set to a null string.

3
Development Tools

The ProvideX development environment is equipped with a number of facilities for creating, modifying, and maintaining ProvideX program code.

Topics

Writing and Modifying Program Code

While ProvideX allows external methods for writing source instructions, a program cannot be converted to tokenized code and executed unless it exists within an active ProvideX session. In Command mode, the ***current*** program represents all the statements in memory that are immediately available for editing and / or execution. In Execution mode, the current program is also called the ***main-line*** program (execution level 1).

Whether you create a new program or **LOAD** an existing program, the changes you make to numbered statements at the command line apply to the current program in memory.

Note: The current program does not have to be saved to be **RUN**. Conversely, saved program files can be run directly without first being loaded; although, the **RUN** will automatically load the specified program if not already loaded.

ProvideX programs can be edited using a variety of techniques, including command line editor, `*E` character-based editor, `*it` graphical editor, the ProvideX Plug-in for Eclipse, as well as any other text editor that saves source code in ASCII format.

Numberless Programs

Line Numbers are considered *optional* in ProvideX. However, numberless programs can only be created using the ProvideX **GUI-based Program Editor (*IT)** or some other text editing tool (*because non-numbered statements would be executed immediately at the command line*). Once created, a numberless program may be loaded and listed (and edited) in Command mode. When issuing a **LIST** of a numberless program, sequential line numbers will be inserted automatically (for reference / editing purposes).

Command Line Editing

The simplest way to create / change a program involves entering numbered statements directly at the prompt **->** in Command mode. As each numbered statement is entered, it is evaluated for syntax, compiled, and then placed in memory to represent a line in the current program. While rudimentary, the command line editor has all the tools necessary for creating, editing, and saving any size of program.

During this process, the prompt changes from **->** to **-:** to indicate that the current program is being edited and has not been saved. The first numbered statement adds the first line to a new current program, the second, adds another line, and so on. Regardless of the sequence entered, the lines in the current program are rearranged internally based on their line numbers, and are always executed in ascending order. This is true whether you create a new program in memory or **LOAD** an existing program from disk.

The basic functionality for using the ***keyboard*** or ***mouse*** is described under **ProvideX Session**, *p.11*. Other command line facilities are described below.

Using the **LIST** *Directive*

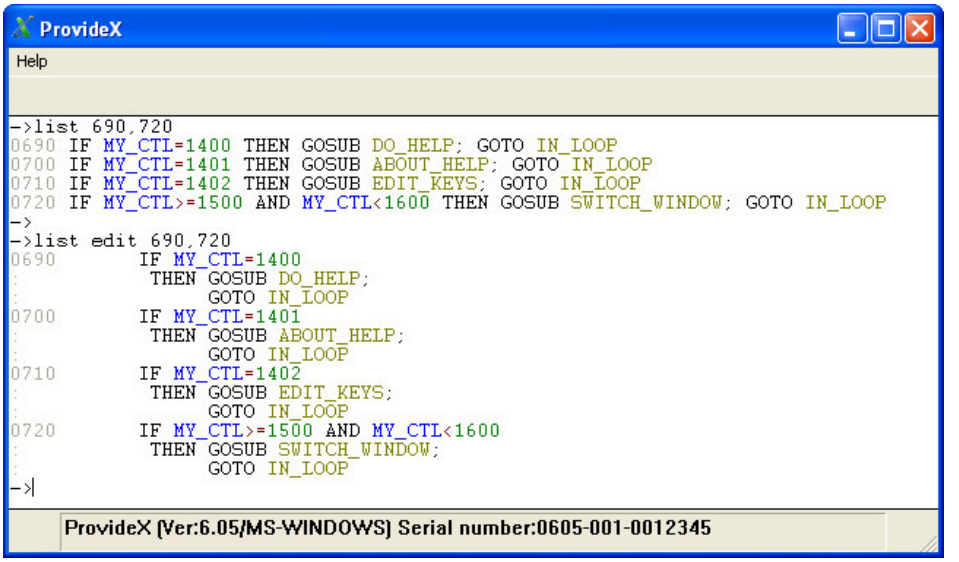
At any point during the editing process, you can review the entered statements in their proper numeric sequence by issuing a **LIST** of the current program. This gives you a listing of the entire contents of a program.

Because a program is maintained internally in compiled (object) format, the **LIST** version of some statements may not always match the original entries, but the actual logic will appear as intended. In the following example, subtle changes are made to line 40 after it is listed:

```
->40 z=1,y=6; goto 10
-:list
0010 LET A=4,B=3
0020 LET C=A+B
0030 PRINT C
0040 LET Z=1,Y=6; GOTO 0010
```

Once listed, the keyword **LET** is inserted, the lowercase variable names are converted to uppercase, and the line number is expanded to four digits.

The **EDIT** keyword can be used with the **LIST** directive to format the display by indenting loops and compound statements (see example below).



The behaviour of the **LIST** directive can be affected by various system parameters. When turned on, **'LC'** lists variables in lowercase, **'MC'** lists variables using mixed case (dependent on how they were initially entered), and **'LE'** causes **LIST** to display the same as **LIST EDIT**.

Statement Errors

If an error occurs during compilation, the flawed statement will be indicated by a **?** *question mark* between the line number and the text; e.g.,

```
->10 A="Hello"
#26 -- Variable type invalid
->LIST
0010?A="Hello"
```

Modifying Existing Lines

When a numbered statement is entered with the same line number as a line in the current program, the existing line is automatically overwritten by the new statement. If you wish to modify rather than overwrite the statement, use the **EDIT** directive (or ` shortcut) to recall the specific line number; e.g.,

```
-:list
0010 PRECISION 2
0020 ROUND OFF
0030 LET A=3*(2/3)
0040 PRINT A
-:edit 30
-:0030 LET A=3*(2/3)
```

Once displayed, the contents of the line (including the line number) can be changed and reentered. Refer to the **EDIT** directive in the *Language Reference, p.107*.

Program code can also be copied, pasted, and edited using a variety of other techniques that include ***command line recall*** (up / down cursor keys), as well as the Edit function, which is accessed via the  drop-down menu in the ProvideX (Windows) console.

To ***append*** text to an existing statement, enter the line number followed by a colon; e.g.,

If an existing statement is

```
0030 PRINT "End-of-pass"
```

then

```
->30:;NEXT I
```

results in

```
0030 PRINT "End-of-pass"; NEXT I
```

Deleting Existing Lines

To remove lines from a program, simply enter the desired line number followed by no directive (e.g., entering a `20` deletes all of line `0020`). Optionally, a range of lines can be deleted using the **DELETE** directive; e.g.,

```
DELETE 260,890 ! Deletes statements from line 0260 to line 0890
DELETE 260, ! Deletes statements from line 0260 to program end
DELETE ,890 ! Deletes from program start to line number 0890
```

Autonumber Sequences

In Command mode, statements must have line numbers to be accepted in the current program, otherwise they are executed as commands. If you prefer to add numbered statements without having to type the actual numbers, use the **AUTO** directive to generate the line numbers automatically; e.g.,

```
->AUTO 100,10
0100 ! ProvideX generates lineno 0100 for you. Add the statement.
0110 ! ProvideX adds 10, generates line 0110.
0120
```

AUTO mode remains active until you enter a null line. You can also backspace over to change the generated line number.

Renumbering

Use the **RENUMBER** directive to change line numbers in an existing program without affecting the logic. All references to the original line numbers will be adjusted as required; i.e., **GOTO** and **GOSUB** statements will point to the new line numbers. The starting line number, increment, and the range of lines affected can be defined using this directive.

Search and Replace

Command mode includes a built-in utility that allows the programmer to search the current program for a specific character string, display it and optionally replace it. The symbols used to implement this functionality are listed in the section **Primary Syntax Elements** , *p.32*. Subject character strings are delimited by square brackets:

`[PRINT]`	Displays the next line that contains the word `PRINT`.
`10,100[PRINT]`	Displays all lines between `10` and `100` containing `PRINT`.
`*[PRINT]`	Displays all lines containing `PRINT`.
`*[PRT$]=[PRINT$]`	Changes all instances of `PRT$` to `PRINT$`. The replacement can also apply to a range of lines (e.g., `10,100[PRT$]=[PRINT$]`) or be limited to the next line only (`[PRT$]=[PRINT$]`).

Saving, Loading, Running

Use the **SAVE** directive to copy the current program in memory to a specified file name on disk; e.g.,

```
SAVE "Myprogram"
```

As mentioned earlier, programs that are saved to disk, may be recalled using the **LOAD** directive. This clears the current program (if any) and replaces it with the program specified; e.g.,

```
LOAD "Myprogram"
```

The **RUN** directive is used to execute the current program. If **RUN** includes a program name, the system will automatically load the specified program from disk before executing it; e.g.,

```
RUN "Myprogram"
```

The above information is covered in more detail in the section **Directives, Statements, and Programs** , *p.22*.

Full-Screen Editors

The command line editor is by no means the only way to create / revise a program in ProvideX. The base system comes with two full-screen editors for program development and maintenance.

Character-Based Program Editor (***E**)

The ***E** utility provides a character-based editor for editing the current program on the screen. One of the primary advantages of using this editor is that it runs in all ProvideX environments – text or graphical.

To invoke the utility, select Edit from the main utilities menu or enter CALL "*E" at the ProvideX prompt **->**. If a program is currently running when the editor is invoked, it will display a termination message; otherwise, the editor displays the current program on the screen. The top of the screen contains the editor menu:

```
F1-Text edit F2-Line edit F3-Program F4-Quit
```

The cursor keys (or mouse) can be used to position the cursor anywhere in the displayed program. Program compile errors are displayed on the screen below the lines on which they occur.

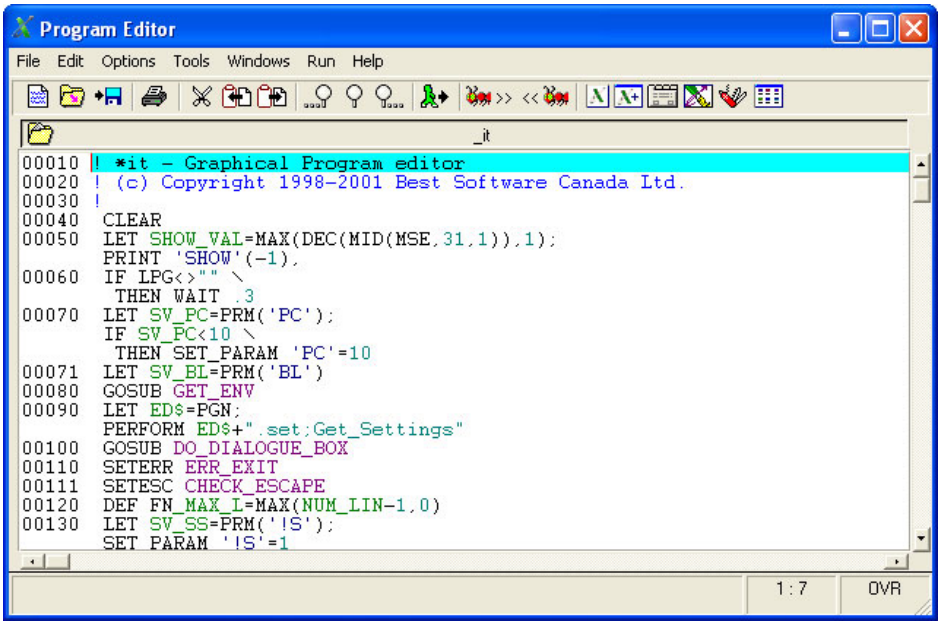
GUI-based Program Editor (***IT**)

The GUI-based editor requires either Windows or the use of one of the ProvideX graphical thin clients. This utility is specifically designed for writing and modifying ProvideX code. It can maintain programs with / without line numbers and allows the source to be loaded / saved either as ASCII text or as program files.

There are multiple ways to invoke the Program Editor:

- Click the Edit menu item in the graphical System Utilities interface.
- CALL "*it" from the command line.
- Enter it (*CMD shortcut).
- Click the editor tool button next to logic entries in NOMADS.

The Program Editor is completely menu-driven but can also be manipulated using various quick-key combinations. The interface appears as follows:



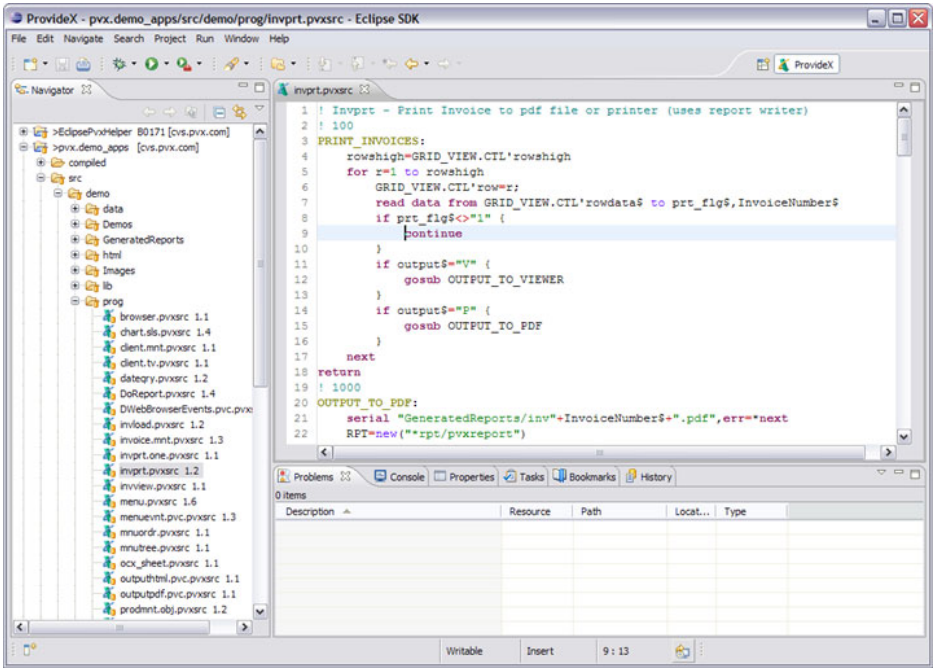
It also includes a number of added features that are not available (directly) to the other editing tools, including:

- Formatted text with color highlighting
- Automatic syntax checking
- Line number removal and insertion facilities
- Auto renumber and bracketing
- Built-in print facility
- Direct access to other ProvideX utilities such as NOMADS and panel execution.

A single user may have the same file opened multiple times in one edit session. The Program Editor keeps track of the last version saved, and displays a warning message if a previous version is being saved over the latest version. If a user attempts to open a file currently being edited by another user, a warning message is presented. If the second user continues and opens the file, neither user can save the file until one relinquishes control by closing the file.

ProvideX Plug-in for Eclipse

In addition to the tools included in the base system, a ProvideX plug-in has been developed for use within the Eclipse development platform.



Tutorials, documentation, and installation instructions are available via the ***ProvideX Plug-in for Eclipse*** product page: www.pvx.com/downloads/eclipse.plugin.

About the Eclipse Platform

Eclipse is an open platform for integrating development tools that has wide support among many of the world's leading technology companies and organizations. It is designed to be easily and infinitely extensible by adding products / tools to the Eclipse SDK (software development kit) in the form of ***plug-ins*** (usually developed separately by third parties).

Plug-ins are the individual software components that make up the Eclipse platform. In fact all Eclipse functionality exists in the form of a plug-in (except for the kernel). Each is a self-contained bundle in the sense that it contains the code and all resources that it needs to run. For more information on the Eclipse platform, refer to *Eclipse Foundation* website:

http://wiki.eclipse.org/index.php/The_Official_Eclipse_FAQs

About the ProvideX Plug-in

This ProvideX plug-in is designed specifically for the development of ProvideX applications within the Eclipse framework and is available for use with ProvideX activations (as of Version 7) at no additional charge. It adds a ProvideX project, nature, and perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and other useful tools.

Pre-requisites for using the plug-in are as follows:

- Minimum of ProvideX v7 activation.
- On MS-Windows a 2 user license (or a demo) is currently required.
- Java SE 5 (1.5.x) run-time environment.
- Eclipse Version 3.2 or higher (MS Vista requires Eclipse Version 3.2.2 or higher).

Feature Summary

Following are some of the key features included with the ProvideX plug-in:

PvxDocs. Automatic generation of documentation for ProvideX Class definition from tags that are embedded in the source code.

XMI Converter. Conversion of UML models saved in XMI format to one or more ProvideX Class files.

XML / ProvideX Keyed Files Converter. Conversion of ProvideX keyed files into XML *and vice versa*.

NOMADS Integration. All actions that are available from the stand-alone NOMADS front-end are accessible from within the plug-in.

Data Dictionary Viewer. Displays tables, records, columns and keys for a ProvideX **Embedded Data Dictionary**, *p.117*.

Message Library Viewer. Displays message codes and their texts from the selected message library.

Super Search Utility. Simple interface for defining complex searches to be performed against the source code for an application.

Style Checker. Mechanism for verifying that the source code for an application follows the style rules established by the developer, and for reporting any discrepancies that are found.

Note: Complete documentation is provided in the *ProvideX Development User Guide* (under Eclipse SDK > Help) included with the ProvideX Plug-in installation. Visit the ***ProvideX Plug-in for Eclipse*** product page: www.pvx.com/downloads/eclipse.plugin

Error Handling and Debugging

Finding errors in code is often the most cumbersome and time-consuming task in programming. ProvideX features several utilities and language elements that will help you locate and handle errors during the coding, testing, and debugging stages of the development cycle.

Topics

Note: For trouble-shooting procedures involving the installation, startup, and configuration of ProvideX, refer to the *Installation and Configuration* guide.

Error Codes and Messages

As noted previously, the interpretive nature of the development environment ensures that statements are automatically checked for correct syntax before they are tokenized. This is true whether you are entering a command at the ProvideX prompt, or program code using a full-screen editor.

When an erroneous statement is entered, an error message is displayed immediately:

```
->print hello world
Error #20: Syntax error ...world
->print "hello world"
hello world
```

The resulting `Error #20:` indicates a syntax error. Refer to the complete list of errors provided under **Error Codes and Messages** in the *Language Reference, p.824*.

Execution Errors

During program execution, ProvideX may detect various errors due to problems with program logic, invalid data, or status conditions (end-of-file, duplicate record, etc.).

All errors detected by ProvideX 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 ProvideX-specific messages. Error codes starting from `256` are operating system (OS) errors. When ProvideX detects an error during program execution:

1. The system variable **ERR** is set to the appropriate error code.
2. The program is halted.
3. ProvideX returns to Command mode (unless the **'XT'** parameter is set, in which case ProvideX is also terminated).

4. The statement that generated the error is displayed.

5. The error code and its associated message text is displayed.

Refer to the complete list of errors provided under **Error Codes and Messages** in the *Language Reference, p.824*.

Displaying Error Messages

As mentioned earlier, the **ERR** variable contains the error code for the last system-detected error. Use the **MSG( )** function to display a description of the error code; e.g.,

```
->PRINT MSG(11)
Error #11: Record not found or Duplicate key on write
->PRINT MSG(275)
No more files
->OPEN(1)"COM10"
Error #12: File does not exist (or already exists)
->PRINT MSG(-1)
IE_BADID Invalid/unsupported device I.D.
```

Descriptive messages can also be returned for OS errors, provided the information is available from the OS.

The statement number where the last error occurred is available in the system variable **ERS**. This information can also be obtained via the **TCB( )** function: `TCB(5)` or `TCB(30)`. The system parameter **'ES'**, if enabled, will display any OS error message returned along with the normal ProvideX error from a command prompt; e.g.,

```
SET_PARAM 'ES'
OPEN (1)"[ODB]Access;FooFoo"
    Error #15: Operating system command failed
    IM002: [Microsoft][ODBC Driver Manager] Data source name not found
           and no default driver specified
```

Extended Error Information

The **ERR( )** function provides additional information for diagnosing the most recent untrapped error. This information is not affected by errors that are programmatically trapped using the procedures described under **Error Processing**, *p.58*.

The syntax for this feature, **ERR(***keyword$***)**, denotes a *keyword$* representing the specific return value. Use `ERR("*")` to display a list of supported keywords. For complete details, refer to the *Language Reference, p.425*.

Error Processing

A program should be designed to handle most of the errors anticipated during normal program operation. This helps avoid an unwelcome drop to Command mode (and / or cryptic messages displayed to the end-user). The procedure for detecting and processing errors without dropping to the system is commonly referred to as ***error trapping*** (or ***error handling***).

There are three methods for dealing gracefully with program errors. The order of precedence for handling errors in ProvideX is defined as follows:

1. **Error Transfer Option – ERR=** syntax within directive or function.
2. **Error Handling Subroutine – SETERR** directive.
3. **Error Handling Program – ERROR_HANDLER** directive.

Error Transfer Option – ERR=

Most ProvideX directives and functions allow an error transfer option to be included within their syntax. This option is denoted by the **ERR=s***tmtref* clause, where **s***tmtref* specifies a line number or label to which to transfer control. When a statement that includes an **ERR=** option generates an error, the branch to **s***tmtref* will be executed; e.g.,

```
0010 OPEN (2,ERR=0100) "CONFIG"
0020 READ (2,ERR=0050) R$
0030 PRINT R$
0040 GOTO 0020
0050 CLOSE (2)
0060 STOP
0100 PRINT "Unable to open file"
0110 END
```

If, in the preceding example, an error occurs on the **OPEN** directive, control will transfer to statement `0100`. If an error occurs on the **READ** (most likely an end-of-file), control will transfer to statement `0050`. For more information on **ERR=** and other control options, refer to the *Language Reference, p.807*.

Error Handling Subroutine – SETERR

Use the **SETERR** directive to define a subroutine for handling errors in your program that are not covered by the **ERR=** option. While **SETERR** is in effect, all errors will cause an immediate transfer of control to the line number or label specified; e.g.,

```
0010 SETERR YIKES
0020 OPEN (2) "config"
0030 LOOP: READ (2) R$
0040 PRINT R$
0050 GOTO LOOP
0100 YIKES: !!! Error handler !!!
0110 IF ERR=2 THEN CLOSE (2); STOP
0120 PRINT "error ",ERR," while printing config"
0130 END
```

In the preceding example, statement 0010 defines the general error handling procedure starting at YIKES (statement 0100). After the execution of the **SETERR**, any error will cause control to transfer to statement 0100. Statement 0110 tests the system variable **ERR** for an END-OF-FILE status which is returned by the **READ** directive on statement 0030. Any other error would cause the generation of an error message.

The execution of a **SETERR** specifying a statement number of 0000 will disable the general error procedure. In addition, the execution of a **BEGIN**, **CLEAR**, **END**, **STOP**, or any directive that causes a program to be loaded will reset the **SETERR** location.

Note: **SETERR** can also be used to specify an error-trapping ***program***, which is similar in functionality to the **ERROR_HANDLER** directive described later in this section. For syntax details, see **SETERR** in the *Language Reference, p.312*.

Retrying an Error

Sometimes it is necessary to re-execute the directive that caused the error after some corrective action has been taken. A typical example of this would be to repeat an I/O request on a record or file that is currently busy.

Whenever an error occurs and an **ERR=** or **SETERR** transfer occurs, the location of the directive that caused the error is saved. To return to the saved location, the error handling procedure can execute a **RETRY**. The **RETRY** directive transfers control back to the statement that initiated the error procedure; e.g.,

```
0010 OPEN (2,ERR=0100) "config"
0020 READ (2,ERR=0050) R$
0030 PRINT R$
0040 GOTO 0020
0050 CLOSE (2)
0060 STOP
0100 REM See if file busy (ERR=0)
0110 IF ERR=0 THEN PRINT "One moment please"; WAIT .5; RETRY
0120 PRINT "Unable to open file"
0130 END
```

In this example, the **RETRY** directive is used in statement 0110 to return to the **OPEN** directive should the error code indicate that the file was busy. The **WAIT** directive suspends the execution of the program for half a second before retrying.

There is one exception to the **RETRY** procedure. When an **ERR=** option is used on a statement to transfer control to the same statement (i.e., to cause the statement to repeat in case of an error), the **RETRY** location is neither saved nor modified.

RETRY also returns control to directives within a compound statement; however, should control be returned to a **LET** directive with multiple assignments, all assignments will be repeated. For example, if the following statement is retried, the value of A would be re-evaluated incorrectly since B will have changed:

```
0010 LET A=B+C, B=2*B, C=B/(D-A)
```

Error Handling Program – ERROR_HANDLER

The **ERROR_HANDLER** directive can be used to assign a generic program for handling untrapped errors. If an error occurs, and it is not handled via **ERR=** option or **SETERR** subroutine, the system will call an error handling program (with optional entry point) specified by:

ERROR_HANDLER *prog$*[;*entry$*]

By placing all common error-trapping procedures in a single program, the same logic can be used by multiple applications throughout the ProvideX session. The called program determines the necessary corrective actions and, once the recovery is performed, returns to the offending instruction via the **EXIT** directive. Should the error handler specify an error code on the **EXIT** directive, ProvideX will return to Command mode allowing the program to be corrected. If desired, the error handler can execute a **START** directive to abort the session and restart.

Note: ProvideX includes a sample error handler program that can be used as a template for creating application-specific routines; e.g., `ERROR_HANDLER "*ERROR"`.

To determine the error handler currently in effect, enter **ERROR_HANDLER READ** *var$*, where *var$* is the string variable to receive the program name.

Avoiding Endless Loops in Error Handling

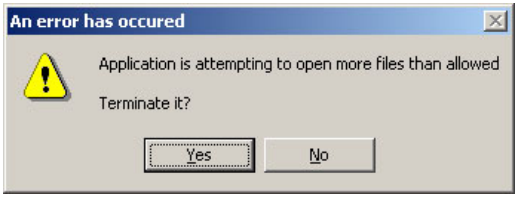
ProvideX can prevent endless loops that are caused by errors within an error handling subroutine. Once a **SETERR** transfer takes place, ProvideX inhibits further **SETERR** transfers until a subsequent **SETERR** is executed or a **RETRY** directive re-executes the statement that caused the error.

However, endless loops can occur on certain ***system errors*** that result in a call to the generic error-handling program. For example, if the **ERROR_HANDLER** is invoked due to an "out of file handles" system error, ProvideX will attempt to load the error handling program. At this point, because the "out of file handles" condition is still in effect, the result would be an endless loop (see **Infinite Loops**, *p.74*).

You can avoid this situation by placing the error handler program in cache memory (via the **ADDR** directive). Use **ADDR** prior to executing **ERROR_HANDLER**, otherwise an endless loop may occur when the system attempts to report `Error #12: File does not exist (or already exists)`.

The error handler program should not generate any un-trapped errors. If the error handler issues a **CALL** or **PERFORM** that causes an error outside of itself, the error handler will be called again, thus creating another possibility for an endless loop.

As an additional safeguard in the event that an un-trapped *out of file handles* condition does occur, ProvideX will first attempt to report the error condition to the application. If no corrective action is taken by the application then the following message is displayed, prompting the user to decide how to proceed:



Tracing a Program

ProvideX provides the ability for a programmer to trace the execution of a program via the **SETTRACE** directive. Each statement will be listed as it is executed. This trace output can either be displayed or written to a file.

The trace output will contain a listing of the statement that is being executed. If a statement repeats itself then only one occurrence of the output will be displayed. If a compound statement executes **GOSUB**, **FOR**, or **WHILE** directive as other than the final directive, it will appear twice in the trace, once when the directive is executed and once when control returns to the statement; e.g.,

```
->LOAD "PRIME"
->LIST
0010 INPUT "Enter test prime:",prime
0020 FOR I = 2 TO INT(SQR(PRIME))
0030 IF INT(PRIME/I)*I = PRIME THEN EXITTO 0100
0040 NEXT I
0050 PRINT "Yup, its a prime"; STOP
0100 PRINT "Nope, it ain't prime"; STOP
->SETTRACE
->RUN
0010 INPUT "Enter test prime:",prime
Enter test prime:11
0020 FOR I = 2 TO INT(SQR(PRIME))
0030 IF INT(PRIME/I)*I = PRIME THEN EXITTO 0100
0040 NEXT I
0030 IF INT(PRIME/I)*I = PRIME THEN EXITTO 0100
0040 NEXT I
0050 PRINT "Yup, its a prime"; STOP
Yup, its a prime
```

The tracing of a program remains active until either an **ENDTRACE** is executed or the program terminates.When trace output is sent to a file by specifying **SETTRACE** (*chan*) make sure that the file chosen does not conflict with any files in use by the program. Make sure the program does not accidentally close the file by executing a **BEGIN** directive. See also **TraceWindow**, *p.63*.

Stepping Operations

Another common debugging technique involves putting an **ESCAPE** directive into a program and then "stepping" through the code in console mode during execution. This is an extremely useful method for tracing and following complex problems within program logic. It allows you to check the value of variables at different points and track program flow.

Use the following shortcut keys followed by ENTER to apply a stepping operation.

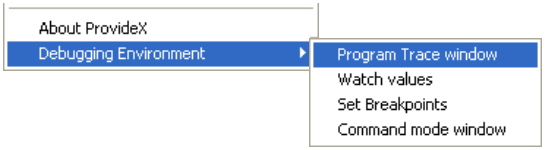
Keys	*Stepping Operation*
.	Executes next line of code and returns to command prompt.
.*n*	Executes *n* lines of code and returns to command prompt.
..	Steps through stack entries (e.g., **FOR..NEXT**, **GOSUB..RETURN**, or **CALL..PERFORM..EXIT**) until completion.
...	Completes execution of (and then exits) the current program level (**CALL..PERFORM..EXIT**) stack.
;	Steps through compound statements.
;*n*	Steps through *n* directives in a compound statement (and then stops).

The F2 key (*without* ENTER) can be used to repeat the last type of stepping operation performed. (It emulates the . *period* by default.) The Step menu option in the **CommandWindow** works in much the same way (See below).

Windows Debugging Environment

The Windows version of ProvideX includes a debugging environment that can be accessed via the drop-down menu. It comprises four separate windows that allow you to display lines of code, set break points, monitor variables and / or expressions, and work in a stand-alone command console while executing a program for testing.

To access these tools, simply press the key combination Alt - SPACEBAR, **D** then select one of the following debugging actions: **TraceWindow**, **WatchWindow**, **BreakWindow**, or **CommandWindow**.



Debugging functionality can also be set on-the-fly in ProvideX using the **'OPTION'** mnemonic; e.g., `PRINT 'OPTION'("DebugWindow","Trace")`. For more information, refer to the **'OPTION'** mnemonic in the *Language Reference, p.622*.

Disabling the Debugging Environment

The debugging facilities are automatically disabled if a lead program (**LPG** system variable) is specified in the command for launching ProvideX. This can be overridden by adding the line `DEBUG=1` to the `[Config]` section of the INI file. (The name of the INI file can be obtained via the **ARG(-1)** function.) The debugging environment can be disabled by setting the `DEBUG` option to a value of `-1` (negative one) in the INI file. These options are fully explained under **INI Files (Windows)** in the *Installation and Configuration* guide.

TraceWindow

TraceWindow is used to trace the execution of an application. It displays (up to) the last 4096 lines executed, which can be saved to a file for subsequent analysis. The menu items for this window include:

Options

Always on top	Displays TraceWindow always on top of ProvideX.
Font	Changes font and font size for text in TraceWindow.
Auto-Start	Auto-activates TraceWindow when ProvideX starts.
Log all Errors	Logs errors trapped by **ERR=**, **DOM=**, **SETERR**.
Suppress Program Trace	Suppress normal program tracing.
Host Trace	*WindX only* - Log All Errors and Trace Programs options for server-side programs. Trace lines from the host are prefixed with `<h>`.
Show Property GET	Enables / disables property **GET** option.
Show Property SET	Enables / disables property **SET** option.
Trace File Opens	Enables / disables trace file opens option.
File Open Failures	Enables / disables trace file open failures option.
File IO Operation Trace	Enables / disables file IO operation trace option.
DebugPlus with Backtrace	Enables / debugplus backtrace option.

Edit

Copy	Copies TraceWindow contents to the Windows clipboard.
Find	Search forward / backwards for search string.
Save to file	Save TraceWindow contents to a file.
Clear trace list	Clear the contents of the TraceWindow.
Trace List Size	Options for setting the size of the trace buffer window: 1K, 2K, 8K, 16K or 32K lines. If the selected trace buffer is smaller than the data currently in the buffer, then the trace buffer will be reset.

The **SETTRACE PRINT** directive can be used throughout an application to output directly to the TraceWindow; however, this directive is ignored if the Suppress Program Trace option is active.

Under WindX, TraceWindow serves a dual purpose. Normal tracing / error logging options are relative to the WindX workstation itself. Host tracing capabilities are controlled by a separate Options item. This allows the application to follow program execution on the server while tracing remote calls back to the workstation.

WatchWindow

WatchWindow allows you to constantly monitor variables and / or expressions during program execution. These settings may be copied to the clipboard or saved in a file for subsequent reload. The menu items for this window include:

Options

Always on top	Displays WatchWindow always on top of ProvideX.
Font	Changes font and font size for text in WatchWindow.
Auto-Load	Auto-loads watch values from the last Save to file.
No data break	Output without breaks.
50 byte data break	Automatic breaking of string data every 50 bytes.
100 byte data break	Automatic breaking of string data every 100 bytes.
Host Watch	*WindX only* - Enables / disables server-side watch values.

Edit

Copy	Copies WatchWindow settings / contents to the clipboard.
Save to file	Save TraceWindow settings / contents to a file.
Load from file	Load settings from a previously saved file.
Clear all watches	Remove all of the current watch values.
Add new watch	Prompts for a variable / expression to add to watch window.
Delete current watch	Delete the currently selected watch value from the window.

BreakWindow

BreakWindow is used to assign logical break points for halting execution in an application. You can specify the name of the program, line reference, and optional condition to test. The menu items for this window include:

Options

Always on top	Displays BreakWindow always on top of ProvideX.
Font	Changes font and font size for text in WatchWindow.
Auto-Start	Auto-activates BreakWindow when ProvideX starts.
Host Break Point	*WindX only* - Enables / disables server-side break points.

Edit

Copy	Copies BreakWindow contents to the clipboard.
Save to file	Save BreakWindow settings to a file.
Load from file	Load settings from a previously saved file.
Clear all breaks	Remove all of the break points.

Add new break	Establish a new break point with the following parameters: *Program* name to add a break point to. *Statement* (number or label) for the break point. *Break when* on variable name or Boolean expression. *Changes* condition regarding variable specified. *Is true* condition regarding Boolean expression specified.
Delete current break	Delete the currently selected break point from the window.

CommandWindow

This window can be used to handle all console commands without disrupting your standard screen display. The menu items for this window include:

Options

Always on top	Displays CommandWindow always on top of ProvideX.
Font	Changes font and font size for text in CommandWindow.
Auto-Start	Auto-activates CommandWindow when ProvideX starts.
Run *or* Halt	Start or suspend execution of the current program.
Step	Single step through the current program.

The CommandWindow facility is also available under ***WindX***.

Structured **SAVE**

By setting the **'SS'** parameter, you can verify the logical integrity of decision and loop structures automatically each time you **SAVE** a modified program. This feature checks programs in logical forward sequence to see if a *start-of-structure* block finishes in order with the correct matching *end-of-structure* directive. This also validates that there are no **CASE** or **DEFAULT** directives outside of a **SWITCH** / **END SWITCH** structure. The following sets of directives are checked for structural integrity:

FOR..NEXT **WHILE..WEND** **REPEAT..UNTIL** **IF..THEN..ELSE**

SWITCH..CASE **SELECT..FROM..NEXT RECORD**

If a logical error occurs (e.g., a **FOR** with no corresponding **NEXT**) the process will result in a `Warning #125: Improper Structure Detected` that indicates the line where the fault was detected; e.g.,

```
0010 BEGIN
0020 FOR I = 1 to 40
0030 .....
0040 WEND
```

The resulting report would indicate the start of the structure block at line 20 and mark the detection point at line 40, since it should have encountered a **NEXT** as opposed to a **WEND**.

Note: There will be no attempt to decipher the logic to determine if a **GOTO** might make the logic work.

For more information on logical structures, see *Chapter 4.* **Programming Constructs**.

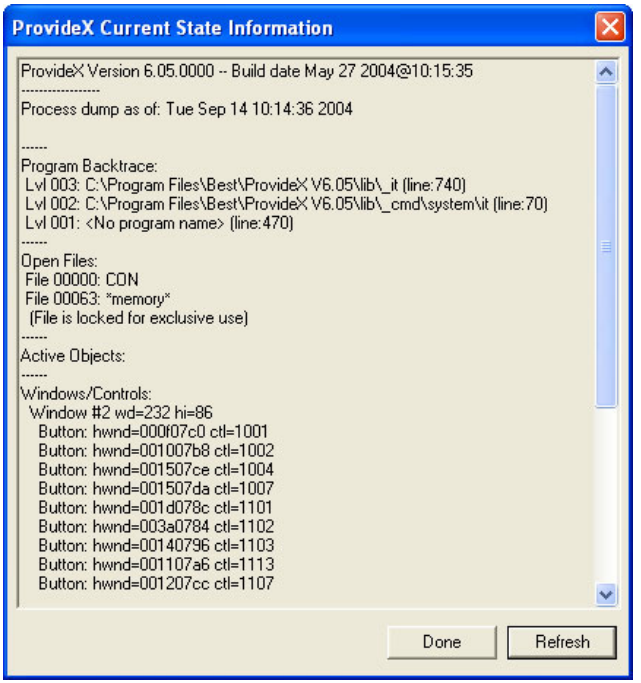
Additional Debugging Procedures/Facilities

While **Error Codes and Messages** and the **Windows Debugging Environment** provide the most obvious means for finding errors in syntax and logic, a few other facilities can be employed to handle or prevent programming problems.

Note: For debugging procedures involving the installation, startup, and configuration of ProvideX, see **Trouble-Shooting** in the *Installation and Configuration* guide.

Current State Information (Windows)

For a quick look at the current state of the ProvideX version installed on your system, select the About ProvideX option from under the PVX icon , then click the Info button.



The information displayed in this window includes the version number and build date, current date, program backtrace, list of all open files, and a list of all active OOP objects. Update contents by clicking the Refresh button. This feature is only available with ProvideX for Windows.

ProvideX Log File

The ProvideX log file can be used to provide a more detailed breakdown of internal errors that are encountered by ProvideX on MS Windows and UNIX/Linux platforms.

In Windows, the log file is activated by adding a `LogFile=`*filename* entry to the ProvideX INI file under the `[Config]` section. For more information, see **INI Files (Windows)** in the *Installation and Configuration*.

The log file feature in the UNIX environment is enabled by the existence of a file called `pvxtrace.log`. If the file exists when ProvideX attempts to add the first entry, then entries are appended to this file. If it does not exist, then the logging feature is disabled for the balance of the ProvideX session. The name of the log file on a UNIX machine is not configurable currently.

All entries in the log file are displayed using the following format:

mmm dd hh`:`*mm* `[`*progname:stmt#*`]`*errormsg*

Where:

mmm dd hh`:`*mm*	Date and time of the entry.
progname:stmt#	Name of the program and statement executing when the entry was added.
errormsg	Actual descriptive message.

Example:

```
Sep 15 16:52 [*ntslave:0320] [TCP][Winsock]Error status:10049 (-1)
```

Entries in the log for TCP/IP issues include a descriptive message followed by the actual TCP/IP error code and the socket number that was being used. A socket value of –1 usually indicates the **OPEN** was not successful.

Logical Line Labels

While logical line labels don't qualify as debugging options, they can reduce potential difficulties caused by renumbering programs and can simplify the development of programs without line numbers. These are described in the *Language Reference, p.812*.

Generic Escape Key Handling

Using the following syntax, an application can intercept Esc keys system wide:

SETESC *prog_name$*

The **SETESC** directive takes precedence over the generic escape key handler.

For ODBC-Related Problems

The **'!Q'=** system parameter can be used to will show all of the SQL commands being executed in a Windows message box. The OK/Cancel option allows a programmer to disable the message boxes by selecting Cancel.

4
Programming Constructs

The discussion now turns to basic programming constructs using the building blocks covered in *Chapter 2.* **Language Elements**.

For more information on the syntax elements discussed in this section, see *Chapter 2.* **Directives** in the *Language Reference*.

General Concepts

While ProvideX is an ***extensible*** language that has the flexibility to incorporate new functionality and sophisticated coding techniques, the more advanced capabilities are built upon universal concepts. Therefore, learning ProvideX begins with some programming fundamentals, as outlined below.

Note: The facilities in ProvideX for writing and modifying program code are discussed in *Chapter 3.* **Development Tools**.

Order of Execution

As described in Chapter 2, directives can be executed individually from the command line or they can be entered as numbered statements to be saved in memory before execution (see **Directives, Statements, and Programs**, *p.20*).

Program statements are grouped into the constructs for receiving and manipulating data, doing calculations, and printing output. During execution, the statements are evaluated and processed in the order they are read by the system. Logic flows through all the statements in one pass, from left to right and top to bottom, until the last statement is processed - this direction remains fixed unless specific commands are used to change this sequence.

Changing the Sequence

ProvideX employs various **Flow Control** mechanisms in order to perform conditional or repetitive instructions or to improve structure and maintainability of a program. Statements may be subject to certain conditions (decisions), executed repeatedly (loops), or packaged into code modules (subroutines/subprograms) that can be accessed from different points in the main program (see **Modular Programming Facilities**, below).

Stack

Like most programming languages, ProvideX maintains a type of data buffer called a "stack", which is used to store dynamic information associated with active counters, loops, and subroutines during execution of a program. For example, the primary purpose of a *subroutine* stack is to keep track of the location in the program (address) to which each procedure will return control when it is completed. At the start of every subroutine, a new return address will be placed on the top of the stack. When the procedure finishes, it *pops* the return address off the stack and transfers control to that address.

This type of information is continuously stacking up and unstacking the buffer as the program requires. An operational error that causes a stack to exceed its buffer allocation is called a *stack overflow*.

Input/Output Operations

Input/Output (I/O) in ProvideX programming refers to activities where source data is accepted into a program for processing, or where resultant data is sent from a program for storage or display. Depending on the process, I/O may take place to or from a device or file, with or without user interaction. Interfaces can include the keyboard, mouse, monitor, printer, and a variety of other connected devices. This chapter introduces **Basic Input/Output** operations (at the ProvideX console). More advanced topics are covered in *Chapter 6.* **Graphical User Interfaces**.

Data can take different forms: defined as **Numeric Values** or **String Values**, presented as **Literals**, or stored in **Variables**. Once a session is terminated, the processed data also disappears. However, if a more permanent storage solution is required, the data can be saved to a data ***file***. For an in-depth look at the ProvideX file system and file I/O operations, refer to *Chapter 5.* **File Handling**.

During execution, files as well as devices are opened for access via the **OPEN** directive. See **Opening/Closing Devices and Files**, *p.87*.

Modular Programming Facilities

As an application becomes larger and more complex it becomes increasingly difficult to keep track of where certain procedures are to be executed. ProvideX allows you to organize and extend the built-in capabilities of the language using modular

programming facilities, referred to as **Called Procedures**. Commonly-used expressions or calculations can be packaged into user-defined functions for reference by name elsewhere in the program. A similar approach includes transferring execution to subroutines (inside) or separately-written subprograms (outside) from points in the main program.

Advanced Concepts

The ProvideX language supports programming techniques where the coded lines are not necessarily written and executed as a fixed series of statements. This flexibility enables the use of different constructs that are better-suited for incorporating new user functionality and advanced programming paradigms; i.e., **Event-Driven Methodology**, **Graphical User Interfaces**, and **Object-Oriented ProvideX**.

Event-Driven Programming

Traditionally, programs operated in a sequential fashion: they received some data, did some processing, produced output, received more data, and so on. In ***event-driven programming***, processing is invoked only in response to specific inputs or *events*. Each call to a single subroutine or function is a sequential process, but the application as a whole does nothing until it is triggered by an event.

Graphical user interfaces are event driven by nature. GUI-based programs must run continuous event loops to check for, capture, and process the different sources of user input; e.g., dragging a mouse, clicking a button, pulling down a menu.

Refer to the *Chapter 6.* **Graphical User Interfaces** for more on this subject.

Object-Oriented Programming

While the functionality of a GUI-based application appears simple and natural to the user, the algorithms behind this type of interface design can be extremely complex. One of the more efficient ways in which graphical, event-handling operations can be expressed in ProvideX is via object oriented programming (OOP)

The idea of object orientation is to keep data and processing methods together as a single indivisible thing—an object. For example, each element of a GUI can be represented by one object. The GUI object determines both the state and the behavior of the corresponding elements. For example, an object representing a window would have data for its position, size and title. If the user closes the window, a message is sent to the window object telling it to close itself. The window then executes a process that erases its image from the desktop.

See *Chapter 11.* **Object-Oriented ProvideX** for complete documentation on the OOP mechanisms used in ProvideX.

Flow Control

Several mechanisms can be used to control program flow to perform conditional or repetitive instructions or to improve structure and maintainability. **Statement References** are sometimes an essential component for these types of procedures. As mentioned earlier, the ProvideX **Structured SAVE** feature can be used to verify the logical integrity of decision and loop structures automatically each time you save a modified program.

This section covers sequence control statements for *repetitive execution*, *conditional execution*, and *sequence overrides*.

Topics

Loop Structures: **FOR..NEXT** **WHILE..WEND** **REPEAT..UNTIL**
SELECT..FROM..NEXT RECORD (**WHERE** *condition*)

Decision Structures: **IF..THEN..ELSE** **SWITCH..CASE** **ON..GOSUB / ON..GOTO**

Flow Overrides: **EXITTO** **CONTINUE** **BREAK** **POP**

Statement References

By definition, a sequence control procedure will generally redirect program flow to some place that is outside the current order of execution. These transfers (branches or jumps) are most often invoked via ***statement references*** – pointers to other locations in the program that are identified by **Line Numbers**, **Line Labels**, or **Logical Statement References**.

Line Numbers

At one time, ***line numbers*** were among the most distinctive features of BASIC programming. In early dialects, they were mandatory for identifying lines of code, and as reference points for any type of sequence control procedure. They still work in ProvideX, but are generally associated with less structured programming techniques. See also **Numberless Programs**, *p.47*.

Currently, with prevalent use of line labels in contemporary programming, line numbers are now considered superfluous. While many developers keep them in their applications, line numbers are really only necessary when working with the **Command Line Editing** in ProvideX (for listing and editing).

Note: Many of the code samples in this documentation use line numbers for *illustration purposes*. These are *optional*, unless they appear in a statement reference.

In *unstructured* line-numbered programs, transfers are normally accomplished via the **GOTO** directive. This provides a direct unconditional jump. For example, when the system encounters a `GOTO 0021`, control automatically transfers to the statement at line 21.

However, if line 21 does not exist, control moves down to the statement with the next higher line number; e.g.,

```
-:0010 LET X=10
-:0020 IF X=10 THEN GOTO 0021
-:0025 PRINT "Where is line 21?"
-:run
Where is line 21?
```

While the use of **GOTO** statements does not mean that a program is unstructured, they are most often used to facilitate unstructured flow control. Generally, you should *avoid* unstructured programming wherever possible.

Line Labels

Like line numbers, ***line labels*** are used to identify a single line of code. Labels can include any combination of characters (plus the _ *underscore* character) but must begin with a *letter* and end with a : *colon*. The colon is not used when referencing a line label. A label name can be up to 127 characters in length.

For example, when the statement `GOTO Total` is processed, the program automatically jumps to the line with the label `Total:` at the beginning of the statement:

```
CHECK_TOTAL: IF A$=B$ THEN GOTO TOTAL
```

The line label can also appear on a line by itself:

```
TOTAL:
! Continue processing
```

While a line label can be referenced multiple times, the label itself can only occur ***once*** in a program. When used in large programs, line labels offer ***improved readability*** and are considered much easier to maintain. As mentioned earlier, references to ***line labels*** make line numbers unnecessary.

While line label references are optional in line-numbered programs, they are mandatory in programs where line numbers are omitted (and in this case, references to line numbers would be invalid).

Logical Statement References

ProvideX includes a set of built-in logical statement references that can be applied as keywords (with leading *asterisk* *) anywhere actual statement references are used; i.e., `*BREAK`, `*CONTINUE`, `*END`, `*ESCAPE`, `*NEXT`, `*PROCEED`, `*RETRY`, `*RETURN`, `*SAME`.

Unlike number or label references, logical references do not point to a *specific* location, but provide a *generic or dynamic* transfer of execution; i.e., advancing to the next line, or returning from a subroutine. These references are particularly useful in line-numbered programs because they help to reduce errors caused by renumbering issues and can clarify the intent of a statement.

Example:

```
READ(1,KEY=CUST_NUM$,DOM=*NEXT)
```

Where the logical statement reference *NEXT automatically transfers execution to the beginning of the next line / statement. For more information, see **Labels/Logical Statement References**, *Language Reference p.812*.

Loop Structures

Loop structures are intended for repetitive execution. They comprise a set of statements that are specified once, but can execute multiple times based on defined counters or condition statements. ProvideX includes the following directives for building controlled loops:

FOR..NEXT	uses a counter to control the number of repetitions made.
WHILE..WEND	tests a condition at the *beginning* the loop before starting.
REPEAT..UNTIL	tests a condition at the *end* of the loop before repeating.

A similar structure is used to open, query, and read records from a data file. For more information on this type of loop, refer to the **SELECT..FROM..NEXT RECORD** statement described in *Chapter 5.* **File Handling**.

ProvideX also includes some directives that allow early termination of a currently active loop. See **Flow Overrides**, *p.79*.

Infinite Loops

Sometimes a loop is constructed so that it executes endlessly. The term ***infinite loop*** most often applies to a situation where this result is *not intended* and is likely due to a logic or system error. Carefully review your code to ensure that your loops are designed to halt normally. For information on dealing with unexpected results during program operation, see **Error Handling and Debugging** , *p.60*.

FOR..NEXT

The **FOR** directive is used to define the start of a *counter-controlled loop*. There are two types of **FOR..NEXT** loops: a *conventional format* and a *simplified iteration format*.

Conventional Format. This **FOR..NEXT** syntax specifies a *control* variable (*var*), an *initial* value (*first*), an *ending* value (*last*), as well as an optional **STEP** value that can be used to set the increment / decrement to a specific value (default is 1); e.g.,

FOR *var*=*first* **TO** *last* [**STEP** *val*] **..NEXT** [*var*]

The **NEXT** directive marks the end of the loop. The *control* variable used with the **NEXT** directive must match the variable in the corresponding **FOR** directive. The **NEXT** *control* variable is optional, but should be used to maintain readability (especially for *nested* loops).

All statements following the **FOR** directive are executed in sequence until a **NEXT** directive is encountered. At this point, the *control* variable is incremented or decremented automatically. If the contents of the *control* variable exceeds the *ending*

value (or falls below it, if decremented), control is transferred out of the **FOR..NEXT** loop, and execution resumes at the statement following the **NEXT** directive. Otherwise, control stays in the loop and is transferred back to the statement following the **FOR** directive.

Example:

```
0010 FOR I=1 TO 10
0020 PRINT I,
0030 NEXT I
-:RUN
1 2 3 4 5 6 7 8 9 10
```

Note: The *conventional* **FOR..NEXT** format does not test the condition until the **NEXT** statement is executed; therefore, the loop is always executed at least once, even if the control variable is initialized to a value exceeding the ending value.

Simplified Iteration Format. The following **FOR..NEXT** syntax is used to specify the number of iterations in the loop:

FOR *var* ..**NEXT [***var***]**

The simplified format executes the logic immediately following the **FOR** by the number of times specified in the numeric value *var*; therefore, `FOR 1` will execute the loop once, and `FOR 5` will execute the loop 5 times. A value of zero causes the loop to be skipped. An error is generated if the numeric value is not an integer or it is less than zero.

If *var* is a simple numeric variable, the system will first set *var* to 1, then increment up to its initial value. When *var* is 5, the loop will execute 5 times, with *var* starting at 1 and incrementing by 1 through each iteration to 5. At the completion of the loop, *var* will equal its initial value.

Example:

```
0010 LET X$="THIS IS A TEST"
0020 LET N=LEN(X$)
0030 FOR N
0040 IF X$(N,1)=" " THEN LET X$(N,1)="_"
0050 NEXT
0060 PRINT X$
-:RUN
THIS_IS_A_TEST
```

If the loop is exited prematurely (via **BREAK**, **POP**, or **EXITTO**), the counter variable *var* will retain the value of the current iteration. Regardless of whether a simple numeric variable is used, **TCB(19)** will contain the current iteration count during the loop.

For syntax details, see **FOR..NEXT** in the *Language Reference, p.133*.

WHILE..WEND

Use the **WHILE** directive to specify a condition (numeric expression) at the beginning of a *condition-controlled loop;* e.g.,

WHILE *expression* **..WEND**

ProvideX executes all statements between **WHILE** and **WEND** repeatedly, until the *expression* returns a ***false*** (0 *zero*) result; e.g.,

Example:

```
0010 INPUT "CAN WE TALK? <Y/N> ",R$
0020 WHILE UCS(R$)<>"N"
0030 INPUT "NAME PLEASE: ",N$
0040 INPUT "WHAT IS YOUR AGE "+N$+"? ",A$
0050 INPUT "CONTINUE? <Y/N> ",R$
0060 WEND
```

When ProvideX encounters a **WHILE** directive, it evaluates the *expression*. If the result is not 0 *zero*, ProvideX continues execution until a corresponding **WEND** directive is encountered, at which point the *expression* is re-evaluated. ProvideX continues to loop back to the directive following the **WHILE** directive until a 0 value is reached. At this point, ProvideX advances to the next **WEND** directive, where it terminates the loop. Then control transfers to the statement following the **WEND**.

For syntax details, see **WHILE..WEND** in the *Language Reference, p.371*.

REPEAT..UNTIL

The **REPEAT** directive is used to begin a *condition-controlled loop.* In this structure, the **UNTIL** directive specifies the condition at the end of the loop*;* e.g.,

REPEAT ..UNTIL *expression*

ProvideX executes all statements between **REPEAT** and **UNTIL** repeatedly, until the *expression* returns a ***true*** (*non-zero*) result; e.g.,

Example:

```
0010 LET X=1
0020 LET C=1
0030 REPEAT
0040 LET X=X*2
0050 LET C=C+1
0060 UNTIL X>100000
0070 PRINT C,X
-:run
 18 131072
```

After the **REPEAT** directive, ProvideX executes all statements until the **UNTIL** directive, it then tests the condition specified. If the result is ***false*** (0 *zero*), ProvideX loops back to the directive following the **REPEAT** directive and resumes execution. If the result is ***true*** (*non-zero*), the loop is terminated and execution continues from the statement following the **UNTIL** directive.

For syntax details, see **REPEAT..UNTIL** in the *Language Reference, p.283*.

Decision Structures

A decision structure is intended for conditional execution. It is designed to change program flow by selecting a new path from among one or more possible branch points. ProvideX includes the following directives for building decision structures:

IF..THEN..ELSE	tests a Boolean expression, then determines what action to take depending on the ***true*** or ***false*** result.
SWITCH..CASE	compares an expression / value against a series of specified values to determine what action to take.
ON..GOSUB / ON..GOTO	counts through a sequence of possible statement destinations based on a supplied value to determine what action to take.

IF..THEN..ELSE

Use an **IF**..**THEN** statement to execute a series of statements based on the result of a Boolean expression that evaluates to ***true*** (*non-zero*). An optional **ELSE** clause can be used to specify statements that are to be executed when the Boolean expression evaluates to ***false***.; e.g.,

IF *expression* **THEN** *...* [**ELSE** *...*] [**END_IF**]

All statements within an **IF..THEN..ELSE** structure exist on the same line and must be separated by semicolons. However, statements can span multiple lines if they are enclosed within ***curly brackets***; e.g.,

IF *expression* **THEN** {*...}* [**ELSE** {*...}*] [**END_IF**]

An optional **END_IF** (*or* **FI**) clause may be used as an **IF** terminator and / or to execute a common closing statement that is outside the **IF** condition. This is particularly useful for separating **ELSE** clauses within a nested **IF..THEN..ELSE** structure. Once the statements that follow an **END_IF** clause are executed, control falls through to the next line, or (if nested) to the previous layer of **IF..THEN..ELSE**.

Examples:

```
00010 IF SW=0 \
       THEN LIGHT$="OFF" \
       ELSE LIGHT$="ON" \
      END_IF;
      PRINT "LIGHT IS ", LIGHT$
-> SW=0
-> RUN
LIGHT IS OFF
```

```
00010 IF SW=0 THEN {
00020 LET LIGHT$="OFF"
00030 } ELSE {
00040 LET LIGHT$="ON"
00050 }
00060 PRINT "LIGHT IS ", LIGHT$
-> SW=1
-> RUN
LIGHT IS ON
```

These example **IF..THEN..ELSE** statements perform similar operations. The one on the right uses ***curly brackets*** to separate statements on different lines. For syntax details, see **IF..THEN..ELSE** in the *Language Reference, p.156*.

As an alternative to a compound **IF..THEN..ELSE** statement, use **SWITCH..CASE**, *p.78*.

SWITCH..CASE

The **SWITCH..CASE** directive executes different statements depending on where it finds a match among a series of values. Unlike the **IF..THEN..ELSE** structure, which chooses between two possible branch points, this structure chooses from multiple branch points; e.g.,

SWITCH *expression* **CASE** *range_1* [*...***CASE** *range_n*] [**BREAK**] [**DEFAULT**] ... **END SWITCH**

Execution continues with the statements that follow any matching **CASE** *range* (until the next **BREAK** or **END SWITCH)**. If there are no matches in any of the **CASE** statements, control falls through to the **DEFAULT** clause (if present), and the statements that follow are executed automatically.

Example:

```
SWITCH UCS(x$)
  CASE "CAT"
  PRINT "Selected cat"
  BREAK
  CASE "DOG","FOX","PIG"
  PRINT "Selected ",x$
  BREAK
  CASE >"ZEBRA"
  PRINT "Selected something Greater than a Zebra"
  BREAK
  DEFAULT
  PRINT "Default code kicks in"
! This executes when all of the above fails
  END SWITCH
```

This can be coded similarly using the following **IF..THEN..ELSE** statement:

```
IF x$="CAT" \
   THEN PRINT "Selected cat" \
   ELSE IF x$="DOG" OR x$="FOX" OR x$="PIG" \
        THEN PRINT "Selected ",x$ \
        ELSE IF x$>"ZEBRA" \
             THEN PRINT "Selected something Greater than a Zebra" \
             ELSE PRINT "Default code kicks in"
             ! This executes when all of the above fails
```

For syntax details, see **SWITCH..CASE** in the *Language Reference, p.327*.

ON..GOSUB / ON..GOTO

These statements provide a variation of the **GOTO** and **GOSUB** directives that allows transfer of control to a choice of multiple destinations; e.g.,

ON *num* **GOSUB** *stmtref,stmtref,...*
ON *num* **GOTO** *stmtref,stmtref,...*

In this decision structure, the branch point is determined by counting *num* places through the sequence of *stmtref*'s. The sequence is 0 *zero* based. If *num* is greater than the number of *stmtref*'s supplied, the last *stmtref* is assumed. If *num* is 0 or less, the first *stmtref* is assumed.

Examples:

```
1000 ON X GOSUB 2100,2200,2200,2300
```

If `X=0`, control transfers to line `2100`; if `X=1`, control transfers to line `2200`; and so on. For syntax details, see **ON..GOSUB** / **ON..GOTO** in the *Language Reference, p.229*.

Flow Overrides

Sometimes it is necessary to exit a running loop (subroutine, procedure) as soon as a specific task is completed, regardless of the value of the control variable or test expression. The following directives allow for immediate termination of a loop or subroutine:

BREAK

The **BREAK** directive terminates an active loop and transfers control to the statement immediately after where the loop normally ends; i.e., **NEXT** or **WEND**.

Example:

```
0010 DIM X$[100](1,"*"); LET X$[50]=""
0020 FOR I=1 TO 100
0030 IF X$[I]="" THEN BREAK
0040 NEXT
0050 ESCAPE
```

If the conditional **BREAK** is executed, the program skips to line `0050`. For syntax details, refer to the **BREAK** directive in the *Language Reference, p.33*.

EXITTO

The **EXITTO** directive terminates the currently active loop or subroutine and transfers control to the statement reference indicated.

Example:

```
0010 DIM X$[100](1,"*"); LET X$[50]=""
0020 FOR I=1 TO 100
0030 IF X$[I]="" THEN EXITTO 50
0040 NEXT
0050 ESCAPE
```

For syntax details, refer to the **EXITTO** directive in the *Language Reference, p.124*.

POP

Use this directive to clear (pop) the top entry from the **FOR..NEXT**, **WHILE..WEND**, **REPEAT..UNTIL**, **GOSUB** stack. **POP** is equivalent to **EXITTO** except that it does not transfer control (to a statement reference), but continues with the next statement in direct execution sequence. For syntax details, refer to the **POP** directive in the *Language Reference, p.244*.

CONTINUE

The **CONTINUE** directive is similar to the **BREAK** directive in that it causes the current iteration of a loop to be terminated, but unlike **BREAK**, it resumes the loop's execution with the next natural cycle.

Example:

```
0010 DIM X$[100](1,"*"); LET X$[50]="",K$="*"
0020 FOR I=1 TO 100
0030 IF X$[I]=K$ THEN CONTINUE
0040 PRINT I
0050 NEXT
0060 ESCAPE
```

For syntax details, refer to the **CONTINUE** directive in the *Language Reference, p.56*.

Called Procedures

Subroutines, ***subprograms***, and ***user-defined functions*** comprise a sequence of statements that are specified once, but can be accessed many times from various points in a main program. The directives discussed in this section are used for building and accessing called procedures in ProvideX.

Topics

GOSUB	transfers control to a ***subroutine***, which exists inside the current program.
CALL *and* **PERFORM**	transfers control to a ***subprogram***, which exists outside the current program.
DEF FN	creates a ***user-defined function*** that can be invoked by name anywhere in the current program.

Called procedures are common features in structured programming. They allow a larger program to be split into smaller logical sections, which makes it easier to maintain and debug. They can also consolidate general-purpose tasks and frequently-used calculations, which eliminate repetitious code and help reduce overall program size. This concept (of reusable blocks of code) is integral to the more advanced programming techniques on which **Object-Oriented ProvideX** is based.

Note: While the **SETESC** and **SETERR** directives perform similar transfers of control, this form of called procedure has a single purpose, and is not intended for reuse.

Each time a subroutine, subprogram, or user-defined function is called, the system remembers where the transfer occurs in the main program and resumes execution at that point when the procedure is completed. However, some subroutines can include statements that redirect control to a destination other than the original transfer point; see **Flow Overrides**, *p.79*.

Some called procedures share variables with the initiating procedure so that they can be defined, modified, or cleared at any point during execution. Use the **LOCAL** directive to restrict or reassign a variable name within a called procedure. This is explained in more detail under **Variables**, *p.34*.

GOSUB

Use the **GOSUB** directive to transfer control to a *subroutine* – a sequence of statements that can be accessed multiple times from anywhere in the main program; e.g.,

GOSUB *stmtref*

When **GOSUB** is executed, ProvideX saves the current location on the **GOSUB** stack then transfers control to the specified program line number or label (*stmtref*), which marks the start of the called subroutine. See **Statement References**, *p.72*.

The **RETURN** directive marks the end of a subroutine and transfers control back to the location saved on the **GOSUB** stack; e.g.,

```
00010  FOR X=1 TO 10
00020  PRINT " GOING TO THAT";
```

```
       GOSUB THAT
00030  NEXT X
00040  PRINT 'LF',"VALUE IN X=",X
00050  END
00060 THAT:
00070  LOCAL X
00080  WHILE X<25
00090  PRINT X,;
       X++
00100  WEND
00110  RETURN
```

All subroutine content exists between the called statement reference and the **RETURN** directive. The subroutine itself can exist anywhere in the program, typically outside the main order of execution.

Alternatively, the **EXITTO** directive may be used to terminate the subroutine before it reaches the **RETURN** statement – this transfers control to the statement identified by the **EXITTO** instead of the location saved on the **GOSUB** stack. There is no limit to the number of locations that can be saved on the **GOSUB** stack.

Note: The same stack is used for **GOSUB**, **FOR..NEXT**, **WHILE..WEND**, and **REPEAT..UNTIL** directives; therefore, a **RETURN** can only be executed after all of these structures in the subroutine have been terminated.

For syntax details, refer to the **GOSUB** directive in the *Language Reference, p.140*.

CALL

Use the **CALL** directive to transfer control to a *subprogram* – a called procedure that exists in a completely separate program file; e.g.,

CALL *subprog$*[*;entry$*] [*,arglist*]

When a **CALL** is executed, ProvideX saves the current location on the stack in the current program, then loads and executes the subprogram (*subprog$*). For syntax details, refer to the **CALL** directive in the *Language Reference, p.40*.

Subprograms

The term ***subprogram*** denotes a program that is called from another program. There are no real differences between subprograms and programs, except that, when a subprogram terminates, processing continues as control is passed back to the program that initiated it. Subprograms can also initiate further subprograms with virtually no limit (apart from memory constraints).

The **EXIT** directive is used to ***terminate*** a subprogram; however, a **STOP** or **END** directive may be used in its place.

A *level number* is maintained within ProvideX that records the number of subprograms currently in progress. When a subprogram is started, the current level is incremented by one – when the subprogram terminates, the level is decremented by one. The value of the current level indicates the number of programs currently active. This information can be accessed via the **TCB( )** function and is displayed by ProvideX at

the prompt line whenever execution is halted while within a program. The return address and programs are maintained in the *subprogram stack*. This information can be retrieved via the **STK( )** function.

Passing Arguments

Arguments (*arglist*) are received in a subprogram via the **ENTER** directive. Each argument in the **CALL** statement corresponds by position and in data type (numeric or string) to an argument in the **ENTER** statement. A complete numeric array may be passed to a subprogram by specifying **{ALL}** following the variable name in both the **CALL** and **ENTER** directives. Changes to any element of the array will affect the corresponding array in the main program.

For example, if a **CALL** to a subprogram named SUBR defines arguments as follows:

```
CALL "SUBR",LEN(A$),N,A$,T{ALL}
```

The subprogram SUBR would require the following **ENTER** statement to receive those arguments:

```
0020 ENTER A,B,Z$,N{ALL}
```

Subprograms can alter the value of the arguments passed to them in this manner. Simple variables passed as arguments become common between the main program and the subprogram – any changes to these variables in the subprogram will directly affect the value of the variable in the main program.

This affects only the variables defined by the **ENTER** directive. All other variables in the subprogram remain completely independent of variables in the main program. If you wish to prevent a variable in the argument list from being changed, place parentheses around it – this makes it an expression rather than a simple variable, so it cannot be changed. The following table defines the conditions under which a **CALL** argument can be changed by a subprogram:

CALL Directive	**ENTER** Directive	***Description***
X	**Y**	**Y** in the subprogram will be assigned value of **X** from the main program. Changes to **Y** will change **X** in the main program.
X+*nn*	**Y**	**Y** in the subprogram will be assigned value of **X+***nn* from the main program. Changes to **Y** will ***not*** effect **X** in the main program.
X(*n*)	**Y**	**Y** in the subprogram will be assigned value of **X**(*n*) from the main program. Changes to **Y** will ***not*** affect variable **X**(*n*) in the main program.
X(*all*)	**Y**(*all*)	**Y** in the subprogram will receive the complete array defined by **X** in the main program. All changes to elements in **Y** will affect the corresponding element in **X**.
X$	**Y$**	**Y$** in the subprogram will be assigned value of **X$** from the main program. Changes to **Y$** will change **X$** in the main program.
X$(...)	**Y$**	**Y$** in the subprogram will be assigned value of the substring **X$**(...). Changes to **Y$** will ***not*** affect **X$** in the main program.
"Fred"	**Y$**	**Y$** in the subprogram will be assigned the value of "Fred".

Note: If the **CALL** statement has fewer arguments than in the **ENTER** statement, make sure to maintain the same relative position and type up to the point where the *arglist* is shortened (and include error handling options). Otherwise, this will result in an `Error #36: ENTER parameters don't match those of the CALL`.

Entry Points

When specifying a program name you can also suffix it with an entry point (*;entry$*) within the called program; e.g.,

```
CALL "PROG01;Add_Record",X$,Y$
```

If the subprogram `PROG01` contains:

```
0010 REM PROG01
0020 INIT:
0030 OPEN (1) "ARCUST",(2) "ARBILL"
0040 EXIT
0100 ! 100 - Add-record logic
0110 ADD_RECORD:
0120 ENTER CST_ID$,CST_NAME$
0130 WRITE (1) CST_ID$, CST_NAME$
0140 EXIT
```

`CALL "PROG01:Add_record"` causes the system to open the subprogram `PROG01` and commence execution at the label `ADD_RECORD` rather than at the beginning of the program. ProvideX internally issues a **GOTO** directive using *entry$* as a statement reference. Use this feature to create subprograms to act as "libraries" (i.e., multiple stand-alone routines, each starting at its own entry point).

PERFORM

A **PERFORM** is similar to a **CALL** directive, in that it also transfers control to a subprogram; however, when a **PERFORM** is used to invoke a subprogram, ***all variables*** are made ***common*** between the initiating and called programs (no *arglist* or corresponding **ENTER** directive is required); e.g.,

PERFORM *subprog$*[;*entry$*]

All variables that are defined, modified, or cleared during execution of the subprogram will be transferred back to the initiating program. For syntax details, refer to the **PERFORM** directive in the *Language Reference, p.242*.

Subroutines within Subprograms

PERFORM can also access ***subroutines*** externally via entry points in the called program. With this feature, the **RETURN** statement that is used to terminate the subroutine in a subprogram automatically returns control to the initiating program. This allows the same block of code to be accessed internally (**GOSUB**) as well as externally (**PERFORM**). If the subprogram `CUSTOMER` contains:

```
0010 ! CUSTOMER - Customer logic
...
0100 GOSUB CHK_TYPE
...
```

```
1000 ! 1000 - Validate customer number
1010 CHK_TYPE: ERR_MSG$=""
1020 IF POS(CST_TYPE$=%CST_TYPE_TBL$)=0 ERR_MSG$="Bad Type"
1030 RETURN
```

The CHK_TYPE subroutine could be accessed externally via:

```
1000 PERFORM "CUSTOMER;Chk_type"
1010 IF ERR_MSG$<>"" GOTO ...
```

DEF FN

When a user-defined function is executed, it transfers control to a function procedure – a single statement (or a sequence of statements) defined for multiple access using the **DEF FN** directive. This type of procedure does not require a calling directive (**CALL** or **GOSUB**) because it is ***invoked via the function name itself***.

The following syntax defines a *single-line* function procedure (function name, parameters, and associated expression):

DEF FN*name*[*$*]([**LOCAL**]*argvar1*[,*argvar2*, …])=*expression*[*$*]

For syntax details, refer to the **DEF FN** directive in the *Language Reference, p.67*.

All user-defined functions are identified by an **FN** prefix. The remaining characters in the function name follow the same rules that apply to **Variables**. When executing a user-defined function, the syntax is consistent with **System Functions**. All functions in ProvideX accept and process values, and return control (with results) to the statement where the function was invoked in the main program; e.g.,

```
-:LIST
0010 DEF FNSUM(A,B,C)=A+B+C
0020 LET D=5,E=6,F=7
0030 LET X=FNSUM(D,E,F)
0040 PRINT X
-:RUN
 18
```

Depending on how they were defined, functions can return either a numeric value or a character string. This is determined by the data type represented in the function name, and by the result of the expression / procedure specified; e.g.,

Numeric: `0010 DEF FNSUM(A,B,C)=A+B+C`

String: `0020 DEF FNNME$(X$)=UCS(X$(1,1))+LCS(X$(2))`

Parameters used in a function definition must match the variables used in the expression it represents, as well as the parameters specified when the function is used. In the numeric example above (FNSUM), if the function uses the arguments 1, 2, and 3 then the variables in the expression A+B+C will receive these values respectively. While there is no limit to the number of parameters that can defined, it is imperative that each time the function is used, the number of arguments and their type (numeric / string) match the parameter list specified in the **DEF FN** directive. Any mismatch will generate an `Error #25: Invalid call to user function (Non-existent or recursive)`. Arguments can also be defined as **LOCAL** for processing exclusively within the function procedure.

Multi-Line Function Procedure

A user-defined function procedure can also be listed over multiple lines. This type of **DEF FN** procedure is very similar in appearance to the contents of a **GOSUB** ***subroutine***. Use the following syntax to define a multi-line function:

DEF FN*name*[*$*]([**LOCAL**]*argvar1*[,*argvar2*, …])
RETURN *expression*[*$*]
END DEF

If the **DEF FN** directive is used without the equals sign / expression in the syntax, it is used to mark the beginning of a multi-line function definition; **END DEF** marks the end. Execution of a multi-line function is terminated via the **RETURN** statement. Once completed, the value specified by the **RETURN** statement will be passed back to the statement where the function was invoked in the main program.

It is also possible to define an error number within a multi-line function by issuing a **ESCAPE** *nnn*, where *nnn* is the error to be returned to the calling expression.

Example:

```
0010 DEF FNPRIME(A)
0020 IF A<2 THEN RETURN 0 ELSE IF A=2 THEN RETURN 1
0030 FOR N=2 TO A/2
0040 LET M=INT(A/N)
0050 IF (M*N)=A THEN EXITTO 0080
0060 NEXT N
0070 RETURN 1
0080 RETURN 0
0090 END DEF
0100 FOR I=1 TO 200
0110 IF FNPRIME(I) THEN PRINT I,
0120 NEXT I
```

In this example, the multi-line function `FNPRIME` checks to see if the value given is a prime number by trying to divide it by all the numbers up to 1/2 of the original number. If all the numbers fail the function returns `1`, otherwise it returns `0`.

Global User-Defined Functions

Global variable names can be used in defining user functions. If a global variable name is used in the **DEF FN** directive, then this function remains defined for the duration of the session and in all programs and subprograms.

Example:

```
0010 DEF FN%TM$(T)=STR(INT(T*60)+INT(T)*40:"00:00")
```

Once this **DEF** instruction is executed, the user function `FN%TM$` is defined and is accessible to all programs for the duration of the user session or until a **START** is issued. Only single-line functions may be defined as global. Multi-line functions can never be defined as global.

Basic Input/Output

The **INPUT** directive is used to enter data interactively while the program is running. It issues prompts to the user and processes the responses. **PRINT** statements may be used to format and output printable data to a monitor or printer as well as into a file. ProvideX I/O statements need to reference a ***channel number*** in order to access a device or file. The relationship between a channel and the physical file or device must first be established via the **OPEN** directive.

Topics

Opening/Closing Devices and Files, *p.87*
Input Statements, *p.88*
Output Statements, *p.94*

Other I/O directives (**READ**, **WRITE**, etc.) used specifically for transferring data in and out of *files* are discussed in *Chapter 5.* **File Handling**. For information on I/O processing in a GUI environment, see *Chapter 6.* **Graphical User Interfaces**.

Opening/Closing Devices and Files

Most input/output operations in ProvideX require a channel number (*chan*) to identify the connection to a specific device, interface, or file. These are established using the **OPEN** directive, which associates a *logical* number (normally between 0 and 127) with a *physical* file or device. Once the *chan* is defined, the program will be able to access the input or generate output simply by referencing that number.

In theory, all I/O statements use channel numbers; however, the *console* (keyboard and display screen) is defined as channel 0 by default and may be omitted from **INPUT** or **PRINT** statements intended for immediate display.

For files, the **OPEN** process also sets a pointer to the beginning of the opened file and allocates system resources. The actual number of files that can be opened at any one time depends on the operating system. For more information on using **OPEN** for file I/O operations, see *Chapter 5.* **File Handling**.

The basic syntax for an **OPEN** statement is provided below.

OPEN (*chan*[,*fileopt*])*string$*

Where:

chan	Logical channel number to be assigned.
fileopt	Various options used for controlling the contents and characteristics of a data file. See **Processing Data Files**, *p.107*.
string$	Name of the file or device to open. The string expression can include a specialty filename or file tag (e.g., ***MEMORY***, **[RPC]**, etc.).

Several **OPEN** keywords are also available for specific types of file access operations, including: **INPUT**, **LOCK**, **PURGE**, and **LOAD**.

As mentioned earlier most I/O operations begin with an **OPEN** directive to establish the relationship with the target device or file; e.g.,

```
PRINT "Today's date is ",DAY
OPEN (1)"*WINPRT*;HP Laser Jet;Orientation=Landscape"
PRINT (1)@(0),"Customer",@(45)," Balance"
```

In this example, the first statement simply outputs text to the screen, which is the default syntax for **PRINT.** The second **PRINT** statement in the example outputs text to ***channel*** `(1)`, which is the identity of a printer made available for use via the preceding **OPEN** directive.

The **CLOSE** directive closes the connection to one or more files/devices and allows their channel number(s) to be reused. The basic syntax is provided below:

CLOSE {**(*)** | **(***chan***)** [,**(***chan***)**...]}

Where:

*****	Asterisk denotes "all **OPEN** local channels" *except* channel 0.
chan	Channel number.

Note: Local files are closed automatically on a **BEGIN** or **END** statement. All files are closed at the end of a user session or whenever a **START** directive is issued.

Input Statements

The ProvideX **INPUT** directive is used to process interactive responses. It can be used to prompt for, receive, and validate input from the user in a character-based application; e.g.,

```
INPUT NUMBER
INPUT "Please enter your name :",NM$
INPUT "Please enter your name",NM$, "Thank You"
```

Note: In cases where it is *undesirable* to echo user input, use **OBTAIN** in place of the **INPUT**. For syntax, see **OBTAIN** in the *Language Reference, p.226*

Input from the user is stored in the variables specified. ProvideX treats any literals or expressions included in the statement as prompts. String variables will be assigned any keyboard characters. Numeric variables must be assigned numeric data only. Non-numeric input in response to a numeric variable (other than commas and decimals) will cause an `Error #26: Variable type invalid`.

For syntax details, refer to the **INPUT** directive in the *Language Reference, p.159*.

Example 1:

```
0010 INPUT "Enter a number:", A
0020 IF A=0 THEN STOP
0030 PRINT "You entered ", A
0040 GOTO 0010
```

When the above example is run, it yields the following as the user enters input to the prompts:

```
Enter a number:5
You entered 5
Enter a number:1,000
You entered 1000
Enter a number:4.X
0010 INPUT "Enter a number:", A
Error #26 - Incorrect variable type
```

Example 2:

```
0010 INPUT "What is your name?", A$
0020 INPUT "How old are you "+A$+"?",A
0030 IF A<18 THEN PRINT "Sorry X-Rated"; STOP
0040 ......
```

When this is run, it yields the following:

```
->RUN
What is your name?MIKE
How old are you MIKE?13
Sorry X-Rated
->
```

The **INPUT** directive can also receive input from other than just the user console by specifying a channel of a device to receive from. Channel number 0 ***zero*** is used to identify the console (keyboard and display); e.g.,

```
0010 INPUT (0,ERR=0100) "Enter amount:", A
0020 LET INTEREST=A*RATE
.......
0100 PRINT "Invalid amount -- Retry"
0110 GOTO 0010
```

If, in the above example, the user enters invalid numeric data in response to the `Enter amount:`, the program would transfer to statement 100. As a different tactic, statement 0010 could have specified the option `ERR=0010`, which would result in the **INPUT** directive being re-issued until valid data was received.

Note: The **INPUT** directive is primarily designed for use with terminals, but it can also be used with other file types. For more information, see **Processing Data Files**, *p.107*.

Default Input Values

Depending on your application, it is sometimes useful to display a default value at the input prompt. The **INPUT EDIT** directive can be used to *pre*-load the input buffer with the current contents of the specified variable; the user may then choose to edit the current value or enter a new one.

If the user starts typing, the contents of the variable is re-initialized with a new value, otherwise the current value can be edited and/or re-entered. In the following example, **INPUT EDIT** places the current contents of NAME$ into the input buffer.

```
0010 NAME$="JOHN DOE", AGE=21
0020 INPUT EDIT "What is your name?", NAME$
0030 PRINT "Hi ",NAME$
0040 INPUT EDIT "How old are you?",AGE:"##0"
->RUN
What is your name?JOHN DOE
```

At this point, the user can choose to edit the default JOHN DOE or enter a new name.

Formatted Input

An **INPUT** statement can establish a format to be used on either numeric or string information during the input cycle. This forces the data to adhere to the format mask specified (and prevents invalid entries).

To specify a format mask to be used during an input statement, place a : *colon* and format string after the input variable; e.g.,

```
0010 INPUT "Enter Amount:",AMT:"$##,##0.00-"
->RUN
Enter Amount:     $0.00
```

In the above example the format mask specified in the **INPUT** directive would be used to control the entering of the value. One major advantage of using formatted **INPUT** statements is that no ERR= clause is required since invalid data cannot be entered. For numeric data within a format mask, ProvideX aligns the data to the decimal point. See **Data Format Masks**, *Language Reference p.809*.

Input Size Control

Two options are available for use with the **INPUT** statement for controlling the size of the input data. The **LEN=** option sets the maximum length of input allowed. If the user attempts to enter more characters than is specified on the **LEN=** option, it is rejected. The **SIZ=** option sets the number of characters which can be entered after which the input automatically terminates.

A special feature of the **INPUT** directive is the ability to allow for scrolling input. By specifying both **LEN=** and **SIZ=** options you can control the maximum length of the data you wish to have entered as well as the size of the data as it is to appear on the screen. For example, if you wish to allow the user to enter 100 characters of input but only wish to have 30 characters appear of the screen, you would specify the following:

```
INPUT (0,LEN=100,SIZ=30) X$
```

Input Validation

INPUT variables can also include options to validate the values as they are being received. To specify validation on input, place a : *colon* followed by conditions (in parentheses) after the input variable; e.g.,

```
0010 INPUT A$:("A"=0100,"B"=0200)
```

This type of validation option is called a ***branchlist***. If the input is "A", control transfers to statement 0100; If the input is "B", control transfers to statement 0200. Any number of entries may be included in the branchlist. Each entry is processed as it appears, from left to right. A branchlist may be specified with either ***string*** or ***numeric*** variables. Numeric data is converted only after the branchlist is processed.

String Validation

For input to a ***string variable***, the validation may contain a branchlist as well as an additional ***length*** option. The length option takes the form:

LEN=*minimum* [, *maximum*]

The input must contain at least the number of characters specified as a *minimum* but no more than the number specified as a *maximum*. If no *maximum* is specified, then the input length must equal the *minimum* (*maximum* set to *minimum*).

In the following **INPUT** statement, if "END" is entered in response to the prompt "Name:", then control transfers to statement 0100, otherwise the input string would have to be between one and twenty characters long:

```
0010 INPUT "Name:",N$:("END"=0100,LEN=1,20)
```

If the input is less than one character long (null) or greater than twenty, an Error #48 is generated. Only one **LEN=** condition may be specified, and it must follow the branchlist (if provided). When validating a string variable, an Error #48: Invalid input is generated if input does not match one of the entries in the branchlist and/or no **LEN=** option is provided.

Examples:

The following statement does not have a branch list – input must be exactly six characters long or an error will occur:

```
0010 INPUT (0,ERR=0010)"Invoice #:",I$:(LEN=6)
```

If the LEN=6 condition is not met, the ERR=0010 clause causes the **INPUT** directive to repeat.

In the following example, the prompt "Yes or No" is displayed and the variable C$ is requested:

```
0010 INPUT (0,ERR=0010) "Yes or No", C$:("Y"=0100,"N"=0200)
```

The input validation options cause control to transfer to 0100 if "Y" is input, and 0200 if "N" is input. Any other input will cause an error, which due to the ERR=0010 clause, causes the **INPUT** directive to repeat.

Numeric Validation

For input to a ***numeric variable***, the validation may contain a branchlist as well as an additional ***range check***. The branchlist is processed before the input data is converted to an internal numeric value; therefore, it is possible for non-numeric input to be processed.

A numeric range check is specified by providing the maximum value to be allowed; e.g.,

```
0010 INPUT "Enter hour to start:",H:(23)
```

In this example, the input to `H` would have to be an integer between 0 and 23. A number outside of this range generates `Error #48: Invalid input`. If the value specified in the range check is positive (greater than zero) then the input must be numeric between zero and the number specified (inclusive).

The range check can also include negative numbers; e.g.,

```
0010 INPUT (0,ERR=0010)"Percent: ",P:("X"=0100,-99)
```

In this example, the input range is between -99 and +99. If the input consisted of the single character `"X"`, then control transfers to statement `0100`. Non-numeric input or a number outside of the specified range would cause the **INPUT** directive to repeat due to the `ERR=0010`.

The number of digits to the right of the decimal point (in the range value) define the maximum number of decimal places to be accepted; e.g.,

```
0010 INPUT (0,ERR=BAD_INP)"Discount:",D:(19.99)
```

This statement would allow the user to enter a number between 0 and 19.99. Two digits to the right of the decimal point are allowed. If the input is non-numeric, outside the range specified, or has more than two digits to the right of the decimal point, an error is generated and control transfers to the line label `BAD_INP`.

Any numeric value (constant, variable, or expression) can be used to specify the range of numeric input. It must follow any branchlist specification within the validation list. If no range check is specified then the input is only checked for valid numeric data.

Submitting Input (*CTL Values*)

From the keyboard, input is received when the user presses Enter. By default, this keystroke signals to ProvideX that **INPUT** has terminated and that the entered values can be submitted for use in the running program. This keystroke also sets the ProvideX **CTL** system variable to a specific *numeric code* (CTL value) representing use of the Enter key.

Other types of keys can be used to terminate an **INPUT** statement. The CTL values assigned to keyboard actions are listed as follows:

0	Enter key (normal)
1 - 4	F1 to F4 keys
5	Input terminated due to **SIZ=** option
6 - 12	F6 to F12 keys

Example:

```
0010 INPUT "Enter customer id:",CST_ID$
0020 ON CTL GOTO 0030,0010,0010,9000,9000
0030 INPUT "Enter date:", DTE_ID$
0040 ON CTL GOTO 0050,0030,0030,0010,9000
0050 .....
```

In fact, all user input (keyboard or mouse) can be mapped to user-specified actions via CTL values and the **CTL** system variable. Programs can test this variable during user interaction to determine what action is to be taken. Use of CTL values and the **CTL** system variable is discussed in other sections later in this manual, most notably, *Chapter 6.* **Graphical User Interfaces** and *Chapter 9.* **External Components**.

Note: In character-based ProvideX, the standard convention for F4 (**CTL** = 4) terminates the program, and F3 (**CTL** = 3) backs up one field.

Positive CTL values represent function / control keys and can also be applied to other input signals that are returned to the user application. These CTL values may be defined / redefined using the **DEFCTL** directive; however, the replacement only occurs when the original value is rejected. See **DEFCTL**, *Language Reference p.77*.

Negative CTL values have special significance in ProvideX. They have fixed definitions and are handled internally by the system. The complete list of **Negative CTL Definitions** are provided in the *Language Reference, p.813*.

Input editing can also be overridden using the **'BI'** and **'EI'** (*Input Transparency*) mnemonics; in which case, you will receive the CTL values directly. A variation is to use the **'ME'** and **'MN'** (*Edit Mode*) mnemonics, which deal with all CTL values that can be handled directly by ProvideX, but will pass any invalid requests on to the program.

CTL Subroutines

The **SETCTL** directive can be used to simplify the processing of CTL values by transferring control to a subroutine whenever a specified CTL value is received by an **INPUT** statement. See **SETCTL**, *Language Reference p.303*. When the subroutine returns, the **INPUT** statement is re-executed; e.g.,

```
0010 SETCTL 4:WRAP_UP
0020 SETCTL 3:BACK_UP
...
0300 BACK_UP:
0310 FLD_NO=FLD_NO-1
0320 EXITTO 0110
...
9000 WRAP_UP:
9010 END
```

This directive can simplify the coding of programs that implement several **INPUT** statements and is ideal for the reuse of **CTL** processing procedures.

Output Statements

The **PRINT** (**?**) directive is used to format and output printable data. If the data is intended for a printer or file, the **PRINT** statement must include a valid ***channel number***; otherwise, the output is displayed immediately at the console. In this section, the primary focus is on output operations as they apply to the console.

Note: Use of the **PRINT** directive to output data to printers or files is covered in *Chapter 5.* **File Handling** and *Chapter 7.* **Printing**.

PRINT supports the ability to position output and allows insertion of special control codes through the use of **Mnemonics**, *p.30*. The basic **PRINT** statement appears as follows:

PRINT *list*

Where *list* is a comma-separated list of variables, literals, expressions, mnemonics, or screen positions. For syntax details, see **PRINT**, *Language Reference p.251*.

All data sent to a display device must be in ASCII format. String variables and string literals are maintained in ASCII and are displayed as is. Numeric data is maintained in binary format and is converted to ASCII automatically. It is important that string variables only contain *printable characters*. Attempting to print control sequences to the screen may cause unpredictable results.

By default, each **PRINT** statement advances one line on the console after outputting the data. This automatic advance can be overridden by terminating the output list with an extra (***hanging***) comma, in which case data from the next **PRINT** is appended to the current line; e.g.,

```
->0010 LET A=4, B=5
-:0020 PRINT "Multiplying ",A," times ",B
-:0030 PRINT " results in ", A*B
-:run
Multiplying  4 times  5
 results in  20
```

If line 20 includes a hanging comma the result will appear on the ***same line***; e.g.,

```
->0010 LET A=4, B=5
-:0020 PRINT "Multiplying ",A," times ",B,
-:0030 PRINT " results in ", A*B
-:run
Multiplying 4 times 5 results in 20
```

Format Masks

A format mask is a character string that is used in an input or output statement to describe how data is to be filtered, displayed, or printed. Masks are most often applied to the display / printing of numeric data (**PRINT** directive). The **INPUT** directive may apply masks to the display of prompts as well as in the filtering of

incoming data. Masking may also apply to the conversion / validation of a string; i.e., the **STR( )** function. For the complete list of numeric and string masks, see **Data Format Masks**, *Language Reference p.809*.

To assign a format mask in ProvideX syntax, place a colon before the mask following the given data value:

val[$]**:***mask$*

Where *mask$* may be a literal string, a string variable, substring, or a string expression (concatenation); e.g.,

```
0010 PRINT "The total is ",A:"$#,###,##0.00CR"
```

Numeric format mask characters are used to convert numeric data (from literals, variables, or numeric expressions) to ASCII. String data can also be converted through the use of format masks. However, unlike numeric format masks, string format masks are typically used to validate that the contents of a string match a pre-defined format.

When more characters exist in the data value than are specified in the format mask, the result will generate an `Error #43: Format mask invalid`; e.g., outputting 1000 with a mask of `"##0"` causes an error. However, the system parameters **'FI'** and **'FO'=** can be specified to handle overflows without generating errors.

Unformatted Output

If no format mask is specified when outputting numeric values, the system formats the value based on certain criteria. More information on this is provided in the *Language Reference, p.809*.

Output Positioning

The **@( )** system function can be used to position output at specific column and line coordinates. This function can be used with directives wherever text is to be sent to an output device, most commonly in a **PRINT** or **INPUT** statement. The format for the **@( )** location function appears as follows:

@(*column*[,*line*]**)**

Where the *column* represents a column position (0 to the number of columns available on the screen -1) and *line* represents an optional line number (0 to the number of lines available on the screen -1).

For example, 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
```

More information see **@( )** in the *Language Reference, p.388*.

Controlling Output

Special control sequences (**Mnemonics**, *p.30*) can be inserted within an output statement (**PRINT** or **INPUT**) to invoke such functions as clearing the screen, positioning the cursor, changing the colour of characters, setting / resetting various attributes, or enabling / disabling I / O modes; e.g.,

```
PRINT 'CS',"File maintenance",'LF'
```

As the above **PRINT** statement is executed, it clears the screen (**'CS'**), displays "File maintenance" in the upper left corner of the screen, then advances one line (**'LF'**).

All mnemonics are enclosed within single quotes. Some require arguments (e.g., `PRINT 'CIRCLE'(720,600,100,1)`). Some are represented by more than one keyword: a long form or short form (e.g., **'PUSH' or 'WC'** can be used to copy the current window).

Use of an invalid mnemonic, or one that is not applicable to a particular device, results in `Error #29: Invalid Mnemonic or position specification`.

Note: Mnemonics are specific to the channel on which they are defined and are only valid while the channel remains open. When the channel is closed, the mnemonics are cleared.

ProvideX developers can also define / redefine their own 2-character control sequences via the **MNEMONIC** directive. For example, to assign settings for the ProvideX mnemonics **'CP'** and **'SP'**:

```
MNEMONIC(0)'CP'="Courier New,-8":120,40
MNEMONIC(0)'SP'="*":80,25
```

When a *defined* mnemonic is encountered in a **PRINT** or **INPUT** statement, the system converts it to the character string specified. For further details on mnemonic definitions, refer to the **MNEMONIC** directive in the *Language Reference, p.210*.

5
File Handling

From a computing perspective, a file is simply a named storage location on disk that contains a collection of data. There are many different types of files: text, programs, documents, directories, ASCII, binary, etc. It is the contents of a file that determines how it will be used; for example, a ProvideX *data file* contains information that is organized specifically to be accessed for processing by a ProvideX program.

This chapter focuses primarily on the ProvideX operations for creating ***data files***, and for transferring data in and out of different data file types.

Topics

As mentioned in *Chapter 1*, all aspects of ProvideX are designed to work seamlessly together while at the same time interface with other external components. This is also true with regards to data handling. Internally, all ProvideX database and file systems are viewed as datasets which can be accessed either sequentially or by key (such as a client or product identifier). ProvideX datasets can be accessed quickly and easily and their format is designed for maximum performance while simplifying accessibility and maintainability.

The native file system in ProvideX is optimized for small to mid-range systems but it includes all the features required to develop and maintain large-scale business applications. ProvideX supports transparent access to external databases. Built-in rollback and recovery is available for maintaining data consistency. Other features include the Views system, the ability to include embedded I/O processes to filtering/handling data, along with dynamic index creation and deletion.

Note: For information on the options that are available for storing and retrieving data, outside access to ProvideX data files, and the handling of third-party data formats (Oracle, Microsoft SQL, etc.) see *Chapter 10.* **Data Integration**.

Data Files

Prior to this chapter, most of the discussion of data has centred on how it is input, manipulated, and output at runtime. However, when processing large volumes of data, it becomes necessary to place output into a separate storage area called a *data file*. This generally involves the use of various program instructions for creating the file, opening the file, writing to or reading from the file, and closing the file.

Each data file contains a collection of data organised in a specific format and for a specific purpose, which is stored somewhere in external storage. This section explains how to create data files and discusses the various **Data File Types** that are available for use in ProvideX.

Topics

Records and Fields

The contents of a data file is usually grouped into logically-related pieces of information called *records*. Records are generally composed of one or more *fields*, each of which contains a single item of information.

For example, a Customer file might contain records, where each record is comprised of three fields: a Name field, an Address field, and a Phone number field. Each record is maintained in the file using a separator between fields. ProvideX uses the character Hex $8A$ as a default field separator:

The default field separator character is a Hex $8A$.

Record 1	Field 1	Field 2	Field 3	Field 4
Record 2	Field 1	Field 2	Field 3	Field 4
Record 3	Field 1	Field 2	Field 3	Field 4
Record 4	Field 1	Field 2	Field 3	Field 4
Record 5	Field 1	Field 2	Field 3	Field 4

A number of different ProvideX programs should be able to access this file for writing and retrieving data.

Note: The SQL environment uses the terms *rows* and *columns* for records and fields. This is further explained in *Chapter 10.* **Data Integration**.

Creating, Deleting, and Renaming Data Files

After input and processing operations are complete, the resultant data can be saved to a file that is structured to facilitate future access. ProvideX supports the creation of different types of data files and record formats. Each of these file types has its advantages and disadvantages for storing and retrieving data:

Serial Native OS records can vary in length and are typically accessed in a *sequential* manner from beginning to end.

Indexed Records are the same length and are accessed by index number.

Keyed The most common file type used in ProvideX. Each record has at least one key field and up to 15 alternate keys for FLR / VLR files (255 for EFF). This type of file may be accessed by any key field, by index, or sequentially. The record formats for keyed files may include FLR (fixed-length records padded with 00), VLR (variable-length records), and EFF (enhanced file format). Keyed files are also categorized according to the how the key is being used:

Direct Files, consisting of an external key plus data.
Sort Files, consisting of keys but no data.
Multi-Keyed Files, consisting of one or more keys plus data.

The ProvideX file creation directives (described in the sections that follow) are designed to not only define data files but to initialize the control information within these files – each is associated with a specific format. The syntax for creating the different file types and record formats is documented in the sections that follow.

ERASE is the directive used to delete a file or directory. This deletes all the records from the file and de-allocates the disc space for the file. Two other directives, **PURGE** and **REFILE** logically delete all records from a file but leave the file defined to the system. The **FILE** directive is used to recreate a file given its file description (returned by **FIB( )** or **FID( )** function). The **RENAME** directive allows the user to rename a file (or on some operating systems a directory).

Serial

These are native OS flat files containing one or more records of variable length. Normal uses for serial files include holding print images or reports, or for interchange with other operating system applications. **READ**, **WRITE**, **PRINT** and **INPUT** directives may be used on a serial file (see **Processing Data Files**, *p.107*).

The **SERIAL** directive is used to create a serial file:

SERIAL *filename$*[,*max_recs*[,*rec_size*]]

Where:

filename$ String variable that defines the name of the serial (sequential) file to create.

max_recs Estimated number of records the file is to contain. The default is no initial allocation of file space, with no limit as to final size. (Not used in most operating system implementations.) Numeric expression.

rec_size Maximum size of the data portion of the record. (Optional on most operating systems.) Numeric expression.

Example: `SERIAL "filename"`

On a serial file, the **READ** directive reads the file one record at a time, from beginning to end. **WRITE** appends data to the end of the file. Any attempt to **READ** after a **WRITE** without having either closed the file or re-positioned the pointer via the **IND=** option will result in an `Error #2: END-OF-FILE on read or File full on write`. A serial file must be locked in order to **WRITE**. The **'LU'** parameter can be used to eliminate the need to lock a serial file before writing to it.

Example:

```
0010 OPEN (2) "SERFIL" ! Open SERFIL as 2
0020 READ (2,END=1000) NAM$, ADR$ ! Read next
0030 PRINT NAM$, ADR$
0040 GOTO 20
1000 CLOSE (2)
1010 END
```

The system provides an internal key for a serial file that may be used to reference records. This internal key is 4 characters long and contains the actual address of the record in binary.

When issuing a **WRITE** to a serial file it must first be locked. The position of the last write prior to the file being closed marks the end-of-file.

Note: The end of line for a serial file is typically OS dependant and is normally a Hex $0A$ on UNIX/Linux systems and Hex $0D0A$ on Windows.

Also, when processing a typical serial file, a **WRITE** directive will append a Hex $8A$ field separator to each record while the **PRINT** directive will not.

Indexed

In this type of file, the records may be accessed by index number. Each record on the file is assigned a record index, starting from 0. While the storage space allocated for each record is the same, the contents of the record must fit within the defined record size, with extra spaces padded with `$00$`. All records in an indexed file are the same length. No keys are maintained for indexed files.

The **INDEXED** directive is used to create an indexed file:

INDEXED *filename$*,[*max_recs*],*rec_size*[,**SEP=***char$*]

Where:

char$ Separator character. Hex or ASCII string value.

filename$ Name of the indexed file to create. String expression.

max_recs Maximum number of records in the file. Optional numeric expression. Default is no initial allocation of file space, with no limit as to final size. 0 *zero* indicates that the number is dynamic.

A positive value indicates that the file is pre-sized to the specified number of records – an `Error #2: END-OF-FILE on read or File full on write` will be generated if an attempt is made to add more than this specified number of records, or access an **IND=** value above this limit.

rec_size Size of the data portion of each record.

Example: `INDEXED "filename",,100`

Indexed files support all I/O directives except **REMOVE** (see **Processing Data Files**, *p.107*). Whenever an indexed file is read or written to, the current file position is updated to the record index selected; e.g., if a read is issued for record index 80, a read of the next index returns record 81.

The **READ** directive without an index specified returns the record with the next record index. Attempting to **READ** a record whose index exceeds the highest index written results in an error.

A **WRITE** directive without an index specified overwrites the record with the next record index. Attempting to write a record whose index exceeds the highest previously written results in all intermediary records being initialized with Hex zeroes except where the index specified exceeds the maximum record size for the file, in which case an error is returned.

Example:

```
0010 OPEN (2) "ORDFIL" ! Assign ORDFIL to channel 2
0020 OPEN (3) "ORDLIN" ! Assign ORDLIN to channel 3
0030 NEXT_LINE: INPUT "Enter order number:",ORD_NUM
0040 IF ORD_NUM = 0 THEN END
0050 READ (2,IND=ORD_NUM,END=0030) ORD_CST$,ORD_DTE$, ORD_LINES
0060 PRINT "Order:", ORD_NUM, " Cust:", ORD_CST$
0070 READ (3,IND=ORD_LINES) LIN_QTY, LIN_ITEM$, LIN_AMT, LIN_NEXT
0080 PRINT LIN_QTY:"##0:", LIN_ITEM$,@(60), LIM_AMT
0090 IF LIN_NEXT = 0 THEN GOTO NEXT_LINE
0100 ORD_LINES = LIN_NEXT
0110 GOTO 0070
```

Keyed

This is the most common file format used in ProvideX. It contains at least one unique (primary key) field that identifies each record on the file. ProvideX supports internally and externally defined keys with record sizes up to 2GB. There are a wide variety of key segment options.

Keyed files support fixed-length (FLR) and variable-length (VLR) records as well as the ProvideX Enhanced File Format (EFF). Up to 16 keys and 96 segments are supported for FLR & VLR based files while EFF increases these to 255 keys and 255 segments per key. EFF files are further discussed under **Enhanced File Format**, *p.106*.

Three types of Keyed files can be created:

Direct Files	consisting of an external key (FLR / VLR, EFF)
Sort Files	consisting of keys but no data (FLR / VLR, EFF)
Multi-Keyed Files	consisting of one or more keys (FLR / VLR, EFF).

All ProvideX I/O directives work with keyed files (see **Processing Data Files**, *p.107*). The **READ** directive without a **KEY=** option reads the record with the next higher key. When reading a file with an external key, it is normal to first retrieve the key of the next record; e.g.,

```
k$=KEY(1,end=done)
read (1,key=k$)
```

The **WRITE** directive without a **KEY=** option is invalid unless the record was previously extracted or has an imbedded key (see **Multi-Keyed Files**, *p.103*).

The **REMOVE** directive deletes the record whose key matches the key specified. When the record is deleted its associated disk space is returned for re-use within the file.

Accessing or attempting to access a record whose key does not exist positions the file pointer to the next higher existing key. The only exception is when using the **FIND** directive. This directive is a variation of the **READ** verb that does not update the current file pointer if the key is not found.

Example:

This assumes the file ACTNME has a key of name with a data field of account number.

```
0010 OPEN (2) "ACTNME" ! Assign ACTNME to file 2
0020 INPUT "Enter name:",NM$
0030 IF NM$="" THEN END
0040 K$=NM$
0050 READ (2,KEY=K$,DOM=0070) A_NUM$ ! Read record info
0060 PRINT A_NUM$, " ", K$
0070 K$=KEY(2,END=0020) ! Get next higher key
0080 IF K$<=NM$+$FF$ THEN GOTO 0050
0090 CLOSE (2)
```

Direct Files

A direct file is a keyed file with an external key. The key size must be specified along with the file name. The **DIRECT** directive is used to create a direct file:

DIRECT *filename$*,*max_len*[,*max_recs*[,*rec_size*]]

Where:

filename$	Filename of the **DIRECT** (Keyed) file. String expression.
max_len	Maximum length of the key for all records in the file. Numeric expression, integer.

max_recs Maximum number of records in the file. Optional numeric expression. Use a comma with no value to set the default (zero - unlimited). If a positive value is supplied, ProvideX creates and pre-allocates disk space for the file (for FLR/VLR formats). With a negative value, ProvideX allocates sufficient disk space for the file (for FLR/VLR formats), but will set *max_recs* back to zero.

rec_size Maximum size of the data portion of each record (excluding the key). Optional. Numeric expression. You can use:
No Value: Default is VLR with maximum size of 256.
Positive Integer: FLR of size specified.
Negative Integer: VLR with maximum length specified.

Example: `DIRECT "CSTFLE",6`

Sort Files

This type of keyed file consists of a key only, with no data record portion. Key-size must be specified along with the file name. The **SORT** directive is used to create a sort file:

SORT *filename$*,*max_keysize*[,*max_rec*]

Where:

filename$ Name of the **SORT** file to create. String expression.

max_keysize Maximum key size to be maintained for this file. Numeric expression.

max_rec Estimated number of records that the file is to contain. Default is no initial allocation of file space, with no limit on final size. Numeric expression.

Example: `SORT "CSTFLE",6`

Multi-Keyed Files

The **KEYED** directive can create a file with one or more keys. The primary key may be external or internal. If the first field after the filename is a number, an external file key is created; if it contains a key definition (enclosed within []) then only internal key fields may be used.

The first key specified is considered the primary key. Every record must have a unique primary key. You can have duplicate secondary keys from record to record.

A ***multi-keyed*** file is a keyed file with one primary record key and up to 15/255 secondary keys. The primary key must be unique within the file, while the secondary keys may be duplicated between records; e.g., where two JONES may exist. Multi-keyed files are defined like any keyed file except that the key definition contains a series of comma-separated key declarations. The first key declaration is for the primary key; the remaining declarations are for secondary keys.

The **KEYED** directive is used to create a file with one or more keys:

KEYED *filename$*,[,*extkey_len*][,*key_def$*][,*max_recs*[,*rec_size*]][,*fileopt*]

Where:

filename$ Name of the file to create. String expression.

extkey_len Numeric expression. Length of the external key for all records in the file.

key_def$ String expression defining the key. The Keyed file can be single- or multi-keyed. A key definition is made up of one or more key field definitions ranging 0 to 15 for FLR / VLR files or 0 to 255 for EFF files.

The key definition formats are as follows:

Single key field:
[["*keyname*":]*field*:*offset*:*len*[:"*attr*"] **]**

Composite key fields (using the + operator):
[["*keyname*":]*field*:*offset*:*len*[:"*attr*"]]+[*field*:*offset*:*len*[:"*attr*"] **]**

Multi-keyed alternate key fields are comma-delimited:
[["*keyname*":]*field*:*offset*:*len*[:"*attr*"] **],** [["*keyname*":]*field*:*offset*:*len*[:"*attr*"] **]**

Where:

keyname	Name of key assigned for use in **KNO=***name$* options.
field	Integer representing specific field number, 0 *zero* for record-based offset, or **KEY** to indicate an *external* key; e.g., `[KEY:1:6]+[2:1:30]`.
offset	Starting position within the field (integer, 1 to 3839).
len	Length, number of characters in the key field (integer).
"*attr*"	Attribute characters.
:	Colon - the separator for elements in a key segment.

Note: The *outer* set of square brackets in the above formats are part of the syntax; the *inner* brackets indicate optional syntax items (i.e., the brackets enclosing the optional [:"*attr*"] are not part of the syntax).

max_recs Maximum number of records the file is allowed. Optional numeric expression. The default is zero (no limit). (Use a comma with no value to set the default.) If a positive value is supplied, ProvideX creates and pre-allocates disk space for the file (for FLR / VLR formats). With a negative value, ProvideX allocates sufficient disk space for the file (for FLR / VLR formats), but will set the *max_recs* count back to zero (unlimited).

rec_size Maximum size of the data portion of the record (excluding the key). A negative value creates a variable-length record (VLR) data file with the maximum record length equal to the positive value of this field. A positive value creates a fixed-length record (FLR) formatted file. If you do not specify size, the default is VLR with a maximum record size of 256. The maximum block size for a VLR file is 31KB and the maximum record size is 31000 bytes. Attempting to create a VLR file with a record size more than 31000 bytes results in an FLR file with the requested record size.

fileopt Supported file options:
BSZ=*num* Block size. Numeric expression (1 - 63).
ERR=*stmtref* Error transfer.
SEP=*char$* Default field separator character. Hex or ASCII string value.
OPT=*char$* Single character parameter; i.e.,
`"C"` - *Compressed*. Adds simple compression to record data.
`"X"` - *Extended Record Size*. Extends record sizes up to 2GB per record.
`"0"` - *Create VLR/FLR files (default if* `'KF'=0`*)*
`"1"` - *Create EFF Files with 2GB limit*.
`"2"` - *Create EFF Files without 2GB limit (supported platforms).*
`"Z"` - *Set ZLib Compression for VLR and EFF Files.*

Note: `OPT="2"` generates `Error #99: Feature not supported` on platforms that do not offer Large File Support (LFS). Using options `"Z"` and `"C"` together will result in an `Error #32`.

Internal keys are defined as follows:

```
KEYED "CSTFLE",[1:1:6],[2:1:10],,500
```

A ***multi-keyed*** file may contain an external and several internal keys:

```
KEYED "CSTFILE",6,[1:1:6],[2:1:10],,500
```

If desired, a composite key can be generated using the '+' operator to concatenate several data fields, which will form a complete key:

```
KEYED "CSTFLE",6,[2:1:10]+[KEY:1:6]
```

Note: If, when generating a key, the descriptor references data outside the data field (either by an offset outside of field specified or a length which exceeds the field length) and the descriptor is other than the last descriptor within a composite key, the key is padded with Hex 00 in order to achieve the length specified.

There is a limit of 96 total data fields allowable in a file for use within keys (on FLR/VLR files). This means that the total number of data fields that can be used to define keys must be no more than 96 with the maximum number of keys being 1 primary and 15 secondary. If, for example one key consists of twenty data fields, the maximum number of additional fields would now be limited to 76 for a total of 15 keys, 96 data fields.

If no external key is desired on the file, only key descriptors would be specified with the first key descriptor being considered the primary key.

Using the Secondary Keys

For the purposes of accessing multi-keyed data files each key is assigned a key number. The primary key is assigned the key number of zero (0), the first secondary key is assigned one (1), and so on. When reading a multi-keyed file the system performs the read based on the current key number being used. The key number is specified by the **KNO=** option in the **WRITE**, **FIND**, **EXTRACT** directives or any of the

key functions; i.e., **KEC( )**, **KEF( )**, **KEL( )**, **KEN( )**, **KEP( )**, **KEY( )**. Once **KNO=** is specified it remains the current key number until changed by a subsequent **KNO=** option. When a file is initially opened, the key number is set to zero (primary key).

All file input directives function basically the same on multi-keyed files as they do on normal single keyed files. One exception is the **READ(*n*,KEY=*xxx*)** directive on secondary keys. This directive will read the first record that contains the desired key. Specifying a **KEY=** on an alternate key that contains duplicates will result in what appears to be an endless loop; i.e., the read will continue to re-position the key pointer to the first of the duplicates. Therefore, do not use the **KEY=** option in this case.

Updating Multi-Keyed files

Adding or updating records on a multi-keyed file uses the same **WRITE** directive as does a single-keyed file. The record to be changed or added is determined by the primary key. If the file contains an external primary key, the **KEY=** option must be specified in the **WRITE** statement. If the file does not contain an external key then the key will be determined by the data fields presented in the **WRITE** statement.

Note: If a **KEY=** option is specified in a **WRITE** directive and no external key exists or conversely if no **KEY=** option is specified and an external key does exist an `Error #80: Invalid key definition, number or name` is generated.

The **REMOVE** statement is used to delete a record from a multi-keyed file. The **KEY=** option is recommended on the **REMOVE** directive. The key specified must be the primary key of the record to be removed regardless of whether the key is external or not. If the **KEY=** option is not specified, then the last record read or extracted will be removed.

Enhanced File Format

Enhanced File Format (EFF) allows for single files up to 504 gigabytes in size. These are 64-bit and are intended for LFS (Large File System) operating systems that provide 64-bit file addressing and locking functions. Operating systems that do not provide 64-bit file functions are also able to read / write this format, but only within a 2GB limit.

The **CREATE TABLE** directive is designed for creating EFF files on platforms that support LFS, 64-bit addressing. It uses the same syntax as the **KEYED** directive (described in the previous section):

CREATE TABLE *filename$*,[,*extkey_len*][,*key_def$*][,*max_recs*][,*rec_size*][,*fileopt*]

EFF files can also be created with the **KEYED** directive by setting the **'KF'** system parameter (`KF"=1` or`"KF"=2`) or by including `OPT="1"` or `OPT="2"` in the syntax.

Processing Data Files

This section covers the different methods that are available for transferring data to/from a ProvideX data file

Topics

As discussed in *Chapter 4*, file access requires an **OPEN** statement to establish a connection before I/O operations can take place. This process involves setting a pointer to the beginning of the opened file, allocating system resources, and assigning a file number from (1 to 63 for *local files*, and 64 to 127 for *global files*). Extended file mode (**'XF'** parameter) expands these ranges from 0-32767 for local files, and 32768-65000 for global files.

Once a file is opened, all further references to that file are handled via the assigned channel/file number. The actual number of files that can be opened at any one time depends on the operating system. Once a file number is assigned to a file it cannot be reused until the file is closed, typically by the **CLOSE** directive.

Most applications will be accessing data and program files in more than one location (different directories, drives, etc.). While files can be referenced directly via their complete path name, dealing with the different locations and path formats can sometimes be problematic. ProvideX has the ability to set default file search rules to simplify this process (see **Prefix Processing**, *p.122*).

ProvideX also has the ability to **OPEN** OS sequential (flat) files in order to transfer of data from non-ProvideX applications. See **Foreign File Access**, *p.126*.

File Processing Directives

The following ProvideX directives are used for handing file input and output:

INPUT Issues prompts to the screen and to process responses. The channel referenced is normally an I/O device but an indexed file may be used.

PRINT Formats and outputs printable data to a file, printer, or display device. This instruction also processes mnemonics and positioning information.

READ Loads data from a file. The parameter list must contain only variables that are loaded from the record read. These receive the contents of each of the record fields in the order in which they are specified in the parameter list. See also, **Processing Records**, *p.111*.

As an alternative to the **READ** directive, use **FIND** to read the file without moving the pointer if the record is not found. The **EXTRACT** directive reads the file and automatically locks the record until the next I/O operation on the channel.

FIND Reads data from a file where data is split into fields (separated by current delimiter or defined by embedded format, headers, etc.) with the contents of the first field placed in variable 1, the second into variable 2, and so on. See also, **Processing Records**, *p.111*

EXTRACT Reads data from file where data is split into fields (separated by current delimiter or defined by current delimiter or in an embedded IOList format) with the contents of the first field placed in variable 1, the second into variable 2, and so on. This locks the record until the next I/O operation on the channel. See also, **Processing Records**, *p.111*

WRITE Writes a record to a file. In the case of indexed or keyed files this may rewrite existing records. The values specified in the parameter list are written into consecutive fields, each separated by a field separator (Hex $8A$) or formatted as per format specifications in the list. See also, **Processing Records**, *p.111*.

REMOVE Deletes a record from a keyed file. No parameter list required.

These directives use similar syntax and options, which are described below. You can also query and read records from a specified data file using the directive **SELECT..FROM..NEXT RECORD**, *p.110*.

Format

As mentioned, all **INPUT**, **PRINT**, **READ**, **FIND**, **EXTRACT**, **REMOVE** and **WRITE** statements have a common format for processing data files:

directive (*chan*[,*fileopt*])[*varlist*]

Where:

directive **INPUT**, **PRINT**, **READ**, **FIND**, **EXTRACT**, **REMOVE**, or **WRITE** directive.

chan Channel/logical file number of target file (see **Processing Data Files**, *p.107*).

fileopt Supported file options. See **Options**, below.

varlist Comma-separated list of variables, literals, expressions, or mnemonics to be processed. See **Input/Output Parameters**, *p.114*.

Complete documentation on each of these directives is provided in *Chapter 2* of the ProvideX *Language Reference*. See **Data Files** for details on how these file processing directives are used for handling the different types of files and record formats.

Options

The following file options are used with various I/O directives to fine-tune the operation and redirect processing:

BSY=*stmtref* Statement number to transfer to if the record is currently locked by another process.

DOM=*stmtref* Indicates the statement number (*stmtref*) to transfer to if the record referenced by the directive is either missing (in the case of **READ**) or already exists (on **WRITE**).

END=*stmtref* Traps the end of the file on a **READ** (`Error #2: END-OF-FILE on read or File full on write`); on a **WRITE** directive causes a transfer if the output file has reached its maximum size or no more file space is available.

ERR=*stmtref* Indicates the statement (*stmtref*) to transfer to if an error occurs during processing of the directive.

IND=*num* Specifies record index value used to uniquely-identify a record in indexed and keyed files. For fixed length keyed files, *num* represents an offset into the data file (first record has an index of 0, second is 1, and so on). For variable length keyed files, *num* represents a logical page address and record index within that page. Used with the **INPUT** directive, **IND=***num* sets the starting position (column number) of the cursor in the input field.

When processing a file in Binary mode via the **ISZ=** option, **IND=***num* identifies the record address to access when the file is opened.

KEY=*string$* Specifies the record key value used to uniquely-identify a record.

LEN=*num* Limits the length of the input data. If this option is specified in an **INPUT** statement, only the number of characters specified by *num* will be read.

KNO=*num|name$* Access key number (*num*) or name (*name$*), where *num* is 0 based (0-15 for VLR/FLR files, 0-255 for EFF files).

RNO=*num* Specifies the record number within the file based on actual key sequence position; first record in the file is RNO#1.

ProvideX checks that any options selected match the directive. The order of the options is irrelevant; inconsistent options (such as having both **IND=** and **KEY=**) are rejected. In the case of multiple error transfers, (**DOM=**, **END=**, and **ERR=**) **DOM=** and **END=** take precedence. A complete list is provided in the *Language Reference, p.806*.

SELECT..FROM..NEXT RECORD

Use the **SELECT** directive to open, read and query records from the specified data file or just to read data from a specified file number. As each record is read, ProvideX processes any logic included between the **SELECT** and **NEXT RECORD** keywords in the statement. When ProvideX encounters **NEXT RECORD** with no records found for a nested **SELECT**, it will locate the corresponding **SELECT** statement.

If a **WHERE** clause is included, ProvideX will process only those records *where* the condition is ***true***. **BEGIN** and **END** are only supported for **KEYED** and ***MEMORY*** files and for use with **External Databases**, *p.320*.

```
0010 SELECT IOL=0100 FROM "CUST_FILE",KNO=1 BEGIN "ABC CO" END
"NEW CO" WHERE CITY$="CLARENDON"
0020 PRINT REC(IOL=0100)
0030 NEXT RECORD
0100 IOLIST CUST$, NAME$, ADDR1$, ADDR2$, CITY$, PROV$, POSTAL$,
INV_DT$, AMT, TERMS, DUE_DT$
0110 PRINT "DONE"; END
```

If ProvideX is instructed to exit the **SELECT** loop early (with an **EXITTO** directive) the file will be closed. **SELECT** can also be used to read data from tables in a SQL database. See **SELECT..FROM..NEXT RECORD** in the *Language Reference, p.295*.

File Processing Functions

There are twelve file processing functions – all expect a file number as first argument.

FID( ) Returns a description of characteristics of the file specified; returns 1 of 5 formats depending on the **'FF'** parameter.

FIB( ) Same as **FID( )** but always returns ProvideX native format.

FIN( ) Returns detailed physical aspects of file specified.

IND( ) Returns an index of the next record in the file. Generates an error at end-of-file.

KEC( ) Returns a key of the current (last read / written) record. Generates an error at start of file.

KEF( ) Returns the key of first record on the file. Generates an error if no records exist.

KEL( ) Returns the key of the last record on the file. Generates an error if no records exist.

KEN( ) Returns the key of the record after the next record to be read.

KEP( ) Returns the key of the previous record - keyed files only. Generates an error at the start of file.

KEY( ) Returns the key of next record. Generates an error at the end of file.

RCD( ) Returns contents of a record – equivalent to **READ RECORD** (see **Processing Records**, *p.111*).

RNO( ) Returns the record number of a specified record, in key sequence.

Processing Records

As mentioned earlier in the chapter, file data is usually grouped into **Records and Fields**, *p.98*. The **File Processing Directives**, described earlier, are typically used to read / write records in a file separating the data into one or more fields.

Record-Specific Directives

In some cases it is necessary to read the contents of a record from a file that does not contain fields; e.g., when reading data files not created by ProvideX or writing data files that are to be processed by some other application. To handle files of this type, ProvideX includes a set of record-specific directives. In these cases only a single string parameter can be specified in the parameter list.

The **READ RECORD** directive reads the record specified and place its contents into the string variable specified in the parameter list. The **WRITE RECORD** directive writes the string specified in the parameter list to the file without the insertion of a field separator. If the file contains fixed length records, the **WRITE RECORD** statement will append nulls (Hex 00) to the record being written. Other record-specific directives include **FIND RECORD** and **EXTRACT RECORD.**

Example:

```
0010 OPEN INPUT (1) "." ! open current directory
0020 READ RECORD (1,END=DONE)R$
0030 PRINT R$
0040 GOTO 0020
0050 DONE: CLOSE (1); END
-: run
```

MEMORY *File*

A ***MEMORY*** logical file is simply a memory-resident queue of records that can be accessed by index or (external) record key. The system functions **KEC( )**, **KEF( )**, **KEL( )**, **KEN( )**, **KEP( )**, **KEY( )**, and **IND( )** will work with a memory file. As well you can access memory files by record number, **RNO( )**. The record index will equal the logical placement of the records in key order sequence. For complete syntax details, see ***MEMORY*** in the *Language Reference, p.737*.

Creating/Deleting the file. Two types of memory files may be created. The following syntax creates a memory-resident queue of records:

OPEN *(chan) "****MEMORY****"**

The following creates a memory-resident multi-key file similar to a regular keyed file:

OPEN (*chan***[,***fileopt***])"*MEMORY* [;KEYDEF=***key_def$***]"**

The following syntax deletes a memory file and returns memory to the system:

CLOSE (*chan***)**

Keyed (Direct File) Handling. The following syntax adds / updates a record:

WRITE (*chan*, **KEY=***string$*) *iolist*

or

WRITE RECORD (*chan*, **KEY=***string$*) *strexpr*

To read a record:

READ (*chan*, **KEY=***string$*) *iolist*

or

READ RECORD (*chan*, **KEY=***string$*) *strvar*

To remove a record:

REMOVE (*chan*, **KEY=***string$*) *iolist*

Indexed Handling. The following syntax writes to the open file:

WRITE (*chan*, **IND=***num*) *iolist*

or

WRITE RECORD (*chan*, **IND=***num*) *strexpr*

This will insert records at the specified point in the memory queue file. It will insert and not overwrite the existing records; if you currently have 5 records in the file and issue a `WRITE (chan,IND=3)`, the record formerly at index 4 will now be at index 5.

To read a record:

READ (*chan*, **IND=***num*) *iolist*

or

READ RECORD (*chan*, **IND=***num*) *strvar*

To remove a record:

REMOVE (*chan*, **IND=***num*)

Accessing Directory Files

Directory files represent the operating system's directory tables that are used to maintain the list of files present in the system. ProvideX provides read access to these. When a directory file is opened, the records read from it contain the names of the files contained within the directory. The names are not returned in any specific order. The sequence of the entries within a directory is controlled by the operating system.

Use the **INPUT** keyword in an **OPEN** statement to access a disk directory:

```
0010 OPEN INPUT (1) LWD ! Open current directory
0020 READ (1,END=1000) FL_NAME$ ! Get file name
0030 PRINT FL_NAME$ ! Display file name
0040 GOTO 0020
1000 CLOSE (1)
1010 END
```

Note: ProvideX does not allow the user to write to a directory file and will otherwise report an `Error #13: File access mode invalid`.

File Locking - Reserving a File for Exclusive Use

By default, all files accessed by ProvideX are shared between users, allowing any number of users to access the same file and data at the same time. Due to design aspects of a file or its contents, it may be desirable to reserve a file for exclusive use. When a file is reserved for exclusive use only the user who placed the reservation on the file may have access to its contents. All other users are denied access to the reserved file.

A file may be reserved for exclusive use via the **LOCK** directive. Once reserved, the file is considered locked. Locking a file can only be accomplished if the user who places the lock is the only user that currently has the file open. If another user has the file open, the **LOCK** request will be denied. Once locked, no other users will be able to open the file. Any attempt to do so will return `Error #0: Record/file busy`.

There are two instances when it is necessary to lock a file within ProvideX:

- ***Writing to a serial file.*** Due to the nature of a serial file and the format of the data it contains, it is mandatory that a serial file be locked before attempting to write to it. Any attempt to write to a serial file without locking it will return an error.
- ***Purging data from a file.*** The **PURGE** directive removes all records from a file. Before this command can be performed the file must be locked in order to assure that no other user is accessing the file. If the **PURGE** directive is used on a file that is not locked, an `Error #82: File must be 'LOCKED' before being 'PURGED'` will be generated.

ProvideX will automatically lock all physical devices when they are opened. This will prevent multiple users from attempting to share printers and tape drives at the same time. An exclusive reservation against a file can be removed by the **UNLOCK** directive and is automatically removed when the file is closed.

Record Locking - Sharing Critical Information

Both keyed files and indexed files provide a record locking facility. Record locking allows the program to gain exclusive access to a record on a file thereby preventing other programs from incorrectly updating the information.

Typical uses of record locking would be updating a product inventory record or adjusting an account balance. In general record locking is necessary whenever modifying a value in a data file. When it is necessary to modify a value the program should first read and lock the current value, perform the adjustment, then rewrite the value back. Locking the record will force other users to wait for the update to complete or until the lock is removed.

The **EXTRACT** and **EXTRACT RECORD** directives can be used to lock a record to prevent other users from having access to it. This lock stays active until the next I/O request for the same file or until the file is closed. Using a **KEY=** option on a **READ**, **FIND** or **EXTRACT** statement to retrieve the next record while a record is locked will result in the locked record being returned instead. You can enable *read through* access for records that have been extracted by setting the **'XI'** parameter.

Input/Output Parameters

Within ProvideX, all **File Processing Directives** (that don't specify **RECORD**) allow for the specification of a parameter list. This list can consist of literals, variables, expressions, and mnemonics. Depending on the type of operation being performed, some types of parameters cannot be used; e.g., you cannot **READ** a literal. The following table describes the parameters each of the file I/O directives will allow:

PRINT Literals, variables, expressions, and mnemonics are all output to the file/device.

INPUT Literal and mnemonics are output to the device. Variables are read from the file/device.

READ Only variables are allowed. (Same for **FIND**, **EXTRACT**.)

WRITE Literals, variables, expressions, and mnemonic can be written.

All of these directives allow for the inclusion of an **{ALL}** or **[ALL]** option following a variable which will cause the system to process all elements of an array (See also **String Arrays** and **Numeric Arrays**, *p.37*.) Assuming an array of `CAT[3,2,1]` the elements in order will be:

```
CAT[0,0,0], CAT[0,0,1], CAT[0,1,0]
CAT[0,1,1], CAT[0,2,0], CAT[0,2,1]
CAT[1,0,0], CAT[1,0,1], CAT[1,1,0]
CAT[1,1,1], CAT[1,2,0], CAT[1,2,1]
CAT[2,0,0], CAT[2,0,1], CAT[2,1,0]
CAT[2,1,1], CAT[2,2,0], CAT[2,2,1]
CAT[3,0,0], CAT[3,0,1], CAT[3,1,0]
CAT[3,1,1], CAT[3,2,0], CAT[3,2,1]
```

Common I/O Parameter List (IOList)

In order to simplify and reduce coding, I/O directives allow the use of a common list of parameters, an ***IOList***. To reference an IOList, the programmer includes the **IOL=** option either as the parameter list or within a parameter list of **File Processing Directives**; e.g.,

```
0100 READ (1,KEY=K$) IOL=1000
0110 WRITE (1,KEY=K$) IOL=1000
```

The line reference refers to a line number or label where an **IOLIST** directive is found:

```
1000 IOLIST A,D,K(1),F$,F1$
```

The **IOLIST** directive is used to define an IOList. An I/O directive may contain as many **IOL=** options as required.

Defining IOLists on **OPEN**

Another method to reduce the requirements of providing IOLists on file I/O statements is to provide the IOLists with the file **OPEN** directive. If you include the option **IOL=** within the **OPEN** statement, all further **READ** or **WRITE** statements (which do not specify any form of IOList) will automatically use the one given in the **OPEN**; e.g.,

```
0010 OPEN(1,IOL=1000) "CSTFLE"
........
0340 READ (1,KEY=K$+"00")
........
0500 WRITE (1,KEY=K$+"00")
........
1000 IOLIST A,D,K(1),F$,F1$
```

The **READ** statement at line 0340 and the **WRITE** statement at line 0500 both will utilize the **IOLIST** at line 1000.

Another advantage of this technique is that as long as the file remains open, all subsequent **READ** or **WRITE** statements will use the **IOLIST** at 1000 even if they are executed from different programs.

Variable IOLists

ProvideX allows you pass IOLists as variables and use variables as IOLists. In order to utilize a variable as an IOList, it must first be initialized with the object code for the desired IOList. This can be accomplished via the **CPL( )** or **PGM( )** functions; e.g.,

```
0100 IOL_1$=CPL("IOLIST A$,B$,D")
```

or

```
0100 IOL_1$=PGM(1000)
1000 IOLIST A$,B$,D
```

However, due to the fact that a **RENUMBER** directive could change the line number of the IOList in the above example, you should use the following piece of logic:

```
0100 IOL_1$=PGM(TCB(4)+1)
0110 IOLIST A$,B$,D
```

This uses a `TCB(4)` function to get the current line number, then adds 1. When this is passed to the **PGM( )** function, the contents of the next line will be returned, which should have the desired IOList. This type of coding will not be effected by a **RENUMBER** directive.

Once a variable has been loaded with the IOList, it may be specified following the **IOL=** option rather than a line number or statement name; e.g.,

```
READ (1) IOL=IOL_1$
```

One of the easiest ways to handle IOLists is to assign them during initialization to **Global Variables**, *p.35*; e.g.,

```
0100 %CST_IOL$=PGM(TCB(4)+1)
0110 IOLIST CST_NM$,CST_ADR$,CST_CITY$,..
```

This will allow you to easily change IOLists without having to make large scale program changes. If desired you could even open the files and assign them global file numbers.

Formatted IOLists

In addition to simply naming the variables to be used in a **READ** or **WRITE** directive, an IOList can also define the exact format of a data record. A format specification may be given immediately following the variable names on an IOList directive. The format specification is used to define the exact size and form that the data has on the file record.

Format specifications should be separated from the variable name by a **:** *colon* and enclosed within **[ ]** *square brackets*. The following are currently supported format specifications:

CHR(*len*)	Variable length string (fixed output or delimited)
CHR(*dlm*)	Variable length string (delimited)
LEN(*len*)	Fixed length string
STR(*dlm*)	Quoted string
NUM(*len*,*scl*)	Fixed length numeric
SGN(*len*,*scl*)	Signed fixed length numeric
BIN(*len*,*scl*)	Binary numeric
INT(*len*,*scl*)	Unsigned integer numeric
BCD(*len*,*scl*)	Packed decimal numeric

len and *scl* are numeric values. *dlm* is a one-character delimiter. In the following example, the IOList on line 1000 would be used for a 60 character record, with the field NAME in positions 1-30 and ADDR1 in positions 31-60:

```
1000 IOLIST NAME$:[CHR(30)],ADDR1$:[CHR(30)]
1010 IOLIST CUST$:[STR(",")],AMNT:[STR(",")],
1010:DUEDT$:[STR("")]
```

No delimiter would exist between the fields. In line 1010, the IOList would be used against a comma-delimited file where the string values would be enclosed in quotes. Following is another example of the STR(",") IOList formatting option:

```
0010 dim CSV$[1:3]
0020 CSV$[1]="Fred",CSV$[2]="Wilma",CSV$[3]="Pebbles"
0030 iolist CSV${all}:[str(",")]
0040 print iol=0030
->run
"Fred","Wilma","Pebbles",
```

Note: Normal unformatted output is equivalent to a format of **CHR(SEP)**.

When using the **NUM( )**, **SGN( )**, **BIN( )**, **INT( )**, and **BCD( )** format specifiers, the *scl* parameter is used to define the scaling factor to be applied to the numeric data. It represents the number of implied decimal places that are to exist in the number as it resides on the file; e.g.,

Value:	*Format:*	*File Format:*
123.45	NUM(8,2)	00012345
1	NUM(8,2)	00000100
-23	NUM(8,2)	00002300
-23	SGN(8,2)	0002300-
1.6	BIN(2,1)	0010

Prefixing Variables in an IOList via REC=

Sometimes when using common IOLists it is desirable to temporarily override the variable names. The **REC=** option provides this capability. When this option is specified, the variable name which is provided (following the **REC=**) is used to prefix all the variables in the IOList; e.g.,

```
0010 LET GL_FL=HFN; OPEN (GL_FL,IOL=8000) "GLTRAN"
0100 INPUT "Which Credit GL account:",CR$:"AA-0000"
0110 READ (GL_FL,KEY=CR$,REC=CR$,DOM=0100)
0120 INPUT " Debit GL account:",DB$:"AA-0000"
0130 READ (GL_FL,KEY=DB$,REC=DB$,DOM=0120)
0140 INPUT "Amount:",AMNT:"$###,##0.00-"
0150 IF AMNT=0 THEN GOTO 0140
0160 CR.BAL=CR.BAL+AMNT, DB.BAL=DB.BAL+AMNT
0170 WRITE (GL_FL,KEY=CR$,REC=CR$)
0180 WRITE (GL_FL,KEY=DB$,REC=DB$)
0190 GOTO 0100
8000 IOLIST DESC$,BAL
```

In the above example, the **REC=** option is used to maintain two separate records in memory. One record will have all its variables prefixed with "DB.", the other will have "CR." prefixes.

Note: The **REC=** option may also appear in the **OPEN** directive.

Embedded Data Dictionary

ProvideX allows for a data dictionary to be directly embedded into keyed files making IOLists within a program unnecessary. To open a file using its embedded IOList use **OPEN (1, IOL=*)** *"File"*. From then on, all **READ** and **WRITE** statements that ***do not*** specify any IOList or variables will utilize the embedded data dictionary; e.g.,

```
OPEN (1, IOL=*) "CSTFILE"
...
READ (1,KEY=K$) ! Would use the embedded IOLIST
READ (1,KEY=K$) A$,B$! Would NOT
```

The IOList can be embedded using the **Data Dictionary Maintenance** facility.

The **REC=** clause can be used to prefix the elements in the IOList if desired; e.g.,

```
OPEN (1,IOL=*) "CTSFILE"
READ (1,REC=CST$)
```

This would read all the fields defined in the `CSTFILE` but prefix the fields with `"CST"`. Optionally the **REC=** clause can be applied in the **OPEN** directive causing the IOList to be prefixed by default.

Embedded I/O Procedures

Embedded Input/Output (I/O) procedures provide the ability to intercept and control all file I/O directives and functions using a user-defined program. The program is called when the file is accessed by an **OPEN**, **READ**, **FIND**, **EXTRACT**, **WRITE**, **REMOVE**, **PURGE**, **LOCK**, **UNLOCK**, **CLOSE**, and all **KEY( )**, **IND( )**, **RNO( )** and **FIB( )**/**FID( )**/**FIN( )** functions. For more information on the various syntax elements in ProvideX for file operations, see **Processing Data Files**, *p.107*.

The program name may be specified in the **Data Dictionary Maintenance** interface, which writes it to the embedded data dictionary structure of the file, or it can be specified using the **SETDEV** directive on an already open channel. Both ***Pre*** and ***Post*** operations are provided for most directives while functions have either a ***Pre*** or a ***Post*** operation.

Possible applications for embedded I/O procedures include:

- Security and data encryption (apply passwords and/or encrypt the data)
- Data integrity
- Data replication
- Prevent a customer from being deleted when there is an outstanding balance
- Remove cross-reference information from all files before deleting a customer
- Normalizing data files (re-direct alternate records types to “normal” files).
- Circular file journalizing (maintain a queue of the last *nnn* # of transactions).

Implementation

Embedded I/O procedures allow the developer to intercept file input and output operations on a ProvideX keyed data file, or on any channel using the **SETDEV** directive. A user-defined program is logically invoked using the **PERFORM** directive during use of any of the above-mentioned I/O directives.

In order for the program to be invoked on an **OPEN** of the file, the name of the program must be written to the file's internal data dictionary. This is only available for keyed files, and is done using the ProvideX **Data Dictionary Maintenance** interface.

The embedded I/O program must exist and be accessible using the standard **PREFIX** search rules in order to open the data file. If the program cannot be located, then an `Error #121: Invalid program format` is reported and the **OPEN** will fail.

To use an I/O procedure on any other type of file, a **SETDEV** directive must be executed after the channel has been opened. The syntax for the **SETDEV** directive appears as follows:

SETDEV(*channel*) **PROGRAM** "*IOProc*"

Where:

channel	Channel number to which the I/O procedure program is assigned.
IOProc	User-defined program to **PERFORM**.

Example:

```
10 OPEN(1)FID(0)
20 SETDEV (1) PROGRAM "ioproc.tst"
```

Note: If the specified program is not accessible, an `Error #121` is not generated on execution of the **SETDEV** directive.

To verify that the program can be located, **OPEN** or **ADDR** the program prior to issuing the **SETDEV** directive to ensure that ProvideX can access it. Another option is to reference the currently executing program; e.g., `SETDEV(1) PROGRAM PGN`.

Pre-Defined Entry Points

When file input/output is performed, the user-defined program will be logically invoked using a **PERFORM** at the following pre-defined entry points:

For directives:

PRE_READ	**POST_READ**
PRE_EXTRACT	**POST_EXTRACT**
PRE_WRITE	**POST_WRITE**
PRE_REMOVE	**POST_REMOVE**
PRE_PURGE	**POST_PURGE**
PRE_LOCK	**POST_LOCK**
PRE_UNLOCK	**POST_UNLOCK**
PRE_CLOSE	**POST_PASSWORD**

For functions:

PRE_KEY	**PRE_KEF**
PRE_KEL	**PRE_KEP**
PRE_KEC	**PRE_KEN**
PRE_IND	**PRE_RNO**

POST_FIB **POST_FIN**
POST_FID

Additional Notes:

- There is no line label entry point used when a file is opened, as the program is simply invoked.
- The **CLOSE** directive only supports the **PRE_CLOSE** entry point.
- If the entry point for a particular function does not exist, no error will be reported.
- All entry point labels are optional, and ProvideX will only **PERFORM** the routines that exist within the program

Execution Environment

The program is invoked logically using the **PERFORM** directive; therefore, it is recommended that all variables within the I/O procedure be declared **LOCAL**. This will prevent variables referenced in the I/O procedure from conflicting with any program accessing the file.

Command mode processing is not recommended within an I/O procedure. Although it may be possible to add an **ESCAPE** to the I/O procedure to have it drop to console mode, doing so can produce undesirable results, which could terminate the ProvideX session.

The I/O procedure can generate an error which in turn will be returned to the function or directive being executed. Errors may be generated by the I/O procedure as follows:

EXIT *error*

Where:

error Error number to report back to the directive or function accessing the channel.

Additional Notes:

- Any error reported by the I/O program is reported as an I/O error on the file.
- While the I/O procedure is executing, any subsequent file access to the same file is *not* passed to the file I/O procedure.
- A normal ProvideX **PERFORM** does not allow an **ENTER** statement; however, there is a special **ENTER** provided for I/O procedures.

The following arguments are passed to the I/O procedure:

ENTER *access_mode, key$, index, value$, access_options, keynumber*

Where

access_mode **0**-*Next*, **1**-*Key*, **2**-**IND=**, **3**-**RNO=**.

key$* Value in *key$ or null.

index	Value in **IND=**, **RNO=**, or **KNO=** (if **KEY=** specified).
value$	Record contents for **READ**/**WRITE.**
access_options	Bit-masked type value indicating the following: **1** - **DOM=** specified **2** - **END=** specified **4** - **NUL=** specified **8** - **BSY=** specified **16** - **FIND** directive **32** - **EXTRACT** directive
keynumber	Value of **KNO=**.

All parameters are read-only, and are not supplied for **OPEN** and **CLOSE** operations.

Note: The **POST_READ** and **POST_EXTRACT** logic is performed prior to the variables in the IOList being populated with data. This allows the data in the record to be modified before it is available for use by the program accessing the file.

If a **TIM=** clause was specified on the File I/O directive, then the value is reported in **TCB(92).**

Changing Return Values

The I/O program can alter the return value using:

RETURN *xxx$*
-or-
RETURN *xxx*

A **RETURN** *anything* in ***Pre*** logic will result in the ***Post*** logic ***not*** being executed.

Sample Code

```
0010 ! ENCRYPT - Embedded I/O Procedure to encrypt data
0100 ! ^100
0110 OPEN: ! This label is for informational purposed only, the program
0111 !"     execution started at line 10 when the open was performed
0120 LOCAL P$
0130 IF %PASSWORD$="OK" THEN GOTO 0190
0140 PRINT 'WINDOW'(10,10,50,6,"Password?",'MODE'($000F$)+'CS'),
0150 OBTAIN (0,ERR=0160)@(0,1),"Please enter the password? ",P$
0160 PRINT (0,ERR=*NEXT)'POP',
0165 ! Next line is an example of forcing an error to be returned using
0166 ! the EXIT err
0170 IF UCS(P$)<>"PASSWORD" OR CTL<>0 THEN EXIT 52
0180 %PASSWORD$="OK"
0190 EXIT ! end of open routine
0200 ! ^100 " Start of POST_READ routine
0210 POST_READ: LOCAL VALUE$,ACCESS_MODE,KEY$,INDEX,V$
0220 ENTER ACCESS_MODE,KEY$,INDEX,V$
```

```
0230 VALUE$=V$; GOSUB ENCRYPT_IT; RETURN VALUE$
0240 END ! End of POST_READ routine
0300 ! ^100
0310 PRE_WRITE: LOCAL VALUE$,ACCESS_MODE,KEY$,INDEX,V$
0320 ENTER ACCESS_MODE,KEY$,INDEX,V$
0330 VALUE$=V$; GOSUB ENCRYPT_IT; RETURN VALUE$
0340 END
1000 ! ^1000
1010 ENCRYPT_IT:
1020 LOCAL ENCRYPT_STRING$
1030 IF VALUE$="" THEN RETURN
1040 ENCRYPT_STRING$="Use this string to randomize the data"
1050 ENCRYPT_STRING$=DIM(LEN(VALUE$),ENCRYPT_STRING$)
1060 VALUE$=XOR(VALUE$,ENCRYPT_STRING$)
1070 RETURN
```

File Naming Conventions

Actually, ProvideX does not impose any naming rules for creating or accessing files. While some naming conventions (specific prefixes/suffixes) may be required by the OS or to interact with non-ProvideX applications, ProvideX developers do maintain full control over how their ProvideX-based files are named.

There are two exceptions: **NOMADS** defaults to a `.EN` extension (English) for naming library files, but this may be changed by the developer. A `.PVC` extension is used in **Object-Oriented ProvideX** to define and/or access a class definition. **ProvideX Search Rules** for locating file names are provided later in this section.

Prefix Processing

The ProvideX **PREFIX** directive allows you to specify up to 10 different prefix strings to be inserted automatically in front of all file name references; e.g.,

PREFIX [*search_string$*]

The prefixes are provided as a string to the **PREFIX** directive using a blank as a separator character; e.g.,

```
0010 PREFIX "DATA/ PROGS/"
0020 OPEN (1) "ARDATA"
```

ARDATA would be searched for first as DATA/ ARDATA, then PROGS/ ARDATA, then as ARDATA. The first occurrence of a file found with the name specified would be used.

The **PREFIX** directive causes all file creation directives to search for the file using these prefixes before considering the file as currently non-existent, and proceeding with the creation. If the file cannot be created in the first directory (permissions denied or no such directory) it will try to create the file in the next directory.

The system variable **PFX** and function **PFX( )** will return the current setting of the **PREFIX** directive and can be used to extend the **PREFIX** search rules; e.g.,

```
0010 PREFIX "TESTDATA/ "+PFX
```

For syntax details, refer to the **PREFIX** directive in the *Language Reference, p.248*.

The use of prefixes should be limited to just one or two at most, with the first entries being the most likely entries to return the file desired. Since ProvideX will try to open the file using each prefix until the request is satisfied, much time and I/O overhead will be wasted if a long list of prefixes is used.

Program Prefix

A special prefix may be defined within ProvideX specifically for use when accessing programs. Using a program prefix can increase system performance by reducing the time it takes to locate and load a program. This prefix is defined as follows:

PREFIX PROGRAM [*search_string$*]

Once a program prefix is defined, it will be the first prefix used for all **LOAD**, **RUN**, **CALL**, **PERFORM**, **PROGRAM** and **SAVE** directives. After the program prefix has been searched, any other prefixes will be checked. All other file accesses will search the other prefixes first, then the program prefix. The current setting of a program prefix may be obtained via the **PFX(PGN)** function.

Dynamic Prefix Assignment

In many systems the file names are standardized; i.e., the leading 2 or 3 characters indicate the subsystem or application that the program/file belongs to. For example `'ARHIST'` or `'ARINFO'` might belong to the ***Accounts Receivable*** subsystem.

The **PREFIX** directive allows for the automatic separation of these files based on the first characters. Substituting the **=** *equals sign* character in the **PREFIX** directive causes ProvideX to take the first character of the file name. Each character of the file name that corresponds by position to an equals sign will be used to form a subdirectory name to be used in the search.

For instance, if you include 1 equals sign, ProvideX will interpret that to mean that the first character of *filename*$ is also the subdirectory name. If you include 2 equals signs, it will take the first 2 characters as matching the subdirectory name, and so on.

ProvideX automatically finds the location to retrieve or create files by looking first for a subdirectory with a name matching, character-for-character, the portion of the filename that corresponds to the equals signs. This allows you to sort files into subdirectories based on automated substitution of the first few characters of your filename.

Example:

```
0010 PREFIX "==/"
...
0100 OPEN (1) "ARDATA"
0110 OPEN (2) "CSTDTA"
...
0300 RUN "ARPOST"
```

The following searches would be performed:

File:	*1st search:*	*2nd search:*
ARDATA	AR/ARDATA	ARDATA
CSTDTA	CS/CSTDTA	CSTDTA
ARPOST	AR/ARPOST	ARPOST

Asterisk Wildcards

You can also use ***** *asterisk* and ****** *double asterisk* as wildcard characters to support the use of filename extensions without modifying your code. With the wildcard characters, you can rename files on disk with a common file extension without modifying the program code.

Using * Single Asterisk. If the **PREFIX** directive includes a single star plus a specified extension as a filename, ProvideX inserts the filename from your **OPEN** command in place of the *asterisk* and searches for the filename with the added prefix; e.g.,

```
PREFIX "c:\somedir\*.PRG"
OPEN(chan)"FOOFOO"
```

In the example above, ProvideX scans the disk for `"c:\somedir\FOOFOO.PRG"` and opens that file if found. If `FOOFOO.PRG` is not found, ProvideX attempts to find and open a file named `"FOOFOO"`. If the filename in the **OPEN** command already includes an extension, no substitution will occur; e.g.,

```
PREFIX "c:\somedir\*.PRG"
OPEN(nahc)"MyFile.Dat"
```

In this case, ProvideX does not add the `.PRG` extension when it executes the search to find and open `MyFile.Dat`.

Using ** Double Asterisk. If the **PREFIX** directive includes two stars plus a specified extension, ProvideX inserts the filename from your **OPEN** command in place of the *asterisks* and searches first for the filename with the extension specified in the prefix; e.g.,

```
PREFIX "c:\somedir\**.PRG"
OPEN(chan)"FOOFOO"
```

In the example above, ProvideX scans the disk for `"c:\somedir\FOOFOO.PRG"` and opens that file if found. If `FOOFOO.PRG` is not found, ProvideX attempts to find and open a file named `"FOOFOO"`.

If the filename in the **OPEN** command already includes an extension, Providex adds the additional extension specified by the **PREFIX** directive when it searches for the filename; e.g.,

```
PREFIX "c:\somedir\**.PRG"
OPEN(chan)"FOOFOO.PRG"
```

In this case, ProvideX scans for `"c:\somedir\FOOFOO.PRG.PRG"`. However, if `FOOFOO.PRG.PRG` is not found, ProvideX attempts to find and open a file named `"FOOFOO.PRG"`.

Prefix File

You can use the prefix file to do dynamic translations of file names using a **Keyed** file as a lookup table. The prefix file consists of a file name (which is the key) and a data portion that contains two fields (the path to the file and the options field used in the **OPT=** clause in **OPEN**). Prefix files must be variable length. The following example creates the prefix file:

```
KEYED "PVX_PFXF",15
```

The following sets search rules using the prefix file:

```
PREFIX FILE "PVX_PFXF"
```

ProvideX Search Rules

The ProvideX default is to search all prefixes, in the following order:

OPEN *Directive*	**LOAD/RUN/CALL/PERFORM/SAVE** *Directives*
1. **PREFIX FILE**, if set; replaces pathname then continues sequence. 2. Current Directory; if '**CD**' system parameter is set. 3. **PREFIX** 0 to 9. 4. **PREFIX PROGRAM**, if set. 5. Current Directory; if '**CD**' system parameter *not* set.	1. **PREFIX FILE**, if set; replaces pathname then continues sequence. 2. Program Cache. 3. Current Directory; if '**CD**' system parameter is set. 4. **PREFIX PROGRAM** if set. 5. **PREFIX** 0 to 9 6. Current Directory; if '**CD**' system parameter *not* set.

The **PREFIX** search rules apply not only to files being found, but also to files being created. ProvideX creates files in the first location that is permitted by the **PREFIX** rules. If '**CD**' (*Search Current Directory*) is on, then all files are created in the current directory (the first permitted location). If the '**CD**' system parameter is off, then ProvideX creates the file in the first location permitted by the search rules above. For syntax details, refer to the **'CD'** parameter in the *Language Reference, p.656*. Windows search rules are used to find DLLs (i.e., not **PREFIX** search rules or current directory).

Use the **ENABLE** and **DISABLE** directives to control which of the numbered prefixes ProvideX will use in the search. (While scanning prefixes 0 to 9, ProvideX ignores any prefix that is disabled.)

The initial check for **PROGRAM** cache checks for a match against the original filenames. If you used `CALL "ABCD"` and you had previously loaded a program with the same name, ProvideX would use the one in cache. This eliminates the directory searches involved, but if you have duplicate program names in your system, it is possible to get the wrong one; i.e., if you `CALL "ABCD"`, change the directory / prefix, then re-`CALL "ABCD"`. If this happens for duplicate program names in your system, either clear the cache or do not use it.

For syntax details, see **DISABLE** and **ENABLE** in the *Language Reference, p.91*.

Foreign File Access

ProvideX has the ability to read and write native OS serial files (sometimes referred to as flat files) for easy transfer of information from other applications. It also has the ability to access any data file regardless of the file format.

The special **OPEN** file option **ISZ=** allows a file to be accessed as if it was a **Indexed** file, specifying a logical indexed record size. This means that the file's contents can be read or written directly. Once the file has been opened (with **ISZ=** set) read and write statements can be issued against it. Optionally, the user can specify the record index via the **IND=** option to access specific areas of the file directly. The size specified via the **ISZ=** option is considered the record size for each record in the file. A record index specified on a file read or write directive with a **IND=** option determines where in the file (at what record index) the I/O function is to be performed.

Use of `ISZ=1` on an **OPEN** provides direct access to a file on a byte-by-byte basis. Whatever is specified on the **IND=** clause identifies the actual byte within the file. The **SIZ=** clause can be used on the **READ** or **WRITE** to define the amount of data to transfer.

Example:

```
0010 OPEN (1,ISZ=100) "CUSTDB"
0020 FOR I = 0 TO 49
0030 READ RECORD (1,IND=I,ERR=0060) R$
0040 PRINT HTA(R$)
0050 NEXT I
0060 END
```

In the preceding example, the file `CUSTDB` is opened with an `ISZ=100`; i.e., the disk space occupied by the file is to be considered as a series of 100 byte records. The **FOR..NEXT** loop then reads the first 50 records (bytes 0 through 5000) and prints them in Hex via the **HTA( )** function. Record index zero (0) in the preceding example would consist of bytes 1 through 100. Index one (1) would be bytes 101 through 200, and so on.

One of the main reasons for accessing files via the **ISZ=** option is to read and/or update databases or other files maintained by other applications such as word processors, spread sheets, etc. Another reason may be to provide *direct access* to disk contents in order to correct a error which may have occurred on a file.

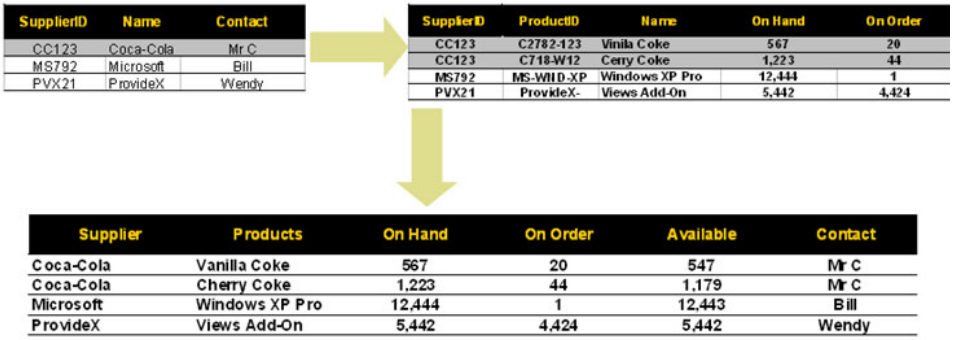
Warning: Use the **ISZ=** feature carefully, as there is no attempt by ProvideX to verify the data you are writing is correct for the type of file being updated. It is possible to accidentally corrupt data if you update the file with the wrong data.

Other options for storing and retrieving external data are discussed in *Chapter 10.* **Data Integration**.

Views System

ProvideX *Views* offer a more intuitive way to look at data and how it inter-relates. They give developers the ability to define logical representations of data sources that are relevant and more accessible to the end-user: i.e.,

- Provide only the data items you need to see.
- Assign more meaningful names to data items.
- Create virtual fields from exisiting data.
- Combine elements from different sources.



The ProvideX Views System provides a simplified interface that allows you to create a hierarchical view of the related tables and fields that is much easier for the end-user to understand. It includes built-in filters that let you define a dataset to retrieve just the records you need and to organize the data into meaningful groups.

Views can be accessed through the View object, the ProvideX **Report Writer**, and the **ODBC** driver. Use of this product may require a separately-purchased activation key apart from your initial ProvideX activation. Contact your local ProvideX dealer/distributor or visit www.pvx.com for product information and licensing.

For complete documentation, refer to the ProvideX *Views* manual.

6
Graphical User Interfaces

The *graphical user interface* (GUI) of your application may be one of its most crucial aspects (at least from the user's perspective) and ProvideX has the all tools necessary to adapt and incorporate powerful, effective GUI functionality. This includes the development of panels (windows or forms), menus, toolbars, buttons, radio buttons, checkboxes, list boxes, and scrollbars—just to name a few. It also has the flexibility to take your GUI application beyond the Windows environment.

This chapter leads you through the basics of GUI programming in ProvideX, the creation of a GUI window, the creation of the graphical components to be used within that window.

Topics

Background

A GUI application is designed so that the user is able to interact with the software in a manner similar to the physical manipulations in the real world. For example, in a typical "windowing" operating system, files are represented by a file icons (tiny pictures with a descriptive label). Data in a file can be moved to a new location by simply moving the icon via the mouse pointer. Behind-the-scenes, this physical interaction is translated into commands sent automatically to the application.

In ProvideX, the reusable GUI tools (menus, buttons, toolbars, check buttons, text entry boxes, and so on) are called **Graphical Controls**, *p.134*. In order to respond to user input, a program needs to lay out various controls within the application window, and set functions to be called when the user performs actions like selecting a menu item or clicking on a button.

Concepts and Terminology

GUI development is just like other types of programming, except for the following:

First, GUIs are almost exclusively *event driven* by nature, which means they perform tasks in response to events. A GUI spends most of the time in an idle state waiting for the operating system to send an event that can arrive in the form of a user action (clicking a mouse) or an operation invoked by the OS itself (screen refresh). For more on this topic, see **Event-Driven Methodology**, *p.131*.

Second, most modern GUIs are designed and built within an *IDE* (integrated development environment). While it is quite feasible to build all your GUI objects programatically in the ProvideX language (see **Syntax Elements**, *p.135*), most developers find it easier to design and implement their GUI applications using an application that is itself GUI-based; i.e., in **NOMADS**, *p.212*.

This section discusses some of the general concepts in ProvideX GUI development. How to create and implement specific GUI components will be covered later in the chapter.

Topics

GUI Terminology

The following terms are used in the context of ProvideX GUI development.

Control — A control is a graphical object used for "controlling" the application in a GUI environment. Controls have properties and generate events. Typical controls include buttons, list boxes, grids, menus, scrollbars, folders, etc. See **Graphical Controls**, *p.134*.

CTL Value — When controls are created, they are assigned a unique CTL identifier. (*ctl_id*). This value is used by a GUI program to determine what actions have been performed by the user or operating system. See **CTL Values**, *p.136*

Dialogue — Typically, a dialogue box is an independent (popup) window object that is used to request information from the user or to supply information the user may need. See **Window Categories**, *p.138*

Event — In GUI programming, an event is the reporting of an action generated by the user (or the GUI operating system itself) to which a program might respond. Examples of events include a mouse click, keystroke, focus change. See **Event-Driven Methodology**, *p.131*.

Mnemonic ProvideX syntax element used to control an application window in a in GUI application. See **Graphical Mnemonics**, *p.135*.

NOMADS Completely GUI-based development environment that simplifies the building and implementation of graphical applications in ProvideX. See **NOMADS**, *p.212*.

Panel In ProvideX, a panel is the primary display area that is under the control of an application at run time. It provides the layout for controls required by the user to interact with an application. Some GUI environments refer to this as the ***application window***. See **Window Categories**, *p.138*.

Property A property is a named attribute of a graphical control object in ProvideX. Each control may be referenced and modified dynamically using it's assigned CTL value followed by the apostrophe operator (tick) and property name. See **Dynamic Control Properties**, *p.137*.

Window A window is the generic term for a rectangular display object that presents its contents (e.g., controls, information, images, etc.) seemingly independent of the rest of the GUI operating system. See **Interface Windows**, *p.138*.

Event-Driven Methodology

This is one of the most significant concepts in GUI programming. Whereas a typical batch program runs all of its operations in a linear fashion from start to finish, an interactive user interface must be designed to run its operations in a fairly *random order*. This behavior is expected by users and is nearly impossible to implement without being "event driven".

The general criteria for designing a GUI application in ProvideX can be broken down into the distinct implementation stages, as outlined below.

Separate the User Interface from the Data Processing. There are several advantages to partitioning an application into GUI and non-GUI components – one being that it is much easier to maintain platform independence when building a distributed / server-based system. For more information on this topic, refer to the *ProvideX Client-Server Reference*. GUI generators (such as the NOMADS toolkit) can help accomplish this.

Determine User Actions. How will the interface allow the user to carry out these actions? Which steps are required by the user to exercise all the necessary functionality?

Determine Events. Determine which events in your code will need to be triggered when each action is taken. A typical GUI is input / output intensive. Each event may be generated by input from the keyboard, the mouse and system devices (e.g., menus, buttons, and scroll bars) and can occur in any order and at any time.

Organize Events. For more complex behavior, it may be useful to map GUI events visually using a ***state-transition*** diagram or table. This would illustrate the various states of an object, the events that cause a transition from one state to another, and the actions that result from a transition.

Associate Events with GUI Controls. Focus for (mouse and keyboard) input events is usually associated with the top window in a GUI desktop. Overlapping windows are hierarchically ordered. Events generated by child windows are frequently delivered up the window hierarchy for handling. See **CTL Values**, *p.136*.

The events can be dispatched in several ways. They can be asynchronously sent to controls as messages. Controls or logic can poll queues or devices. Controls may register a *callback*, a pointer to a function for handling each event – the window manager invokes the functionality whenever the specified event type occurs.

Note: The activities described above emphasize the event-driven aspects of GUI development. Designing a GUI tends to be an incremental process. However, in practice, these steps may proceed concurrently and in a different order.

General Design Principles

Although the functionality of your application is important, the way in which it delivers that functionality may be more important to your users. ProvideX includes several tools for performing similar GUI tasks, some may be better suited to your user's requirements than others. But how do you make the right usability choices in your design? The sections below outline some basic guidelines for designing a usable graphical user interface.

Know Your Users. Developing any user interface requires careful thought about how your users intend to use it. One excellent way to verify your design choices is to test your interface with potential users.

Ensure Consistency. Whether they are arbitrary, precisely task-oriented, or follow an OS standard, the "rules" behind a graphical element should be readily perceived by the intended user. Put your buttons in consistent places on all your windows, use the same wording in labels and messages, and use a consistent color scheme throughout.

Provide Clear Labeling. Assist navigation by providing good textual clues on or near each graphical control. The interface may be "graphical", but most users recognize words faster than they recognize icons, especially first-timers. Terminology should be defined so that the same term always has the same meaning anywhere in the application.

Use Feature Layering and Navigation Mechanisms. Crowded panels can overwhelm users with extraneous details. Most people would prefer to see only the important features exposed at any time, saving advanced or peripheral functionality until it is actually needed. Provide clear and easy routes between the different panels users will need to for accessing particular tasks. By convention, certain graphical components are designed specifically for feature layering; i.e., menus, toolbars, tabs and dialogues.

Apply Redundancy. Offer alternate methods to perform common tasks. While it is important to build on a familiar look and feel, the GUI should not trap users into a single linear path to everything in the application. As users learn the steps within the larger process, the seemingly "helpful" aspects of your interface may actually work against its usability. By providing several access methods (menu items, toolbar icons, and quick key sequences) you allow users to control their own productivity.

Expect Mistakes, Allow for "Undo". Give users a clean way out of a confusing (possibly destructive) situation. When a GUI provides navigation that is easily reversible, new users will feel much more confident about exploring the capabilities of your application. Of course the best remedy is to also include an *undo* or *cancel* option. This way, if users reach a point where they've made some serious mistakes, they could simply go back to their earlier work.

User-Centered Design - Resources

The above guidelines are not 'hard-and-fast' rules and can apply to a variety of interface applications. There are plenty of good detailed resources out there on the principles of high-level interface design. Check the internet for titles specific to the design of usable interfaces; i.e., web applications, mobile devices, and traditional desktop GUIs.

GUI Development in ProvideX

Not only does ProvideX offer complete GUI functionality, it is easily adapted for building graphical components on top of existing (non-GUI) programs. ProvideX is flexible, in that it allows more than one approach to GUI development. You also have the option to create and position controls programmatically or to use the visual, interactive building blocks provided under **NOMADS**, *p.212*.

Depending on your situation, there are four approaches to the development and implementation of GUI-based applications in ProvideX:

1. Retrofit GUI functionality into an existing program by introducing graphical **Syntax Elements**, *p.135* (directives, properties, etc.). With this approach, the onus is on the ProvideX developer to create the necessary control objects and trap events. The conditional logic will presumably already be in the existing code, so this portion of the development has been completed.
2. Construct your GUI application from the start using all the necessary graphical **Syntax Elements**, events, etc.
3. Employ the **NOMADS** toolkit to create and implement the desktop GUI. This development environment provides a fast visually-oriented framework for creating panels, populating them with GUI controls, and then associating event-driven code with these controls (pointing to logic in the already written program).

4. Develop your GUI at the file or database level. Using **NOMADS**, you can define the database or files, apply all the controlling data to each field, and then use screen generation facilities (**Dictionary-Based Development**) to automatically create parts of the application based on these file specifications.

Windowing Environment

Coding a GUI application begins with the implementation of an **Interface Window**. This is the interactive display area that is under the control of your ProvideX GUI application when running in a graphical operating system; i.e., MS Windows. It also defines the layout for the controls required by the user to interact with your application. ProvideX uses special GUI mnemonics for the creation and management of windows and dialogues. Multiple windows may be opened and closed at runtime.

Graphical Controls

These are the objects that can be manipulated by the user within the window. They provide different methods for displaying information, inputting data, and handling event processing within an application window. ProvideX supports a full range of **Graphical Directives** for creating and maintaining various **Control Objects**, *p.150*. Interactive properties are identified via **CTL Values**, which generate specific events in your application at runtime. Various **Control Options** allow you to define the initial appearance and functionality of each control when it is first created. Once they are created, controls may be further modified via **Dynamic Control Properties**, *p.137*.

Graphical Objects

Other graphical object types are used to produce images and text, or define the layout of your application. These are output on the graphic plane using **Graphical Mnemonics** via the **PRINT** directive, and have no events associated with them.

Object Focus

A key concept in ProvideX GUI programming is the handling of *focus*, the condition where a window or control object has the exclusive ability to receive input via keyboard or mouse actions.

One simple example of focus is when an input-capable field shows that it has the cursor – this means that the field *holds focus* and is ready to receive the next input from the keyboard. In a windowing environment, several windows may be visible at the same time but usually only one has focus to accept and display user input. The focused window is usually positioned on the top of the stack, overlapping other non-focused windows. (However, **NOMADS** can be set to allow *multiple active windows*.)

There are several ways that focus can be enabled in ProvideX; i.e.,

- Upon creation of an control object or window.
- When an object is selected by the user (mouse or keyboard) at runtime.
- **SET_FOCUS** directive (forcing transfer to a control object).

• **'GOTO' or 'WG'** mnemonic (forcing transfer to a window).

Focus is also used in event handling; i.e., certain (get focus, lose focus) events can be associated with GUI objects to generate a CTL value for triggering application logic.

Portable User Interfaces

ProvideX supports Windows, Java, and browser access to ProvideX GUI applications running on UNIX, Linux, Mac OS X, or Windows. ProvideX syntax elements for GUI development were originally designed for a Windows-only environment and were added to the language back when there were few other options. However, with the availability of ProvideX products such as WindX, JavX and UltraFX, it is now feasible to build a high-quality GUI in ProvideX that is truly OS (and device) independent. For a discussion on the full range of ProvideX multi-platform options, see *Chapter 8.* **Client-Server**.

Syntax Elements

The ProvideX language includes a variety of commands for creating GUI windows, controls and other display components – the basic building blocks used in the creation of a GUI application. This section provides an overview of the syntax elements (**Graphical Directives**, **Graphical Mnemonics**, **CTL Values**, **Control Options**, and **Dynamic Control Properties**) used to develop GUI applications at the *language-level*. If you prefer a graphical approach to GUI development, where the underlying code is generated for you automatically, look into **NOMADS**, *p.212*.

Graphical Directives

ProvideX supports a full range of directives for creating and maintaining various control objects in a GUI application: **BUTTON**, **CHART**, **CHECK_BOX**, **DROP_BOX**, **GRID**, **LIST_BOX**, **MULTI_LINE**, **RADIO_BUTTON**, **TRISTATE_BOX**, **VARDROP_BOX**, **VARLIST_BOX**, **V_SCROLLBAR**, and **H_SCROLLBAR**. For descriptions and examples on the use of these types of directives, see **Control Objects**, *p.150*.

Other GUI directives include **GET_FILE_BOX,** for generating a standard files / directory selection box, and **MSGBOX**, for launching a popup message box. These are discussed in the section **Special Function Windows**, *p.146*.

Graphical Mnemonics

Generating non-interactive graphical output from a ProvideX application requires the use of mnemonics, such as **'TEXT'**, **'FONT'**, **'PICTURE'**, **'ARC'**, **'PIE'**, **'CIRCLE'**, **'LINE'**, **'POLYGON'**, **'RECTANGLE'**, and **'IMAGE'**. Mnemonics are inserted within a **PRINT** statement to draw these types of components on the graphical plane. For descriptions and examples, see **Display Objects**, *p.201*

CTL Values

When each graphical control object is created, it is assigned a unique CTL value (*ctl_id*) which is returned when an action has been performed on the control. Your program processes this value to discover what actions have been performed by the user. CTL values are also used to identify the control for assigning **Dynamic Control Properties**, *p.137*.

CTL values can signal various events, including:

- Button pushed
- Menu / tool bar items selected
- Scroll bars adjusted
- List / drop boxes selected, edited.

These events are sent to the program to deal with one at a time. Eventually the user will perform an action that terminates the program. For further information, see **Event-Driven Methodology**, *p.131*, and **Submitting Input (CTL Values)**, *p.92*.

Control Options

Several options (*ctrlopt*) are available for use in GUI directives to specify format type, customize appearance, and define the behavior of graphical objects. These are outlined below. For information on how these are used, see **Control Objects**, *p.150*.

ERR=*stmtref*	Defines program line number / line label for on-error transfer. Can be used with any directive.
HLP=*string$*	Help message identifier for defining ***AutoComplete*** and ***Calendar*** functionality in a **Multi-Line** control.
FNT=*"font,size[,attr]"*	Sets font name, size, and optional attributes for control object directives that support text.
FMT=*def$\|mask$*	Specifies the format definition (*def$*) in directives used to create different styles of **Chart**, **Grid**, and **List Box** controls. Used for character string *mask$* in a **Multi-Line**.
KEY=*char$*	Defines hot key character in a **Drop Box**, **List Box**, **Multi-Line**, **Variable Drop Box**, and **Variable List Box**.
LEN=*num*	Defines maximum number of input characters allowed in a **Multi-Line**, **Variable Drop Box**, and **Variable List Box**.
MSG=*text$*	Supplies the text that is to appear on the message line when the control object has focus.
MNU=*ctl*	Assigns *ctl* value to be associated with a right-click **Popup Menu** event for when the mouse pointer is positioned over the GUI control object.
NUL=*string$*	Defines the null field display value in a **Multi-Line** control.
OWN=*name$*	Name assigned to a control for automated testing purposes. This will be visible to programs that use the Microsoft Active Accessibility (MSAA) interface. Refer to documentation on ProvideX *GUI Testing Automation*.

OPT=*char$*	Applies single character attribute / behaviour settings. Some characters may be combined; e.g., the combination `OPT="VUTf"` on a button creates an HTML-style hotspot.
SEP=*char$*	Sets the input line or column separator for **Chart**, **Grid**, **List Box**, and **Multi-Line** controls. Hex or ASCII value; e.g., `SEP=":"` or `SEP=$3A$ ...`
TBL=*char$*	Table of single character values to represent the displayed options. The 1st character represents the first entry, etc....
TIP=*text$*	Provides text to be used as the floating tip message when the mouse pointer hovers over the control.

Dynamic Control Properties

GUI control objects have a variety of properties associated with them. The apostrophe operator (***tick***) allows ***dynamic*** access to the property names used to define these attributes for a given GUI control. Any numeric variable containing the CTL number associated with a control can be used with the apostrophe operator. This feature allows you to redesign a control object dynamically in your application by assigning or resetting its properties / attributes.

For example, a button's location, size, text, and colour would be represented by properties named **Col**, **Line**, **Text$**, **TextColour$**, etc. You could change the text of a button control (called `MyButton`) as follows:

```
MyButton.ctl'Text$ = "Hit me now"
```

Other common property names include:

'Col	Starting Column
'Line	Starting line
'Cols	Width of control
'Lines	Height of control
'Tip$	Tip message
'Msg$	Message line
'Fmt$	Format mask
'BackColour$	Background colour
'Enabled	Enabled State
'Lock	Locked State
'Value$	Current value / state of control.

To obtain a complete list of properties for a given control, read the logical attribute *****; e.g.,

```
X=100
PRINT X'*
```

This results in a list of the attributes for the control defined with a CTL value of 100.

There is virtually no end to what can be changed on the fly within your controls. Most control commands can be changed by accessing attributes. For more information on dynamic properties and the apostrophe operator, see **Control Object Properties** in the *Language Reference, p.699*.

Warning: NOMADS stores the current value of a control in a variable with the same name as the control. If you change a control value using the **'Value$** property, you must also change the control's variable to keep NOMADS in sync.

Interface Windows

An interface window is an area that is under the control of your application when running in a graphical operating system. Within a typical "windowing" environment, there can be a variety of window types, each displaying certain characteristics with a unique purpose for interacting with your application. The ProvideX syntax for implementing the different window formats is covered in this section.

There are as many ways to invoke a ProvideX GUI application as there are windowing formats. One key rule to remember about creating a ProvideX GUI, is that ***ProvideX must be running*** in order to execute the application – that includes the primary application window (whether inside or outside the ProvideX console).

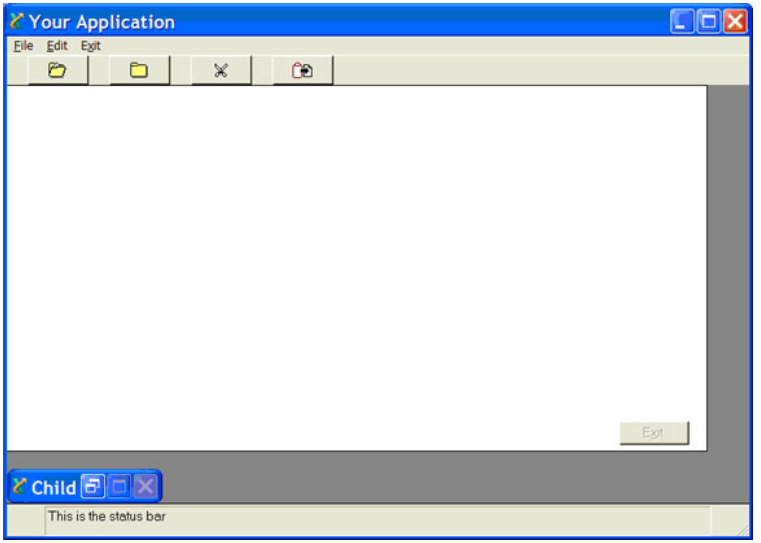
Note: For more information on creating the objects that make up the contents of an interface window, see **Control Objects**, *p.150*, and **Display Objects**, *p.201*.

Window Categories

In this documentation, the terms panel, dialogue, and form may be used to identify the functionally of different interface windows. While each windowing environment has its own rules and terminology, ProvideX window types generally fall within the categories described in the sections below.

ProvideX Console Window

As described in *Chapter 1.* **Getting Started**, the ProvideX environment under MS Windows, is itself invoked within a GUI window. Some applications may be designed to appropriate this window directly (the ProvideX console populated with various control objects) as the primary window for the application. The NOMADS toolset is itself an example of how an application would use the ProvideX console in this context.



The characteristics of the ProvideX console window are as follows:

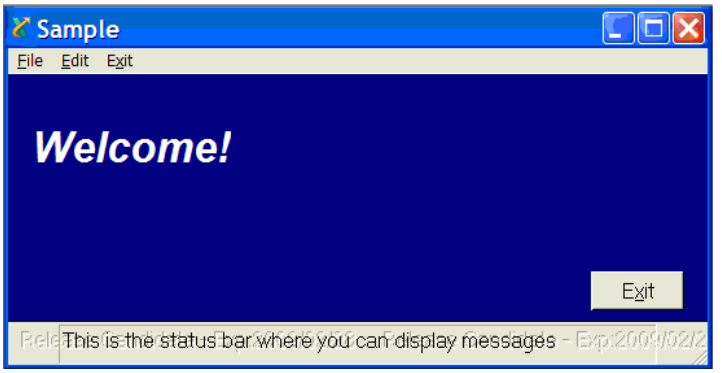
- Free-standing window that can be moved, resized, minimized, maximized and restored.
- Accessible workspace is not resizable after creation, leaving a gray area surrounding it when the window is enlarged.
- Title bar to display application name.
- Displays PVX icon at the top left corner of the window which controls session-related actions / characteristics.
- Supports optional menu, tool and status bars, including global menus and buttons.
- Supports creation of dependent windows in a parent-child relationship (See **Child Window** below).
- Has an icon in the Windows task bar.

Due to the static nature of the workspace area presented by the console window, or to take advantage of the more flexible display options offered by the dialogue window, many developers may prefer to launch a dialogue window for their main application window, either hiding the ProvideX console completely or using it only as a launch / login screen during the initial startup of their ProvideX application. (See **Dialogue Window**, below).

Note: Use the `-HD` command line option to hide the console at ProvideX startup. See **Launching ProvideX** in the *Installation and Configuration* guide To hide the console once your application is running, use the mnemonic `'SHOW'(-1)`.

Dialogue Window

A dialogue window is a free-standing independent window.



Following are the characteristics of a *dialogue* window:

- Free-standing window that can be moved and optionally resized, minimized, maximized and restored.
- Optional title bar to display application name.
- Accessible workspace area is resized when the window is resized.
- Optionally displays PVX icon at the top left corner of the window which controls session-related actions / characteristics.
- Supports optional menu and status bars.
- Does not support global menus and buttons.
- Supports the creation of dependent windows in a parent-child relationship (See Child Windows below).
- Entry in the Windows task bar.

Creating a Dialogue

A dialogue window is created by inserting the **'DIALOGUE'** mnemonic within a **PRINT** statement; e.g.,

```
PRINT 'DIALOGUE'(0,0,60,10,"Sample",'WHITE'+'_BLUE',OPT="-mMSXZ")
```

When creating a dialogue, the size and position of the window is defined via four coordinates. The first pair of numbers establishes the upper left starting position of the window (column and line coordinates). These are absolute values relative to the top left corner of the monitor screen. The third number indicates the window width (in columns) and the fourth is the window height (in lines). All are integer values.

Following the title you may also specify an optional default attribute string for the window. This string is comprised of one or more mnemonics, such as foreground / background colours.

The many options that are available in the **OPT=** clause make this a very flexible display window. The menu and status bars are optional, as are the icons and resizing boxes in the title bar. Even the title bar is optional. For details on the syntax and options for this mnemonic, see **'DIALOGUE'** in the *Language Reference, p.598*.

Example Dialogues

The following code creates the dialogue window shown on the previous page, with the title Sample:

```
PRINT 'DIALOGUE'(0,0,60,10,"Sample",'WHITE'+'_BLUE',OPT="-mMSXZ"), \
      'SR','CS',
PRINT 'MESSAGE'("This is the status bar where you can display messages"),
MENU_BAR 100,"-[&File,&Edit,E&xit],F:[&Open,&Close],E:[&Cut,&Paste]"
BUTTON 101,@(50,8,8,1.6)="E&xit"
PRINT 'FONT'("Arial,2,IB"),'TEXT'(@X(2),@Y(2),"Welcome!"),
```

Blue is set as the default foreground colour by specifying it as an attribute of the window. The options set in the OPT= clause include minimize and maximize buttons, menu and status bars, and will create a resizable window. By default, only one window is active at any time. ProvideX allows the creation of multiple active windows by specifying OPT="&". This creates a window that logically attaches to the current windows. See **Handling Multiple Windows**, *p.144*.

When a dialogue is created, the default scroll area leaves a column free on each side of the window and a line at the top and bottom. To reset the scroll region so that it extends to the edges of the window, use the **'SR'** mnemonic.

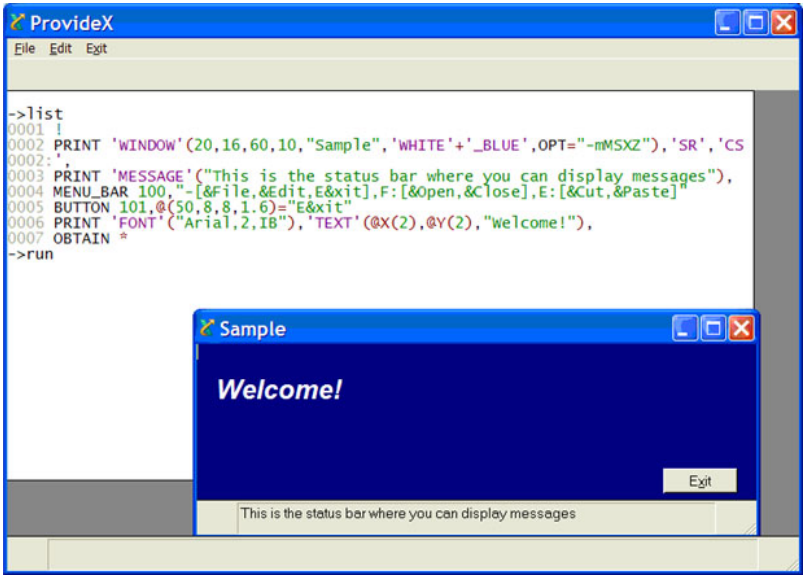
The following code centres a dialogue on the monitor screen:

```
panelWidth=40,panelHeight=10
x$=MSE ! use to determine max columns/rows on monitor
charWidth=DEC($00$+x$(10,1))
charHeight=DEC($00$+x$(11,1))
screenCols=INT(DEC(x$(27,2))/charWidth)
screenRows=INT(DEC(x$(29,2))/charHeight)
c=INT(screenCols/2-(panelWidth/2))
l=INT(screenRows/2-(panelHeight/2))
PRINT 'DIALOGUE'(c,l,panelWidth,panelHeight,"Title"),'SR','CS',
```

Child Window

A *child window* is a window that is launched and contained inside of a parent window. By definition, a child window is one that is dependent on a parent window and is minimized or closed when the parent minimizes or closes. These windows usually share the main application menu bar and can only be moved or resized within the parent window. If you attempt to move a child window beyond the

primary window frame, any part that is outside of the frame is no longer visible. A child can be any window launched by another window (including one that is itself a child window).



The characteristics of a *child* window are as follows:

- Totally contained within the parent window
- Minimized / restored when the parent window is minimize / restored
- Closed when the parent window is closed
- When minimized, they are identified as icons within the parent window
- Supports global menu and buttons created on the parent window
- Does not have an entry in the Windows task bar.

Creating a Child Window

A child window is created by inserting the **'WINDOW' or 'WA'** mnemonic within a **PRINT** statement; e.g.,

```
PRINT 'WINDOW'(0,2,60,10,"Sample",'WHITE'+'_BLUE',OPT="-mMSX")
```

When creating a child window, the size and position of the window is defined via four coordinates. The first pair of numbers establishes the upper left starting position of the window (column and line coordinates). These values are relative to the top left corner of the parent window. In the case of the console window, line 0 begins at the top of the tool bar area, so the line coordinate must compensate for this. The third number indicates the window width (in columns) and the fourth is the window height (in lines). All are integer values.

Following the title you may also specify an optional default attribute string for the window. This string is comprised of one or more mnemonics, such as foreground / background colours.

There are a variety of options for the **OPT=** clause, although fewer than for the **'DIALOGUE'** mnemonic. The status bar is optional, as are the icons and resizing boxes in the title bar. You must include a title when using the **'WINDOW'** mnemonic if you want a title bar and borders. When creating a child window to a **ProvideX Console Window** parent, the `OPT="c"` option is not necessary. This option must, however, be used to create a child launched from a **Dialogue Window** or another child window. For details on the syntax and options for this mnemonic, see **'WINDOW' or 'WA'** in the *Language Reference, p.645*.

Example Child Windows

The following code creates the child window shown on the previous page, with the title Sample:

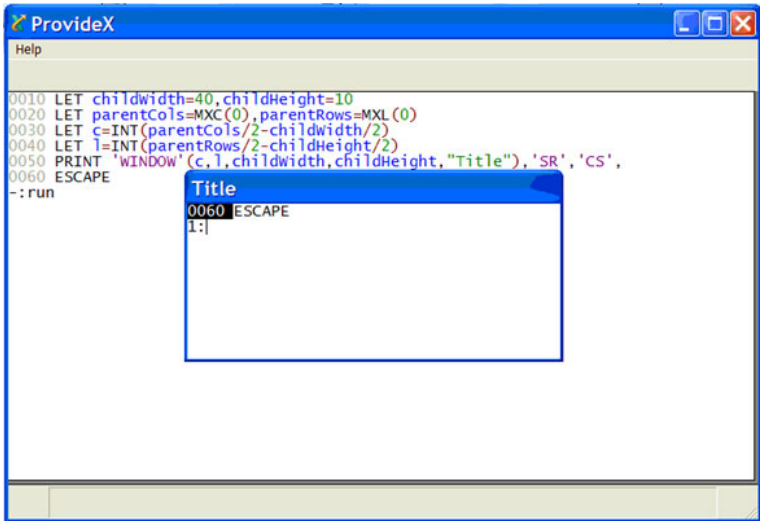
```
PRINT 'WINDOW'(20,16,60,10,"Sample",'WHITE'+'_BLUE',OPT="-mMSXZ"), \
      'SR','CS',
PRINT 'MESSAGE'("This is the status bar where you can display messages"),
MENU_BAR 100,"-[&File,&Edit,E&xit],F:[&Open,&Close],E:[&Cut,&Paste]"
BUTTON 101,@(50,8,8,1.6)="E&xit"
PRINT 'FONT'("Arial,2,IB"),'TEXT'(@X(2),@Y(2),"Welcome!"),
```

Blue is set as the default foreground colour by specifying it as an attribute of the window. The options set in the `OPT=` clause include minimize and maximize buttons the PVX system icon, menu and status bars, and create a resizable window.

When a child window is created, the default scroll area leaves a column free on each side of the window and a line at the top and bottom. To reset the scroll region so that it extends to the edges of the window, use the **'SR'** mnemonic.

The next example centres a child window in the workspace area of the parent window:

```
childWidth=40,childHeight=10
parentCols=MXC(0),parentRows=MXL(0)
c=INT(parentCols/2-childWidth/2)
l=INT(parentRows/2-childHeight/2)
PRINT 'WINDOW'(c,l,childWidth,childHeight,"Title"),'SR','CS',
```



The parent-child relationship may also be extended to a window created outside of the window that launched it. This is accomplished by creating the child window using the **'DIALOGUE'** mnemonic and specifying the `OPT="C"` option. While this type of window is not confined to the borders of its parent, it is nonetheless a child that is dependent on its parent, so the same rules apply.

Handling Multiple Windows

There are several options available for controlling how windows objects are arranged in your GUI application for viewing and input access.

Controlling Display

The **'SHOW'**(*n*) mnemonic can be used to minimize, maximize, and restore the display of specific windows programmatically. Code values for *n* include 0=Minimize current window, 1=Restore current window to normal display state, 2=Maximize current window, 3=Resize current window to previous display state, -1=Hide current window. The **'SIZE'** and **'MOVE'** mnemonics can also be used to control the size and location of the window.

Transferring Focus

If multiple windows / dialogues are to be displayed at the same time within your GUI application, there are two different methods available to you for switching focus from one window to another (without having to hide / drop one of the windows). These are described below.

Method 1. The ProvideX language handles window focus via the **'GOTO' or 'WG'** mnemonic.

Using the **'GOTO'** mnemonic requires the pre-requisite that a unique *window ID* be assigned to all windows upon creation; e.g.,

```
LET WindowNumOne=HWN(0)
PRINT 'DIALOGUE'(10,2,40,10,WindowNumOne,"NumberOne
      Window",OPT="cSX*"),'CS','SB',
LET WindowNumTwo=HWN(0)
PRINT 'DIALOGUE'(41,2,40,10,WindowNumTwo,"NumberTwo
      Window",OPT="cSX*"),'CS','SB',
```

In this example, **HWN( )** is used to supply the highest unused window number available. A **'GOTO'** would now be able to transfer focus to either of these windows by specifying one of the window IDs assigned; i.e., `WindowNumOne` or `WindowNumTwo`.

Method 2. The ProvideX ***WINAPI** utility can be used to switch focus between windows in your application. This method requires the pre-requisite that a unique *window title* be assigned to all windows upon creation. This is a two step process:

First, ***WINAPI** is used to retrieve the system ID (*handle*) associated with the title assigned to the window; e.g.,

```
Title$="WindowNumOne"
CALL "*WINAPI;FindWindowA",Title$,HANDLE
```

Once the handle has been established, you can use this value to position the selected window to the foreground (transferring focus); e.g.,

```
CALL "*WINAPI;SetForegroundWindow",HANDLE,RESULT
```

If multiple concurrent dialogue windows are displayed (using `OPT="&"`), then all the concurrent windows are active, and the user can move among them by clicking on the window.

Closing/Removing

Windows can be closed in two ways. The first is to use the **'POP' or 'WR'** mnemonic. This removes the currently-focused window from the top of the stack and restores the previously-focused window. Sometimes however, the window you want to remove is not the currently-focused window. To provide better control, the **'DROP' or 'WD'** mnemonics allows you to close a *specific* window by *window ID* reference.

Consider the following:

```
LET WindowNumOne=HWN(0);PRINT 'DIALOGUE'(10,2,40,10,WindowNumOne,"Special
      Number One Window",OPT="SX*"),'CS','SB',
LET WindowNumTwo=HWN(0);PRINT 'DIALOGUE'(41,2,40,10,WindowNumTwo,"Special
      Number Two Window",OPT="SX*"),'CS','SB',
```

To close the first window using **'POP'** would require two steps: Switch focus via `PRINT 'GOTO' (WindowNumOne)`, then `PRINT 'POP'`.

The **'DROP'** mnemonic handles this in one step: `PRINT 'DROP'(WindowNumOne)`. Good programming practice would be to actually do the following:

```
PRINT (0,ERR=*NEXT)'DROP'(WindowNumOne)
```

Special Function Windows

Two pre-formatted window types are also available to perform special functions in your GUI application. The directives discussed below are used for creating a dialogue box for selecting files and directories (**GET_FILE_BOX**) and a popup message box (**MSGBOX**).

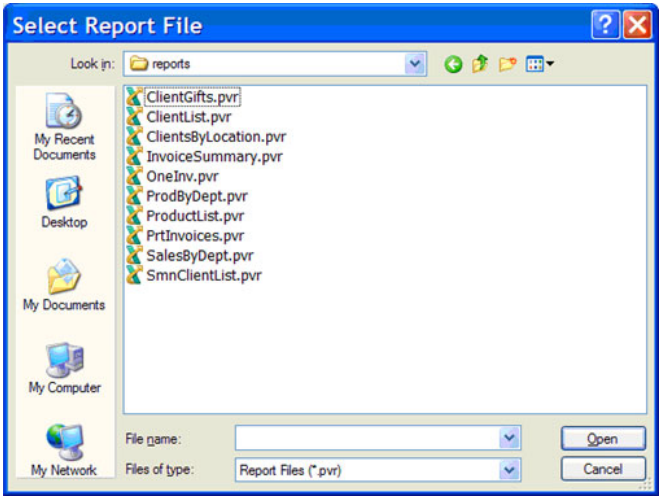
GET_FILE_BOX - File Selection Dialogue

The **GET_FILE_BOX** directive displays a standardized window to enter or select a file or directory on the system. On a Windows system, the standard Windows file selection dialogue is used, while WindX/JavX/UltraFX systems use a dialogue box created by a ProvideX utility program. There are different formats of the **GET_FILE_BOX** directive to accommodate selecting a file to **READ** or to **WRITE** or to select a **DIRECTORY**. For complete syntax details, refer to the **GET_FILE_BOX** directive in the *Language Reference, p.138*.

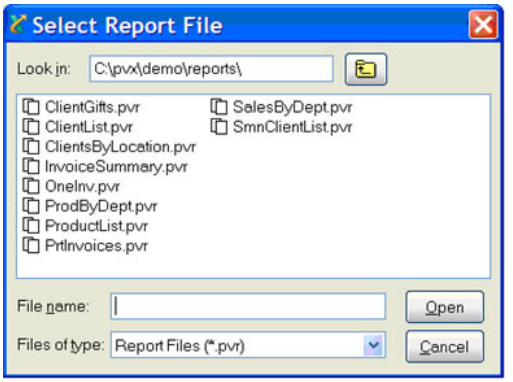
Examples:

```
GET_FILE_BOX READ PathName$,CurDir$,"Select Report File",
"Report Files (*.pvr)|*.pvr,All Files (*.*)|*.*","pvr"
```

On a Windows XP system, the above code would appear as follows:



In a WindX environment, the example would appear as follows:



MSGBOX - *PopUp Message Box*

The **MSGBOX** directive displays a window with a message in the middle of the screen. Various options are available as to which buttons, icons and behaviour combinations may to be applied to the window:

Buttons:

OK	Ok only
CANCEL	Ok, Cancel
RETRYCANCEL	Retry, Cancel
ABORT	Abort, Retry, and Ignore.
YESNO	Yes, No
YESNOCANCEL	Yes, No, Cancel
1 (or **2** or **3**)	Default button number

Icons:

STOP	Stop sign
INFO	Lowercase 'i' in a circle.
QUESTION or **?**	Question mark
EXCLAMATION or **!**	Exclamation Mark

Miscellaneous:

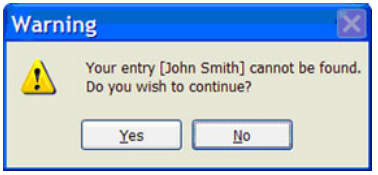
BEEP	Send associated sound
TIM=*num*	Maximum time-out value in integer seconds for closing automatically. The returned value is `"TIMEOUT"`.
TOP or **^**	Always on top (forces foreground window)

The user's message box button selection is returned in a string variable. Possible associated return values are **ABORT**, **CANCEL**, **IGNORE**, **NO**, **OK**, **RETRY**, or **YES** depending on the button selection required.

For syntax details, refer to the **MSGBOX** directive in the *Language Reference, p.212*.

Examples:

```
X$="John Smith"
MSGBOX "Your entry ["+X$+"] cannot be found."+SEP+"Do you wish to
      continue?", "Warning","YESNO,2,!",yesno$
PRINT X$
NO
```



In the above example, a **SEP** has been inserted in the message to break it to a second line. The Yes and No buttons have been specified (**YESNO**). The default set to 2. An **!** *exclamation* icon is displayed

```
MSGBOX "The report is completed","F.Y.I.","TIM=3"
```

In this example, the message box will self-destruct in 3 seconds.

Customizing the Message Box

It is possible to override the Windows standard message box and instead use a message box that you develop yourself. The **'MX'** system parameter will redirect to subprograms `*ext/msgbox.gui` (user-defined) or `*ext/system/msgbox.gui` (ProvideX-supplied) to process the request instead of the system message box.

By default, ProvideX sets the **'MX'** parameter to ***On*** when `*ext/msgbox.gui` is found to exist. The `msgbox.gui` subprogram creates and displays a message box that is virtually identical to the standard Windows system message box but will use XP-style (or Vista) buttons if the **'4D'** mnemonic is enabled. In addition, it will use the currently-selected windows graphic font.

Note: When **'MX'** is set, **MSGBOX** commands entered in console mode or executed within an **EXECUTE** command ***cannot*** be followed by any other command (as **MSGBOX** will be executing a **CALL** without a return address).

Several internal bitmap names for standard Windows bitmaps are available for displaying the embedded OS icons used by the normal message box API call: `!Sys_Stop`, `!Sys_Question`, `!Sys_Info`, and `!Sys_Exclamation`.

The button text ABORT, CANCEL, IGNORE, NO, OK, RETRY, or YES are included in the system message library file (i.e., `*mlfile.en`) to provide support for multi-lingual systems. Message numbers are defined as follows:

MSG( ) *Number*	*String*
160	"OK"
161	"OK,Cancel"
162	"&Retry,Cancel"

```
163             "&Abort,&Retry,&Ignore"
164             "&Yes,&No"
165             "&Yes,&No,&Cancel"
```

Use the **DEF MSG** directive to temporarily override the message text associated with the **MSG( )** number. This directive allows messages to be changed on the fly. For example, **MSG(164)** "&Yes,&No" can be changed to another language:
`DEF MSG(164) = "&Oui,&Non"`

Using Customized Messages with WindX

Use of this facility under **WindX** requires some additional effort by the developer; i.e., will the subprogram be running on the host or on the workstation. If the program runs on the host, it will transmit the screen drawing information to the workstation just like any other ProvideX program. If the program is to run on the workstation, the host will simply send the **MSGBOX** parameters to WindX, which in turn runs the program locally (assuming it is present).

The setup for WindX is described as follows:

- To run the host's `msgbox.gui`, set the **'MX'** system parameter. No change is required for workstation software.
- To run the workstation's `msgbox.gui`, ensure that the program exists on the workstation, then execute `[wdx]Set_param 'MX'` to set the parameter locally.

This takes advantage of the fact that ProvideX automatically sets **'MX'** based on the existence of a (user-defined) `*ext/msgbox.gui`. Simply copy the `msgbox.gui` from `*ext/system` to the `*ext` directory on the host, the system will use it and send screen drawing directives to the workstation. If it is copied (or installed) to `*ext` on the WindX workstation, the system will automatically use it, assuming it is not overridden by the host.

This customizable **MSGBOX** also takes advantage of the **'BEEP'** mnemonic.

Control Objects

Control objects are the graphical components within a panel that allow users to interact with the application. This section describes the different types of controls that can be added to a panel and how various attributes, settings, and logic may be applied to controls:

Topics

Descriptions and examples of the associated directives and other graphical **Syntax Elements** are provided in the sections that follow.

When control objects are manipulated, they can generate CTL values (control signal codes) into the input buffer to signal events that have occurred, such as receiving focus or changing values. **EOM** values (end-of-message strings) are also updated to indicate how an event occurred; i.e., via mouse-click or by pressing Enter.

The CTL value of each object is determined when the control object is defined. A program generally monitors the input queue for these values and triggers logic associated with the various events. The sample programs at the end of this chapter show how to build an event-driven application (see **Example Programs**, *p.208*).

When monitoring the input queue for CTL values, you can also monitor for negative CTL values that indicate certain keystrokes, mouse-clicks and other events such as resizing the panel. See **Negative CTL Definitions** in the *Language Reference, p.813*.

Note: When focus is on certain control objects, such as a list box or grid, many keystrokes (**UP-ARROW**, **DOWN-ARROW**, **PAGE UP**, **PAGE DOWN**, etc.) are utilized directly by the control object itself and are not available to the input queue.

Button

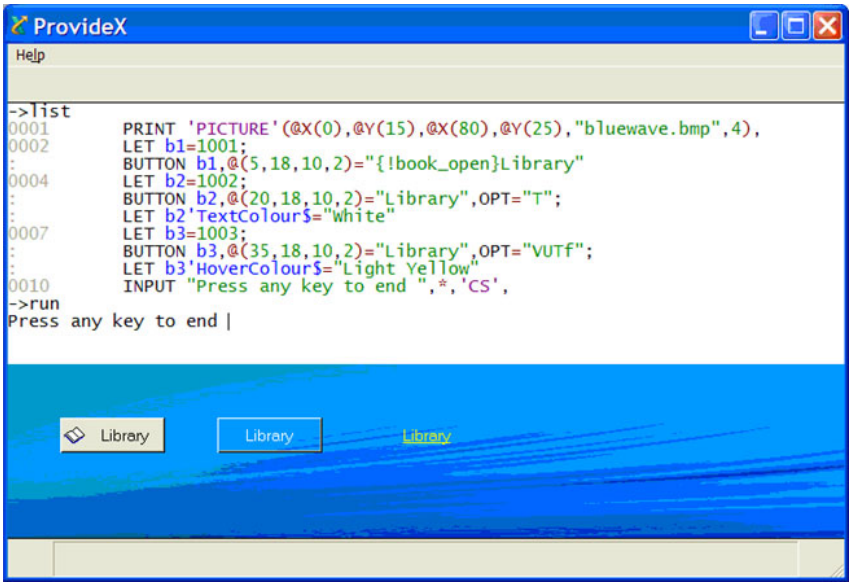
BUTTON [*****]*ctl_id*,**@(***col,ln,wth,ht***)**=*contents$*[,*ctrlopt*]
BUTTON {**REMOVE**|**DISABLE**|**ENABLE**|**ON**|**OFF**} [*]*ctl_id*
BUTTON {**HIDE**|**SHOW**|**GOTO**} [*****]*ctl_id*
BUTTON SET_FOCUS [*****]*ctl_id*,*ctl_val*
BUTTON READ [*****]*ctl_id,mode$*

When a user clicks a button in a graphical application, it is usually defined to send a signal (*ctl_id*) to the application to perform a specific action (see **CTL Values**). Several options are available for creating a button control in ProvideX.

For syntax details, refer to the **BUTTON** directive in the *Language Reference, p.34*.

Note: Other similar "button" behaviour may be defined using **Radio Buttons**, **Check Box**, or **Tristate Box** controls, which are described in the sections that follow.

The **BUTTON** directive can also be used to place an **Taskbar Notification Icon** in the bottom-right corner of the MS Windows desktop (a.k.a. the *System Tray*). See also, **Handling Images and Icons**, *p.398*.



Example:

```
PRINT 'PICTURE'(@X(0),@Y(15),@X(80),@Y(25),"bluewave.bmp",4),
B1=1001;
BUTTON B1,@(5,18,10,2)="{!book_open}Library"
B2=1002;
BUTTON B2,@(20,18,10,2)="Library",OPT="T";
B2'TEXTCOLOUR$="White"
B3=1003;
BUTTON B3,@(35,18,10,2)="Library",OPT="VUTf";
B3'HOVERCOLOUR$="Light Yellow"
INPUT "Press any key to end ",*,'CS',
```

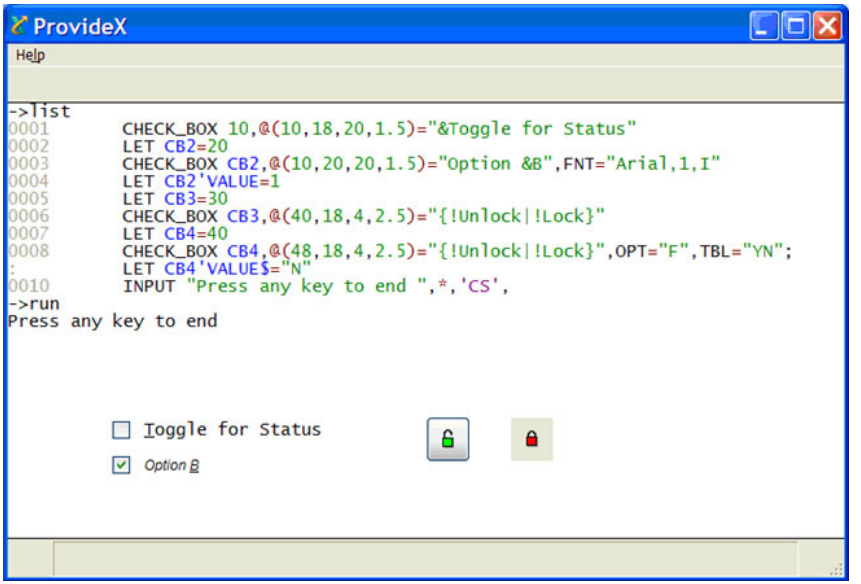
This program creates three buttons with control IDs of 1001 to 1003, 10 columns wide and 2 lines high. The first button contains a bitmap of an open book and the text library. The ProvideX installation includes a group of internal bitmaps available for use in control objects identified by a leading **!** *exclamation point* in the syntax. Braces indicate that the bitmap is to be placed on the button.

The second button is specified with the `'T'` option, making it transparent so that the image below shows through and the text colour has been changed to white using a dynamic property. The `"VUTf"` options are applied to the third button to make it transparent and flat with no border. It also has underlined text that changes colour to yellow when the mouse hovers over it.

Check Box

CHECK_BOX [*]*ctl_id*,**@(***col,ln,wth,ht***)**=*contents$*[,*ctrlopt*]
CHECK_BOX {**REMOVE**|**DISABLE**|**ENABLE**|**ON**|**OFF**}[*]*ctl_id*
CHECK_BOX {**GOTO**|**HIDE**|**SHOW**} [*]*ctl_id*
CHECK_BOX SET_FOCUS *ctl_id*,*ctl_val*
CHECK_BOX READ [*]*ctl_id,state$*[,*mode$*]
CHECK_BOX WRITE [*]*ctl_id,state$*

The check box is a button type control that allows users to toggle between states: ***On*** to select it, ***Off*** to de-select it. Check boxes may include a text label and graphics. For complete syntax, refer to the **CHECK_BOX** directive in the *Language Reference, p.47*. This directive can also be used to create a **Taskbar Notification Icon**, *p.197*.



Example:

```
CHECK_BOX 10,@(10,18,20,1.5)="&Toggle for Status"
CB2=20
CHECK_BOX CB2,@(10,20,20,1.5)="Option &B",FNT="Arial,1,I"
CB2'VALUE=1
CB3=30
CHECK_BOX CB3,@(40,18,4,2.5)="{!Unlock|!Lock}"
CB4=40
CHECK_BOX CB4,@(48,18,4,2.5)="{!Unlock|!Lock}",OPT="F",TBL="YN";
CB4'VALUE$="N"
INPUT "Press any key to end ",*,'CS',
```

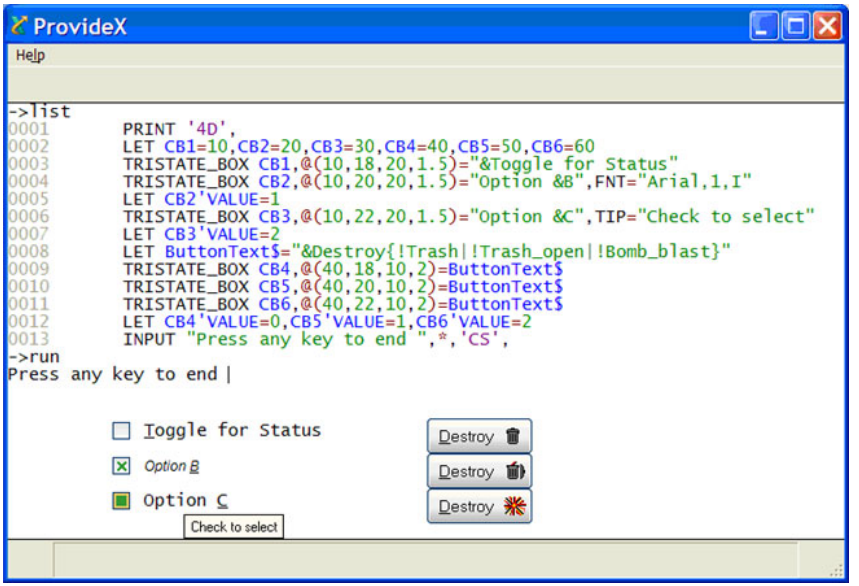
This program creates four check box controls. The first two are standard check boxes, in the ***off*** (default) and ***on*** states. Notice the use of the **&** *ampersand*. This makes the `T` in `Toggle` a *hot key* which when used in conjunction with the **Alt** key will transfer focus to this control. The last two check boxes contain images. When an image is included, the check box takes on the appearance of a button, in this case a regular

and a flat button. These check boxes are created with two images, one for each state. The last check box is also created with a translation table, "YN". This means that the ***off / on*** values are **Y** and **N** rather than 0 and 1.

Tristate Box

TRISTATE_BOX [*]*ctl_id*,**@(***col*,*ln*,*wth*,*ht***)**=*contents$*[,*ctrlopt*]
TRISTATE_BOX {**REMOVE**|**DISABLE**|**ENABLE**|**ON**|**OFF**} [*]*ctl_id*
TRISTATE_BOX {**GOTO**|**HIDE**|**SHOW**} [*]*ctl_id*
TRISTATE_BOX READ [*]*ctl_id*,*state$*
TRISTATE_BOX WRITE [*]*ctl_id*,*state$*

A tristate box is a type of **Check Box** that allows a ***third*** state to be activated; e.g., the third state may have the control greyed-out to indicate that the option is unavailable. Refer to the **TRISTATE_BOX** directive in the *Language Reference, p.340*. This directive can also be used to create a **Taskbar Notification Icon**, *p.197*.



Example:

```
CB1=10,CB2=20,CB3=30,CB4=40,CB5=50,CB6=60
TRISTATE_BOX CB1,@(10,18,20,1.5)="&Toggle for Status"
TRISTATE_BOX CB2,@(10,20,20,1.5)="Option &B",FNT="Arial,1,I"
CB2'VALUE=1
TRISTATE_BOX CB3,@(10,22,20,1.5)="Option &C",TIP="Check to select"
CB3'VALUE=2
ButtonText$="&Destroy{!Trash|!Trash_open|!Bomb_blast}"
TRISTATE_BOX CB4,@(40,18,10,2)=ButtonText$
TRISTATE_BOX CB5,@(40,20,10,2)=ButtonText$
TRISTATE_BOX CB6,@(40,22,10,2)=ButtonText$
CB4'VALUE=0,CB5'VALUE=1,CB6'VALUE=2
INPUT "Press any key to end ",*,'CS',
```

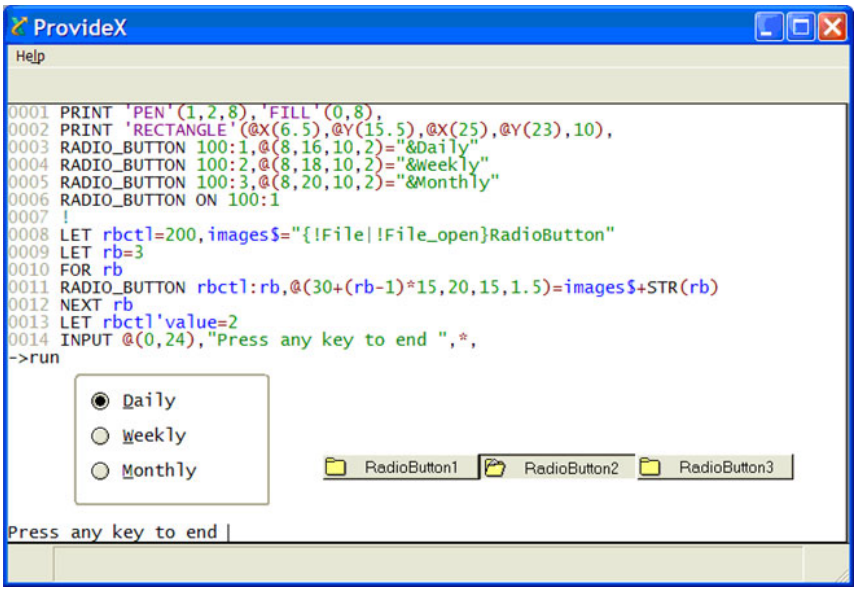
Radio Buttons

RADIO_BUTTON [*]*ctl_id:sub_id*,**@(***col,ln,wth,ht***)=***contents$*[,*ctrlopt*]
RADIO_BUTTON {**REMOVE**|**DISABLE**|**ENABLE**|**ON**|**OFF**} [*]*ctl_id:sub_id*
RADIO_BUTTON {**GOTO**|**HIDE**|**SHOW**} [*]*ctl_id:sub_id*
RADIO_BUTTON READ [*]*ctl_id,var,mode$*

These control types are laid out in a group of one or more related buttons, each representing one possible value for the variable the group represents. Only one button can be active at a time; i.e., when a user selects one of the radio buttons, that selection is activated ***on*** and all other related radio buttons are automatically set to ***off***. For syntax details, refer to the **RADIO_BUTTON** directive in the *Language Reference, p.261*.

Example:

```
PRINT 'PEN'(1,2,8),'FILL'(0,8),
PRINT 'RECTANGLE'(@X(6.5),@Y(15.5),@X(25),@Y(23),10),
RADIO_BUTTON 100:1,@(8,16,10,2)="&Daily"
RADIO_BUTTON 100:2,@(8,18,10,2)="&Weekly"
RADIO_BUTTON 100:3,@(8,20,10,2)="&Monthly"
RADIO_BUTTON ON 100:1
!
rbctl=200,images$="{!File|!File_open}RadioButton"
rb=3
FOR rb
RADIO_BUTTON rbctl:rb,@(30+(rb-1)*15,20,15,1.5)=images$+STR(rb)
NEXT rb
rbctl'value=2
INPUT @(0,24),"Press any key to end ",*,
```



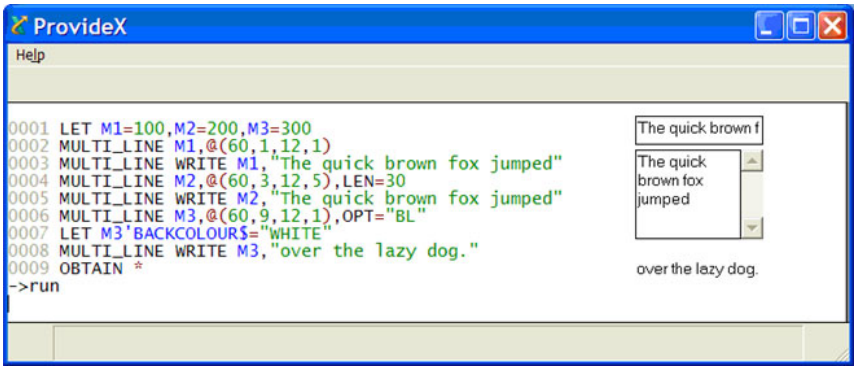
In the above example, the standard set of radio buttons (left side examples) appears as a series of of circular / radio-knob buttons, of which only one button is active at a time. When images are defined within the button content (right side examples) they appear as regular buttons, with the exception that when one is selected it remains depressed and the other related buttons are automatically de-selected.

When referencing a specifc radio button, the individual index must be used to identify it, as in `RADIO_BUTTON 100:1,@(8,16,10,2)="&Daily"`. When referencing specific radio buttons using **Dynamic Control Properties**, you must set the **'ID** property first to identify it, as in `rbctl'id=2,rbctl'text$="{!File}New text"`.

Multi-Line

MULTI_LINE *ctl_id*, **@(***col,ln,wth,ht***)**[,*ctrlopt*]
MULTI_LINE {**REMOVE**|**DISABLE**|**ENABLE**|**LOCK**|**UNLOCK**} *ctl_id*
MULTI_LINE {**GOTO**|**HIDE**|**SHOW**|**AUTO**} *ctl_id*
MULTI_LINE SET_FOCUS *ctl_id*,*ctl_val*
MULTI_LINE READ *ctl_id*,*var$*[,*mode$*]
MULTI_LINE WRITE *ctl_id*,*contents$*

A multi-line input area is used to enter and display text. If the multi-line input field is more than one line in height, ProvideX adds scrollbars and applies word wrapping. If the height of the multi-line is less than or equal to the height of a line of text, no scrollbars are drawn and the text will automatically scroll to the left as the input box is filled. Input lengths and formats, as well as **AutoComplete** and **Calendar** functionality, may be applied to multi-line controls. For syntax details, refer to the **MULTI_LINE** directive in the *Language Reference, p.214*.



Examples:

```
M1=100,M2=200,M3=300
MULTI_LINE M1,@(60,1,12,1)
MULTI_LINE WRITE M1,"The quick brown fox jumped"
MULTI_LINE M2,@(60,3,12,5),LEN=30
MULTI_LINE WRITE M2,"The quick brown fox jumped"
MULTI_LINE M3,@(60,9,12,1),OPT="BL"
M3'BACKCOLOUR$="WHITE"
```

```
MULTI_LINE WRITE M3,"over the lazy dog."
OBTAIN *
```

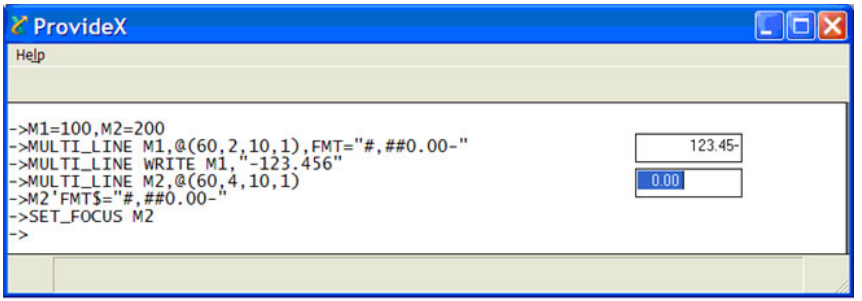
The examples show multi-lines that are a single line and several lines high. When a multi-line with a height of one is displayed, it is actually drawn with a height greater than 1 (one) to accommodate the height of the text plus white space above and below it. A third multi-line is also included that is locked and borderless. This type of multi-line is often used for display-only text that may need to be changed dynamically. Locked multi-lines are normally grayed-out, so the background colour may need to be changed to match the panel background.

MULTI_LINE WRITE is used to load a string into the input area. You can set the **'Value** property to do this as well.

Formatting Input

An input format can also be imposed on the multi-line using the **FMT=** clause or by setting the **'FMT$** property; e.g.,

```
M1=100,m2=200
MULTI_LINE M1,@(60,2,10,1),FMT="#,##0.00-"
MULTI_LINE WRITE M1,"-123.456"
MULTI_LINE M2,@(60,4,10,1)
M2'FMT$="#,##0.00-"
SET_FOCUS M2
```



For details on format masks, refer to **Data Format Masks** in the *Language Reference, p.809*.

AutoComplete

The ProvideX *AutoComplete* feature enables multi-line controls to show suggested text based on partially-typed entries in an attached dropbox. With this functionality turned on, a multi-line field can be set up to remember what a user has entered so that the next time they begin to type, it will display a list of previously entered words/ phrases that match what the user has typed thus far.

Note: **NOMADS** offers a utility for specifying auto-complete definitions that can be assigned to multi-lines on a panel. Refer to the *ProvideX NOMADS* manual.

Auto-complete is also useful for handling repetitive data entry. In this case, you can populate the data in advance so that the user will receive suggestions from a preset list of matches as they type in the field. The preset data may be retrieved from application data files accessed in read-only mode.

To set up a multi-line to use auto-complete functionality, a definition string must be built and specified using the **HLP=** clause when the multi-line is created, or using the **'AutoComplete$** property (see **Dynamic Control Properties**, *p.137*). The definition string consists of a number of parameter settings separated by semicolons:

AUTOPURGE=YES|NO
: Automatically purges expired records. Expired words or phrases are only purged when the multi-line is accessed. Default is **NO.**

DATAFILE=*path$*
: Name of keyed file that contains the words/phrases. This file should be resident and accessible on the local workstation.

EXPIRED=*num*
: Number of days a given record will be used before expiry. If this is not set or set to 0, the words/phrases do not expire.

FIELD=*num*
: Field that is being displayed.

KNO=*num*
: Key number to be used.

LENGTH=*num*
: Maximum number of characters that will be displayed.

OFFSET=*num*
: Starting position within the field to be displayed.

PREFIX=*string$*
: Prefix that will be used for searching matching words/phrases.

READONLY=YES|NO
: ProvideX does not automatically update the key file when user enters a new word and/or phrase. Default is **YES**.

When setting *AutoComplete* using the **HLP=** clause, the definition string must be prefixed with `"[AutoComplete]"`; e.g.,

```
Parameters$="DATAFILE=ac_clname.dat;KNO=1;FIELD=2;READONLY=YES"
MULTI_LINE ctl_id, @(10,2,15,1)),HLP="[AutoComplete]"+parameters$
```

When using the **'AutoComplete$** property, just specify the parameters; e.g.,

```
ctl_id'AutoComplete$="DATAFILE=ac_clname.dat;KNO=1;FIELD=2;READONLY=YES"
```

For multi-lines that are intended for remembering previous entries, a keyed file must be set up to to store the entries. Generally, a keyed file should be created by you for each set of multi-line entries required for your application; e.g.,

```
0010 LET F$="ac_clname.dat"; ERASE F$,ERR=*NEXT
0020 KEYED F$,[1:1:30:"C"],0,-40
```

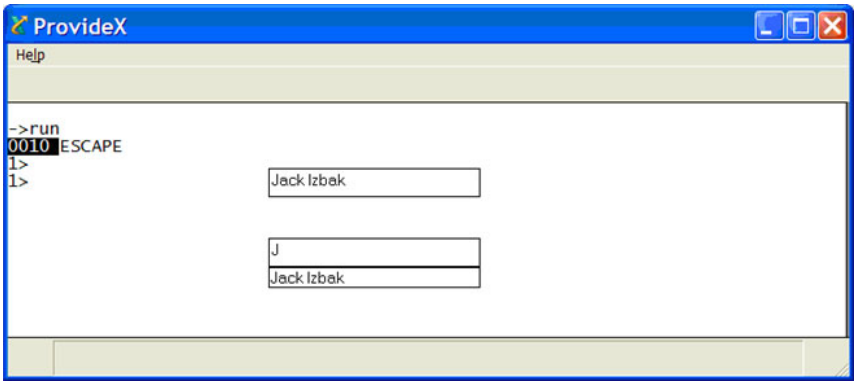
AutoComplete will be based on the internal key of the keyed file which must be case insensitive for this functionality to work correctly. If the key is case sensitive, all lowercase keys will be ignored. For information on creating keyed files in ProvideX, refer to the **KEYED** directive in the *Language Reference, p.165*.

Once the user enters new text in the multi-line, each word / phrase will be saved to the keyed file. When the user tries to type the same information again, they will be presented with a match from the associated keyed file before they finish typing.

For multi-lines that are intended for preset entries, the associated keyed file will already be populated with data. This could be an existing source data file used in your application. It is recommended that the **READONLY=** parameter be set to **YES** when defining this type of auto-complete scenario; otherwise, the existing data may become corrupted as ProvideX tries to update the file with new user entries.

Example:
In the following code, the auto-complete definition for `MultiLineA` is specified using the **MULTI_LINE** directive declaration and for `MultiLineB` it is specified using the **'AutoComplete$** property. They both use the same key file.

```
F$="AutoComplete.dat";
ERASE F$,ERR=*NEXT
KEYED F$,[1:1:30:"C"],0,-40
MULTILINEA=1000;
MULTILINEB=1001
AUTODEF$="Datafile=Autocomplete.dat;Readonly=NO"
MULTI_LINE MULTILINEA,@(25,4,20,1),HLP="[AutoComplete]"+AUTODEF$
MULTI_LINE MULTILINEB,@(25,8,20,1)
MULTILINEB'AUTOCOMPLETE$=AUTODEF$
ESCAPE
```



The result appears similar to the address edit box in a web browser. Initially, the key file is empty, so nothing will happen when the user types. When the user enters new text in the multi-line, each word / phrase will be saved in the keyed file. If the user tries to type the same word / phrase again, it will find a match from the list before they finish typing.

Client-Server Environment. In this scenario, the file used by the auto-complete logic to store and / or retrieve data must be on the client machine by default. If you wish to retrieve data from a *read only* file on the server, you must use additional program logic to accomplish this.

Set the '**AutoCTL** property of the **MULTI_LINE** with a *ctl_id* to be generated when content is needed for loading values in the auto-complete dropbox. The final character of the list is used as the list item delimiter. Assign the list to the **MULTI_LINE**'s **AutoValue$** property. This will cause the dropbox to be loaded for selection.

It is also possible to create a list with tab-separated display / return value pairs that displays the first item of the pair in the drop box, but uses the second item to load the multi-line when selected. For example, a list consisting of

```
"12345 - Barry's Bargain Bistro"+$09$+"12345"+$0A$+"12121 -
ABC Company"+$09$+"12121"+$0A"
```

... would display the following:

```
12345 - Barry's Bargain Bistro
12121 - ABC Company
```

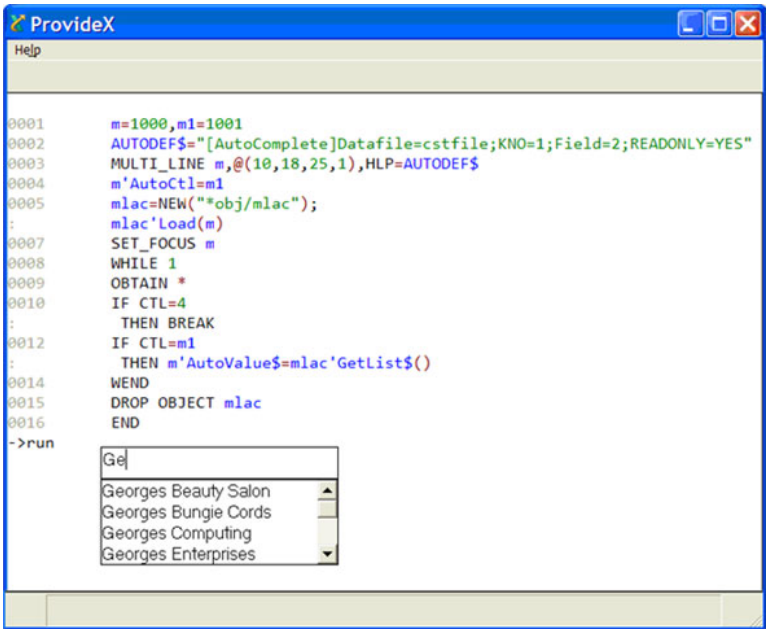
... but would load `12345` or `12121` into the multi-line depending on the selection.

Providex supplies an object class definition called `*obj/mlac.pvc` to build the list of entries for the drop box. The object class should be instantiated after the multi-line it services has been created. The class possesses two methods, a **Load(***ctl_id***)** method which uses the multi-line's assigned CTL value as an argument. Once loaded, the **GetList$( )** method can be called to create the contents of the auto-complete drop box associated with the multi-line.

The contents of the dropbox may be a simple list defined by the **FIELD=**, **OFFSET=** and **LENGTH=** parameters, and where the selected value is loaded back into the multi-line. It may also be a more complex combination of literals and fields defined using the **DISPLAY=** and **RETVAL=** parameters, which also allows for a different value to be loaded into the multi-line than the one displayed.

Example:

```
m=1000,m1=1001
AutoDef$="[AutoComplete]Datafile=cstfile;KNO=1;Field=2;READONLY=YES"
MULTI_LINE m,@(10,18,25,1),HLP=AutoDef$
m'AutoCtl=m1
mlac=NEW("*obj/mlac");
mlac'Load(m)
SET_FOCUS m
WHILE 1
OBTAIN *
IF CTL=4 \
  THEN BREAK
IF CTL=m1 \
  THEN m'AutoValue$=mlac'GetList$()
WEND
DROP OBJECT mlac
END
```



Calendar

The ProvideX *Calendar* feature provides a user-friendly way to enter date information into a multi-line input area. When applied to a multi-line, a button will be added to the control that can be clicked to invoke a *month-calendar* pop-up. This will allow users to pick a date to be inserted automatically into the multi-line input area. The format of the date inserted is based on the formatting rules of the **DTE( )** function.

To set up a multi-line to use *Calendar* functionality, a definition string must be built and specified using the **HLP=** clause when the multi-line is created, or by setting the **'Calendar$** property (see **Dynamic Control Properties**, *p.137*). The definition string consists of a number of parameter settings separated by semicolons:

CALENDAR=YES|NO
: **YES** turns on the calendar support. **NO** turns it off. Default is **NO.**

CONTENTS=*string$*
: Text or graph appearing on the button, default is {!DATE}.

DTE=*date$*
: Date formatting rules. Default is based on the **DTE( )**. *Semi-colon* cannot be part of this parameter (if used, the string following will be ignored). Date code should include **%** *percent*; however, if not used, input will be parsed based on format provided. If a time formatting string is included, the current time is used.

HEIGHT=*num*
: Height of button. Defaults to height defined for **MULTI_LINE**.

SHOWBUTTON=YES|NO
: **YES** shows the calendar button. **NO** hides it. Default is **YES.**

WIDTH=*num* Width of the button. Default width is equal to the height defined for the **MULTI_LINE**; i.e., the default size is a *square*.

When setting the ***Calendar*** feature using the **HLP=** clause, the definition string must be prefixed with `"[Calendar]"`; e.g,

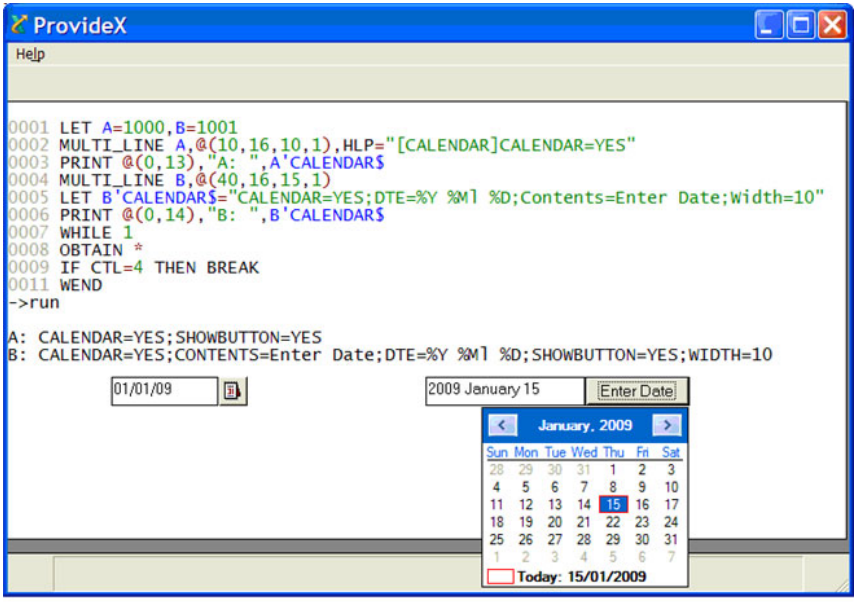
```
MULTI_LINE ctl_id, @(10,2,15,1)),HLP="[Calendar]CALENDAR=YES;DTE=%Y%M%D"
```

When using the **'AutoComplete$** property, just specify the parameters; e.g.,

```
ctl_id'Calendar$="CALENDAR=YES;DTE=%Y%M%D;CONTENTS={!Calendar}"
```

Example:

```
A=1000,B=1001
MULTI_LINE A,@(10,16,10,1),HLP="[CALENDAR]CALENDAR=YES"
PRINT @(0,13),"A: ",A'CALENDAR$
MULTI_LINE B,@(40,16,15,1)
B'CALENDAR$="CALENDAR=YES;DTE=%Y %Ml %D;Contents=Enter Date;Width=10"
PRINT @(0,14),"B: ",B'CALENDAR$
WHILE 1
OBTAIN *
IF CTL=4 \
  THEN BREAK
WEND
```



To invoke the calendar, the user can click on the calendar button, press SPACEBAR or ENTER when the calendar button has focus, or Shift - F2 when the multi-line or button has focus. For syntax details on the ProvideX *Calendar* feature, refer to the **MULTI_LINE** directive in the *Language Reference, p.214*.

Menu Bar

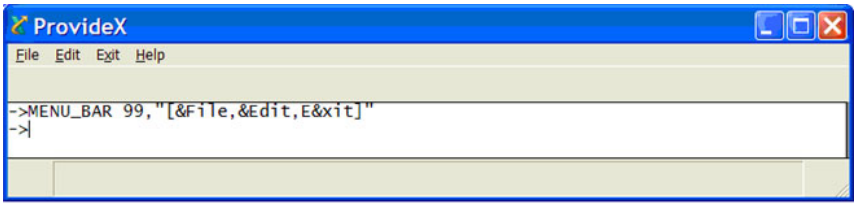
MENU_BAR *ctl_id*,*menu_def$*
MENU_BAR {**REMOVE**|**DISABLE**|**ENABLE**|**ON**|**OFF**} *element*[$]
MENU_BAR GOTO
MENU_BAR READ *var$*
MENU_BAR CLEAR
MENU_BAR RESET

Use the **MENU_BAR** directive to define and control the menu bar options across the top of a window / panel. Top level menu items, sub-menus, subordinate entries, hot keys, and menu images / icons can all be defined using this directive. For syntax details, see **MENU_BAR** in the *Language Reference, p.202*. For related functionality, refer to the **Popup Menu**, *p.165*.

Defining Menus and Sub-Menu Entries

The top level of the menu consists of a list of comma-separated entries enclosed in square brackets, each entry containing a unique selection character prefixed with an **&** *ampersand* used to identify it as a *hot key*; e.g.,

```
"[&File,&Edit,E&xit]".
```



The **MENU_BAR** directive adds the Help menu bar option by default. You can remove this option by specifying a dash immediately after the opening quote; e.g.,

```
MENU_BAR 99,"-[&File,&Edit,E&xit]
```

Sub-menus are identified by the hot key(s) used to open them, followed by a colon and a list of subordinate entries enclosed in square brackets; e.g.,

```
MENU_BAR 99,"-[&File,&Edit,E&xit],F:[&Open,&Close,&Print,E&xit]"
```

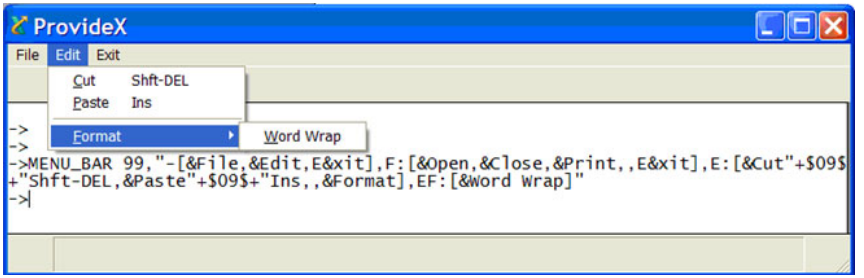
The text for subordinate entries may also contain tab characters (09) to separate additional item information, such as alternate hot keys. You can include lines to separate items by inserting an additional comma between items:

```
MENU_BAR 99,"-[&File,&Edit,,E&xit],F:[&Open,&Close,&Print,E&xit],
E:[&Cut"+$09$+"Shft-DEL,&Paste"+$09$+"Ins,&Format]"
```

Further sub-menus are identified by combining the higher level hot keys; e.g.,

```
MENU_BAR 99,"-[&File,&Edit,E&xit],F:[&Open,&Close,&Print,,E&xit],
```

```
E:[&Cut"+$09$+"Shft-DEL,&Paste"+$09$+"Ins,,&Format],EF:[&Word Wrap]"
```



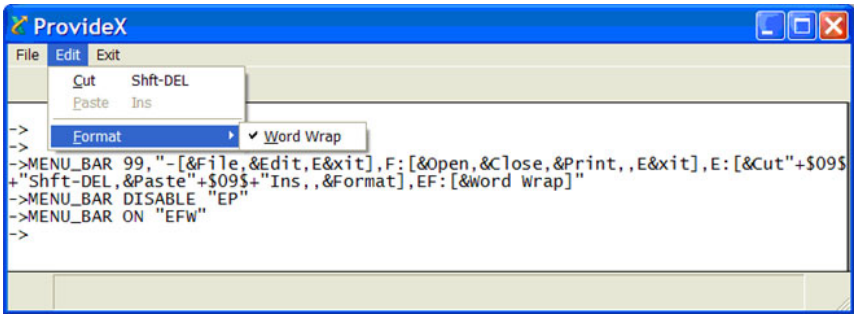
Normally, when a user selects any of the items from a menu bar, ProvideX generates the *ctl_id* you assigned to the menu bar. You can return the selected menu item code in a string variable using **MENU_BAR READ**. For example, if the user selected Word Wrap from the above menu, **MENU_BAR READ** would return EFW. It is also possible to assign specific CTL values to individual menu entries. To do this, append an equals sign and the CTL value to any item in the menu selection list; e.g., `E&xit=4`.

Changing Menu Appearance

Menu items can be disabled, displayed in bold, or show a checkmark, by placing a `"D"`, `"B"`, or `"C"` after the equal sign and before the assigned CTL value; e.g., `"[&One=1,&Two=BC2,&Three=D3]"`.

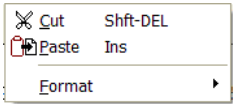
Items can also be enabled / disabled or checked / unchecked after the menu is created using **MENU_BAR DISABLE|ENABLE** and **MENU_BAR ON|OFF** directives; e.g.,

```
MENU_BAR DISABLE "EP"
MENU_BAR ON "EFW"
```



To include images with each item in the menu, enclose the image name in curly braces and place it in the menu definition just prior to the specific item text. Use a leading **!** *exclamation point* to identify the image as internal, or specify the relative path and filename to access an image file that is external. The first bitmap determines the dimensions used to display menu items (up to 64x64); e.g.,

```
E:[{!Cut}&Cut"+$09$+"Shft-DEL,{!Paste}&Paste"+$09$+"Ins,,&Format]
```

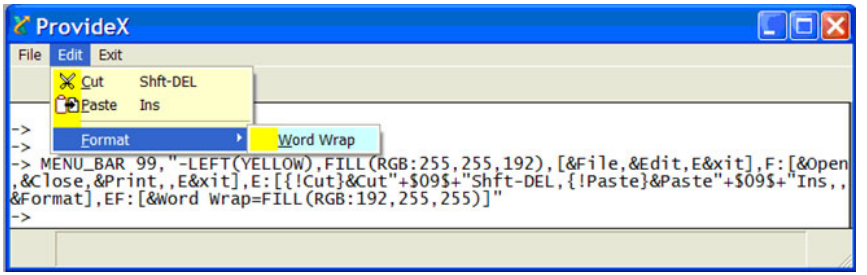


Transparency options are also available. `"T"` indicates use of the upper left most pixel colour, and `"G"` means use colour `RGB:192,192,192`.

Colours may be applied to the left edge of the menu where images are displayed, as well as to the text background area. (Colours do not apply to the top-level of the menu). The **LEFT(*colour*)** and **Fill(*colour*)** parameters are used to do this. The colour for the left edge applies to the entire menu and is specified prior to the menu items. The text area colour may apply to the entire menu if specified prior to the menu items, or to a specific item.

Example:

```
MENU_BAR 99,"-LEFT(YELLOW),FILL(RGB:255,255,192),[&File,&Edit,E&xit],
F:[&Open,&Close,&Print,,E&xit],
E:[{!Cut}&Cut"+$09$+"Shft-DEL,{!Paste}&Paste"+$09$+"Ins,,&Format],
EF:[&Word Wrap=FILL(RGB:192,255,255)]"
```



Processing Menu Bar Actions

To process a menu bar, you would trap its CTL value or the CTL value of an individual item. In the first case, you would then have to execute a **MENU_BAR READ** directive to determine which item was selected, then process as desired; e.g.,

```
PRINT 'CS',
MENU_BAR 99,"[&File,&Edit,E&xit=4],F:[&Open,&Close],E:[&Cut,&Paste]"
WHILE 1
OBTAIN *
IF CTL=4 \
   THEN BREAK
IF CTL=99 \
  THEN MENU_BAR READ X$;
       PRINT X$
WEND
END
```

Popup Menu

POPUP_MENU [**@(***col,ln***)**], *list$*, [*strvar$*|*numvar*]

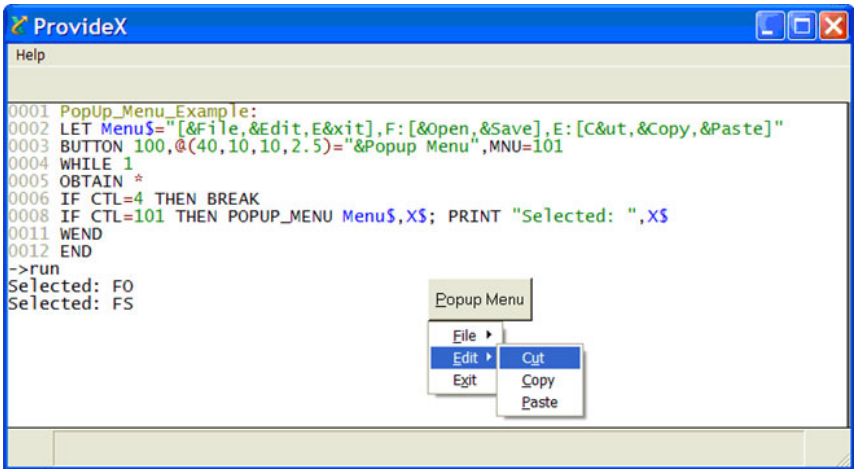
Popup menus are designed to "pop up" over the current window when you right-click on a selected control (**Button**, **Multi-Line**, **List Box**, etc.). Popups are similar to menu bars except for the fact that they can be placed anywhere on the panel and are only visible when invoked. The control objects that these popups are attached to are created using the **MNU=** option, which defines the CTL value that will be generated when the user right-clicks on the control.

The definition of a popup menu is similar to to that of a **Menu Bar** control. For syntax details, see **POPUP_MENU** in the *Language Reference, p.245*.

Example:

```
Menu$="[&File,&Edit,E&xit],F:[&Open,&Save],E:[C&ut,&Copy,&Paste]"
BUTTON 100,@(40,10,10,2.5)="&Popup Menu",MNU=101
WHILE 1
OBTAIN *
IF CTL=4 \
 THEN BREAK
IF CTL=101 \
 THEN POPUP_MENU Menu$,X$;
      PRINT "Selected: ",X$
WEND
END
```

Once a popup menu is created and assigned to a control, it remains invisible until the user right-clicks while the mouse is over the enabled component. In the example above, a CTL value of 101 will be triggered when you right-click on the associated button. At this point, the application will issue the **POPUP_MENU** directive; e.g.,



List Box

LIST_BOX *ctl_id*,**@(***col,ln,wth,ht***)**[,*ctrlopt*]
LIST_BOX {**REMOVE**|**DISABLE**|**ENABLE**} *ctl_id*
LIST_BOX {**GOTO**|**HIDE**|**SHOW**|**AUTO**} *ctl_id*
LIST_BOX SET_FOCUS *ctl_id,ctl_val*,
LIST_BOX LOAD *ctl_id,dlm_list$*
LIST_BOX LOAD *ctl_id,array_name$***{ALL}**
LIST_BOX LOAD *ctl_id,index,{element$ | *}*
LIST_BOX FIND *ctl_id,index,var$*
LIST_BOX READ *ctl_id,var$*[,*mode$*]
LIST_BOX READ *ctl_id,var*[,*mode$*]
LIST_BOX WRITE *ctl_id,element$*
LIST_BOX WRITE *ctl_id,index*
LIST_BOX WRITE *ctl_id*, ""

List box controls allow users to select items from a displayed list. Users can select *but not enter* values in a list box. Use a **Variable List Box** to implement a list box that allows both variable input as well as selection from a list. The **LIST_BOX** directive can be used to create and load several different **List Box Styles**.

List Box Styles

Standard list boxes contain a single column of data with no formatting. Other available list box styles are described as follows:

Formatted Displays multiple elements in different columns with alignment and width formatting, allowing colour mnemonics to be inserted into the data.

List View Lists a single element over multiple columns, where data wraps from the bottom of one column to the top of the next.

Report View Displays multiple elements in different columns (like a formatted list box) and allows column headings, sorting, bitmaps and other attributes.

Tree View Displays data grouped hierarchically into a collapsible tree-like structure which optionally may include + and - buttons to expand tree levels, dotted lines, etc. State indicators may be applied to ***Tree View*** list boxes (see **State Indicators**, *p.173*).

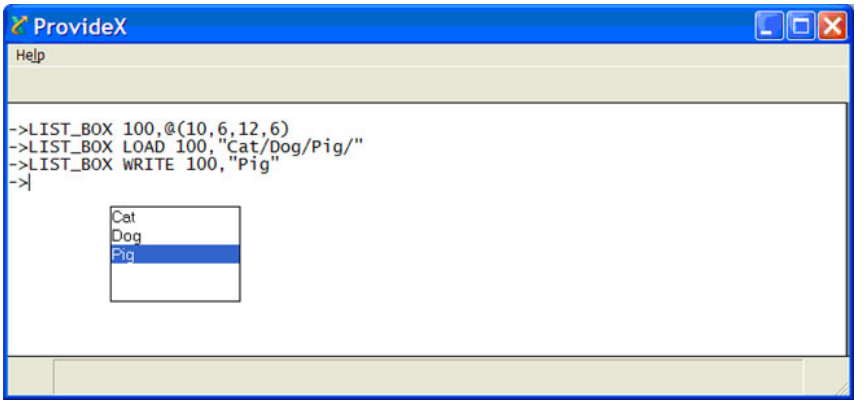
Alternate styles are defined using the *ctrlopt* settings **OPT=** and **FMT=**. For syntax details, refer to the **LIST_BOX** directive in the *Language Reference, p.178*.

Example 1 - Standard List Box

The following example displays a simple *Standard* list box with a single column of data and no formatting:

```
LIST_BOX 100,@(10,6,12,6)
LIST_BOX LOAD 100,"Cat/Dog/Pig/"
```

```
LIST_BOX WRITE 100,"Pig"
```



The **LIST_BOX LOAD** directive loads the list box in one step using a single variable that contains all the list items separated by delimiters. In this case, the "/" character is used to delimit the individual entries. (The delimiter is derived from the last character in the string and could be any character.)

LIST_BOX LOAD can also be used to clear a list box, e.g., `LIST_BOX LOAD 100,""`. Also, it can be used to remove an item; e.g., `LIST_BOX LOAD 100,n,*`. Where *n* specifies the index number (i.e., *sequential position*) of the entry to delete, and ***** signals the delete function.

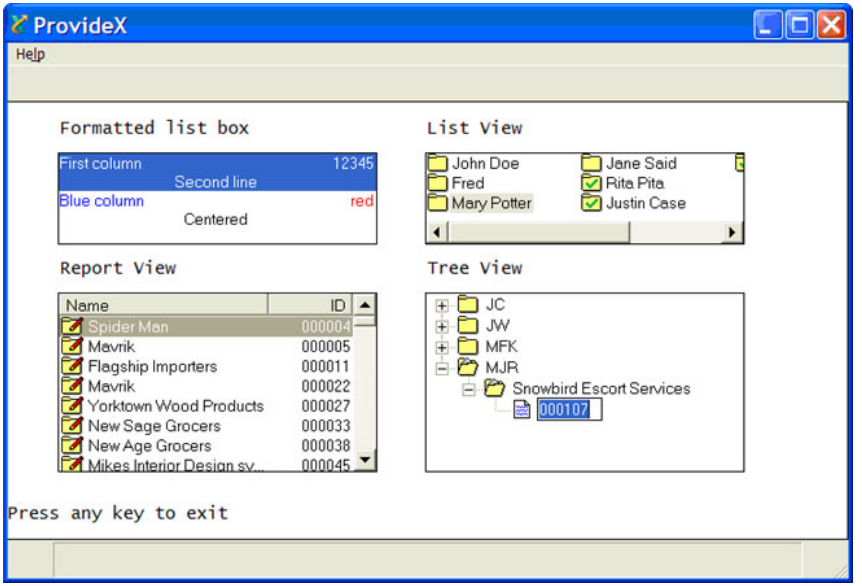
The **LIST_BOX WRITE** directive is used to select an entry, and can be set utilizing either the text of the entry or its index number.

Example 2 - List Box Styles

The next example displays four different types of list boxes: a ***formatted*** list box, using different colours and items displayed over multiple lines; a ***list view***, using images; a ***report view***, with multiple columns; and a ***tree view***.

```
PRINT 'CS',@(5,1),"Formatted list box"
LIST_BOX 100,@(5,3,30,5),OPT="~",FMT="L20 R10/C30",SEP="/"
LIST_BOX LOAD 100,0,"First column/12345/Second line"
LIST_BOX LOAD 100,0,'BLUE'+"Blue column/"+'RED'+"red/"+'BLACK'+ \
      "Centered"
LIST_BOX WRITE 100,"First column/12345/Second line"
!
PRINT @(40,1),"List View"
DIM Items$[1:7]
Items$[1]="!File,John Doe",Items$[2]="!File,Fred"
Items$[3]="!File,Mary Potter",Items$[4]="!File,Jane Said"
Items$[5]="!Done,Rita Pita",Items$[6]="!Done,Justin Case"
Items$[7]="!Done,Harry Aipe"
LIST_BOX 300,@(40,3,30,5),OPT="l",FMT="{}",SEP=","
LIST_BOX LOAD 300,Items${ALL}
```

```
LIST_BOX WRITE 300,"!File,Mary Potter"
!
PRINT @(5,9),"Report View"
LIST_BOX 200,@(5,11,30,10),OPT="rV",FMT="{} [Name]L20 [ID]R8"
SELECT cst_id$,cst_name$ FROM "cstfile"
LIST_BOX LOAD 200,0,"!file_edit"+SEP+cst_name$+SEP+cst_id$
NEXT RECORD
LIST_BOX WRITE 200,1
!
PRINT @(40,9),"Tree View"
LIST_BOX 400,@(40,11,30,10),OPT="e|!E",FMT="{!Page|!File|!File_open}"
SELECT cst_id$,cst_name$,*,*,*,cst_smn$ FROM "cstfile",KNO=2
item$=cst_smn$+SEP+cst_name$+SEP+cst_id$
LIST_BOX LOAD 400,0,item$
NEXT RECORD
LIST_BOX WRITE 400,item$
!
INPUT @(0,23),"Press any key to exit ",*,'CS',
END
```



For more information on how to create these list box styles, refer to the sections below. See also **Loading List Boxes**, *p.170*, **Load on Demand**, *p.171*, and **Selecting Items From a List Box**, *p.171*.

Formatted Style

The ***formatted*** list box allows you to display multiple data elements over multiple columns in a table format. This type of list box is created by adding **FMT=** settings to the **LIST_BOX** definition; e.g.,

```
LIST_BOX 100,@(5,3,30,5),OPT="~",FMT="L20 R10/C30",SEP="/"
```

The list of columns is a space-separated string enclosed in quotation marks. Each column is formatted with an alignment code for left, right or centre (L*n*, R*n*, C*n*). The width in the format definition is the display (window) width, not the number of characters in the text. Each new row is delineated by a */slash*. To hide data, use "S" to indicate that a column is to be skipped – data is present, but not displayed and the user cannot gain access to the column.

Details for creating a **Formatted** list box are provided under the **LIST_BOX** directive in the *Language Reference, p.187*.

List View Style

A ***list view*** is similar to a standard list box, but it displays the data as a continuous list over multiple columns. This type of list box is created by adding OPT="l" (lowercase L) to the **LIST_BOX** definition; e.g.,

```
0200 LIST_BOX 100,@(2,14,12,6),FNT="*",OPT="l"
```

Use **FMT=** to override the default column sizing of the list view (only "L*n*", "R*n*", and "C*n*" alignment are supported). A bitmap/icon may be placed to the left of the data element, by including {} *curly braces* in the **FMT=** string.

Details for creating a **List View** list box are provided under the **LIST_BOX** directive in the *Language Reference, p.189*.

Report View Style

A ***Report View*** displays multiple data elements in table form (similar to a **Formatted** list box), but it also includes headings, sorting, and other attributes. This type of list box is created by adding OPT="r" to the **LIST_BOX** definition; e.g.,

```
LIST_BOX 200,@(5,11,30,10),OPT="rV",FMT="{} [Name]L20 [ID]R8"
```

The report view in **Example 2 - List Box Styles** displays a bitmap at the beginning of each row/line and uses full-row highlighting. Other **OPT=** settings to refine a report view include "b"(suppress column header buttons), "p" (highlight partial matches) "q" (disable sorting), "v" (first column highlight), and "V" (full row highlight).

Use **FMT=** to define column alignment, titles, sorting, and bitmap placement. The format definition is similar to that described for **Formatted** list boxes; however, "L*n*", "R*n*", and "C*n*" are the only alignment options, and there are additional sorting options available for date and numeric sorting. A bitmap/icon may be placed to the left of the data element, by including {} *curly braces* in the **FMT=** string.

Details for creating a **Report View** list box are provided under the **LIST_BOX** directive in the *Language Reference, p.189*.

Tree View Style

A ***Tree View*** displays the data hierarchically in a tree-like structure. This type of list box is created by adding `OPT="e"` to the **LIST_BOX** definition; e.g.,

```
LIST_BOX 400,@(40,11,30,10),OPT="e|!",FMT="{!Page|!File|!File_open}"
```

Several other **OPT=** settings can be used in the definition: `"!"` (use bitmaps/icons), `"|"` (show connecting lines), `"b"` (suppress expand/collapse buttons); `"E"` (enable direct editing), `"q"` (disable sorting). Use **FMT=** to define default images to be used in the tree. The order of the images determines when they are used:

1. Default overall bitmap or icon: always used with any listed entries that do not have subordinates.
2. Default bitmap or icon for items with subordinates.
3. Default bitmap or icon for items with subordinates if the tree level is expanded (i.e., shown) in tree view.
4. Bitmap or icon for entries that do not have any subordinates when the item is selected.
5. Bitmap or icon for entries that have subordinates when selected.
6. Bitmap or icon for entries that have subordinates when selected and level is expanded.

Note: If images are set up for *individual elements* in tree view **LIST_BOX LOAD** and **WRITE** statements, these will ***override*** the default **FMT=** images for the individual element.

Details for creating a **Tree View** list box are provided under the **LIST_BOX** directive in the *Language Reference, p.192*.

Loading List Boxes

The code in **Example 2 - List Box Styles** shows different ways to load list boxes. There are examples of loading a list box one item at a time (***Formatted***, ***Report View***, and ***Tree View***), loading it all at once using an array (***Standard***), and building the entire list in a string variable and loading it all at once (***List View***).

Any method may be used with any type of list box. Loading the list box all at once is faster than loading one item at a time, but you must wait for the compilation to complete before anything is displayed. When loading one item at a time, the items are displayed immediately and the list box is accessible while it loads. Load time can be decreased by hiding the list box control while it is loading.

When loading ***Formatted*** and ***Report View*** list boxes, the values in the individual columns must be separated using a column delimiter. By default, the **SEP** value is used, but this can be changed by specifying a **SEP=** clause when creating the list box, or by setting the **'SEP$** property (see **Dynamic Control Properties**, *p.137*). If loading one item at a time, a line delimiter is not used, but if loading the entire contents via a single load, then the last character of the string value will be used as the line delimiter; e.g.,

```
LIST_BOX LOAD 200,0,"!file_edit"+SEP+cst_name$+SEP+cst_id$
```

When loading a ***Tree View***, all the levels of each entry must be specified, separated by delimiters as described above.

```
item$=cst_smn$+SEP+cst_name$+SEP+cst_id$
LIST_BOX LOAD 400,0,item$
```

Load on Demand

Another method is available for speeding up the load process for list boxes. On-demand loading allows an application to load a list box with only those items the user actually scrolls into view. This reduces network traffic and file access since a list box is only loaded with those items required by the user. Also, it assures proper function of the scrollbar and its relationship to the list.

The following properties are used to implement load-on-demand logic:

'Itemcount	Defines number of items.
'ItemNeededCtl	CTL issued when data needed.
'ItemNeededFrom	Lowest item needed.
'ItemNeededTo	Highest item needed.

This feature requires the developer to pre-declare the number of items that the list box is to have (by setting the **'ItemCount** property). When the user scrolls items into view, the system generates a CTL event.

Upon receiving the CTL event (set by **'ItemNeededCtl**), the application queries **'ItemNeededFrom** and **'ItemNeededTo** to determine the index number and the number of items. The application then loads the list box with the contents of the specified items by setting **'Item** and **'ItemText$**. If no elements are needed then **'ItemNeededFrom** and **'ItemNeededTo** will be zero. Once the value has been loaded into the **'ItemNeededTo**, ProvideX checks if further items are required and if so, it generates another CTL event.

In the case of a ***Report View*** list box, should the user request the list be sorted or attempt to auto-size the width of a column, the system will force a load of all list box elements before processing the request.

In some instances the contents of the list box may need to be shown prior to the contents being loaded, in which case the system will display 5 dots in place of the data.

Selecting Items From a List Box

To select an item from a list box, the user can use the mouse to double-click an item, or highlight the item if the auto-signal option is specified (`OPT="A"`). An item can also be selected by highlighting it and then moving focus to another control. To determine the value of the selection, the **LIST_BOX READ** directive can be used, or the control's **'VALUE$** property can be queried. The value that is returned consists of all the column entries including the column separators and image references. In the case of ***Formatted*** list boxes, embedded mnemonics such as colours are included as well. ***Tree Views*** return the item and its parent branches.

```
LIST_BOX 100,@(5,3,30,5),OPT="~",FMT="L20 R10/C30",SEP="/"
LIST_BOX LOAD 100,0,"First column/12345/Second line"
LIST_BOX LOAD 100,0,'BLUE'+"Blue column/"+'RED'+"red/"+'BLACK'+ \
      "Centered"
LIST_BOX WRITE 100,2
LIST_BOX READ 100,x$
PRINT x$
Blue column/red/Centered
Item$=STP(MNEMONIC x$)
PRINT Item$
Blue column/red/Centered
```

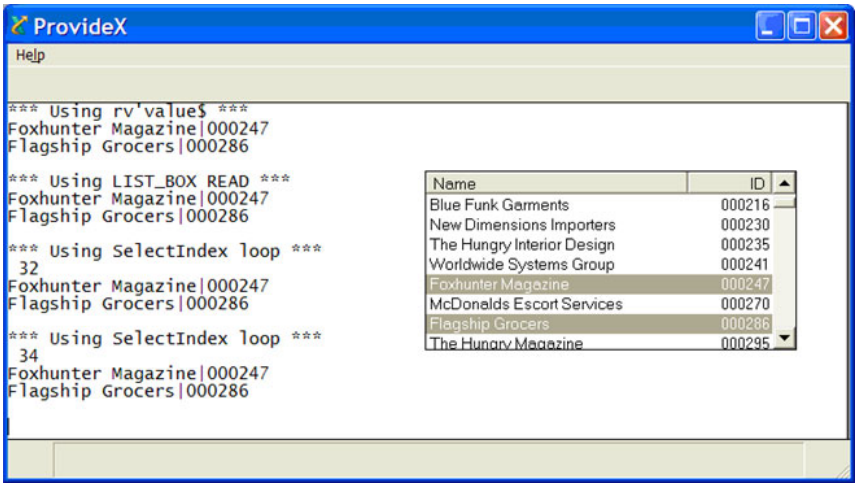
Standard, ***Formatted***, ***List View*** and ***Report View*** list boxes can support multiple item selection. This is enabled by specifying `OPT="#"`. If items were loaded in a single string, then when you read / write the element(s) highlighted in the list box, the item(s) will be returned in the variable using either the delimiter from the **LIST_BOX LOAD** statement or, as default delimiter, the **SEP** character.

To determine which items were selected, you can use the **LIST_BOX READ** directive or look at the **'VALUE** property. The resulting value will contain all the selected items which must then be parsed to access the individual items. Another way to access the selected items is to use the **'SelectCount**, **'SelectIndex** and **'SelectItem** properties to spin through the selections individually. **'SelectCount** contains the number of items / cells selected. (Set this property to zero to de-select all.) **'SelectIndex** is the index to point to a selected element; i.e., set to 1 to point at the first item selected, 2 to point at the second item selected, etc. After **'SelectIndex** has been set, then **'SelectItem** will contain the sequential location within the list of the item being pointed at by the **'SelectIndex** property.

Example:

```
rv=100
LIST_BOX rv,@(40,4,35,10),OPT="rV#",FMT="[Name]L25 [ID]R8",SEP="|"
SELECT cst_id$,cst_name$ FROM "cstfile"
LIST_BOX LOAD rv,0,cst_name$+"|"+cst_id$
NEXT RECORD
!
! Select several items (normally done by mouse clicks)
LIST_BOX WRITE rv,32
LIST_BOX WRITE rv,34
!
! Determine selections using 'value$ property
PRINT 'CS',"*** Using rv'value$ ***",'LF',rv'value$
!
! Determine selections using LIST_BOX READ directive
LIST_BOX READ rv,x$
PRINT "*** Using LIST_BOX READ ***",'LF',x$
!
! Determine selections using 'SelectCount/'SelectIndex properties
```

```
i=rv'SelectCount
FOR i
rv'SelectIndex=i
PRINT "*** Using SelectIndex loop ***",'LF',rv'SelectItem, \
      'LF',rv'value$
NEXT
!
OBTAIN *,
```



Tree Views also support multiple selections when ***state indicators*** are used. This is discussed in the next section.

State Indicators

State indicators are basically images that appear in front of a ***Tree View*** entry that can be used to indicate whether the item has been selected or not, or what state the item is in. State indicators are currently supported for ***Tree View*** list boxes only.

The following properties are used to create and process state indicators:

'ItemState	State of **'Item**.
'StateBitmaps$	List of images used to display states.
'AutoState	Control auto toggling of state.
'CascadeState	Control cascading of states.

Assigning Images. The application must set the **'StateBitmaps$** property in order to define the number of images that will used in the display of state indicators. A maximum of 15 images, separated by *pipe* character delimiters (|), can be assigned. All images must be of the same size/format and may specify transparency options. These images can be external or internal. The order of the images will correspond to the state values, and may include an additional image for use with cascading states; e.g.,

```
tv'StateBitmaps$="!EmptyBox|!CheckedBox|!HalfCheckedBox"
```

Toggling Between States. Once the bitmaps are set, each item / row / entry may set its 'ItemState property to determines what image is to appear next to the row text depending on the state. A maximum of 15 states can be assigned for each image. A state of 0 zero causes no state indicator to be displayed. For example, assuming the list box is defined with 3 images. The first image will appear if the item state is 1, the second image will appear if the item state is 2 and the third image will appear if the item state is 3.

A CTL event will return `EOM="S"` if the property is set to a non-zero value. This is used to identify that the user clicked over the indicator state portion of the line, as opposed to elsewhere in the item. Applications that add state indicators to their existing logic should add a check for this EOM code.

Auto Toggling Of States. **'AutoState** is a numeric property that controls auto toggling of states. If this property is set, state indicators can automatically be toggled without generation of a CTL event with `EOM="S"`.

The number of states that the system will toggle through is determined by the value set in this property or, if the property is set to 1, the number of bitmaps assigned to the tree view. In addition, when the user toggles a state indicator while holding down the Shift, all entries between the current entry and the last will be toggled to the new state of the current entry (in effect allowing for group select / deselect).

Cascading States. If the **'CascadeState** property is set to non-zero, the system automatically cascades parent states to their children and correspondingly makes parent states representative of all of their children. Setting a parent state, either under program control or using the **'AutoState** property in the tree view definition, will result in all subordinate children being set to the same state.

When a child state is set, its parent state will be set according to the state of all of the child's siblings; i.e., if all children are in a consistent state, the parent will be set to the same state. If a parent has children of various states (some on, some off), the parents state will be set to the value set in the **'CascadeState** property.

For example, you could have three state indicators - *Off* (state 1), *On* (state 2), and *Partial* (state 3). You would set **'AutoState** to 2 and **'CascadeState** to 3 to have children that automatically toggle off / on and parents that will be *On* if all children are on, *Off* if all children are off, and *Partial* (state 3) if the children are not in a consistent state.

When cascading, only items with states will be affected. In addition, items without states will not affect their parents states, nor will changing the parent of an item without a state affect the children of that item.

Loading the Tree View. When loading an item with a state indicator into the tree view, the item content must contain an image clause inside { } *curly braces.* This clause may contain an optional image reference to be displayed next to the state image. It must also contain a `;state_value` expression for setting the initial state of the item; e.g.,

```
LIST_BOX LOAD tree,0,"{!Stop;2}category$+sep+name$
LIST_BOX LOAD tree,0,"{;1}parent$+sep+child$
```

When the ***Tree View*** is loaded, the syntax for loading an item includes {optional_image;state_value}, allowing you to include an additional image to be displayed next to state image. The ;state_value is required for setting the initial state of the item.

Selection. In addition to the state indicator properties described earlier, ***Tree View*** listboxes support the following:

'SelectedChildren Number of child items

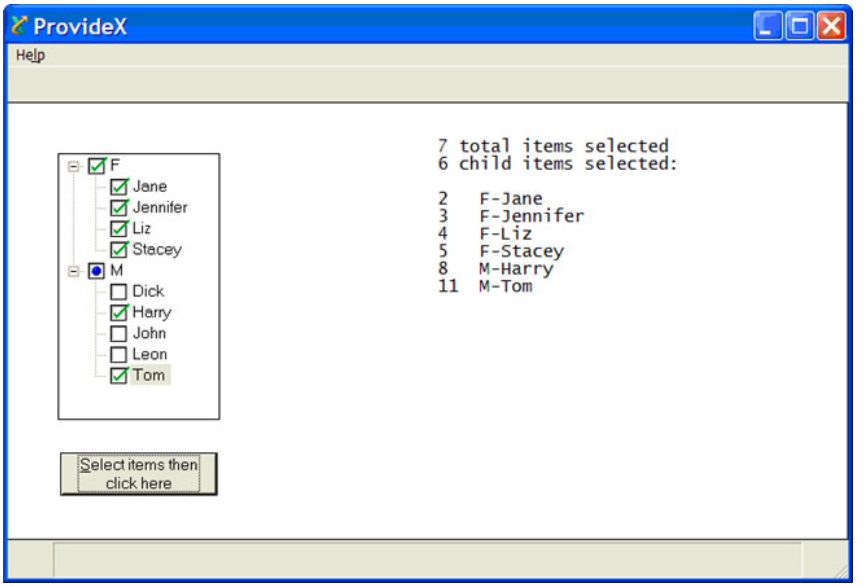
'SelectStateMask State filter to apply

Use **'SelectedChildren** in conjunction with **'SelectedStateMask** to return the number of child items with the desired state. When **'SelectedStateMask** is set, the **'SelectCount**, **'SelectIndex**, and **'SelectItem** properties will reflect only those items that have the specified state; e.g., to find all items that have a state of one, set **'SelectStateMask** to 1. **'SelectCount** will then indicate the number of items that have this state and sequencing through **'SelectIndex** and **'SelectItem** will return their item numbers.

Example:

```
! Tree view with state indicators
 PRINT 'CS'
 BUTTON 1000,@(5,20,15,2.5)="&Select items then click here"
 tree=2000
 LIST_BOX tree,@(5,3,15,15),OPT="e|A!"
 tree'StateBitmaps$="!EmptyBox|!CheckedBox|!DotInBox"
 tree'AutoState=2
 tree'CascadeState=3
 ! Load the treeview from the data statements
 WHILE 1
 READ DATA category$,name$,END=*BREAK
 LIST_BOX LOAD tree,0,"{;1}"+category$+SEP+name$
 WEND
 ! Selection loop - Select items and click 'OK'
 WHILE 1
 OBTAIN *
 IF CTL=4 \
  THEN BREAK
 IF CTL=1000 \
  THEN GOSUB Display_Selected
 WEND
 END
 !
 Display_Selected:
 tree'SelectStateMask=2
 PRINT 'LF',@(40),tree'SelectCount," total items selected",'LF',
 PRINT @(40),tree'SelectedChildren," child items selected:",'LF'
 n=tree'SelectCount
```

```
FOR n
tree'SelectIndex=n
LIST_BOX FIND tree,tree'SelectItem,item$
READ DATA FROM item$ TO category$,name$
IF name$<>"" \
 THEN PRINT @(40),tree'SelectItem,@(45),category$+"-"+name$
NEXT
RETURN
!
DATA "M","Tom","M","Dick","M","Harry","M","John","F","Jane"
DATA "F","Jennifer","M","Leon","F","Stacey","F","Liz"
```



This example creates a ***Tree View*** with state indicators. The `"!"` option is required to allow bitmaps, and, in this example, the `"A"` option allows selection with a single mouse click. The ***Tree View*** has two states for the child items (`'AutoState=2`), displaying either an `!EmptyBox` or `!CheckedBox` image. The third image, `!DotInBox`, is used by a parent item when its children have mixed states (`'CascadeState=3`).

When the ***Tree View*** is loaded, the state value for each item is initialized to 1. The selection loop allows the user to click on items, and clicking the button displays those selected by executing a loop based on the `'SelectCount` and `'SelectIndex` properties. Notice that the display logic filters out the parent-only entries and just displays the selected children.

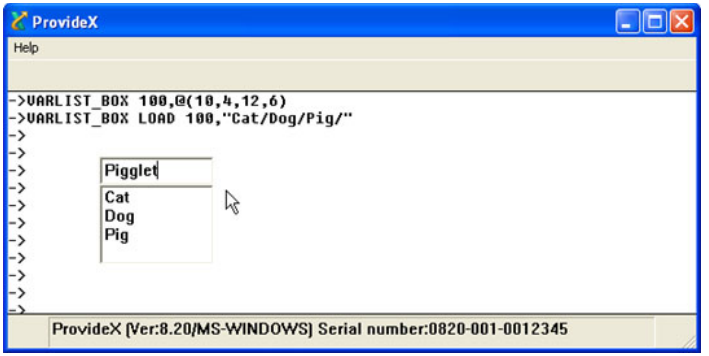
Variable List Box

VARLIST_BOX *ctl_id*,**@(***col*,*ln*,*wth*,*ht***)**[,*ctrlopt*]
VARLIST_BOX {**REMOVE**|**DISABLE**|**ENABLE**} *ctl_id*
VARLIST_BOX {**GOTO**|**HIDE**|**SHOW**|**AUTO**} *ctl_id*
VARLIST_BOX SET_FOCUS *ctl_id*, *ctl_val*
VARLIST_BOX LOAD *ctl_id*,*dlm_list$*
VARLIST_BOX LOAD *ctl_id*,*array_name$***{ALL}**
VARLIST_BOX LOAD *ctl_id*,*index*,*{element$* | ***}**
VARLIST_BOX FIND *ctl_id*,*index*,*var$*
VARLIST_BOX READ *ctl_id*,*var$*[,*mode$*]
VARLIST_BOX READ *ctl_id*,*var*[,*mode$*]
VARLIST_BOX WRITE *ctl_id*,*element$*
VARLIST_BOX WRITE *ctl_id*,*index*
VARLIST_BOX WRITE *ctl_id*,""

A *variable* list box is a ***Standard*** **List Box** that allows the user to select any element from the list of items and / or enter any other value. For syntax details, refer to the **VARLIST_BOX** directive in the *Language Reference, p.356*. For examples of how to process a ***Standard*** list box, see **List Box**, *p.166*.

Example:

```
VARLIST_BOX 100,@(10,4,12,6)
VARLIST_BOX LOAD 100,"Cat/Dog/Pig/"
```



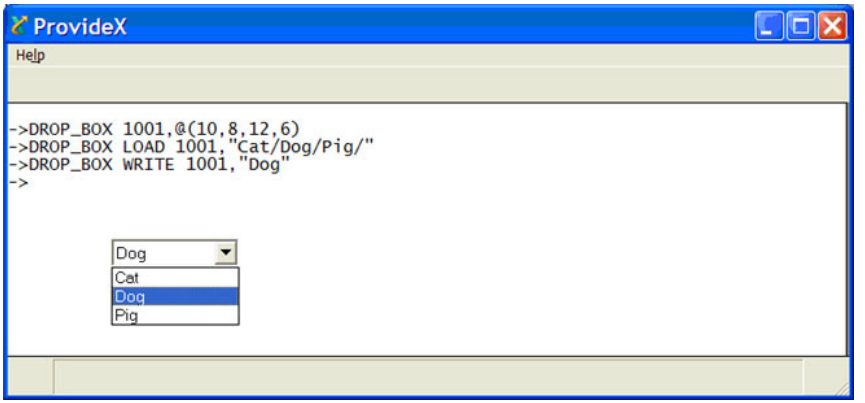
Drop Box

DROP_BOX *ctl_id*,**@(***col*,*ln*,*wth*,*ht***)**[,*ctrlopt*]
DROP_BOX {**REMOVE**|**DISABLE**|**ENABLE**|**ON**|**OFF**}*ctl_id*
DROP_BOX {**GOTO**|**HIDE**|**SHOW**|**AUTO**} *ctl_id*
DROP_BOX SET_FOCUS *ctl_id*,*ctl_val*
DROP_BOX LOAD *ctl_id*,*dlm_list$*
DROP_BOX LOAD *ctl_id*,*array_name$***{ALL}**
DROP_BOX LOAD *ctl_id*,*index*,{*element$* | ***}**
DROP_BOX FIND *ctl_id*,*index*,*var$*
DROP_BOX READ *ctl_id*,*var$*[,*mode$*]
DROP_BOX READ *ctl_id*,*var*[,*mode$*]
DROP_BOX WRITE *ctl_id*,*element$*
DROP_BOX WRITE *ctl_id*,*index*
DROP_BOX WRITE *ctl_id*, ""

A drop box control is similar to a ***Standard*** **List Box**, but the default state only displays a single line of text. The remaining list is made available by clicking the button. The user can select any element from a list of items you assign to the drop box, but variable input is not allowed. For syntax details, refer to the **DROP_BOX** directive in the *Language Reference, p.95*. For examples of how to process a ***Standard*** list box, see **List Box**, *p.166*.

Example:

```
DROP_BOX 1001,@(10,8,12,6)
DROP_BOX LOAD 1001,"Cat/Dog/Pig/"
DROP_BOX WRITE 1001,"Dog"
```



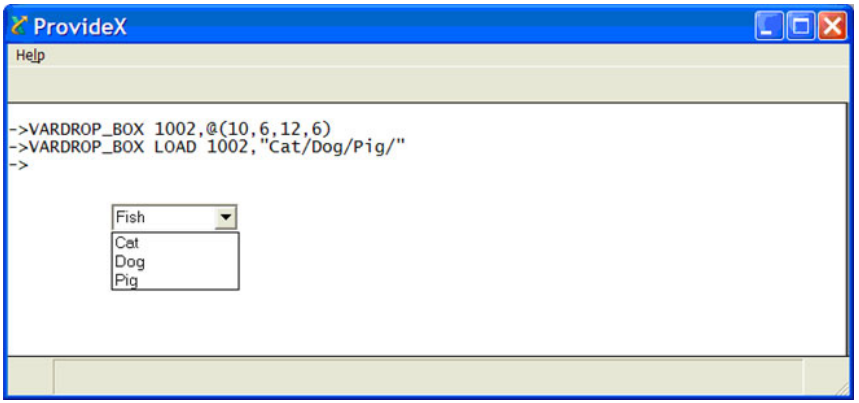
Variable Drop Box

VARDROP_BOX *ctl_id*,**@(***col*,*ln*,*wth*,*ht***)**[,*ctrlopt*]|
VARDROP_BOX {**REMOVE**|**DISABLE**|**ENABLE**} *ctl_id*
VARDROP_BOX {**GOTO**|**HIDE**|**SHOW**|**AUTO**} *ctl_id*
VARDROP_BOX SET_FOCUS *ctl_id*,*ctl_val*
VARDROP_BOX LOAD *ctl_id*,*dlm_list$*
VARDROP_BOX LOAD *ctl_id*,*array_name$***{ALL}**
VARDROP_BOX LOAD *ctl_id*,*index*,{*element$* | ***}
VARDROP_BOX FIND *ctl_id*,*index*,*var$*
VARDROP_BOX READ *ctl_id*,*var$*[,*mode$*]
VARDROP_BOX READ *ctl_id*,*var*[,*mode$*]
VARDROP_BOX WRITE *ctl_id*,*element$*
VARDROP_BOX WRITE *ctl_id*,*index*
VARDROP_BOX WRITE *ctl_id*, ""

A *variable* drop box normally displays a single line on the screen with a **DOWN-ARROW** on the right side and allows variable input. That is, the user can select any element from a list of items associated with the variable drop box or can enter *any other value*. To view the list, the user clicks on the **DOWN-ARROW.** For syntax details, refer to the **VARDROP_BOX** directive in the *Language Reference, p.350*. For examples of how to process a ***Standard*** list box, see **List Box**, *p.166*.

Example:

```
VARDROP_BOX 100,@(10,6,12,6)
VARDROP_BOX LOAD 100,"Cat/Dog/Pig/"
```



Grid

GRID *ctl_id*,**@(***col*,*ln*,*wth*,*ht***)**[,*ctrlopt*]
GRID {**REMOVE**|**DISABLE**|**ENABLE**|**LOCK**|**UNLOCK**} *ctl_id*
GRID {**GOTO**|**HIDE**|**SHOW**|**AUTO**} *ctl_id*
GRID SET_FOCUS *ctl_id*,*ctl_val*
GRID ADD *ctl_id*,*col*,*row*
GRID LOAD *ctl_id*,*col*,*row*,*contents$*
GRID DELETE *ctl_id*,*col*,*row*
GRID CLEAR *ctl_id*,*col*,*row*[,*width*,*height*]
GRID READ *ctl_id*,*col*,*row*,*var$*,*eom$*
GRID READ NEXT *ctl_id*,*col*,*row*,*var$*,*eom$*
GRID SELECT *ctl_id*,*col*,*row*[,*width*,*height*]
GRID SELECT READ *ctl_id*,*col*,*row*
GRID SELECT READ NEXT *ctl_id*,*col*,*row*
GRID SELECT RESET *ctl_id*
GRID FIND *ctl_id*,*col*,*row*,*var$*
GRID WRITE *ctl_id*,*col*,*row*,*contents$*

ProvideX includes the ability to create a highly-flexible and complex grid control. Grids provide a spreadsheet-style format – basically a two-dimensional array of editible cells in columns and rows that may comprise multi-line (default), check box, button, drop box, or any combination of formats (each cell can be a different type). For syntax details, refer to the **GRID** directive in the *Language Reference, p.142*. A more simplistic version of a grid style control can be created using a *Report View* style **List Box**, *p.166*.

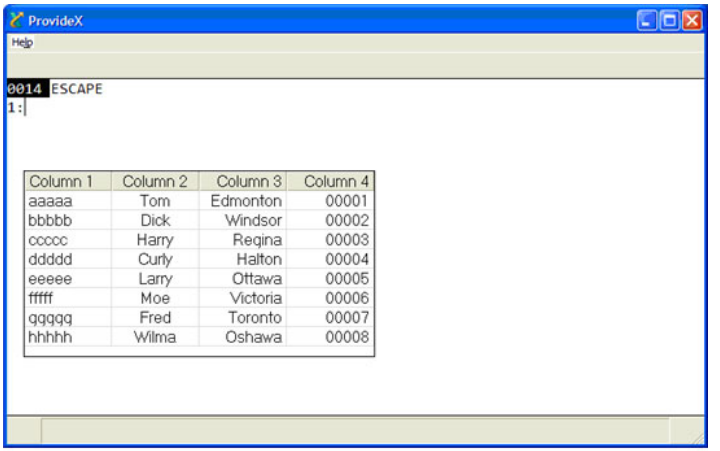
Once it is created, much of the format and operation of a grid is handled using different combinations of **Dynamic Control Properties**, *p.137*. Details are provided in the sections below.

Topics

Example:

```
! Create a GRID starting at column 2 line 5, 40 characters wide.
GRID_ID=1010
GRID GRID_ID,@(2,5,40,10),FMT="[Column 1]L10 [Column 2]C10 [Column 3]R10
      [Column 4]R10"
GRID_ID'COLUMN=-1 ! This selects the rows header column.
GRID_ID'COLUMNWIDTH=0 ! This sets column width to 0
```

```
GRID LOAD GRID_ID,1,1,"aaaaa"+sep+"Tom"+sep+"Edmonton"+sep+"00001"+esc
GRID LOAD GRID_ID,1,2,"bbbbb"+sep+"Dick"+sep+"Windsor"+sep+"00002"+esc
GRID LOAD GRID_ID,1,3,"ccccc"+sep+"Harry"+sep+"Regina"+sep+"00003"+esc
GRID LOAD GRID_ID,1,4,"ddddd"+sep+"Curly"+sep+"Halton"+sep+"00004"+esc
GRID LOAD GRID_ID,1,5,"eeeee"+sep+"Larry"+sep+"Ottawa"+sep+"00005"+esc
GRID LOAD GRID_ID,1,6,"fffff"+sep+"Moe"+sep+"Victoria"+sep+"00006"+esc
GRID LOAD GRID_ID,1,7,"ggggg"+sep+"Fred"+sep+"Toronto"+sep+"00007"+esc
GRID LOAD GRID_ID,1,8,"hhhhh"+sep+"Wilma"+sep+"Oshawa"+sep+"00008"+esc
ESCAPE
```



Formatting a Grid

When creating a grid, **FMT=** settings allow you to format columns by defining column names, titles, widths, alignment and cell types. The format string consists of a space-separated list of column definitions laid out as [*col_title*](*cell_type*:*col_name$*) *Alignment Width*; e.g.,

```
"[Client ID](Multi_line:CST_ID$)L10 [Name](Normal:CST_NAME$)L10"
```

The column title is displayed in the column header line. Cell types include various types of input boxes, buttons, check boxes, and drop boxes. For a full list, see **Cell Types**, *p.183*. The default cell type is `"Normal"` which is a text cell containing one line of data. The column name assigns a variable name (string or numeric) to the column. The alignment character may be **L** (*left*), **C** (*centred*), **R** (*right*). Default: **L**.

It is also possible to format the grid after creation using **Dynamic Control Properties**, *p.137*. This method allows you to format the grid by rows, columns, individual cells, or all cells. Use of the grid **'Row** and **'Column**($) properties to designate affected rows / columns / cells, then set the desired formatting properties.

Referencing Rows, Columns and Cells

For some **GRID** syntax (e.g., **GOTO**, **FIND** keywords), both column and row coordinates are required to act as pointers to particular cells; e.g.,

```
GRID GOTO GRID_ID,2,1 ! Puts focus on column 2 row 1
GRID FIND GRID_ID,3,4,val$ ! Retrieves value of column 3 row 4
```

Other syntax allows you specify a particular row or column (or the entire grid) as well as a particular cell. To specify an entire row, set the column to 0 *zero*. To specify an entire column, set the row to 0. To specify the entire grid, set both to 0; e.g.,

```
LIST_BOX LOAD GRID_ID,0,0,"" ! Clears the entire grid
LIST_BOX DELETE 100,0,2 ! Deletes row 2
LIST_BOX DELETE 100,3,0 ! Deletes column 3
```

When setting properties for rows, columns, or individual cells within the grid, **'Row** and **'Column**($) properties must be specified to identify the cell(s) involved. The special 0 *zero* values can be used to specify rows, columns or the entire grid; e.g.,

```
GRID_ID'Row=n,GRID_ID'Column=0,GRID_ID'Lock=1 ! Locks all cells in row n
```

There are also special values for referencing column and row headers. To reference a column header, set row to -1. To reference a row header, set column to -1; e.g.,

```
GRID_ID'Column=-1,GRID_ID'ColumnWidth=0 ! Row header is now gone!
```

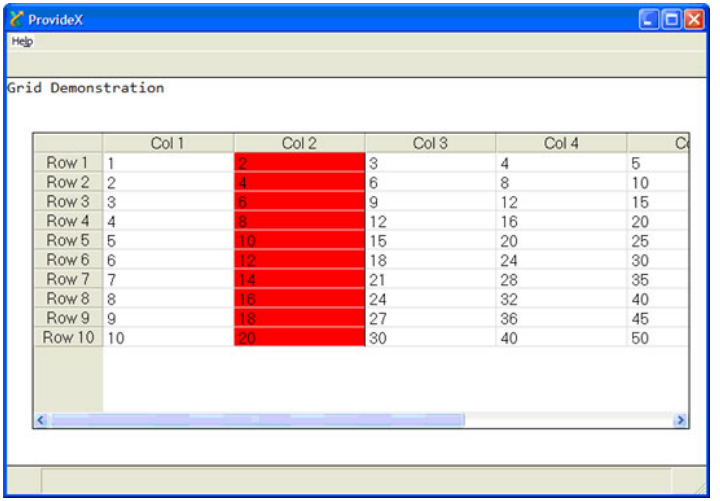
To set properties that pertain to the general behaviour of the grid control (**'Auto**, **'Sep$**, **'LockColumns**, **'LockRows**, **'EnterMode**, **'TabMode**, **'AutoTrack**, **'ExcelStyle**, **'MenuCtl**, etc.) the **'Row** and **'Column**($) values do not apply; e.g.,

```
GRID_ID'EnterMode=2, GRID_ID'TabMode=1, GRID_ID'AutoTrack=3
```

The grid in the following example is created without a **FMT=** string. All formatting is handled using dynamic properties:

```
! GRIDDEMO - Grid demo program
BEGIN ;
PRINT 'CS',"Grid Demonstration"
GRID 10,@(3,3,75,16)
GRID LOAD 10,0,0,""
FOR R=1 TO 10
R$=""
FOR C=1 TO 5
R$+=STR(C*R)+SEP
NEXT C
GRID LOAD 10,1,0,R$
NEXT R
X=10
FOR I=1 TO 5;
X'ROW=-1,X'COLUMN=I;
X'VALUE$="Col "+STR(I);
NEXT
FOR I=1 TO 10;
X'ROW=I,X'COLUMN=-1;
```

```
X'VALUE$="Row "+STR(I);
NEXT
X'COLUMN=-1,X'COLUMNWIDTH=8
X'COLUMN=0,X'COLUMNWIDTH=15,X'ROW=0,X'LEN=7
X'COLUMN=1,X'ROW=0,X'LEN=5
X'COLUMN=2,X'ROW=0,X'LEN=10
X'COLUMN=2,X'ROW=2,X'LEN=3
X'COLUMN=2,X'ROW=0,X'BACKCOLOR$="Light Red"
X'LOCKCOLUMNS=2
INPUT X$;
IF CTL<>4 \
 THEN GOTO *SAME
END
```



For the complete list of dynamic properties, see **Control Object Properties** in the *Language Reference, p.699*.

Cell Types

Several different cell types can be assigned to the **GRID** using the **FMT=** option or via the **'Celltype$** property:

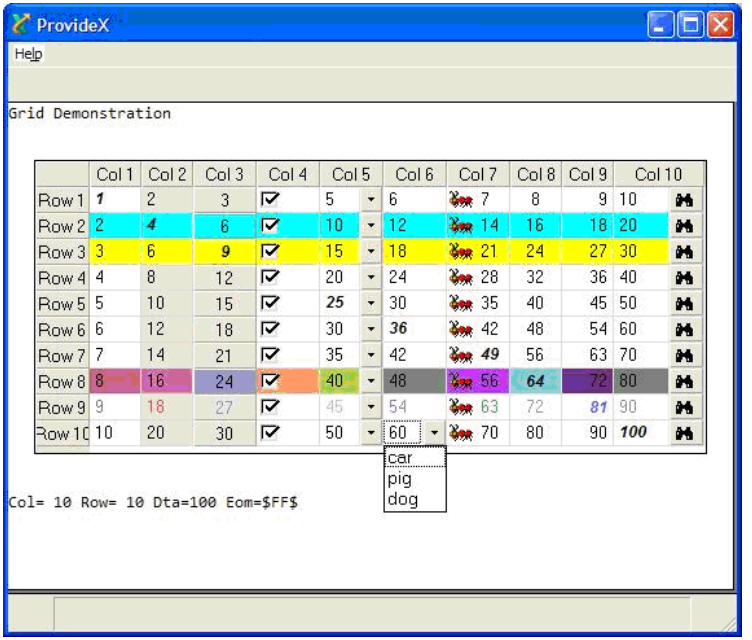
```
"Button", "CheckBox", "CheckBoxRaised", "CheckBoxRecessed",
"CheckMark", "CheckMarkRaised", "CheckMarkRecessed", "DropBox",
"DropBoxHideBtn", "Ellipsis", "EllipsisDrop", "Lookup",
"LookupHideBtn", "Multi_line", "Normal", "Query", "QueryHideBtn",
"SingleLine", "UseTextNormal", "UseTextSingleLine",
"UseTextEllipsis", "VarDropBox", "VarDropBoxHideBtn"
```

For details on these, see **Cell Types** in the *Language Reference, p.145*. While some of these appear similar to other graphical controls, be aware that a grid will only accept **Dynamic Control Properties** that are listed specifically for use in **Grid** controls (***not***

multi-lines, check boxes, buttons, drop boxes, etc.). The following example creates a grid with "Normal", "Button", "DropBox", "Lookup", "CheckMarkRecessed", and "DropBoxHideBtn" cell types:

```
BEGIN
PRINT 'CS',"Grid Demonstration"
GRID 10,@(3,3,74,15)
GRID LOAD 10,0,0,""
FOR R=1 TO 10
R$=""
FOR C=1 TO 10
R$+=STR(C*R)+SEP
NEXT
GRID LOAD 10,1,0,R$
NEXT
X=10
FOR I=1 TO 10;
X'ROW=-1,X'COLUMN=I;
X'VALUE$="Col "+STR(I);
NEXT
FOR I=1 TO 10;
X'ROW=I,X'COLUMN=-1;
X'VALUE$="Row "+STR(I);
NEXT
X'MENUCTL=11
X'COLUMN=-1,X'COLUMNWIDTH=6
X'COLUMN=2,X'ROW=0,X'LOCK=1
X'COLUMN=3,X'ROW=0,X'COLUMNWIDTH=7,X'CELLTYPE$="Button",X'LOCK=1
X'COLUMN=4,X'ROW=0,X'COLUMNWIDTH=7,X'CELLTYPE$="CheckMarkRecessed"
X'COLUMN=5,X'ROW=0,X'COLUMNWIDTH=7,X'CELLTYPE$="DropBox",X'TEXT$="car/pig/dog/"
X'COLUMN=6,X'ROW=0,X'COLUMNWIDTH=7,X'CELLTYPE$="DropBoxHideBtn", \
      X'TEXT$="car/pig/dog/"
X'COLUMN=7,X'ROW=0,X'COLUMNWIDTH=7,X'BITMAP$="!bug"
X'COLUMN=8,X'ROW=0,X'ALIGN$="C"
X'COLUMN=9,X'ROW=0,X'ALIGN$="R"
X'COLUMN=10,X'ROW=0,X'COLUMNWIDTH=10,X'CELLTYPE$="Lookup",X'BITMAP$="!Binoculars"
X'COLUMN=0,X'ROW=2,X'BACKCOLOUR$="LIGHT CYAN"
X'COLUMN=0,X'ROW=3,X'BACKCOLOUR$="LIGHT YELLOW"
X'ROW=8;
FOR I=1 TO 10;
X'COLUMN=I;
X'BACKCOLOUR$="RGB:"+STR(RND(200)+55)+" "+STR(RND(200)+55)+" "+ \
      STR(RND(200)+55);
NEXT
X'ROW=9;
FOR I=1 TO 10;
X'COLUMN=I;
X'TEXTCOLOUR$="RGB:"+STR(RND(200)+55)+" "+STR(RND(200)+55)+" "+ \
      STR(RND(200)+55);
```

```
NEXT
FOR I=1 TO 10;
X'COLUMN=I,X'ROW=I,X'FONT$="Arial,1,BI";
NEXT
WHILE 1
INPUT X$;
IF CTL=4 \
 THEN BREAK
IF CTL=11 \
 THEN POPUP_MENU "[&One=1,&Two=2,&Three=3,&Four=4]",X;
      IF X \
       THEN MSGBOX STR(X),"Menu Selection"
IF CTL<>10 \
 THEN PRINT @(0,20),'CL',"Recv'd CTL=",CTL,;
      CONTINUE
GRID READ 10,C,R,ZZ$,E$
PRINT @(0,20),'CL',"Col=",C," Row=",R," Dta=",ZZ$," Eom=$",HTA(E$), \
        "$",
WEND
END
```



Named Columns

Columns can be given logical names. This feature allows applications to assign names to columns that are the same as the variables used within the underlying application; i.e., ITEMID$,DESC$,PRICE,QTY. It also facilitates dealing with swapped columns. Naming is handled in the **FMT=** clause or via dynamic properties. The name can represent a string or numeric variable; e.g.,

```
GRID 10,@(10,10,60,10),
FMT="[Item](Normal:ITEMID$)L15,[Description](Normal:DESC$)L20,[Price](Nor
      mal:PRICE)L12,[Qty](Normal:QTY)L10"
```

Dynamic properties can also be used to set column names. This is done by setting '**Row** property to -1 and assigning a variable to the '**Text$** property; e.g.,

```
GRID 10,@(5,3,70,10) ! Define grid
X=10
X'COLUMNSWIDE=4,X'ROW=-1
X'COLUMN=1,X'VALUE$="Item",X'TEXT$="ITEMID$",X'COLUMNWIDTH=15
X'COLUMN=2,X'VALUE$="Description",X'TEXT$="DESC$",X'COLUMNWIDTH=20
X'COLUMN=3,X'VALUE$="Price",X'TEXT$="PRICE",X'COLUMNWIDTH=12
X'COLUMN=4,X'VALUE$="Qty",X'TEXT$="QTY",X'COLUMNWIDTH=10
```

Columns can be referenced either by column number, using the '**Column** property or by column name using the '**Column$** property. If you set '**Column$** with the name of a variable, then internally, the column number is used as a pointer. If you read the property '**Column**, then the column number is returned. '**Column$** returns the name of the column. Column swapping has no impact on an application that uses logical column names to identify a column; e.g.,

```
X'COLUMN$="QTY",X'ROW=4,X'VALUE$="200" ! Assign cell value 200 to qty column
```

Loading the Grid

Once created, the grid can be updated using **GRID LOAD**, and **GRID WRITE** directives. These, like most other **GRID** statements, require the specification of both the target row and column.

All column and row specification are base 1; therefore, the top most cell is 1,1. By default, there are also column and row headers that can be accessed by column and / or row -1. These headers may not be included in any range specifications.

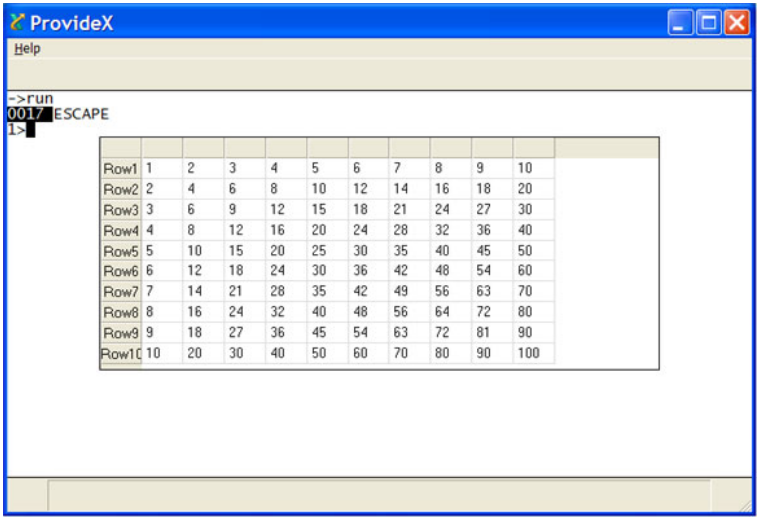
If you write to a grid when the column number is 0 *zero*, then all columns will be affected by the change. If the row number is 0, then all rows will be affected by the change. If both are 0, then all cells will be affected.

Data may be loaded into a grid in three ways, using the **GRID LOAD** directive, dynamic properties, and named columns).

In order to make the load process easier, the **GRID LOAD** directive will accept multiple rows and columns of data. The column data must be separated by the character identified in the column separator in the grid creation and the rows must be separated by the delimiter contained in the last data character of the loaded data. Rows can also be loaded individually like a list box.

The column number and row number are used to indicate the starting point within the grid. If both are 0, then the grid will be cleared before loading. If just the row is 0, then the data will be appended after the last existing row on the grid. In the following example, the entire grid is populated in a single load:

```
gr=1000
GRID gr,@(10,4,60,15),SEP="/"
GRID LOAD gr,0,0,""
R$="",TOTAL=0
FOR R=1 TO 10
R$+="Row"+STR(R)+"/"
FOR C=1 TO 10
R$+=STR(C*R)+"/" ! / is the column separator
TOTAL+=(C*R)
NEXT
R$=STP(R$,1,"/")
R$+=SEP ! sep is row delimiter
NEXT
GRID LOAD gr,-1,0,R$
ESCAPE
```



Row1	1	2	3	4	5	6	7	8	9	10
Row2	2	4	6	8	10	12	14	16	18	20
Row3	3	6	9	12	15	18	21	24	27	30
Row4	4	8	12	16	20	24	28	32	36	40
Row5	5	10	15	20	25	30	35	40	45	50
Row6	6	12	18	24	30	36	42	48	54	60
Row7	7	14	21	28	35	42	49	56	63	70
Row8	8	16	24	32	40	48	56	64	72	80
Row9	9	18	27	36	45	54	63	72	81	90
Row10	10	20	30	40	50	60	70	80	90	100

Cells in a grid may also be loaded using dynamic properties. This is done by specifying **'Row** and **'Column**($) to identify the cell desired, then setting the content using the **'Value$** property. The following code creates and loads a grid similar to the previous example, but uses dynamic properties instead.

```
gr=1000
GRID gr,@(10,4,60,15),SEP="/"
gr'ROWSHIGH=10,gr'COLUMNSWIDE=10 ! define cells
R$="",TOTAL=0
FOR R=1 TO 10
gr'COLUMN=-1,gr'ROW=R,gr'VALUE$="Row"+STR(R)
FOR C=1 TO 10
R$=STR(C*R)
gr'COLUMN=C,gr'ROW=R,gr'VALUE$=R$
TOTAL+=(C*R)
NEXT
NEXT
RETURN
```

Note: Setting the **'Value$** property in this example is the same as doing a **GRID WRITE**; therefore, `gr'COLUMN=C,gr'ROW=R,gr'VALUE$=R$` could also be written as `GRID WRITE GRID_1.CTL,C,R,R$`.

Assigning a Row of Data

The **'LoadList$**, **'LoadIolist$** and **'RowData$** properties can be used to load rows of data into a grid. **'LoadList$** returns a list of column names in the order in which they physically appear within the grid object. When a grid is defined using **FMT=** in the **GRID** directive, or columns names are assigned to columns in a grid, **'LoadList$** returns a list of those column names in their current display order. The compiled version, **'LoadIOLIST$** simplifies the loading of grid rows from direct file contents or other IOList-based items.

The property **'LoadList$** allows you to define the order of the columns that are loaded from the data sent to the grid by a **GRID LOAD**. Historically, **GRID LOAD** loads the columns in sequence; i.e., column 1, column 2, etc. Unfortunately, this causes a problem with the ability to alter column order (swap columns). To avoid this issue, you can define the order of the column data that is sent.

When you read this property, it returns either column order list (if previously defined) or the column order by name as it exists currently in the grid.

The property **'LoadIOList$** contains the compiled version of **'LoadList$**. The grid's column separator (**'SEP$**) is included between each variable in the IOList. Using this list enables the user to load data by record as well as overcome problems involved with swapping columns; e.g.,

```
GRID LOAD Grid1,1,Row,REC(Grid1'LoadIOList$)
```

Note: Use of this property assumes that the grid columns have been named using proper variable names. Failure to use proper variable names will result in an invalid IOList being returned.

The **'RowData$** property sets a row of data in accordance with the names defined in the **'LoadList$**. The row number must first be identified using the **'Row** property; e.g.,

For example, assuming X contains the CTL value for the grid:

```
X'ROW=5
X'ROWDATA$=REC(X'LOADIOLIST$)
```

This loads the contents of the IOList into row 5. The example below loads fields from a data file directly into the grid using field names as column names:

```
g=10,row=0
f1$="[Cust ID](Normal:cst_id$)L8,"
f2$="[Name](Normal:cst_name$)L30,"
f3$="[Balance](Normal:cst_amt)R10"
GRID g,@(10,3,60,12),FMT=f1$+f2$+f3$
SELECT cst_id$,cst_name$,*,*,*,*,cst_amt FROM "cstfile"
row++
g'rowshigh=row
g'row=row,g'RowData$=REC(g'LoadIolist$)
NEXT RECORD
```

	Cust ID	Name	Balance
	000192	The Hungry Incorporated	0.55
	000213	Torpedo's Away Body Shop	85.59
	000215	Ugly Duckling Incorporated	55.68
	000216	Blue Funk Garments	7.74
	000230	New Dimensions Importers	4.17
	000235	The Hungry Interior Design	1.57
	000241	Worldwide Systems Group	62.56
	000247	Foxhunter Magazine	92.82

Reading Values from the Grid

When the contents of a cell in a grid are modified, a CTL value is generated for the event. The location and value of modified cells can be retrieved using the **GRID READ** and **GRID READ NEXT** statements. These directives include the column and row of the cell that has been affected by a change.

In order to assure that no change ever gets lost, all changes are placed in a **READ** queue. Whenever a **GRID READ** request is executed, if there is additional input in the queue, another CTL event is initiated. This ensures that, should any CTL events be lost, the changed data will be preserved. However, it does pose a problem due to

potential race conditions when the host is unable to keep up with the user input. Superfluous CTL events may be received. To avoid this problem, the **GRID READ** directive receives an `Error #2 End of file if no data` is in the **READ** queue.

Warning: Under NOMADS, the read is handled automatically and should not be attempted by your application.

Retrieving Data

To retrieve the contents of a single cell, the control attribute **'Value$** can be used. You must first specify the **'Column($)** and **'Row** properties to identify the cell desired.

For example, where X is the CTL value.

```
X'Row=r,X'Column=c
CellVal$=X'Value$
```

To retrieve the contents of the cell which currently has focus, use the **'CurrentRow** and **'CurrentColumn** properties to specify the cell; e.g.,

```
gr'row=gr'CurrentRow,gr'column=gr'CurrentColumn
cur_value$=gr'VALUE$
```

The grid allows multiple cells to be selected across multiples columns and rows. To return the contents of each selected cell, the **'SelectCount**, **'SelectIndex**, **'SelectRow**, **'SelectColumn** , **'SelectValue$** and **'SelectText$** properties can be used:

'SelectCount Number of items / cells selected. Set to zero to de-select all.

'SelectIndex Index the selected cell. Set to point to a selected cell; e.g., set to 1 to point at the first cell selected, 2 to point at the second cell selected, etc.

The following properties are used to access selected cells based on the location defined by **'SelectIndex**:

'SelectRow Row number of selected cell.

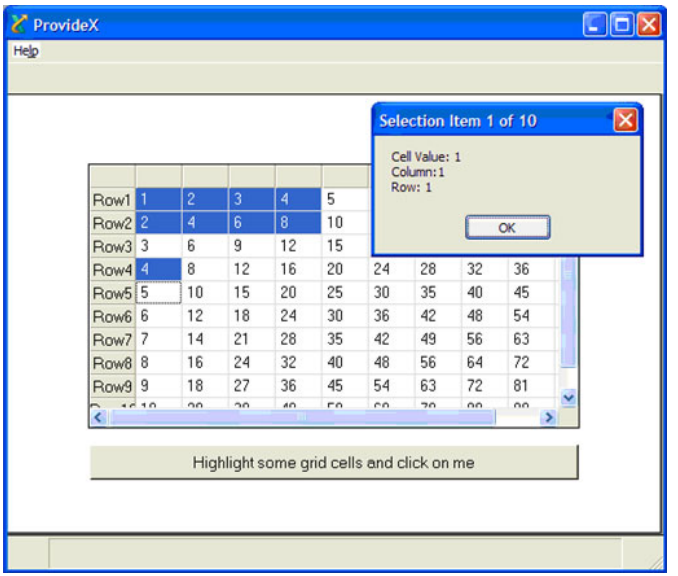
'SelectColumn Column number of selected cell.

'SelectValue$ Value within selected field.

Example:

```
! Create and load the grid
gr=1000
GRID gr,@(10,4,60,15),SEP="/"
GRID LOAD gr,0,0,""
R$="",TOTAL=0
FOR R=1 TO 10
R$+="Row"+STR(R)+"/"
FOR C=1 TO 10
R$+=STR(C*R)+"/" ! / is the column separator
TOTAL+=(C*R)
NEXT
```

```
R$=STP(R$,1,"/")
R$+=SEP ! sep is row delimiter
NEXT
GRID LOAD gr,-1,0,R$
! Select some cells and click the button
BUTTON 10,@(10,20,60,2)="Highlight some grid cells and click on me"
WHILE 1
INPUT *
IF CTL=10 \
 THEN BREAK
WEND
! Display the selected cells
TotalSelected=gr'SelectCount
IF gr'RowsHigh<1 OR TotalSelected=0 \
  THEN EXIT
FOR T=1 TO TotalSelected
gr'SelectIndex=T
SLval$=gr'SelectValue$
SLcol=gr'SelectColumn
SLrow=gr'SelectRow
MSGBOX "Cell Value: "+PAD(SLval$,50)+SEP+"Column:"+STR(SLcol)+SEP+"Row: "+ \
      STR(SLrow),"Selection Item "+STR(T)+" of "+STR(TotalSelected)
NEXT T
END
```



The property **'RowData$** will return a row of data in accordance with the names defined in the **'LoadList$**. The row number must first be identified using the 'Row property.

For example, assuming X contains the CTL value for the grid:

```
X'ROW=5
READ DATA FROM X'ROWDATA$ TO IOL=X'LOADIOLIST$
```

This will loads the IOList with the data from row 5.

Multi-Property Set/Get for Controls

Reading more than one property at a time is not typically a problem. However, when accessing properties across a **WindX** / **JavX** / **UltraFX** connection, one packet is required for each property read. This can become a performance issue, especially when dealing with objects that use a large number of properties (i.e., a grid).

To reduce the amount of network traffic, three special "common" properties can be used to retrieve and set the values from multiple properties at the same time:

'_PropList$	Comma separated list of property names to read / write.
'_PropValues$	String that contains values for each of the properties in **_PropList$**.
'_PropSep$	Character used as a field separator between values.

Note: These are considered to be "common to all" and do not appear in the list of properties when querying a particular object (via **'*** *tick asterisk*).

To retrieve the value of multiple properties, first set **'_PropList$** and then read **_PropValues$**; e.g.,

```
G1'_PropList$="CurrentColumn,CurrentRow,Value"
x$ = G1'_PropValues$
```

Here x$ would receive a string containing the values of `CurrentColumn`, `CurrentRow`, and `Value` with each field separated by either the standard **SEP** field separator, or with the `'_PropSep$` character. Data can then be extracted using a **READ DATA** directive, e.g.,

```
READ DATA FROM G1'_PropValues$ TO IOL=MYIOL
MYIOL: IOLIST Col, Row, Value$
```

To set values, simply set the value in `'_PropValues$`; e.g.,

```
G1'_PropValues$ = "1"+SEP+"2"+SEP+"Data" or G1'_PropValues$ =
      REC(IOL=MYIOL)
```

The advantage here is that fewer packets will need to be sent to the thin-client for setting or retrieving the values of multiple properties. It is important to remember that the fields **'_PropList$** and **'_PropSep$** should not be constantly read from the control in order to parse the data returned by **'_PropValues$**, as this will defeat the purpose.

For example, do not...

```
READ DATA FROM G1'_PropValue$,SEP=G1'_PropSep$ TO IOL=CPL("IOLIST
      "+G1'_PropList$)
```

This would actually result in three exchanges with WindX: one to get '**_PropSep$**, one for '**_PropList$**, and then one for '**_PropValues$**.

The separator character and list should be maintained within the application code and not retrieved from the control.

Drag and Drop Properties

The grid interface supports drag and drop functionality. Grids have four properties that are used to support drag and drop:

'**DraggedColumn**	Column where the drag started from
'**DraggedRow**	Row where the drag started from
'**DroppedOnColumn**	Column dropped on
'**DroppedOnRow**	Row dropped on

When dragging off of a grid, the starting point must be a column or row header. If a column header is used and column swapping is enabled using the '**SwapEnabled** property, then the drop target must be a control other than the grid itself. This is because dragging a column header within the grid is the method used to swap columns.

Example:

```
1600 ! 1600 - Drag and drop rows from g1 (grid 1) to g2 (grid 2)
1610 ROW_DROP:
1620 LET R=G1'DRAGGEDROW; IF R<1 THEN RETURN
1630 LET G1'ROW=R
1640 READ DATA FROM G1'ROWDATA$ TO IOL=G1'LOADIOLIST$
1650 GRID DELETE G1,0,R
1660 LET R=MAX(1,MIN(G2'DROPPEDONROW,G2'ROWSHIGH))
1670 GRID ADD G2,0,R; LET G2'ROW=R
1680 LET G2'ROWDATA$=REC(G2'LOADIOLIST$)
1690 RETURN
```

Scrollbars

V_SCROLLBAR *ctl_id*,**@(***col*,*ln*,*wth*,*ht***)**=*contents$*[,*ctrlopt*]
V_SCROLLBAR *ctl_id* **WINDOW [**,*ctrlopt***]**
V_SCROLLBAR {**REMOVE**|**DISABLE**|**ENABLE**} *ctl_id*[,**ERR=***stmtref*]
V_SCROLLBAR {GOTO|**HIDE**|**SHOW}** *ctl_id***[,ERR=***stmtref***]**
V_SCROLLBAR READ *ctl_id*,*setting*,*max*,**[***rgn_chg***][,***arrow_chg***][,ERR=***stmtref***]**
V_SCROLLBAR WRITE *ctl_id*,*marker*,*max***[,ERR=***stmtref***]**

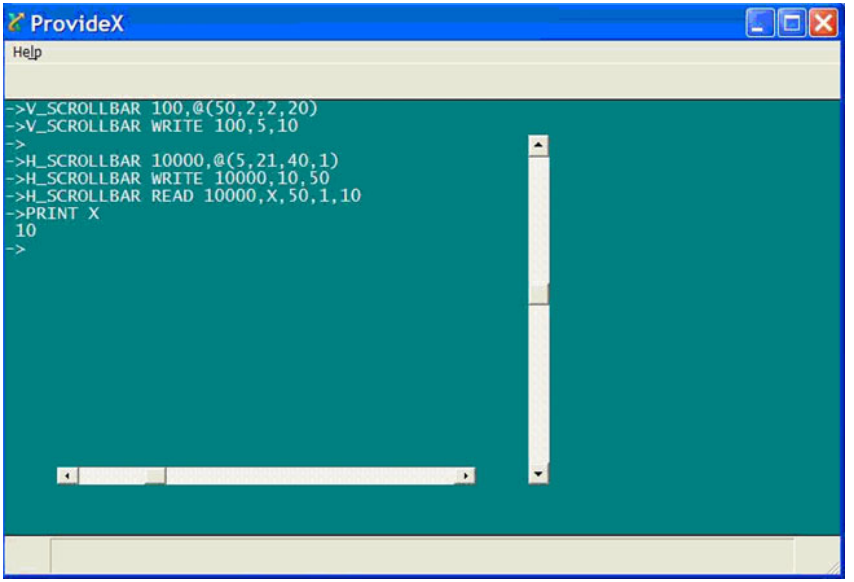
H_SCROLLBAR *ctl_id*,**@(***col*,*ln*,*wth*,*ht***)**=*contents$*[,*ctrlopt*] **...**

Vertical and horizontal scrollbars are controls that can be used create a slider, spinner, progress bar object. For syntax details, refer to the **H_SCROLLBAR** and **V_SCROLLBAR** directives in the *Language Reference*.

Examples:

```
V_SCROLLBAR 100,@(50,2,2,20)
V_SCROLLBAR WRITE 100,5,10

H_SCROLLBAR 10000,@(5,21,40,1)
H_SCROLLBAR WRITE 10000,10,50
H_SCROLLBAR READ 10000,X,50,1,10
PRINT X
10
```



Chart

CHART *ctl_id*,**@(***col*,*ln*,*wth*,*ht***)**,[,*ctrlopt*]
CHART {**REMOVE**|**DISABLE**|**ENABLE**} *ctl_id*
CHART {**HIDE** | **SHOW**} *ctl_id*
CHART LOAD *ctl_id*,*strvar$*
CHART CLEAR *ctl_id*
CHART DELETE *ctl_id*
CHART FIND *ctl_id*,*dataset*, *point*,{*numvar*|*label$*}
CHART READ *ctl_id*, *dataset*,*eom$*
CHART WRITE *ctl_id*,*dataset*,*point*,{*numvar*|*label$*}

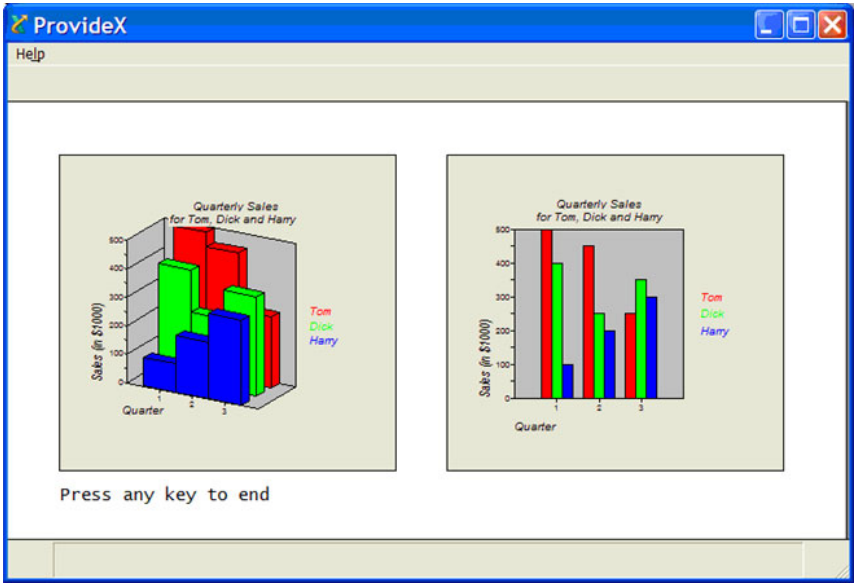
The charting control feature in ProvideX can be used to create a variety of chart illustrations with 2D or 3D effects. This object type is generally for display purposes only and responds to few events. Available chart types include area, bar, column, line, pie, ribbon, scatter, and stack. For syntax details, refer to the **CHART** directive in the *Language Reference, p.43*.

This feature requires activation of the ProvideX ***Charting Control*** add-on package.

Example:

```
! Chart example
  BEGIN
  PRINT 'CS',
  FOR i=1 TO 2
  GOSUB Draw_Chart
  NEXT
  INPUT @(5,22),"Press any key to end ",*,'CS',
  END
 Draw_Chart:
  IF i=1 \
   THEN c=10,x=5,format$="3DCOLUMN" \
   ELSE c=20,x=42,format$="2DCOLUMN"
  CHART c,@(x,3,32,18),FMT=format$,SEP=","
  c'indexmode=1
  c'AutoScale=0,c'font$="Arial,-10,I"
  c'BackColour$="Light Gray"
  c'LegendLocation$="Right"
  c'Title1$="Quarterly Sales",c'Title2$="for Tom, Dick and Harry"
  c'xAxisTitle$="Quarter",c'yAxisTitle$="Sales (in $1000)"
  CHART LOAD c,"Tom=500,450,250/Dick=400,250,350/Harry=100,200,300/"
  RETURN
```

This example creates two and three-dimensional charts, using properties to add legends, change font and colours as well as other display attributes, as illustrated below.


ProvideX
Help
Quarterly Sales
for Tom, Dick and Harry
Sales (in $1000)
Quarter
Tom
Dick
Harry
Quarterly Sales
for Tom, Dick and Harry
Sales (in $1000)
Quarter
Tom
Dick
Harry
Press any key to end

Taskbar Notification Icon

In ProvideX for Window, the **BUTTON**, **CHECK_BOX** and **TRISTATE_BOX** directives (with **OPT="Y"** setting) can be used to place an icon in the *Taskbar Notification Area* (a.k.a the *System Tray*) in the bottom-right corner of the MS Windows desktop.

Different aspects of this type of control may be defined, including the icon used, floating tip, balloon information, and right mouse menu. Depending on the number of *states* required, **BUTTON** is used to define one icon, **CHECK_BOX** toggles between two different icons, **TRISTATE_BOX** cycles between three icons. The ability to cycle between states is controlled using the '**Value** property.

Creation Format

Apart from the fact that each directive defines a specific number of images (states), the creation format for a taskbar notification icon is basically the same regardless of which directive is used; e.g.,

BUTTON [*****]*ctl_id*,**@(***col,ln,wth,ht***)=***contents*$[,*ctrlopt*],**OPT="Y"**

In the above syntax example, **OPT="Y"** identifies this control as an icon to be placed in the *Taskbar Notification Area*. The *contents*$ argument defines the icon image inside { } *braces* followed by text information to be used as the title for a balloon message (explained in the sections below). Following is an example creation command for a single-state taskbar notification icon:

```
BUTTON *10,@(0,0,0,0)="{}ProvideX Tray Control",opt="Y",tip="A
Floating Tooltip",mnu=11,msg= "{pvxwin32.exe@pvxstop}This is
line 1 of a tray balloon"+$0d0a$+ "And this would be line
2"+$0d0a0d0a$+"Click on the balloon to continue"
```

Coordinates are ignored for taskbar notification icons. A leading * asterisk denotes it as a "*global*" icon; in which case, the icon is tied to the whole ProvideX session and remains visible no matter which window is considered current. Otherwise, the icon is tied only to the window where it was created and is hidden / shown as necessary when moving to a different window. See also, **Other Formats**, *p.199*.

Assigning an Icon Image

The string assigned on the creation command may contain one or more icon images enclosed in { } *braces* and separated by | *pipe* symbols. If no icon is specified, or a null icon is specified, then the current icon in use by the ProvideX session is used. Only `.ico` images may be used in the *Taskbar Notification Area*. Other file formats (`.bmp`, `.jpg`, etc.) are ignored.

Floating Tip

Floating tips may be specified by adding **TIP=***text$* to the creation command.

Right Mouse Events

A CTL value may be assigned to any right mouse clicks by adding a **MNU=***ctl* to the creation command.

Balloon Text

Balloons may be specified via the control option **MSG=** during creation, or may be changed on the fly via the **'Msg$** property. The string data may include an icon for use within the balloon itself, as well as lines of text and other options to control specific balloon behaviour. Format of the **MSG=** or **'Msg$** string is:

"{*icon*}*text*;**TIM=***num*;**QUIET;REALTIME**"

Where:

{icon}	Balloon icon. Optional, must appear before *text* when used. If no braces are present, then no icon appears in the balloon. If **{}** (braces with no contents) are given, then the icon currently in use by the ProvideX session will be used. Any other icon specification follows ProvideX's icon specification conventions. {**!ERROR**} , {**!INFO**} or {**!WARNING**} define built-in icons that use the Windows standard icons and potentially any sound associated.
text	Balloon text. This may contain CRLF ($0D0A$) to specify line breaks. The maximum length is 255 characters.
;TIM=*num*	Time in seconds that the balloon is visible. Optional, must appear after *text* when used. ***Windows Restrictions.*** The OS controls the minimum / maximum time (XP defaults to 10 minimum and 30 maximum). It can override the **TIM=** setting; i.e., if more balloons are queued, then the display time drops to the minimum time regardless of the setting. Also, user (keyboard / mouse) activity may be necessary for the time to actually count down. However, under Vista, balloon display seems to last for a minimum of 7 seconds, and keyboard / mouse movement is no longer required to have the time count down.
;QUIET	Turns off sound associated with the balloon tip (if any). Optional, must appear after *text* when used.
;REALTIME	Tells the OS not to display the balloon if it cannot be displayed right away. Optional (Vista only), must appear after *text* when used.

Control Object Properties

Most **Dynamic Control Properties** will have no effect on a notification icon; i.e., you cannot change items such as **BackColour**, **Left**, **Line** etc. The properties which do have an effect are as follows:

Enabled	Can be read, or set to disabled (0) or enabled (1).
Eom$	Contains the character that caused the last CTL event for this control.

Focus	May be set to 1, which sets focus to this control within the notification tray and away from the current ProvideX window.
ImageCount	Returns the number of icons associated with this control.
MenuCTL	Can be read or set to a CTL value that will be generated when the user right mouse clicks on this control.
Msg$	Can be read or set to change the balloon information on the fly.
OnFocusCTL	Can be read or set to a CTL value that will be generated when the user clicks on the balloon while the balloon is visible.
Text$	Can be read or set to change the icon and text (bolded title in the balloon).
Tip$	Can be read or set to change the floating tool tip.
Value	Can be set to 1, 2 or 3 to switch between icons specified in the text of the control (**BUTTON** has one icon, **CHECK_BOX** may have two, **TRISTATE_BOX** may have three).
Visible	Can be used to make the notification icon hide/show itself.

Other Formats

These other formats are used to control the actions of a taskbar notification icon once it has been created. As mentioned before, the syntax elements described below apply to all three directives; i.e., substitute the **BUTTON** keyword for **CHECK_BOX** or **TRISTATE_BOX** as required.

1. **BUTTON REMOVE** [*****]*ctl_id*[,**ERR=***stmtref*]

   Deletes a global or non-global taskbar notification icon.

2. **BUTTON** {**DISABLE | ENABLE**} [*]*ctl_id*[,**ERR=***stmtref*]

   Enables/disables a global or non-global taskbar notification icon. When enabled, CTL events will fire when the user clicks on the button. When disabled, all mouse or keyboard actions on the control are ignored. The control remains visible whether enabled or disabled.

3. **BUTTON** {**HIDE | SHOW**} [*****]*ctl_id*[,**ERR=***stmtref*]

   Makes an icon visible or invisible.

4. **BUTTON GOTO** [*]*ctl_id*[,**ERR=***stmtref*]

   Forces focus away from ProvideX and into the taskbar notification area. The icon will get focus and any floating tool tip will automatically be displayed.

5. **BUTTON** {**ON | OFF**} [*****]*ctl_id*[,**ERR=***stmtref*]

   Displays the icon's balloon when **ON** is used, and removes the balloon when **OFF** is used. The balloons will automatically time out themselves, so it is not necessary to use **OFF** unless the developer wishes to force an early close of the balloon

6. **BUTTON READ** [*****]*ctl_id,mode$*[,**ERR=***stmtref*]

   Returns the EOM of the last CTL event.

CTL Values for Taskbar Notification Icons

CTL values will be generated when the user clicks with the mouse, or uses the keyboard with the tray control. The events include the following:

- **LEFT-MOUSE-CLICK** *or* SPACEBAR generates *ctl_id* of control, EOM=01
- **LEFT-DOUBLE-MOUSE-CLICK** *or* Enter generates *ctl_id* of control, EOM=02

If the **MNU=***ctl* option is specified on creation or the **'MenuCtl** property is set:

- **RIGHT-MOUSE-CLICK** *or* **RIGHT-DOUBLE-MOUSE-CLICK** generates value of **MNU=***ctl*, **'MenuCtl** property, or EOM=11.

If the **MNU=***ctl* option is *not* specified or the **'MenuCtl** property is *not* set:

- **RIGHT-MOUSE-CLICK** generates *ctl_id* of control, EOM=11
- **RIGHT-DOUBLE-MOUSE-CLICK** generates *ctl_id* of control, EOM=12

If **'OnFocusCTL** property is set:

- Any click on the balloon generates value of **'OnFocusCTL** property, EOM=FF

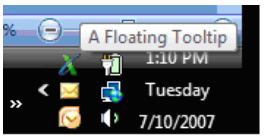
No event will be fired when clicking on the balloon if **'OnFocusCTL** is not set.

Note: When the user clicks on the notification icon or balloon, focus is *not* automatically returned to the ProvideX application window. Focus remains in the taskbar notification area, as the user may hit escape to cancel a menu selection.

Examples

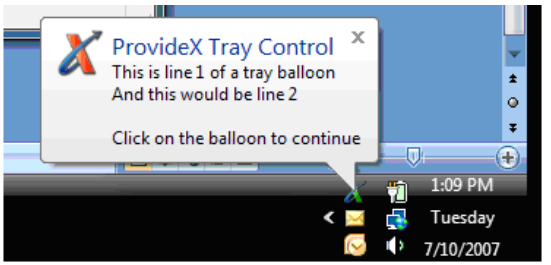
The following code creates a taskbar notification tray icon:

```
BUTTON *10,@(0,0,0,0)="{}ProvideX Tray Control",opt="Y",tip="A
Floating Tooltip",mnu=11,msg= "{pvxwin32.exe@pvxstop}This is
line 1 of a tray balloon"+$0d0a$+ "And this would be line
2"+$0d0a0d0a$+"Click on the balloon to continue"
```



To display the balloon:

```
BUTTON ON *10
```



Display Objects

Besides the interactive **Control Objects** which add functionality to a GUI panel, other graphical components are needed to help with the usability, organization, and layout of your graphical application.

Topics

These are display-only objects, such as text and labels, images, and shapes drawn on the canvas of the graphical panel. As mentioned earlier, these types of object are created with mnemonics (rather than directives) and have no associated events.

Mnemonics for creating graphical objects include: **'TEXT'**, **'FONT'**, **'PICTURE'**, **'ARC'**, **'PIE'**, **'CIRCLE'**, **'LINE'**, **'POLYGON'**, and **'RECTANGLE'**. There is also a mnemonic for creating a graphical group, **'IMAGE'**. These objects are output on the graphic plane via the **PRINT** directive. By default, **PRINT** outputs a line feed to the text plane unless the statement is terminated by a trailing comma. It is therefore recommended that the trailing comma be used when drawing graphical elements to prevent unwanted scrolling of the text plane. For syntax details, refer to the **PRINT** directive in the *Language Reference, p.251*.

Placement and Size

The various display objects are located on the panel using graphical units. Graphical units are based on 1/16 of the width of a character cell. Since the size of character cells change based on the session's fixed base font, the size of a graphical unit will change proportionately. Therefore, the relative location of a graphical element on a panel will be same whether the base font is large or small. However, base fonts with different width to height aspect ratios may result in a different value when measuring vertically.

Unlike **Interface Windows** and **Control Objects** (which are sized by specifying width and height in terms of columns and lines) the dimensions of graphical display elements are determined by specifying the graphical coordinates of the vertices that define the element. For example, text areas and rectangles are defined by specifying an x/y coordinate for the top left corner and another for the bottom right.

Horizontal and vertical graphical units are consistent, such that a rectangle drawn from 0,0 to 100,100 will form a square. Shapes such as arcs, pies and circles are defined by specifying the x/y coordinate for the centre of the shape and a radius measured in graphical units.

@x( ) and @y( ) Functions

When programming, it is not obvious how many graphical units represent a particular location. (The top left corner of the panel is location 0,0.) Therefore, ProvideX includes two system functions to convert the more familiar line and column locations into graphical units. The **@x( )** function returns the x-axis graphical coordinate of a given column, and **@y( )**returns the y-axis graphical coordinate of a given line. For syntax details, see **@X( ) / @Y( )** in the *Language Reference, p.389*.

The values passed to these functions can be fractional, but the return value will be an integer representing the number of graphical units. Because fractional values may be passed to these functions, they can be used to locate graphical display elements in precise locations on the panel.

MXC( ) and MXL( ) Functions

To determine the width and height of a panel in graphical units, you can use the **MXC( )** and **MXL( )** functions in conjunction with **@x( )** and **@y( )**; e.g.,

```
PRINT @X(MXC(0)+1) ! Width of panel in graphical units
1280
PRINT @Y(MXL(0)+1) ! Height of panel in graphical units
844
```

While the number of horizontal graphical units for a panel 80 columns wide will remain consistent, the number of vertical graphical units for a panel 25 lines high may differ based on the width:height ratio of the base font. For syntax details, see **MXC( ) / MXL( )** in the *Language Reference, p.486*.

Text

Plain text (i.e., output to the text plane) can be displayed on a graphical panel, but placement is restricted to lines and columns, and the font is restricted to the session's fixed text plane font. It is therefore more common to draw text in graphics mode when displaying text on a graphical panel. Location of the text can be defined with precision, and any available font can be utilized. To draw text in graphics mode, the **'FONT'** mnemonic is used to control the font, and the **'TEXT'** mnemonic is used to locate and display the text.

'FONT'(*name$*,*size*[,*attrib$*[,*angle*]]

Defines the current font and specifications. The font specified by the font *name$* must exist on the system or the default system font is used. Optional *size* values may be positive or negative. Positive font sizes are relative to the current default, e.g. .5 for half size, 2 for double, etc. Negative font sizes are used for absolute font size in points. The optional *attrib$* string is composed of codes:

& Underscore the character following the '&' (as in hot keys)
B Bold
C Centre text
F Show focus lines around text
I Italics
N Numeric data alignment

R Right justify text
S Applies background colour to area directly behind text.
U Underscore ("_")
W Word wrap
Same as **N**

For syntax details, see the **'FONT'** mnemonic in the *Language Reference, p.607*.

Examples:

```
PRINT 'FONT'("Courier New"),
PRINT 'FONT'("Verdana",1,"BR"),
PRINT 'FONT'("Arial",-12,"CI"),
```

'TEXT'(*x*,*y*[,*x*,*y*],*text$*,*attrib$***)**
Draws text in graphics mode. The first set of x/y graphical coordinates provide the starting point (top left corner) of the text. The optional second set of x/y coordinates define the bottom right corner of a rectangular region for displaying the text. The second set of coordinates are necessary to centre, right justify or word-wrap the text. The optional attribute string is composed of codes that include:

& Underscore the character following the '&' (as in hot keys)
C Centre text
F Show focus lines around text
N Numeric data alignment
R Right Justify
S Applies background colour to area directly behind text.
W Word wrap
Same as **N**

For syntax details, see the **'TEXT'** mnemonic in the *Language Reference, p.640*.

Examples:

```
PRINT 'FONT'("Arial",-11),
PRINT 'GREEN','TEXT'(@x(2),@y(3),"&Hello","&"),
PRINT 'TEXT'(@x(2),@y(4.5),@x(15)@y(6),T$,"R"),
```

The **TXH( )** and **TXW( )** functions can be used to determine the size of the text to be displayed. For syntax details, see **TXH( )**/**TXW( )** in the *Language Reference, p.542*.

Images

The '**PICTURE**' mnemonic is used to draw a picture on the panel.

'PICTURE'(*x*,*y*,*x*,*y*,{*name$*|*#chan*[,*transp_opt*]}[,*display_opt*]**)**
The images can be bitmaps, or with the ProvideX **Multiple Image Support** add-on, other image formats (*gif*, *jpg*, *png*, *tif*, etc.). A number of display options are available:

0=Align at top-left
1=Centre/crop within region
2=Scale to fit
3=Tile bitmaps to fill the given area
4=Halftone for enhanced legibility (may lighten black images)
5=Scale with proper aspect ratio but output in top left
6=Scale with proper aspect ratio but centred in the region.

For options 0, 1, and 3, the image is cropped to fit within the specified region on the screen. When scaling an image, options 5 and 6 are recommended when it is important that the aspect ratio be maintained and the picture is not to be stretched unnaturally. If it is not important to maintain the aspect ratio when scaling, option 4 generally results in a cleaner image than option 2; e.g.,

```
PRINT 'PICTURE'(0,0,@X(MXC(0))+1,@Y(7),"people.bmp",4),
```

For syntax details, see the **'PICTURE'** mnemonic in the *Language Reference, p.640*.

Shapes

The graphical shapes that are available to ProvideX include the arc, pie, circle, line, rectangle and polygon. All of these shapes use the current attributes for the **'PEN'** and **'FILL'** mnemonics to determine their outline and fill characteristics.

'PEN' (*style,width,colour***)**
Defines the pen style to outline subsequently drawn shapes. Styles include 0=No pen, 1=Solid pen, 2=Dashed line, 3=Dotted line, 4=Dash-dot, 5=Dash-dot-dot. Width is in graphical units. Colour can be a colour code (0-15) or name of one of the ProvideX standard colours, the name of a user-defined colour, or an RGB setting (**RGB:** *n n n*). ProvideX colour codes / names are as follows:

0 - Black	8 - Dark Gray
1 - Light Red	9 - Dark Red
2 - Light Green	10 - Dark green
3 - Light Yellow	11 - Dark Yellow
4 - Light Blue	12 - Dark Blue
5 - Light Magenta	13 - Dark Magenta
6 - Light Cyan	14 - Dark Cyan
7 - White	15 - Gray

For syntax details, see the **'PEN'** mnemonic in the *Language Reference, p.628*.

Examples:

```
PRINT 'PEN'(1,10,6),
PRINT 'PEN'(1,1,"RGB: 192,192,192"),
PRINT 'PEN'(0,3,"Light Red"),
```

Note: The current **'PEN'** and **'FILL'** settings are reset after a **'CS'** mnemonic (clear screen) has been output.

'FILL' (*pattern***,***colour1***[,***colour2***])**
Defines the fill pattern to be used with subsequently drawn shapes. Two types of fill patterns are available, *standard* and *gradient* fill patterns.

Standard patterns include 0=No fill, 1=Solid fill, 2=Horizontal lines, 3=Vertical lines, 4=Crossed lines, 5=Diagonal bottom left to top right, 6=Diagonal top left to bottom right, 7=Diagonal crossed lines. Patterns 4 and 7 require two colours to be specified, the first for the lines and the second for the background.

Gradient patterns include 2=Top to bottom, 3=Left to right, 5=Top left to bottom right, 6=Bottom left to top right. Two colours must be specified for a gradient pattern, with the direction being derived from the code.

Colours can be a colour code (0-15) or name of one of the ProvideX standard colours (see **'PEN'** above), the name of a user-defined colour, or an RGB setting (**RGB:** *n n n*). If a second colour is specified, then both colours must be specified using the same format.

For syntax details, see the **'FILL'** mnemonic in the *Language Reference, p.605*.

Examples:

```
PRINT 'FILL'(1,"RGB: 192,192,192"),
PRINT 'FILL'(0,"Light Red"),
PRINT 'FILL'(3,"UserColour32","UserColour51"),
```

'ARC'(*x,y,radius,aspect,angle_1,angle_2***)**
Draws an arc centred at the given *x* and *y* graphical coordinates with the specified radius (also in graphical units) that extends from starting *angle_1* to *angle_2*. If an aspect ratio other than 1 is specified, then the arc is tilted into an elliptical shape. Since an arc is not a closed figure, fill patterns do not apply. For syntax details, see the **'ARC'** mnemonic in the *Language Reference, p.584*.

Example:

```
PRINT 'PEN'(1,2,1),'ARC'(400,200,100,1,0,120),
```

'PIE'(*x,y,radius,aspect,angle_1,angle_2***)**
Draws a pie slice centred at the given *x* and *y* graphical coordinates with the specified radius (also in graphical units) that extends from starting *angle_1* to *angle_1*. If an aspect ratio other than 1 is specified, then the arc is tilted into an elliptical shape. For syntax details, see the **'PIE'** mnemonic in the *Language Reference, p.630*.

Example:

```
PRINT 'PEN'(0,0,0),'FILL'(1,3),'PIE'(400,200,100,1,45,0),
```

'CIRCLE'(*x,y,radius,aspect*)
Draws a circle centred at the given *x* and *y* graphical coordinates with the specified radius (also in graphical units). If an aspect ratio other than 1 is used, then the circle is tilted into an elliptical shape. For syntax details, see the **'CIRCLE'** mnemonic in the *Language Reference, p.594*.

Example:

```
PRINT 'PEN'(1,3,1),'FILL'(2,6),'CIRCLE'(224,450,90),
```

'LINE'(*x1,y1,x2,y2* [,*x,y*…])
Draws a line (or lines) joining the sequential pairs of *x* and *y* graphical coordinates. Fill patterns do not apply to lines. For syntax details, see the **'LINE'** mnemonic in the *Language Reference, p.616*.

Example:
```
PRINT 'PEN'(1,3,1),'LINE'(0,@y(5),@x(80),@y(5)),
```

'POLYGON'(*x,y,x,y,x,y*…)
Draws a polygon by drawing joining the sets of *x/y* graphical coordinates. For syntax details, see the **'POLYGON'** mnemonic in the *Language Reference, p.631*.

Example:
```
PRINT 'PEN'(1,3,8),'FILL'(2,6),
PRINT 'POLYGON'(224,450,100,100,400,200,390,390),
```

This example creates an irregular four-sided figure by setting the coordinates for the four corners.

'RECTANGLE'(*x1,y1,x2,y2,* [*radius*])
Draws a rectangle defined by two sets of *x/y* graphical coordinates. An optional radius may be specified as a rounding factor for the corners. For syntax details, see the **'RECTANGLE'** mnemonic in the *Language Reference, p.633*.

Example:
```
PRINT 'PEN'(1,3,8),'FILL'(4,6,8),
PRINT 'RECTANGLE'(100,100,400,600),
PRINT 'RECTANGLE'(700,100,820,220,30),
```

Display Object Sampler

Following is some simple code to illustrate the variety of GUI **Display Objects**.

```
PRINT 'DIALOGUE'(0,0,80,25,"Shapes"),'SR','CS',
panelWidth=@X(MXC(0)+1)
panelHeight=@Y(MXL(0)+1)
PRINT 'PICTURE'(0,0,panelWidth,@Y(7),"people.bmp",4),
PRINT 'PEN'(0,0,0),'FILL'(2,4,6),
PRINT 'RECTANGLE'(0,@Y(7),panelWidth,panelHeight),
PRINT 'FONT'("Arial",3,"BI"),
PRINT 'TEXT'(@X(2),@Y(9.5),"Shapes"),
PRINT 'PEN'(1,3,2),
PRINT 'ARC'(@X(10),@Y(18),@X(5),1,0,180),
PRINT 'PEN'(1,2,3),'FILL'(1,3),
PRINT 'PIE'(@X(25),@Y(18),@X(5),1,25,340),
PRINT 'PEN'(1,1,0),'FILL'(2,2,10),
PRINT 'CIRCLE'(@X(40),@Y(18),@X(5),1),
PRINT 'PEN'(1,3,5),
PRINT 'LINE'(@X(50),@Y(15),@X(60),@Y(15),@X(50),@Y(21), @X(60),@Y(21)),
PRINT 'PEN'(1,3,7),'FILL'(4,7,1),
```

```
PRINT 'POLYGON'(@X(65),@Y(18),@X(68),@Y(15),@X(72),@Y(15), \
       @X(75),@Y(18),@X(72),@Y(21),@X(68),@Y(21)),
OBTAIN *
PRINT 'POP',
```

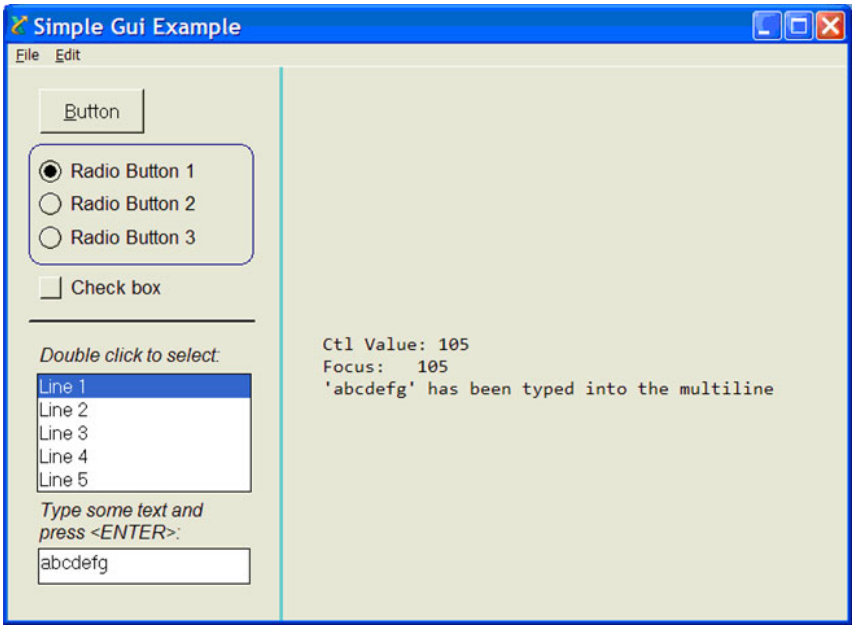
Example Programs

This section provides you with some working program code to illustrate the creation and implementation of ProvideX GUI controls and display objects.

Topics

GUI Program

Following is a short program for demostrating use of ProvideX **Control Objects**. It draws a new window within the ProvideX console, then creates several interactive controls within that window. At runtime, a message on the right will display the CTL value assigned to each control selected via the mouse.



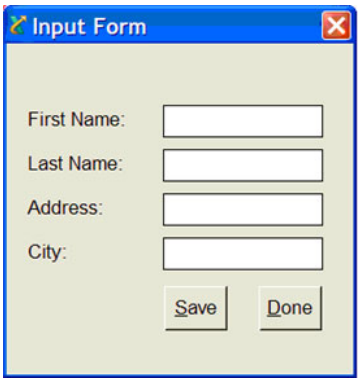

```
! Sample GUI program
! Create window and populate with controls
  PRINT 'DIALOGUE'(10,10,80,25,"Simple Gui Example",OPT="*-mXM"),'B?','SR','CS',
  PRINT 'FONT'("Arial,1,I"),'DF',
  MENU_BAR
       100,"-[&File,&Edit],F:[&Open=901,&Save=902,&Quit=4],E:[&Add=903,&Delete=904]"
  BUTTON 101,@(3,1,10,2)="&Button",TIP="Push the button"
  PRINT 'PEN'(1,1,"Dark Blue"),'FILL'(0,0),;
  PRINT 'RECTANGLE'(@X(2),@Y(3.5),@X(23.5),@Y(9),20),
  RADIO_BUTTON 102:1,@(3,4,20,1.5)="Radio Button 1",FNT="Arial,1"
  RADIO_BUTTON 102:2,@(3,5.5,20,1.5)="Radio Button 2",FNT="Arial,1"
```

```
RADIO_BUTTON 102:3,@(3,7,20,1.5)="Radio Button 3",FNT="Arial,1"
RADIO_BUTTON ON 102:1 ! Turn the default button on
CHECK_BOX 103,@(3,9.5,20,1)="Check box",FNT="Arial,1"
PRINT 'PEN'(1,2,0),'LINE'(@X(2),@Y(11.5),@X(23.5),@Y(11.5)),
PRINT 'TEXT'(@X(3),@Y(12.5),"Double click to select:"),
LIST_BOX 104,@(3,14,20,6);
LIST_BOX SET_FOCUS 104,204 ! set alternate CTL for focus
LIST_BOX LOAD 104,"Line 1/Line 2/Line 3/Line 4/Line 5/"
LIST_BOX WRITE 104,1
PRINT 'TEXT'(@X(3),@Y(19.5),@X(20),@Y(22),"Type some text and press <ENTER>:","W"),
PRINT 'PEN'(1,3,"RGB: 100 200 200"),'LINE'(@X(26),0,@X(26),@Y(25)),
MULTI_LINE 105,@(3,22,20,1);
MULTI_LINE SET_FOCUS 105,205 ! set alternate CTL for focus
! Initialize
CtlMsg$="",Event$=""
Lo_CTL=101,Hi_CTL=105,Current_ID=Lo_CTL;
SET_FOCUS Current_ID ! init focus
! Event Loop
WHILE 1
SET_FOCUS READ x;
IF x \
 THEN Current_ID=x \
 ELSE SET_FOCUS Current_ID ! Keep focus on the controls
OBTAIN (0,SIZ=1)'ME',*,'MN';
TheCTL=CTL
IF TheCTL=4 OR TheCTL=-1999 \
 THEN BREAK ! Exit the loop by pressing F4 or X in top corner of window
CtlMsg$="Ctl Value: "+STR(TheCTL),Event$=""
SWITCH TheCTL
CASE 101 ! Button
Event$="The button has been pressed"
BREAK
CASE 102 ! Radio button
RADIO_BUTTON READ 102,SUB_ID
CtlMsg$="Ctl Value: "+STR(TheCTL)+" Sub_ID: "+STR(SUB_ID)
Event$="A radio button has been selected"
BREAK
CASE 103 ! Checkbox
Event$="The check box has been clicked"
BREAK
CASE 104 ! Listbox
LIST_BOX READ 104,x$,mode$
IF mode$=$02$ \
 THEN Event$=x$+" has been selected from the list box" ! Select only by double
     click
BREAK
CASE 105 ! Multiline
MULTI_LINE READ 105,x$
Event$="'"+x$+"' has been typed into the multiline"
BREAK
CASE 204,205 ! Keep track when focus is on a listbox or multiline
Current_ID=TheCTL-100
```

```
BREAK
CASE 901,902,903,904
x=TheCTL-900;
MenuFunc$=TBL(x,"Unknown","Open","Save","Add","Delete")
Event$="'"+MenuFunc$+"' has been selected from the menu"
BREAK
CASE -1015 ! Tab forward
Current_ID++;
IF Current_ID>Hi_CTL \
 THEN Current_ID=Lo_CTL
SET_FOCUS Current_ID
BREAK
CASE -1016 ! Shift-Tab backward
Current_ID--;
IF Current_ID<Lo_CTL \
 THEN Current_ID=Hi_CTL
SET_FOCUS Current_ID
BREAK
DEFAULT
Event$=""
END SWITCH
PRINT @(30,12),CtlMsg$,'CL',@(30,13),"Focus:
     ",Current_ID,'CL',@(30,14),Event$,'CL',
WEND
! The end
PRINT 'CS',"Gogui.pgm example complete"
END
```

Form Input Program

Following is a simple form input program.




```
BEGIN
PRINT 'DIALOGUE'(0,0,33,15,"Input Form",'B?'+'FONT'("Arial,1")), \
      'SR','CS',
PRINT 'CURSOR'("OFF"),
PRINT 'TEXT'(@X(2),@Y(3),"First Name:"),
PRINT 'TEXT'(@X(2),@Y(5),"Last Name:"),
```

```
 PRINT 'TEXT'(@X(2),@Y(7),"Address:"),
 PRINT 'TEXT'(@X(2),@Y(9),"City:"),
 MULTI_LINE 100,@(15,3,15,1),LEN=15,OPT="",ERR=*NEXT
 MULTI_LINE 200,@(15,5,15,1),LEN=15,OPT="",ERR=*NEXT
 MULTI_LINE 300,@(15,7,15,1),LEN=15,OPT="",ERR=*NEXT
 MULTI_LINE 400,@(15,9,15,1),LEN=15,OPT="",ERR=*NEXT
 BUTTON 500,@(15,11,6,2)="&Save"
 BUTTON 600,@(24,11,6,2)="&Done"
 SET_FOCUS 100
 tab_list$="100200300400"
!
 WHILE 1
 OBTAIN (0)'ME',invar$,'MN';
 got_ctl=CTL,got_eom$=EOM
 SWITCH got_ctl
 CASE 4,600,-1999;
 EXITTO DONE
 CASE -1015;
 tab_list$=tab_list$(4)+tab_list$(1,3);
 SET_FOCUS NUM(tab_list$(1,3));
 BREAK
 CASE -1016;
 tab_list$=MID(tab_list$,-3)+tab_list$(1,LEN(tab_list$)-3);
 SET_FOCUS NUM(tab_list$(1,3));
 BREAK
 CASE 500;
 GOSUB SAVE_ROUTINE;
 BREAK
 END SWITCH
 WEND
!
DONE:
 MULTI_LINE REMOVE 100
 MULTI_LINE REMOVE 200
 MULTI_LINE REMOVE 300
 MULTI_LINE REMOVE 400
 BUTTON REMOVE 500
 BUTTON REMOVE 600
 PRINT 'CS',
 END
!
SAVE_ROUTINE:
 MULTI_LINE READ 100,first$
 MULTI_LINE READ 200,last$
 MULTI_LINE READ 300,address$
 MULTI_LINE READ 400,city$
 MSGBOX "Saving: "+SEP+"First: "+first$+SEP+"Last: "+last$+SEP+ \
       "Addr: "+address$+SEP+"City: "+city$
 RETURN
```

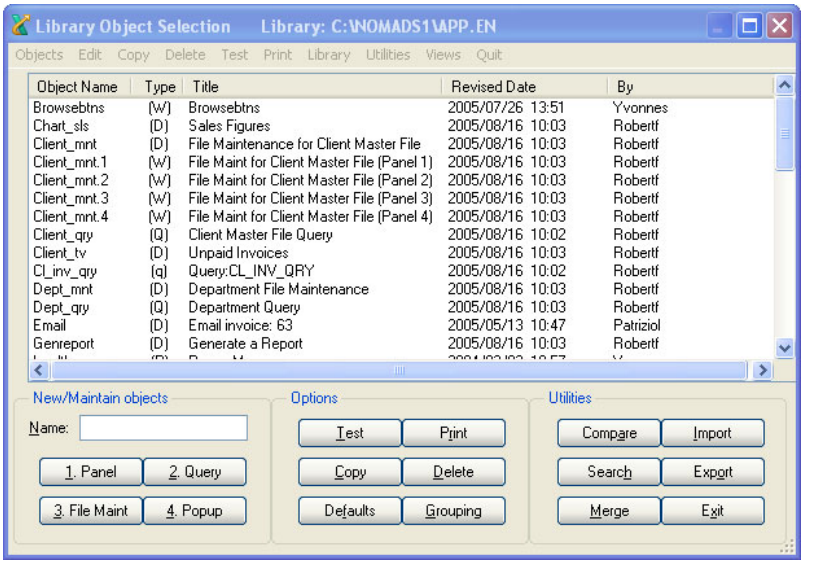
NOMADS

As mentioned, the easiest (and fastest) method for creating a graphical user interface in ProvideX is to use NOMADS (*Non-procedural Object Module Application Development System*). Following is an introduction to the tools and functionality of NOMADS. For complete documentation on the topics discussed below, refer to the *ProvideX NOMADS* manual. For information on building GUI components at the language level, see **Control Objects**, *p.150*, and **Display Objects**, *p.201*.

NOMADS is an application development facility that can be used in conjunction with ProvideX and other languages to create GUI applications. It also includes the ability to separate data, logic, and presentation text and images within the applications. A common data dictionary provides the ability to view and maintain data files without having to write any application code. NOMADS also allows for the integration of both NOMADS-designed applications with non-NOMADS designed applications.

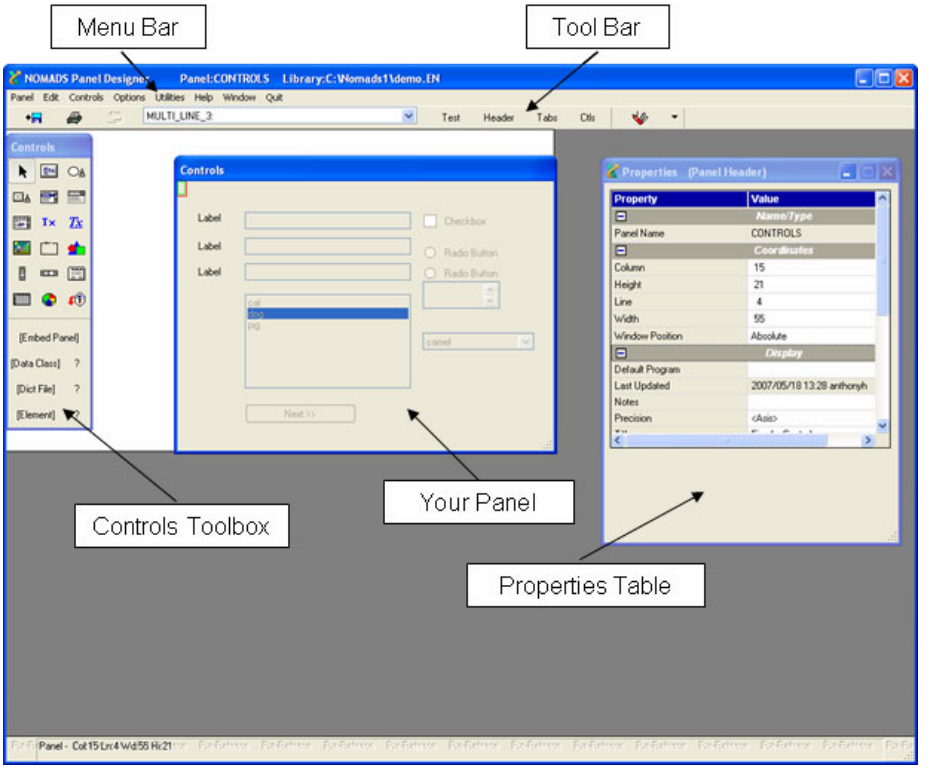
Development Environment

The toolset side of NOMADS provides an integrated working environment for building and assembling object definitions for the panels, dialogues, and windows to be used in your GUI application. These definitions are stored in keyed data files called *libraries*, which are accessed for editing via the Library Object Selection console.



When you develop components in NOMADS, the bulk of the programming is handled for you. Instead of typing lines of code in ProvideX to define size, placement, and functionality, you simply draw the objects using your mouse, then select the desired attributes from associated dialogues and menus.

The NOMADS Panel Designer is the main work area in NOMADS for designing panel layoutS and for drawing and defining **Control Objects**.



NOMADS Engine

At the heart of NOMADS is a powerful runtime engine, *winproc. This is the central controller that operates in ProvideX behind the scenes to process the information stored in library objects, generate GUI components, and execute associated event-handling logic. Your ProvideX application requires a **PROCESS** directive to run NOMADS-based components. At runtime, ProvideX converts this statement into a **CALL** to *winproc, which then generates the GUI for your ProvideX application.

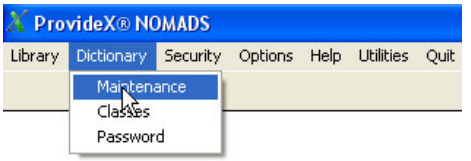
Dictionary-Based Development

NOMADS includes a set of *Rapid Application Development* (RAD) tools that can be used to easily create an integrated ***File Maintenance*** and ***Query*** system without writing a single line of code. These tools use the file and field definitions set up in a **data dictionary**, to create new applications or to convert existing legacy systems.

Data Dictionary Maintenance

The Data Dictionary Maintenance interface allows you to define data files by entering all the pertinent information for the elements (e.g. variable name, type, length, delimiter, etc…). Once this information is entered, the individual data dictionary information is embedded into the file, as well as written to the definition files, `providex.ddf` and `providex.dde`.

To access the interface for building and maintaining data dictionary definitions, select Dictionary > Maintenance from the menu bar on the NOMADS Session Manager.



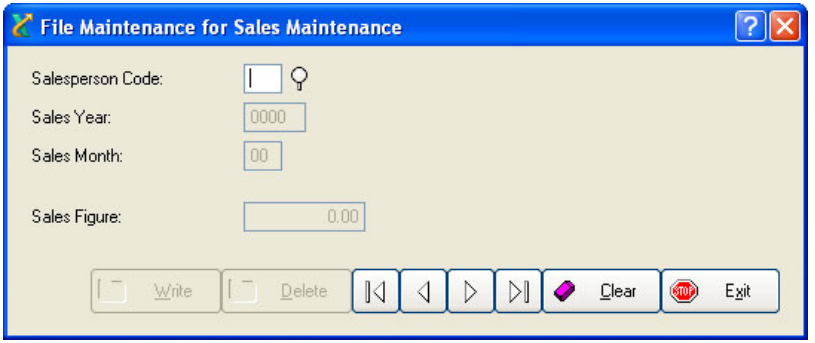
Alternately, you can type DD (or DD *tablename*) at the ProvideX Command prompt. Refer to the ProvideX *Data Dictionary* manual for detailed documentation on the Data Dictionary Maintenance interface.

File Maintenance Generator

Using this facility, you can automatically generate file maintenance panels with built-in editing and browse functionality based on data files defined in an embedded data dictionary.

Options are provided to create a single panel or a panel with multiple folders, to place edit and browse buttons at the bottom or side of the panel, to determine whether acknowledgement and confirmation messages are to be displayed, and to determine what update logic should be used.

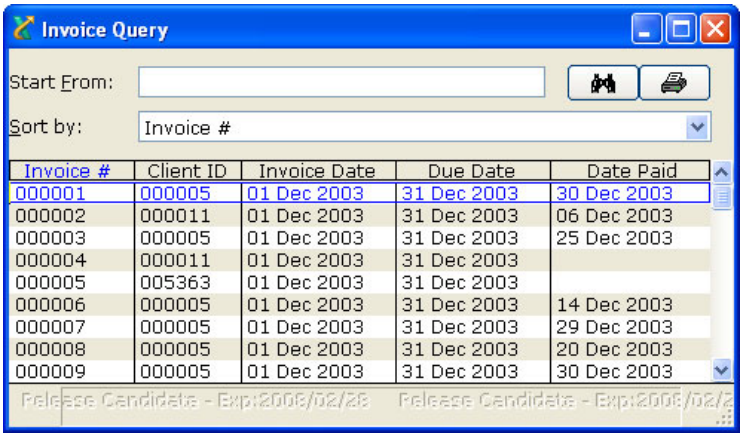
The following file maintenance panel was auto-generated using this facility.



Query Panel Generator

A query object in NOMADS consists of a panel that displays records from a data file and returns a value associated with a record selected by the user. Query objects may be based on an existing data dictionary definition, an ODBC data source, or a manually-defined file with no embedded dictionary.

The panels created using this facility are designed so that users can enter a starting point or switch to a maintenance program or sub-query, print a hard copy of the record list, select a different sort key, or switch to the record display area; e.g.,



7
Printing

PRINT (**?**) is described in *Chapter 5* as the directive for formatting and displaying data at the console (*screen*, *display*). However, as the keyword suggests, **PRINT** can also be used to render text and images on paper using a hardcopy device (***printer***). Actually, the definition of "printing" is not limited to hardcopy format. In ProvideX this can mean any process by which the data (reports, documents, or images) are sent to an output destination (device, interface, or file format). This chapter explains the various aspects of "printing" in ProvideX.

Topics

Unless the output is intended for immediate display, a **PRINT** statement must include a valid ***channel number*** to indicate a destination other than the ProvideX console. Channels are established using an **OPEN** directive, which identifies the connection to a specific device, interface, or file. See **Opening/Closing Devices and Files**, *p.87*.

Printing requires the use of both the **OPEN** directive and the **PRINT** directive:

OPEN to assign a channel number to an output destination

PRINT to format and direct output data to the destination defined via **OPEN**.

Printing Options

The ProvideX environment is designed for graphical application development, platform and device independence, and for the migration of older code to newer technologies. As a result, it comes equipped with a variety of **PRINT Destinations** to accommodate a broad range of application printing requirements.

ProvideX allows you to control all the steps in the printing process, but the end result is contingent on which method is used and on how the different parts work together. *So how do you determine which tools are right for the job?*

It is important that you understand the capabilities, limitations and the intended purpose of the different printing methods before you try to incorporate them into your ProvideX application:

- If your ProvideX application is designed run in Microsoft Windows, then you should be able to handle most hardcopy printing tasks using the standard Windows printer interface ***WINPRT***. Output may be controlled using various **Graphical Mnemonics** and through defaults established by the **WINPRT_SETUP** directive. See **Printing in MS Windows**, *p.221*.
- Legacy or converted character-based programs can use ***WINPRT*** just as easily, provided they are running under Windows. However, output that requires direct-to-device access (via raw escape sequences) should be sent to the ***WINDEV*** interface instead. See **Character-Based Printing**, *p.227*.
- UNIX/LinuX-based ProvideX programs can access either ***WINPRT*** or ***WINDEV*** via WindX. If you are printing over a network print spooler or use a local port connection, the specific UNIX/Linux print routines may be maintained using custom print drivers and link files (explained in the next point).
- When your application has specific output requirements that need to be customized for different environments and/or multiple devices, then you should make use of a custom print driver. This is a basic ProvideX file that allows you to maintain initialization code, mnemonics, and device-specific control sequences ***outside your application***. Devices, device drivers, and other settings can be easily maintained (and are interchangeable) when defined using a ***link***. See **Print Drivers and Link Files**, *p.230*
- **PRINT Destinations** are not limited to hardcopy. ProvideX has an extensive list of output choices, which include the generation of HTML, 24-bit colour bitmap images, PDF-compatible files, and the Print Preview facility. These are described under **Logical Printers**, *p.233*.
- Under WindX, ***WINPRT*** and ***WINDEV*** interfaces are accessible on the Windows system that issues the command. The **[WDX]** prefix is required to ensure that print jobs and dialogues are directed to the client. JavX and UltraFX thin-clients do not have access to local print facilities; however, PDFs may be created on the client machine, which can then be directed to the appropriate local printer. See **Printing via Thin-Clients**, *p.238*.

Character-Based versus Graphical Printing

In *Chapter 4*, the **PRINT** directive is first introduced within the context of a character-based implementation of ProvideX. This is also referred to as ***character mode***, and it defines syntax elements for generating ***line-oriented output*** on both the printer and screen.

By default, character-based **PRINT** statements will only advance by one line at a time when they print to the page. Automatic advance can be overridden by placing a ***hanging comma*** after the statement. ***WINPRT*** has been designed to minimize changes to existing text-based applications and allow virtually any printer to be used. For more information, see **Character-Based Printing**, *p.227*.

Graphical mode in ProvideX takes the original character mode behavior and has added syntax elements that are capable of graphical, ***page-oriented*** output. This implementation is ideally suited for GUI (graphical user interface) applications and is designed for sending graphical data to a Windows printer.

Note: Character and graphical mode methods may be mixed to produce desired output; however, be careful to ensure that the mixed elements are compatible. For example, using **'FONT'** and **'DEFAULT' or 'DF'** mnemonics to change character mode fonts can affect the x/y positioning of graphical output.

PRINT Destinations

Physical printers, printer interfaces, and output files can all be classified as **PRINT** destinations in ProvideX, if they are made available for use (in an **OPEN** statement). There are several options for defining how and where and in which format the output will be sent.

Most of the **PRINT** destinations listed below are discussed in further detail later in the chapter. Syntax options for logical file names (i.e., ***WINPRT*, *PDF***...) are provided in the *Language Reference,* Chapter 8.

WINPRT Enables standard API access to the Windows print subsystem/spooler (*WindX or Windows only*); e.g.,

```
OPEN (1)"*WINPRT*"
```

This provides access to the standard Windows Printer dialogue for both graphical and character-based data; however, raw escape sequences are not permitted in graphical printing. For raw printing, see ***WINDEV***. Use the **WINPRT_SETUP** directive to establish default printing options for the Windows printer; i.e., paper size, margins, copies, etc. For more information, see **Printing in MS Windows**, *p.221*.

WINDEV Sends character-based data to the Windows print subsystem / spooler (*WindX or Windows only*); e.g.,

```
OPEN (1)"*WINDEV*;HP Laser Jet on \\Main_Server\HPLaser"
```

This permits the Windows equivalent of sending data to a directly-connected printer. It provides an interface to the Windows API in a *pass-through* mode that accepts raw data and device-specific escape sequences. For more information, see **Raw Printing**, *p.228*. For graphical printing, use ***WINPRT***.

PDF Generates a PDF file from ProvideX output; e.g.,

```
OPEN (1) "*pdf*;FILE=/tmp/pvx.pdf; FORM=Letter:8.5in:11in"
```

If the file name is omitted, a dialogue appears for users to specify the path, PDF name, and properties. See **Logical Printers**, *p.233*.

VIEWER Allows users to preview reports (*Windows/WindX*);

```
OPEN (CHAN)"*VIEWER*"
```

This renders output to a screen "viewer" exactly as it would print out in hardcopy format via ***WINPRT***. See **Logical Printers**, *p.233*.

BITMAP Generates 24-bit colour bitmap images in memory (*WindX or Windows only*); e.g.,

```
OPEN (12)"*bitmap*"; PRINT 'CS'
```

BITMAP contents can be accessed via the **'PICTURE'** mnemonic and the **SAVE FILE** directive. See **Logical Printers**, *p.233*.

HTML Generates HTML-formatted reports; e.g.,

```
OPEN (1,OPT="FILE=Sample.htm;SHOW;FONT=Courier
New;TITLE=Sample;BACK=FFFFFF;TEXT=000000")"*HTML*"
```

Reports that are formatted using normal fixed fonts may be read using any HTML viewer (browser). The system prompts for a file name to store the resulting HTML document. See **Logical Printers**, *p.233*.

UNC Name Specifies the location of a shared server / printer on the network (*WindX or Windows only*); e.g.,

```
OPEN (1)"\\Print_Server\HP_Laser"
```

The output device can be any shared resource that is identified via Universal Naming Convention. This format accepts raw data along with printer escape sequences, but does not support graphical printing.

LPT Access Specifies the port number of a directly-connected printer device; e.g.,

```
OPEN (1)"*LPT1*"
```

This format accepts raw data along with printer escape sequences, but does not support graphical printing. See **Raw Printing**, *p.228*.

Printing in MS Windows

ProvideX for Windows include two logical device names for accessing Windows printers: ***WINPRT*** and ***WINDEV***. When used in an **OPEN** statement, these devices will invoke the print subsystem API at runtime. Logical device names for printing to ***file types*** in Windows are described under **Logical Printers**, *p.233*.

The standard interface, ***WINPRT***, is designed to handle typical hardcopy print requests. It allows your application to print any variety of character and graphical output (i.e., bitmap images and proportional fonts) and employ a full suite of **PRINT** options and mnemonics.

WINDEV supports the direct-to-device printing methods associated more with legacy and / or converted applications. This interface accepts the *raw* print escape sequences that are otherwise ignored in the Windows print system. It sends output using a *pass-through* mode, which tells the printer driver to read the raw data and queue the job. For more information, see **Raw Printing**, *p.228*.

This section focuses on the access and control of hardcopy print destinations from ProvideX in Windows (and WindX), primarily through ***WINPRT*** and ***WINDEV***.

Selecting a Printer

When ***WINPRT*** (or ***WINDEV***) are used in an **OPEN** statement without options or queue names, the user will be presented with the default Windows print manager ***selection dialogue*** at runtime; e.g.,

```
OPEN (1)"*WINPRT*"
```

If a print queue name is specified along with ***WINPRT*** (or ***WINDEV***), then the dialogue will not be displayed and output is sent to the printer automatically at runtime. To assign a specific printer, use the print queue's name as it appears in the *Printers* folder of the *Control Panel* on the local system (this is assigned to the printer when it is first installed on the OS). Omit any mention of ports. For instance, if the *Control Panel* folder records the name as `HP LasertJet on LPT1`, simply use `HP LaserJet` as the queue name in your directive; e.g.,

```
OPEN (1)"*WINPRT*;HP LaserJet"
OPEN (2)"*WINPRT*;Bills Desk - Canon;orientation=landscape;copies=3"
OPEN (3)"*WINDEV*;HP LaserJet;copies=2"
```

ASIS

Use the **ASIS** keyword to access the most recently selected printer and its previous settings. ***All settings for queue options will be identical to the previous selection.*** Users do not have access to the printer selection dialogue at runtime and cannot alter the settings; e.g.,

```
OPEN (2)"*WINDEV*;ASIS"
```

WINPRT allows you to override printer settings with the **ASIS** option, but the changed properties must be valid for the appropriate driver.

```
OPEN (1)"*WINPRT*;ASIS;copies=5;offset=500:500"
OPEN (2)"*WINPRT*;ASIS;file=hello.txt"
```

This option does not apply to ***WINDEV***.

DEFAULT

Use the **DEFAULT** keyword to **OPEN** the printer that is currently set as the default in Windows. Again, users will not have access to the printer selection dialogue and can't alter the settings at runtime.

For example, if the default printer is *"Bills Desk - Canon"*, then the print jobs in the examples below will be sent to that Canon printer:

```
OPEN (1)"*WINPRT*;DEFAULT"
OPEN (2)"*WINPRT*;DEFAULT;copies=5;orientation=landscape;offset=300:600"
OPEN (3)"*WINDEV*;DEFAULT"
```

WINPRT allows you to override printer settings with the **DEFAULT** option, but the changed properties must be valid for the appropriate driver. This option does not apply to ***WINDEV***.

NORMAL

Use the **NORMAL** keyword with your **OPEN** directive when you want the user to be presented with a printer selection dialogue that includes a page range option (*but no paper size or source tray options*); e.g.,

```
OPEN (30)"*WINDEV*;NORMAL"
```

All settings will be displayed in the printer selection dialogue when the user sees it. Users can accept the current settings or alter them.

SETUP

Use the **SETUP** keyword in your **OPEN** directive when you want the user to be presented with a printer selection dialogue that includes paper size and source tray options (*but no page range option*); e.g.,

```
OPEN(1)"*WINPRT*;SETUP"
OPEN(2)"*WINDEV*;SETUP"
OPEN(3)"*WINPRT*;SETUP;range=1:5"
```

All settings will be displayed in the printer selection dialogue – users can accept the current settings or alter them.

Note: The **SETUP** and **NORMAL** options must be included with ***WINPRT*** in order to display a printer selection dialogue with preset printer properties. For example, `OPEN (14)"*WINPRT*;NORMAL;orientation=landscape"`.

Initializing a Windows Printer

Before sending data to ***WINPRT*** you should set the initial font and point size of the printer to a known state, and make it the default (using the **'FONT'** mnemonic and the **'DEFAULT' or 'DF'** mnemonic); e.g.,

```
OPEN(chan)"*winprt*"
PRINT (chan)'FONT'("Courier New",-10),'default',
```

A negative number defines the absolute point size; e.g.,

```
PRINT (1)'FONT'("Arial",-7),'DF')
```

All standard **PRINT** statement data will use the selected font, with spacing based on logical CPI values (pitch). Use the following guide to equivalent measures:

Point Size	*= Logical LPI*	*= Logical CPI*
-12	=6 LPI	=10 CPI
-10	=7.2 LPI	=12 CPI
-7	=10 LPI	=16 CPI

Column and row addressing are based on the default font. Switching fonts will not alter the column or row addressing unless you change the default font. It is recommended that you use the **'CPI'** and **'LPI'** mnemonics to set text alignment after issuing a **'DEFAULT' or 'DF'** to ensure that the output will have consistent columns/lines per page regardless of the font chosen.

Example 1:

```
OPEN (1)"*winprt*"
PRINT (1)'FONT'("Courier New,-12"),'DF','CPI'(10),'LPI'(6),
```

The above example sets default font and then sets CPI (characters per inch) and LPI (lines per inch) to establish line/column alignment for the output. Now, regardless of font size, the location of the output will remain consistent.

Example 2:

```
0010 FONT$="Times New Roman",COLS_REQD=132,REALLY_SMALL_FONT=200
0020 CHAN=UNT; OPEN (CHAN)"*WINPRT*"
0030 PRINT (CHAN)'FONT'(FONT$,-12),'DF', ! Set base to 10 CPI
0040 CUR_COL=MXC(CHAN)+1,INCHES_WIDE=CUR_COL/10
0050 PRINT (CHAN)'FONT'(FONT$,CUR_COL/REALLY_SMALL_FONT),'DF',
0060 PRINT (CHAN)'CPI'(COLS_REQD/
0070 FOR Z=0 TO 12
0080 PRINT (CHAN)@(Z*10),"Hello
0090 NEXT
```

The above example scales the text to the page based on 132 characters. It is inadvisable to reset the default font or alter the initial cpi/lpi settings after beginning to output the data as this will alter the relative positioning of the output.

To further assure proper alignment of text, the **OPEN** directive has an option clause **FORCE6X10=YES|NO**; e.g,

```
OPEN(1,OPT=FORCE6X10=YES")"*winprt*"
```

When set to YES, this option automatically adjusts the column width to 60% of the line height defined in font size specifications. This setting solves some minor alignment issues when printing proportional fonts to a graphical print device. Historically, most fixed-width fonts adhered to a 6x10 ratio of 6 lines to the inch and 10 characters per inch; i.e., for a character width that is 60% of the line height.

Setting Up Defaults

Depending on the printing requirements, some applications may want certain print criteria (i.e., paper size, margins, copies, etc.) to remain constant regardless of how specific output data is to be formatted. ***WINPRT*** allows you to configure various printer settings; however, if you want to establish defaults that remain in effect throughout the application, it is better to use **WINPRT_SETUP**.

The **WINPRT_SETUP** directive is designed to read / define a default printer and its properties. It can also be used to produce a list of the printers that are currently available to the system. This functionality has multiple uses in printer selection and initialization routines and permits some repetitive options to be removed from specific printer **OPEN** statements. To enable access to printers defined on a Windows server under WindX, use the keyword **SERVER** in the **WINPRT_SETUP** statement; e.g., `WINPRT_SETUP SERVER ...`

WINPRT_SETUP READ *var$*

Returns the name of the current default printer.

WINPRT_SETUP WRITE *printer$*

Changes the current default printer.

WINPRT_SETUP INPUT *var$*

Prompts the user with the Windows printer selection dialogue and returns the name of the printer selected by the user.

WINPRT_SETUP LIST *var$*

Generates a list of all the printers defined in the system.

WINPRT_SETUP DIRECTORY *var$*

Generates a list of all printers with just the printer portion of the names (excludes the "`ON Device`" portion).

WINPRT_SETUP READ PROPERTIES *var$*

Returns the property settings associated with your current default printer.

WINPRT_SETUP WRITE PROPERTIES *settings$*

Changes one or more of the default printer properties. Multiple options can be specified by separating settings with semi-colons.

COLLATE=YES | NO | AUTO
COLOUR=YES | NO
COPIES=*num*
DUPLEX=*num*
FILE=+*filename***|-** *filename*
MARGINS=*left***:***top***:***right***:***bottom*
OFFSET=*x***:***y*
ORIENTATION=PORTRAIT|LANDSCAPE
PAPERLENGTH=*num*(1/10 *mm*)
PAPERSIZE=*num*
PAPERWIDTH=*num*(1/10 *mm*)
QUALITY=*num*
RANGE=*from***:***to*
RESOLUTION=*num***:***num*
SCALE=*num*(%)
SOURCE=*num*
TRUETYPE=*num*

For an expanded list of options, see **WINPRT_SETUP Properties**, *p.372* in the *Language Reference*.

WINPRT_SETUP *Examples*

Default properties are determined by the individual printer manufacturers. The following example lists the properties set for the current default printer:

```
->OPEN (1)"*WINPRT*"
->WINPRT_SETUP READ PROPERTIES PROP$
->?PROP$
RANGE=ALL;COLLATE=NO;COPIES=1;ORIENTATION=PORTRAIT;PAPERSIZE=1;SOURCE=1;
RESOLUTION=300:300;OFFSET=0:0;TRUETYPE=2;DRIVER=WINSPOOL
```

Use the following statement to set up a 1-inch margin on all sides of the page:

```
WINPRT_SETUP WRITE PROPERTIES "MARGINS=1000:1000:1000:1000"
```

The currently selected default printer name can be determined as follows:

```
-: WINPRT_SETUP READ PRTR$
-: ?PRTR$
HPLaserJet III on \\machine_1\hp
```

Aborting a Windows Print Job

The **'AB'** mnemonic is used to abort a print job from the Windows spooler; e.g.,

```
PRINT (30)'AB' ! To cancel the current job
```

Look for the name (title bar caption) of your current ProvideX window to find your print job in the Windows print spooler job listing. How much gets printed after the abort is determined by the printer driver and by the print spooler options *Start Printing After the First Page* or *Start Print After the Last Page*.

Graphical Printing

The ProvideX environment includes several **PRINT** destinations for handling true graphical (bitmap / vector) data. The standard interface to the Windows print manager ***WINPRT***, is designed for printing ***hardcopy*** from graphical output. Other options include ***PDF***, ***VIEWER***, and ***BITMAP***.

Graphical mode is not just for images – the graphical capabilities of the ProvideX environment allow applications to deliver many more printing options than from a strictly character-based environment. Because graphical printing is ***page-oriented*** rather than line-oriented, the line positioning (column / row) is dynamic. ProvideX applications that are set up to print or display graphical output can access any part of the page at any time – top-to-bottom ordering is not required.

ProvideX also allows you to mix and match fonts and attributes, apply **Graphical Mnemonics** as well as maintain older **Character-Based Printing** formats all on the same page.

Graphical Mnemonics

All graphical printing from a ProvideX application requires the use of specific API calls to the Windows print manager – only certain ***predefined*** graphical mnemonics are mapped to Windows API calls for this purpose; e.g., **'ARC'**, **'CIRCLE'**, **'FONT'**, **'FRAME'**, **'LINE'**, **'PICTURE'**, **'PIE'**, **'POLYGON'**, **'RECTANGLE'**, and **'TEXT'**. For the complete list of graphical mnemonics in ProvideX, see **Graphical Display/Printer**, *p.580* and **Graphical Printer**, *p.581* in the *Language Reference*.

Positioning within a graphical mnemonic requires the use of X and Y coordinates. These are often derived using the **@X( ) / @Y( )** functions, which convert column and row values into graphical units. When a graphical mnemonic is applied against a channel, the channel number should be passed as an argument to the **@X( ) / @Y( )** functions. The following routine illustrates how to use some of the mnemonics available for printing graphics in ProvideX:

```
! Printing Graphics
PRINT 'CS',
ch=UNT ! OPEN (ch)"*viewer*"
PRINT (ch)'FONT'("Times New Roman",3,"BI"),'BLUE',
PRINT (ch)'TEXT'(@X(0,ch),@Y(0,ch),@X(70,ch),@Y(5,ch),"Printing
      Graphics","C"),
PRINT (ch)'PEN'(1,2,4),'ARC'(@X(7,ch),@Y(5,ch),@X(1,ch),1,0,180),
PRINT (ch)'PEN'(1,3,1),'FILL'(1,4),'RECTANGLE'(@X(2,ch),@Y(9,ch),@X(12,ch),
      @Y(13,ch)),
PRINT (ch)'PEN'(1,1,7),'LINE'(@X(2.5,ch),@Y(9.5,ch),@X(11.5,ch),@Y(9.5,ch),
      @X(11.5,ch),@Y(12.5,ch),@X(2.5,ch),@Y(12.5,ch),@X(2.5,ch),
      @Y(9.5,ch)),
PRINT (ch)'PEN'(1,2,4),'FILL'(7,1),'CIRCLE'(@X(54,ch),@Y(10,ch),@X(6,ch),1.8),
L=@X(24,ch),R=@X(33,ch),S=@X(4,ch),X=@Y(6,ch),Y=@Y(14,ch)
PRINT (ch)'PEN'(1,3,2),'FILL'(1,12),'POLYGON'(L-S*2,X,R+S*4,Y,L,Y,R-S,X,L,X),
PRINT (ch)'PICTURE'(@X(0,ch),@Y(15,ch),@X(30,ch),@Y(25,ch),"*win/nomads2",2),
```

Note: Only *predefined* mnemonics are mapped to Windows API calls from ProvideX. User-defined mnemonics (via the **MNEMONIC** directive) are not accepted.

Printing with Colours

You can use all foreground and background colours with colour printers. Use the **'PEN'** and **'FILL'** mnemonics to control the colour settings for graphics mnemonics. The following routine prints each of the available 16 colours.

```
0010 ! Colour Printing
0020 FONT$="MS Sans Serif",COLS_REQD=80
0030 CHAN=UNT; OPEN (CHAN)"*WINPRT*"
0040 PRINT (CHAN)'FONT'(FONT$,-12),'DF', ! Set base to 10 CPI
0050 CUR_COL=MXC(CHAN)+1; IF CUR_COL=0 THEN CUR_COL=80
0060 INCHES_WIDE=CUR_COL/10
0070 PRINT (CHAN)'FONT'(FONT$,CUR_COL/COLS_REQD),'DF',
0080 PRINT (CHAN)'CPI'(COLS_REQD/INCHES_WIDE),'LPI'(6),
0090 FOR Z=0 TO 15
0100 INTENSITY$=TBL(Z>7,"","Light ")
0110 COLOUR$=TBL(MOD(Z,8),"Black","Red","Green","Yellow","Blue","Magenta",
0110:"Cyan","White")
0120 COLOUR_MNEM$=ESC+"F"+CHR(ASC("0")+Z)
0130 PRINT (CHAN)@(0),COLOUR_MNEM$,INTENSITY$,COLOUR$
0140 NEXT
```

Character-Based Printing

The syntax elements available for character-based printing in ProvideX are intended primarily to service ***converted*** or ***legacy*** applications that are limited to ***line-oriented*** output. If a ProvideX application is designed to run on Windows, then it should employ complete graphical (display and print) functionality. See **Printing in MS Windows**, *p.221*.

When character-based printing is required, ProvideX allows you to format and print the output in several different ways, depending on the code, the operating system, and on which printer interface is used.

Standard Printing

The standard interface to the Windows print manager, ***WINPRT***, is designed to minimize any changes to print from legacy applications. This allows text-only ProvideX applications to have transparent access to any print destination that is available to the current Windows system.

Since the source of the output is character-based, the hardcopy will also be character-based. However, ***WINPRT*** does not accept any ***raw*** control sequences that may be required to print from legacy or converted applications (see below).

Raw Printing

If an application includes printer-specific control sequences in its print routines, the output will be stripped out by the Windows print system. Therefore, the standard ProvideX printer interface, ***WINPRT***, cannot be used for this type of output data.

Use one of the following methods for handling device-specific instructions and direct-to-output printing in ProvideX:

- ***WINDEV***, interface for printing *Raw Mode* in Windows
- **UNC Name**, Universal Naming Convention
- **LPT Access**, direct to the local LPT port.

WINDEV allows you to send output data via Windows in a *pass-through* mode. This instructs the printer driver to accept the raw escape sequences and queue the printer. However, these instructions will only be valid if they are *supported* by the specified printer and print driver. Refer to the printer's documentation (supplied by the manufacturer) for the specific PCL (Printer Control Language) syntax. Raw printing mode may be controlled using the **'RP'** parameter.

Sets of mnemonics and / or escape sequences used to initialize different printers can be maintained outside your ProvideX application via **Print Drivers and Link Files**.

Fixed-Pitch Fonts

Fixed-pitch fonts are easily applied via ***WINPRT***. The example below adjusts the font to allow for the number of columns in COLS_REQD. The **MXC( )** and **MXL( )** functions return maximum available column and line for the channel, based on the current default settings for paper size, printable area, offset, margin, default font height, width, pitch.

Example:

```
0010 LOOP=0,COLS_REQD=132,FONT$="Courier New"
0020 CHAN=UNT; OPEN (CHAN)"*WINPRT*"
0030 PRINT (CHAN)'FONT'(FONT$,-12),'DF', ! Set Font to 10 CPI
0040 PRINT (CHAN)'FONT'(FONT$,MXC(CHAN)/COLS_REQD),'DF', ! Scale it
0050 IF MXC(CHAN)<COLS_REQD THEN IF LOOP++<5 THEN GOTO 0040
0060 PRINT MXC(CHAN),LOOP
```

This method of printing works with any Windows printer and with no adjustments for legacy code. To control fonts on a text mode device, send raw escape sequences to the printer using ***WINDEV***, UNC, or LPT ports. However, the choice of fonts using the direct-to-output method is limited to the fonts supported by the given printer. For further information, see **MXC( ) / MXL( )**, *Language Reference p.486*.

Proportionally-Spaced Fonts

WINPRT also allows the use of proportional fonts with text-only reports and can scale them to fit the page. When applying proportional fonts to character-based reports:

- Fields should be printed individually to ensure proper alignment.
- Decimal alignment is supported for the **STR( )** function and format masks.
- Typical line drawing characters (_ , **-** , **=**) can be replaced by solid lines automatically using the **'+S' & '-S'** mnemonic.

Applications should print fields individually for proper alignment; i.e., use the `@(col)` position for each element or field. A dimensioned string like `DIM X$(80); X$(1)="Customer",X$(46)=" Balance"` would not be properly aligned. String variables are left-justified at the specified column based on the default font. Decimal alignment is supported for the **STR( )** function or **Data Format Masks**. Numeric variables print right-justified with decimal alignment, provided a format mask is used.

Example:

```
10 LOOP=0,COLS_REQD=78,FONT$="MS Sans Serif"
20 CHAN=UNT; OPEN (CHAN)"*WINPRT*"
30 PRINT (CHAN)'FONT'(FONT$,-12),'DF', ! Set Font to 10 CPI
40 PRINT (CHAN)'FONT'(FONT$,MXC(CHAN)/COLS_REQD),'DF', ! Scale it
50 IF MXC(CHAN)<COLS_REQD THEN IF LOOP++<5 THEN GOTO 0040
60 PRINT (CHAN)@(0),"Customer",@(45)," Balance"
70 PRINT (CHAN)@(0),"Mustangs Unlimited",@(45),19.67:"####,##0.00-"
80 PRINT (CHAN)@(0),"CJ Pony Parts Inc.",@(45),289.67:"####,##0.00-"
```

However, the following style of coding ***will not*** work properly when using a proportionally-spaced font because the variables are not printed individually:

```
10 DIM X$(80); X$(1)="Customer",X$(46)=" Balance"
20 PRINT (CHAN)X$
30 DIM X$(80); X$(1)="Mustangs Unlimited",X$(46)=STR(19.67:"####,##0.00-")
40 PRINT (CHAN)X$
```

You can also use text justification mnemonics to override the normal handling of string and numeric data. Normally, the following mnemonics are required only when the individual fields are grouped into a single variable that is being sent to a printer using a proportionally-spaced font.

'JC' Justify Centre
'JD' Justify Decimal-Aligned
'JL' Left-Justify Text
'JN' Right-Justify for Numeric
'JR' Right-Justify Numeric
'JS' Left-Justify String

The **'+S'** mnemonic automatically replaces the underscore, dash and equals sign (_ **-** and **=**) with solid lines. The solid line technique also only applies to fields that are printed separately. See **'+S' & '-S'**, *Language Reference p.635*.

Example:

```
10 DIM A$(10,"_"),B$(10,"-"),C$(10,"=")
20 LET CHAN=UNT; OPEN (CHAN,ERR=*END)"*winprt*"
30 PRINT (CHAN)'FONT'("MS Sans Serif",1),'DF',
40 PRINT (CHAN)'-S',@(0),A$,@(12),B$,@(24),C$,@(36),"No solid lines"
50 PRINT (CHAN)'+S',@(0),A$,@(12),B$,@(24),C$,@(36),"Solid lines"
60 PRINT (CHAN)'+S',@(0),A$+" "+B$+" "+C$,@(36),"No solid lines"
```

Print Drivers and Link Files

While it is possible to communicate with a printer directly, most output data is delivered through one or more interfaces, software liaisons between the source application and the printer.

Printer manufacturers generally supply their own interfaces (printer drivers) to make it easier for operating systems to communicate with their products. Windows supplies its own interface (print subsystem API) for directing access to the various printer drivers installed on the system. It also handles other tasks, including ***print manager***, which allows users to select from a list of available printers to which they may send their output, and ***spooler***, which acts as a buffer / queue for multiple print jobs so that printers can receive data at their own rate.

There are different printing interfaces within the ProvideX environment itself. ProvideX includes several ready-made interfaces (***WINDEV***, ***WINPRT***, ***PDF***, etc.) that allow you to print directly from an application. However, you can also assemble your own interfaces for tailoring print requirements to different environments: a **Custom Driver** consolidates mnemonics and (legacy) control sequences into a single executable file to be called from the main program whenever the target device is opened; a **Link File** allows you maintain a common printer name (***alias***) that will represent all possible print destinations from your application.

Note: While this chapter focuses primarily on printers, custom drivers can be created for use with any device to be accessed by the ProvideX environment. For more information, see **Device Drivers**, *p.390*.

Custom Driver

A custom driver is a ProvideX program that is automatically loaded and run when the target device file is opened. They reside outside the main program in the ProvideX ***DEV** directory (`/pvx/lib/_dev`). Driver names are limited to 12 characters in length.

Most custom drivers are used to consolidate a set of **Mnemonics** or the escape sequences that apply to a specific file or printer. They are often used to initialize printers in **Character-Based Printing**, but they can also be used to redirect output from one device / file to another, prompt the user for additional information, and / or execute operating system commands.

The **DEFPRT** directive at the beginning of the code tells Providex that the channel is a printer. For example, the following printer driver, ***DEV/HPLASER**, defines the maximum columns and rows for the printer, then defines all of the mnemonics and initializes the printer:

```
0010 ! HP Laser Jet: 10 cpi, 6 lpi, Portrait
0020 DEFPRT (LFO)80,60
0030 MNEMONIC (LFO)'*C'=ESC+"E" ! Close printer mnemonic -- Resets all
0040 MNEMONIC (LFO)'FF'=$0C$ ! <formfeed>
0050 MNEMONIC (LFO)'CR'=$0D$ ! <cr>
0060 MNEMONIC (LFO)'LF'=$0D0A$ ! <cr><lf>
0070 MNEMONIC (LFO)'NP'=ESC+"&k0S":80,0 ! 10 cpi
0080 MNEMONIC (LFO)'SP'=ESC+"(s12H":96,0 ! 12 cpi
0090 MNEMONIC (LFO)'CP'=ESC+"&k2S":132,0 ! 16.66 cpi
0100 MNEMONIC (LFO)'LT'=ESC+"&l12D":0,120 ! 12 Lines per inch
0110 MNEMONIC (LFO)'L8'=ESC+"&l8D":0,80 ! 8 lines per inch
0120 MNEMONIC (LFO)'L6'=ESC+"&l6D":0,60 ! 6 lines per inch
0130 MNEMONIC (LFO)'PM'=ESC+"&l0O" ! Portrait mode
0140 MNEMONIC (LFO)'LM'=ESC+"&l1O" ! Landscape mode
0150 MNEMONIC (LFO)'RM'=MNM('PM',LFO)+MNM('NP',LFO)+MNM('L6',LFO):80,60
0160 X$=MNM('PS',0); IF X$<>"" THEN MNEMONIC (LFO)'PS'=X$ ! Start Slave
0170 X$=MNM('PE',0); IF X$<>"" THEN MNEMONIC (LFO)'PE'=X$ ! End Slave
0180 X$=FIB(LFO); IF X$(19,1)="S" THEN LOCK (LFO,ERR=*NEXT)
0190 PRINT (LFO,ERR=0200)'*C','RM',
0200 END
```

The system variable **LFO** is used to identify the affected channel/file. It will contain the value of the last file opened, which in the case of a device driver, is the channel for which the driver is responsible.

Once created, ***DEV/HPLASER** could be accessed in an application as follows:

```
open (1)"*WINDEV*;HP Laser Jet"
call "*dev/hplaser"
```

For general information on device drivers in ProvideX, see **Device Drivers**, *p.390*.

Link File

A link file is a device descriptor that contains a single line of code that simply points to a device or device driver; e.g.,

```
[Pvxdev]*WINDEV*;Generic/Text Only;FILE=ABC.PRN                hplaser
```

Where the format of the device descriptor is as follows:

Location	Length	Contents
1	8	"[Pvxdev]": Device descriptor ID.
9	60	True path to the device.
69	12	Device type (name of driver).
71	184	Reserved for future (blank).

When ProvideX opens a link file it redirects to the path specified in bytes (9,60). This is the actual file to be opened. Once the file is opened, ProvideX will automatically **CALL** the device driver specified in bytes (69,12).

The link file name serves as an "alias" that can be used in place of where the associated file or device information appears in an application's **OPEN** statement. For example, if the HP Laser Jet device and driver (described above) were defined in a link named "LP", the device could then be accessed from the application as follows:

```
open (1)"LP"
```

The application now sends data to the printer defined via "LP". Provided "LP" is always used in the **OPEN** statement, the contents of the Link file itself can be changed – tailored to any device or driver that will be used with the application. If device information changes, the application will direct output to the new destination, provided the contents of the "LP" Link file is updated, the device exists, and the custom driver is located in the ***DEV** directory.

Link files are simply plain text files that may be produced using any text editor. ProvideX also supplies a *prompt-driven utility* (***UCL**) that simplifies the creation and maintenance of link files. This utility allows you to choose between three slightly different link header types (File, Device, or Attached Printer):

[PVXLNK]	To point to a file without calling a program. For example, if you moved the GLDETAIL file to another directory, you could create a link file using the original name (GLDETAIL) then, with the file name portion of the link, point to the new location of the *actual* GLDETAIL file.
[PVXDEV]	To open a file as well as call a program. Commonly used to associate printers and printer drivers.
[PVXAPR]	Specifically for pointing to a printer that is physically attached to the computer. This issues an additional escape sequence wrapped around every line. ProvideX opens a file, calls the program in question, and automatically prints the **'PS'** mnemonic to the printer. It will also issue a **'PE'** mnemonic when it attempts to close the file.

***UCL** prompts for the device information that will appear on the descriptor line. The following examples illustrate some device information that could be written to Link files via the ***UCL** utility for various printer setups:

```
Link File Name: LP
Other File to Open: LPT1
Prog to Call: epson
```

```
Link File Name: /MYAPP/P1
Other File to Open: >lp -d queuename -c -s 2>/dev/null
Prog to Call: hplaser
```

```
Link File Name: Googles
Other File to Open: /tmp
Prog to Call: spooler
```

```
Link File Name: WINDOWS
Other File to Open: [wdx]*winprt*
Prog to Call: myprog
```

Pages and Form Feeds in Printing

You can access an entire page and print to it. However, you can only print to the current page at any one time. To complete a page, either use a form feed or close the channel. For proper spooler operation, do not include leading or trailing form feeds within the print job. Spoolers always assume the paper is at top-of-form when a job begins.

Note: A form feed is automatically appended to every print job sent.

Logical Printers

A **PRINT** destination does not necessarily mean a hardcopy device. In ProvideX, you have the option to send output data to an image file, HTML, PDF, or preview facility just as easily as any physical printer. In these alternative formats, your output can be further manipulated, merged with other documents, displayed / distributed as is, or simply "printed out" at a later time.

The following ***logical* PRINT** destinations can be made available for use (in an **OPEN** statement) using standard file options. For specific syntax, refer to **Special Files and Devices** in the *Language Reference, p.733*.

HTML Output - *HTML*

HTML generates HTML (Hypertext Markup Language) documents from character-based output in ProvideX (Windows or UNIX / Linux).

Only reports that are formatted using *fixed-pitched fonts* (e.g., Courier) can be converted to HTML. However, the resulting .htm files are in a *universal format* that can be viewed in any browser or incorporated into a web page for posting on the internet; e.g.,

```
OPEN (1)"*HTML*;FILE=Sample.htm;SHOW;FONT=Courier New;TITLE=Sample;
      BACK=FFFFFF;TEXT=000000"
```

This example creates an HTML file called Sample.htm. It generates a report titled "Sample" that is formatted in the Courier New font in black text on a white background. The keyword **SHOW** indicates that the current default browser will be automatically invoked to display the file once it is created. If a file name is omitted, the system prompts for a path / file name to store the resulting HTML document.

Full syntax details are described under ***HTML***, *Language Reference p.736*.

Virtual Bitmap - *BITMAP*

BITMAP captures graphical output to a 24-bit colour bitmap image stored in memory. The resulting ***BITMAP*** image can be retrieved for display using the **'PICTURE'** mnemonic, which accepts the assigned ***BITMAP*** channel number preceded by a **#** in place of a file name; e.g.,

```
OPEN (1)"*BITMAP*"
PRINT (1)"HELLO"
PRINT 'PICTURE'(@X(col1),@Y(row1),@X(col2),@Y(row2),"#1"),
```

The **SAVE FILE** directive can also save ***BITMAP*** contents directly to a `.bmp` file; e.g.,

```
OPEN (1)"*BITMAP*"
PRINT (1)"HELLO"
SAVE FILE (1) TO "C:\test.bmp"
```

For more information, refer to documentation on ***BITMAP***, the **SAVE FILE** directive, and the **'PICTURE'** mnemonic in the *Language Reference, p.734*.

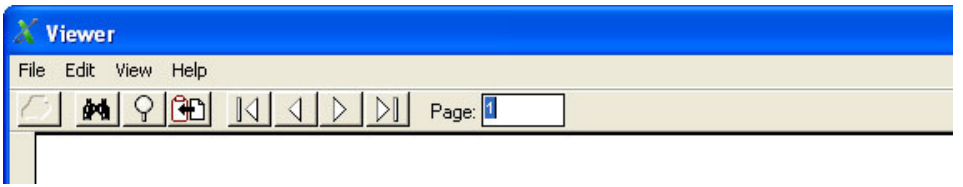
Print Preview - *VIEWER*

VIEWER directs graphical and character-based output to the internal *print preview* facility (Viewer).

The Viewer is a customizable *user interface* in ProvideX for Windows (and WindX) that renders output for display as it would appear in hardcopy form using ***WINPRT*** or any other print destination. When a report is loaded into the Viewer interface, it can then be displayed, manipulated, and "printed out" using a wide variety of format settings, including:

- 1-page, 2-page, 2-page book style, and 4-page viewing.
- 10 to 400% zoom increments and Fit-To-Width / Fit-To-Window display.
- Page-specific paper size and (landscape / portrait) orientation.
- Find and Find Again search funtionality.
- Full report scrolling.
- Watermarks for display or print.
- Banner information for display or print.
- Background spooling.

The above settings, as well as many other actions, are controlled directly from the Viewer panel menubar, GUI buttons and drop-downs:



A full suite of options representing various graphical and character-based output properties may also be specified on the **OPEN** command, either in the **"*VIEWER*"** syntax or by issuing a **OPT=***string$*; e.g.,

```
OPEN(chan)"*VIEWER*"
OPEN(chan)"*VIEWER*;Title=My Report;Orientation=Landscape"
```

Full syntax details for setting output properties are described under ***VIEWER***, *Language Reference p.748*.

PDF Output - *PDF*

PDF generates documents in PDF, Postscript Display Format, from any graphical and / or character-based output in ProvideX (Windows or UNIX / Linux). The `.pdf` files created using this **PRINT** destination are fully compatible with Adobe Acrobat as well as with any other PDF reader; e.g.,

```
OPEN (1) "*PDF*;FILE=/tmp/pvx.pdf; FORM=Letter:8.5in:11in"
```

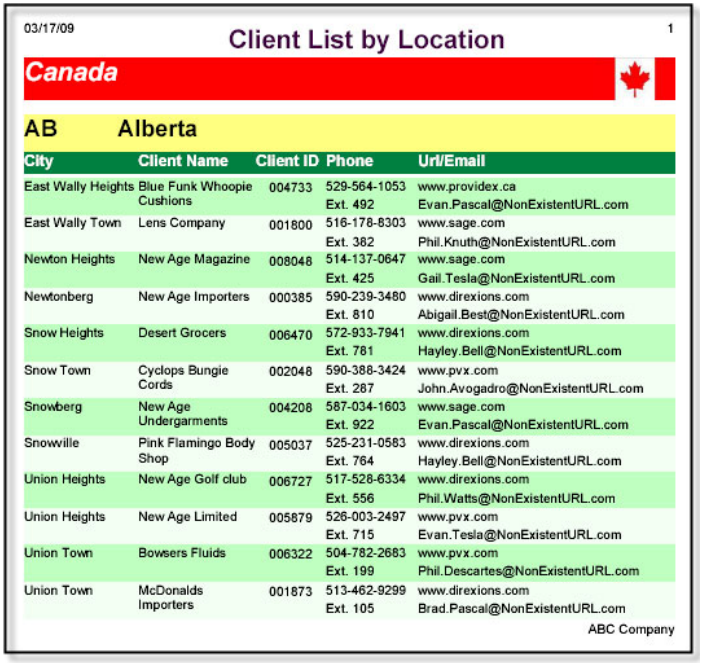
If the file name is omitted, the system prompts for a path / filename to store the resulting PDF.

Output to ***WINPRT*** can be automatically intercepted for PDF using the **'AP'** system parameter. When **'AP'** is set, if the user selects *Output To File* and includes a filename ending in `.pdf`, ProvideX looks at the option selected during the ***WINPRT*** Printer Selection dialogue and processes the output through ***PDF*** instead of ***WINPRT***.

Full syntax details are described under ***PDF***, *Language Reference p.740*.

Report Writer

The *Report Writer* is a ProvideX add-on product for designing and generating formatted reports from ProvideX data. It allows developers and end-users to construct professional reports with the ease and functionality of a spreadsheet application. Built-in features include data drag-and-drop, column and row sizing, computational values, cell formatting (fonts / colours / borders / alignment), image support, sorting rules, data filters, run-time parameter settings, and more.



03/17/09 Client List by Location 1

Canada

AB Alberta

City	Client Name	Client ID	Phone	Url/Email
East Wally Heights	Blue Funk Whoopie Cushions	004733	529-564-1053 Ext. 492	www.providex.ca Evan.Pascal@NonExistentURL.com
East Wally Town	Lens Company	001800	516-178-8303 Ext. 382	www.sage.com Phil.Knuth@NonExistentURL.com
Newton Heights	New Age Magazine	008048	514-137-0647 Ext. 425	www.sage.com Gail.Tesla@NonExistentURL.com
Newtonberg	New Age Importers	000385	590-239-3480 Ext. 810	www.direxions.com Abigail.Best@NonExistentURL.com
Snow Heights	Desert Grocers	006470	572-933-7941 Ext. 781	www.direxions.com Hayley.Bell@NonExistentURL.com
Snow Town	Cyclops Bungie Cords	002048	590-388-3424 Ext. 287	www.pvx.com John.Avogadro@NonExistentURL.com
Snowberg	New Age Undergarments	004208	587-034-1603 Ext. 922	www.sage.com Evan.Pascal@NonExistentURL.com
Snowville	Pink Flamingo Body Shop	005037	525-231-0583 Ext. 764	www.direxions.com Hayley.Bell@NonExistentURL.com
Union Heights	New Age Golf club	006727	517-528-6334 Ext. 556	www.direxions.com Phil.Watts@NonExistentURL.com
Union Heights	New Age Limited	005879	526-003-2497 Ext. 715	www.pvx.com Evan.Tesla@NonExistentURL.com
Union Town	Bowsers Fluids	006322	504-782-2683 Ext. 199	www.pvx.com Phil.Descartes@NonExistentURL.com
Union Town	McDonalds Importers	001873	513-462-9299 Ext. 105	www.direxions.com Brad.Pascal@NonExistentURL.com

ABC Company

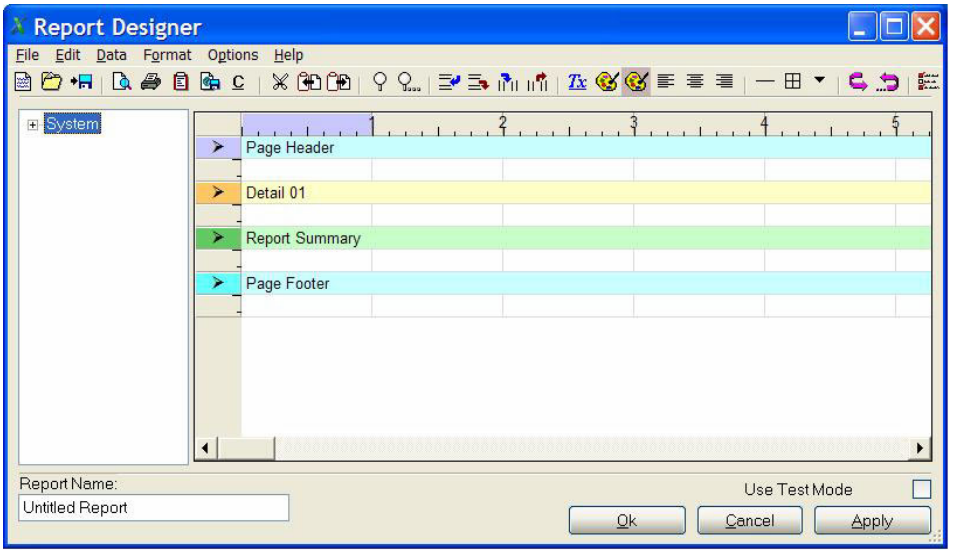
Input sources can be any native ProvideX file with an **Embedded Data Dictionary**, a ProvideX View, or any other source whose data can be accessed using a custom data source object. ***Output*** may be channeled to a variety of physical and logical **PRINT Destinations**, or to a user-defined output object.

Reports are generated based on report definitions, which are produced and modified in the **Report Designer** interface. Users may prefer to start their new report definitions using the interactive **Report Wizard** facility. A **pvxreport** object interface provides developers with programmatic access to report definitions for changing data and format on the fly.

Use of this product may require a separately-purchased activation key apart from your initial ProvideX activation. Contact your local ProvideX dealer / distributor or visit www.pvx.com for complete product information and licensing. For more information, refer to the ProvideX *Report Writer* manual.

Report Designer

The *Report Writer* is used for designing the format and layout of a report. The data is defined in terms of data source selection, sorting rules, and selection criteria for filtering the data. Report layouts are saved as ProvideX report definition files, which can then be used to generate the report.



Report Wizard

This provides a quick and simple way to create a report that does not require any special formatting. The wizard walks you through a series of eight steps that result in the creation of a new report definition.

Printing via Thin-Clients

Print requests are handled differently, depending on the thin-client type. Methods for printing via **WindX**, **JavX**, or **UltraFX** are outlined below. For more information on the options available, refer to the ProvideX *Client-Server Reference*.

Under WindX

Windows printers can be accessed via **WindX** by prefixing the name of the print destination with the **[WDX]** tag; e.g.,

```
OPEN (30)"[WDX]*WINPRT*"
OPEN (1)"[WDX]\\Print_Server\HP_Laser"
```

Under UNIX/Linux, the **[WDX]** tag is not required for access to ***WINPRT*** and ***WINDEV*** because these print requests default to the WindX client automatically.

Access to ***WINPRT*** and ***WINDEV*** under a Windows server will open the printer relative to the host. The **[WDX]** tag is required in order to direct print requests to the WindX client's print system.

To determine if WindX is being used, use the **TCB( )** function or test the **MSE** system variable:

```
%WINDX$=""; IF TCB(88)>0 THEN %WINDX$="[wdx]"
```

or

```
%WINDX$=""; IF DEC(MID(MSE,22,1))>0 THEN %WINDX$="[wdx]"
```

To make all **OPEN** printer requests WindX aware:

```
OPEN (1)%WINDX$+"*WINPRT*;AsIs"
OPEN (1)%WINDX$+"*WINDEV*;AsIs"
OPEN (1)%WINDX$+"*WINPRT*;HP Laser Jet on \\Main_Server\HPLaser"
OPEN (1)%WINDX$+"*WINPRT*;HP Laser Jet;Orientation=Landscape"
```

Under JavX and UltraFX

Unlike WindX, the JavX and UltraFX thin-clients do not have direct access to any of the print facilities on the local machine. The new ProvideX UltraFX print driver ***DEV/ULTRAFXPTR** was developed for use with UltraFX but will work with JavX as well. It automatically sends print jobs to a linked printer device that sends the PDF result to the client machine, which can then be directed to the appropriate local printer.

A personal device file may be created (via the ***UCL** utility) to accommodate this feature. However, you can also use the special print interface ***UFXPTR***, which supports the same output parameters as ***PDF***; i.e.,

OPEN (*chan***[,***fileopt***])"*UFXPTR*[;***option***][;***option***] [...]"**

For format details, see ***PDF*** in the ProvideX *Language Reference*.

8
Client-Server

ProvideX includes several facilities that allow your processing workload and / or data to be distributed over network-connected systems, client workstations, and shared servers. They allow you to maintain heavy processing and data storage on a secure central server while delivering a flexible user-oriented interface to multiple desktop systems and / or portable devices.

This chapter discusses how the various technologies (**WindX**, **JavX**, **UltraFX**, **Application Server**, etc.) will work with your ProvideX applications. For detailed information on this subject, refer to the *ProvideX Client-Server Reference*.

Topics

Background

Since IT departments first made the transition from large mainframes to PC systems (in the 1980s and 90s) the ***client-server*** model has been a central concept in distributed computing. It describes a network architecture where a *client* process requests / receives data or services, and a *server* process provides the data or services. Most internet applications, including Email products, FTP (file transfer) clients, and web browsers, are based on this client-server model. The terms "client" or "server" can apply to hardware and / or software components on either side of the configuration.

TCP/IP and Remote Processing

Transmission Control Protocol / Internet Protocol (TCP / IP) is the primary communication language for governing the services that everyone uses over the *Internet*, including file transfer, electronic mail, and remote logon.

When a TCP / IP connection is established between client and server, the client requests a connection to a specific server by giving its IP address and a service identifier (port / socket number). On the host computer, an application establishes itself by identifying its service number which typically ranges from 1 to 65535. Typically, numbers below 2000 are reserved for specific Internet applications, while numbers above are available for general use.

ProvideX has built-in support for TCP/IP protocol, which allows developers to create new services or to interact with existing services over a network. The **[TCP]** interface is used to communicate directly via TCP/IP. The **[RPC]** *remote processing control* provides a basic method to read/write data in ProvideX files as well as to execute ProvideX background tasks on one or more remote servers via TCP/IP.

These TCP/IP-based facilities are not required when using the client-server deployment options described below. The ProvideX thin-client and hosting products described in this chapter are distributed with pre-built connection/processing functionality that is independent of your hardware and network implementations. Another consideration for remote file access is ProvideX **ODBC Client-Server**, *p.342*.

Client-Server Deployment Options

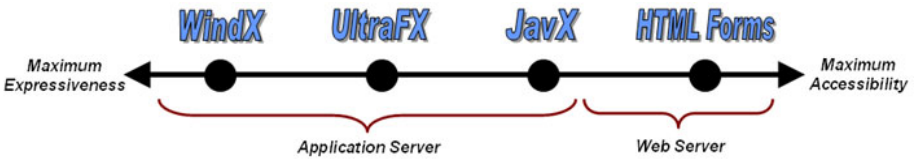
Client-server functionality is tightly-integrated into the language itself. ProvideX offers a range of software extensions for building distributed systems and for optimizing the deployment of your applications across your user base.

A typical client-server application may be configured as follows:

- *On the server side*, a ProvideX host program (either *NTHost or the Application Server) launches a copy of ProvideX and monitors a TCP/IP socket, "listening" for incoming requests from clients.
- *On the client side*, JavX, WindX, or UltraFX opens the client end of the same socket the host is listening to, and passes requests through to the server.

Choosing the Right Configuration

Different customers have different needs; therefore, choosing the appropriate client software requires careful analysis of the trade-offs between *accessibility* and *expressiveness*, as illustrated in the table below.



WindX Full-featured thin-client that takes advantage of the local OS to deliver a rich graphical environment (along with auto-update capability) from any remote ProvideX host system to any MS Windows client.

A WindX client is most effective when:

- end-users are accessing the application from a conventional workstation
- long complex tasks are to be performed on a regular basis
- application developers determine the client platform (not end-users).

UltraFX Platform-independent GUI environment that includes many built-in features such as mutli-threading, toolbars, dockable-stackable-moveable windows, split panes, and an embedded web browser. This thin-client is built on the powerful Eclipse RCP framework. (*ProvideX applications do not need to be developed in Eclipse to use UltraFX*).

JavX *SE* The *JavX Swing Edition* is a Java-based thin-client with similar functionality to WindX that enables ProvideX applications to run on any client platform that supports the Java 2 Standard Edition (J2SE) runtime. With JavX SE, the web browser is promoted to a universal ProvideX client—users can navigate to a JavX SE-enabled web page which uses a Java applet to interface with the server application. However, JavX SE implementations offer slightly less functionality than WindX and have limited access to the local machine.

A JavX SE client is most effective when:

- end-users are accessing the application from outside the office
- long complex tasks are to be performed occasionally
- end-users require more choice in platforms.

JavX *AE* The *JavX AWT Edition* provides a simpler GUI designed specifically for handheld devices that support the Java 2 Micro Edition (J2ME) runtime.

A JavX AE client is most effective when:

- end-users are accessing the application from outside the office
- more general tasks are to be performed occasionally
- typical platform is a WinCE PDA.

JavX AE functionality is upwardly-compatible with JavX SE.

JavX *LE* The *JavX Light Edition* represents a limited non-GUI version of JavX that is intended primarily for fixed-purpose industrial or consumer products.

A JavX LE client is most effective when:

- end-users may be performing a few simple tasks using an interactive device / appliance.

JavX LE functionality is upwardly-compatible with JavX AE / SE .

HTML Forms A ProvideX WebServer / HTML implementation allows for a basic web user environment that has no access to the local machine.

An HTML Forms implementation is an effective client when:

- end-users require brief access to fill in a few form fields that are presented using a standard web page.

Refer to the ProvideX WebServer documentation for information on using HTML with ProvideX to create a web application.

Hosting Facilities

In a direct TCP/IP client-server environment, JavX and WindX thin-clients would be set up to access server-side applications using the ProvideX Application Server or *NTHost/*NTSlave.

*NTHost/*NTSlave

These facilities are supplied with the ProvideX base system for establishing a very simple TCP/IP connection. On the host computer, the program *NTHost is run to monitor incoming requests from client PCs and to initiate new processes to service these requests. On a Windows client system, the program *NTSlave begins the initial connection to the host by requesting a new session to be started.

While the *NTHost/*NTSlave combination is sufficient for establishing quick client-server connections, they are not designed for use "as is" outside of a closed local area network. For exposure to large networking environments or the Internet, it is much safer to run the ProvideX Application Server.

Application Server

The ProvideX Application Server provides an enhanced, configurable alternative to the built-in client-server processes supplied with the ProvideX base system. It extends the usability of your server-based applications and delivers an all-in-one solution for protecting/maintaining your data over large networks including the Internet:

- ***Simplified Interface.*** User-friendly utilities are provided for creating, configuring and administering the different characteristics of your ProvideX client-server applications.
- ***Uses only one TCP /IP Socket.*** Default client-server processes in ProvideX can end up using many different sockets depending on the implementation and the number of clients. The Application Server architecture allows you to direct all connections through a single socket.
- ***Designed for firewalls***. If it is too difficult for a firewall to determine which sockets it needs to block, it might as well be disabled. By reducing the number of sockets needing access, the Application Server ensures that your firewall is fully operational.
- ***Session Adminstration.*** One of the primary security features in the Application Server is its ability to monitor and control access through session administration, user authentication, and limiting client access.
- ***Optional SSL encryption***. The Application Server supports use of a TCP/IP-level Secure Socket Library, a security protocol that allows you to encrypt communications.

Thin-Clients

Once one of the **Hosting Facilities** is configured and running on the server, an installed/configured thin-client should be able to establish communication with the server-based ProvideX system.

Topics

The ProvideX thin-clients described below allow application processing and data storage to be maintained on a secure, centralized server, while delivering graphical interface components to the client desktop.

WindX runs in conjunction with a copy of ProvideX for Windows, so WindX can issue virtually any command of which ProvideX is capable.

JavX uses a set of Java equivalents to match ProvideX language features – and while JavX permits ProvideX GUI applications to be run on a wider range of platforms, it cannot replicate some Windows-specific functionality.

UltraFX, the newest ProvideX thin-client, takes JavX to the next level, offering a self-contained user environment for launching your ProvideX application on any Java-enabled platform.

These products may be integrated directly into your application and shipped as an integral component.

Programming for Thin-Clients

With a ProvideX thin-client in place, most content is routed automatically to the client without the need for special programming techniques. Code related to printing, keyboard input, mouse movement, printer selection and GUI elements are routed to the client or server automatically. ProvideX client-side functionality is designed to minimize network traffic and improve system performance.

Following is an overview of ProvideX thin-client functionality. For programming information that is specific to a particular thin-client, refer to the **WindX**, **JavX**, and **UltraFX** sections documented later in this chapter. See also **Printing via Thin-Clients**, *p.238*. For more detailed documentation, refer to the *ProvideX Client-Server Reference*.

Client or Server?

ProvideX includes some server-side indicators for detecting a thin client session. **MSE** bytes 32 for 1 byte, will be a `W` for Windows-based (WindX), `J` for Java-based (JavX-UltraFX) or 00 for neither. **TCB(88)** will be zero for no client, or will contain the revision level of the WindX/JavX/UltraFX communications protocol that the client is using.

Once ProvideX on the server recognizes the existence of the client station, it changes internal settings that route graphical requests to the thin-client. Graphical directives and functions invoked by the application are **tokenized** and sent to the client.

Code that may be executed on either side needs to indicate if it is to execute on the server or the client. Execution defaults to the server side. To perform an operation on the client side, use the **[WDX]** tag; e.g.,

```
CALL "[WDX]*windx.utl;get_num","tcb(29)",number
OPEN(channel)"[WDX]Path\Filename"
```

When using the **Application Server**, an OOP object **%APS** is created on both the server side and the client side of the connection. This contains properties about both server and client instances of ProvideX that may be used in your code.

Mnemonics

Thin-clients respond directly to the internal form of all mnemonics. Therefore, unlike conventional terminals, no translation table is required. Mnemonics, such as **'CS'**, are transmitted as `$1B$+"CS"` and screen position commands, such as **@(1,2)**, are sent as `$1B$+"@2"+CHR(1)+CHR(2)`. Long-form mnemonics, such as **'WINDOW'** and **'DROP'**, are sent in their native form as well.

Graphical Control Requests

ProvideX tokenizes all graphical directives and references, then forwards them to the client for processing. Access to the control attributes (e.g., `BackColour$, Height, Enabled`) is tokenized as well and forwarded to the client for processing. Under JavX-UltraFX it may be better to use directives rather than attributes when interfacing with controls.

Turbo Mode

During normal operation, each tokenized message sent by the host to the client requires an acknowledgment. While this process guarantees that the application and client are synchronized fully, it can slow down overall transmission speeds. ***Turbo*** mode in ProvideX (**'TU'** system parameter) allows the thin-client to receive and process many requests locally without the need to acknowledge each transmission from the host. However, since no return result is acknowledged, be aware that some error reporting may be ignored.

Help

Help documents with their associated application may be launched from the client via the **SYSTEM_HELP** directive.

Notes on Efficient Coding

While thin-client functionality is fully integrated into ProvideX, how you design your application, and how it handles data, may have an impact on performance. Inefficient code can lead to sluggish application response, overall network congestion, and higher CPU usage on the server. It's best to avoid:

- Requesting the same information more than once.
- Sending the same data repeatedly, especially long strings of information.
- Loops of code to set / retrieve properties or characteristics on the server side.
- Loops of code to calculate values based on items that need to come from the PC.

Platform diversity, available bandwidth, and other network issues should also be considered when developing a ProvideX application for a client-server environment. Regardless of your network constraints, it's best to keep the following in mind:

- The less data you send, the less bandwidth you use.
- Loading 10,000 lines of 100 characters each in a list box on the client will be slow. The slower the bandwidth, the slower the application.
- The fewer questions you ask of the client the better, as latencies can build up.
- If you need to get data from a client (**FIN( ) i**nformation or object properties) ask the question once and keep the answer on the server in case you need it again
- Each time information is needed, the request must be packaged, sent, decoded, a result gathered, the result packaged then sent back to the server and decoded..

For further discussion on this topic, see *Optimizing Your Code* in the *ProvideX Client-Server Reference.*

WindX

Fundamentally, ***WindX*** is designed to provide a feature-rich, graphical user interface to a Windows client from any server-based ProvideX application, even if the host system does not support that type of interface (e.g., UNIX). WindX can also be configured to take full advantage of each platform's functionality by allowing processing and file access to be handled on either side of the client / server configuration.

Two versions of WindX can be obtained from your dealer / distributor or downloaded from the ProvideX website, **www.pvx.com**:

- WindX ***Standalone***, which requires an individual license per client installation and is able to interact with any ProvideX application on any server. Each license has its own *serial number, user count, expiry date* and *activation key.*
- WindX ***Plug-in***, which is freely distributable for clients but must connect with a ProvideX application on a server that maintains a multi-user Professional or eCommerce license. If the server is not licensed for plug-in access, server

file / directory permissions are incorrect, or a ProvideX session cannot be established within 2 minutes of startup, then the plug-in will terminate automatically.

Note: The Standalone and Plug-in downloads for WindX are for installation on Windows client machines only.

For complete WindX installation and activation details, see **WindX Thin Client** in the *Installation and Configuration* guide.

Running ProvideX Applications in WindX

As noted earlier, any file type that ProvideX can access may be accessed locally across a WindX connection (serial, keyed, indexed, ODBC, OCI, TCP etc.). WindX allows you to **CALL** programs that exist on the client; however, you may not **LOAD** or **RUN** local programs. The *WINDX.UTL utility may be used to simplify many client requests.

OOP objects and COM controls may be utilized by the server for WindX clients via:

```
NEW("[WDX]Classname")
DEF OBJECT X,"[WDX]RemoteCOMObject"
```

DLL( ) functions may be used provided they are wrapped in a ProvideX program that is installed on the client that can be called from the server.

JavX

The ***JavX*** thin-client was originally designed to be a Java version of WindX. JavX takes full advantage of Java's portability and platform independence to provide a ***flexible alternative*** to the Windows-only thin-client. However, it is not possible to recreate the entire Microsoft Windows environment within the Java framework, so some WindX functionality may not be available under JavX.

JavX uses Java 2 GUI components to replicate the complex graphical features available in ProvideX. But unlike WindX, the Java-based thin-client is able to run on any machine that has the appropriate Java Runtime Environment (JRE). The Java JRE is a free download that can be installed on any machine just prior to running JavX.

Three editions of JavX are currently available for different target platforms:

- JavX SE (*Swing Edition*) designed for desktop systems that run the Java 2 Standard Edition (J2SE) runtime environment — this includes Windows, Linux and UNIX X-Windows, and Apple Mac OS X systems.
- JavX AE (*AWT Edition*) designed for small devices that run the Java 2 Micro Edition (J2ME) Constrained Device Context (CDC) Personal Profile — this includes a variety of personal digital assistants (PDAs).
- JavX LE (*Light Edition*) designed for task-specific devices that run the J2ME CDC Foundation Profile — this includes a range of consumer products, automotive and other interactive components.

For complete JavX installation and activation details, see **JavX Thin Client** in the *Installation and Configuration* guide.

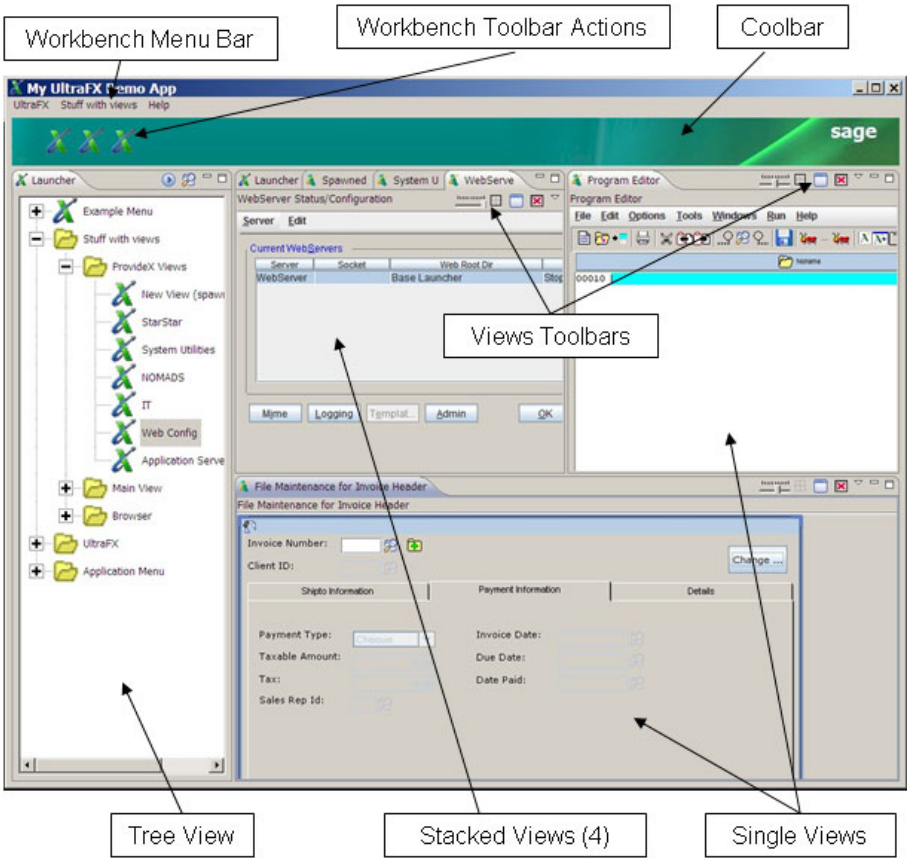
Running ProvideX Applications in JavX

As noted earlier, local ProvideX programs cannot be called on the JavX client. However, several of the entry points in *WINDX.UTL have been coded specifically into JavX so that some specific **CALL** commands may be made. Serial files may be created, accessed or deleted across the connection, but other file types are not available. Java classes may be utilized for JavX clients via the ProvideX COM interface:

```
DEF OBJECT X,"[WDX]JavXClass"
```

UltraFX

This is a Java-based thin-client that provides a self-contained user environment for ProvideX applications. Built on the Eclipse *Rich Client Platform* (RCP), UltraFX delivers a rich GUI environment with built-in features like mutli-threading, toolbars (*coolbars*), dockable-stackable-moveable windows, split panes, an embedded web browser. It is entirely platform independent, a relatively small download, and it requires little or no changes to your ProvideX applications.



While UltraFX employs the Eclipse RCP framework, your ProvideX applications do not need to be developed in the Eclipse IDE. In fact, any ProvideX application can run in UltraFX. For more on RCP, refer to the website http://wiki.eclipse.org/RCP

UltraFX Workbench. The central windowing environment in UltraFX is called the *workbench.* This provides the *overall container* for all views, menus, tool/coolbars, shortcuts, and other interface control objects needed for running applications. From the user's perspective, the workbench represents a desktop-style front end for running multiple application interfaces at the same time. Each ProvideX process running in UltraFX will be launched in the form of a separate view.

For complete UltraFX installation and activation details, see **UltraFX Thin Client** in the *Installation and Configuration* guide.

Running ProvideX Applications in UltraFX

Implementation procedures for running a ProvideX application in UltraFX are virtually identical to those described for JavX. Applications running within UltraFX use standard ProvideX syntax (directives, mnemonics, etc.) to create GUI widgets like windows, buttons, menu bars, etc. On top of that, UltraFX provides a customizable platform and independent framework. It also allows for advanced branding, and unique GUI components such as a coolbar/toolbar, status bar, progress monitors/indicators, as well as pre-built views like the tree view and browser within the UltraFX workbench.

Upgrading Client Software

There are multiple options available for upgrading/updating ProvideX-based client-server applications. Developers may take advantage of 3rd party update services such as those supplied by *InstallShield* or they may create their own software update packages. Also, ProvideX comes with an ***AutoUpdater*** for updating any WindX client from software repositories on the server they connect to.

AutoUpdater

This utility is included with the ProvideX base activation to provide a means for automatically updating (or downgrading) and repairing client installations of WindX. It can be configured to check for and install critical patches/upgrades on all client workstations whenever they are connected to the server. It is not available for JavX or UltraFX clients.

For more information, refer to the *AutoUpdater* documentation in the *ProvideX Client-Server Reference.*

9
External Components

As explained in *Chapter 4*, **Called Procedures** allow code to be reused within ProvideX applications to help increase efficiency and maintainability, as well as reduce program size. The same principles apply to the functionality that allows *external* (third-party) software modules to be accessed and incorporated into your ProvideX applications from anywhere in the local operating environment or over a network. This chapter discusses the use of external objects / custom controls in ProvideX applications.

Topics

Function Library Model

Calling procedures from external files is almost exactly the same as calling line label entry points from within ProvideX, for example,

```
0010 ! CUSTMAINT
0020 READ_CUST:
0030 READ (%CUST_FILE,KEY=CST_ID$...
...
0090 EXIT
0100 !
0110 UPDATE_CUST:
0120 WRITE (%CUST_FILE,KEY=CST_ID$...
...
0190 EXIT
0200 !
0210 REMOVE_CUST:
0220 REMOVE (%CUST_FILE,KEY=CST_ID$...
...
0290 EXIT
```

The above program contains several “functions” that can be re-used at run time for the maintenance of a customer data file.

Concepts and Terminology

In order to better understand the facilities in ProvideX for accessing external components, it is necessary to understand the general concepts and some of the history behind the technologies being discussed.

Note: Some terms may have mixed meanings in the industry due to an enduring misnomer, remnants in the evolution of a technology, or a direct marketing strategy. The definitions below apply to the use of external components in ProvideX.

API

An API (Application Program Interface) is a set of functions and protocols, for building and implementing software applications. Most operating systems provide an API so that programs designed to run on them can access system services and stay consistent within the operating environment. Some common APIs include:

- ***Windows API,*** Microsoft's core set of interfaces for running software within the Windows operating system.
- ***Single UNIX Specification,*** a standardized set of interfaces for running software within versions of UNIX and Linux.
- ***Java API***, a set of standard interfaces and classes grouped into packages such as `java.awt` for building GUIs, `java.io` for handling I/O requests, etc.

These APIs also represent the specific calling conventions that define how OS services are to be invoked. There can be thousands of API calls in a full-blown operating system. While APIs are primarily intended to assist and accelerate program development, they provide a huge benefit to end users as well. By maintaining a set of common interface elements, APIs also make it easier for users to learn new programs that are designed to run on the same OS.

While an API is designed for interaction between the OS and applications, it can also establish standards for interaction between applications; e.g., Microsoft introduced various interface technologies to assist communication between applications running under Windows; i.e., **DDE**, **OLE**, **COM**, and .NET. Some of these are explained below.

DLL

DLL (Dynamic Link Library) files contain executable code that can be shared by several different applications running under MS Windows. Basically, they serve as external code repositories. Unlike executable files, DLLs are not launched directly by the user but are called for by a running program or by other DLLs to provide services not built into the application. They can also save memory space because they don't get loaded into RAM until they are actually needed.

Some DLLs are used only by a specific application, while others may be used by several. For example, a variety of programs would likely call the same Windows DLL for handling user interface tasks to create common toolbars, text boxes, scroll bars, etc.

The DLL files installed to support specific device operations are also called ***device drivers***. The UNIX equivalent of a DLL is referred to as a ***shared library*** or ***shared object module***. For more information, see **Calling DLLs from ProvideX**, *p.252*.

DDE

Dynamic Data Exchange is an early Windows technology used to exchange data, commands, and status information automatically between different applications. While some DDE implementations are still in operation today, this technology has largely been superseded by the more robust **OLE**/**COM** ***Automation*** used in more current versions of Windows.

OLE

OLE (Object Linking and Embedding) is a Microsoft Windows technology that enables objects created in one application to be imported by reference into the documents of another; e.g., an Excel spreadsheet placed inside an MS Word document. These are referred to as ***compound documents***. Making changes to an OLE compatible object in the original editor automatically updates the imported version within the compound document.

An extension of OLE, referred to as ***OLE Automation***, provides an infrastructure for applications to access and manipulate shared automation objects. This technology is now a part of the Microsoft **COM** implementation explained below.

OCX

An OCX (OLE Control eXtension) control is a special-purpose program object that can be re-used by several applications running on Microsoft's Windows systems. This technology began as VBX (Visual Basic eXtension) controls, which were VB-only in the early days of Microsoft Windows. "OLE controls" or "OLE custom controls" were then created to run on Windows 95/NT supporting 32-bit applications. OCX has now been replaced by ***Activex*** (see below).

ActiveX

ActiveX refers to several object-oriented technologies in Microsoft that enable component sharing by many application programs within a computer or among computers in a networked environment. Historically, the definition of ActiveX emerged from the implementation of earlier OCX, and OLE custom controls. The term now encompasses several subsets of Windows component technologies and its meaning changes depending on the application.

When most people say "ActiveX", they are likely talking about ***ActiveX controls*** — specific components that provide applet-like functionality for web pages. Similar to ***Java*** applets, ActiveX controls can be accessed and executed via web browsers and other applications over the internet. However, ActiveX offers little cross-platform support compared to Java, and is limited to software based on Microsoft's Component Object Model (**COM**).

COM

COM (Component Object Model) is the framework for developing and supporting program component objects in Microsoft Windows. While COM originally evolved from Microsoft's OLE technology, which provided services primarily for compound documents, it now includes much more. COM provides the specification for developing reusable software components as well as an underlying language-neutral implementation for these objects to communicate with each other. *Distributed COM* (DCOM) extends COM technology across networked computers. An in-depth discussion of ProvideX COM is provided in the section describing **ProvideX COM Support**, *p.261*.

Calling DLLs from ProvideX

As mentioned earlier, a **DLL** is essentially a free-standing library of functions contained in a single file that allows different applications to run various shared services. Applications use DLLs to access external functionality, such as that provided as part of an **API**, by opening a DLL file and by calling each function within the DLL as it is required.

Topics

The functions that provide the services available within a large operating system API are likely to be contained in a number of DLL files. For example, access to the ***Microsoft Windows API***, is handled via calls to the following primary DLL files:

`gdi32.dll` for graphics-oriented functionality
`kernel32.dll` for access to low level operating system features
`user32.dll` for controlling most visible screen controls.

As of Windows XP, these basic controls reside in `comctl32.dll`, together with the common controls (Common Control Library).

Programmers must already know which functions are available and how to access them if they plan to include DLL calls in their program's code. It is also important to ensure DLL version compatibility and to acquire/consult the necessary API system documentation in order to implement DLLs successfully.

Warning: You can use third-party DLLs, but be certain of what you're passing and getting back. Sage Software Canada Ltd. offers assistance on how to call a DLL, but will not provide support for third-party DLLs. There is no validation on what you pass. Bad pointers are liable to cause memory and data corruption, and may result in a GPF.

ProvideX DLL Interface

In ProvideX, the **DLL( )** function is used to load, find, and execute functions within external OS-loadable modules. It provides a direct interface for both MS Windows ***DLL files*** and UNIX/Linux ***shared object modules***. The function **TCB(196)** will return 1 to indicate that **DLL( )** is supported for a particular UNIX/Linux environment.

The general methods for using this function involve the following formats:

Accessing the DLL by name:

lib_num=**DLL(ADDR** *lib_string$*[,**ERR=***stmtref*]**)**

Accessing the DLL using its internal library identifier:

DLL(*lib_num,fnc_name$,arg*[*,arg,arg...*][**,ERR=***stmtref*]**)**

Accessing a function within the DLL by its memory address:

fnc_addr=**DLL(FIND** *lib_num,fnc_name$*[**,ERR=***stmtref*]**)**

Other formats for calling by string, library number, or function address are also available. For complete syntax details, see **DLL( )**, *Language Reference p.416*.

Note: The **[DLL]** special command tag is *not* associated with the ProvideX **DLL( )** function (DLL interface). It is used as a prefix in an **OPEN** statement to denote that ProvideX is to route all file I/O requests via an external (user-defined) DLL file.

Loading DLLs into Memory

Typically, a DLL would be loaded and unloaded for each use; however, when multiple calls are required to complete an operation, then that DLL should be loaded into memory for the duration. This also speeds up DLL calls considerably.

The following provides an example of a ***single-use DLL***. It sends a message (number 1047) to a ProvideX Window, then ProvideX internally traps a 1047 and exits.

```
x=DLL("User32.dll","SendMessageA",Handle,1047,0,0)
```

When ***multiple DLL*** calls are expected, then the **ADDR** directive can be used to load the specified executable into memory. The example below shows a Windows Registry operation where the DLL call involves placing the DLL into memory (via **ADDR**). To perform such an operation, the DLL requires a handle to the registry. That handle can then be used in subsequent DLL calls for specific operations. Failure to return a handle would be considered a "resource leak".

```
! Read the Windows registry for a Key\SubKey\Name and
! return its current Value and Type of Value if you want it
!
function GetRegistryValue(RegKey%,RegSubKey$,Name$,Value$,ValueType%)
enter (RegKey%),(RegSubKey$),(Name$),Value$,ValueType%
if not(IsWin32) \
 then return 0
if RegKey%>=0 \
 then RegKey%+=dec($80000000$)
RegSubKey$=sub(RegSubKey$,$00$,"")
Name$=sub(Name$,$00$,"")
Value$=""
ValueType%=0
if mid(RegSubKey$,1,1)="\" \
 then RegSubKey$=RegSubKey$(2)
if mid(RegSubKey$,-1)="\" \
 then RegSubKey$=RegSubKey$(1,len(RegSubKey$)-1)
RegSubKey$+=$00$
Name$+=$00$
```

```
SecurityMask%=9 ! KEY_QUERY_VALUE | KEY_ENUMERATE_SUB_KEYS
dim KeyHnd$(4)
DllHnd=dll(addr "advapi32.dll",err=*proceed)
if DllHnd=0 \
 then return 0
if dll(DllHnd,"RegOpenKeyExA",RegKey%,RegSubKey$,0,SecurityMask%,KeyHnd$) \
 then result=dll(drop DllHnd,err=*proceed);
return 0
KeyHnd%=dec(swp(KeyHnd$))
dim DataValueType$(4,$00$)
dim DataValue$(256,$00$)
DataValueLength$=swp(bin(len(DataValue$),4))
error=dll(DllHnd,"RegQueryValueExA",KeyHnd%,Name$,0,DataValueType$,DataVa
      lue$,DataValueLength$)
if not(error) then \
{ ValueType%=dec(swp(DataValueType$))
switch ValueType%
case 1,2,7
Value$=DataValue$(1,dec(swp(DataValueLength$))-1)
break
case 4,5,11
Value$=swp(DataValue$(1,dec(swp(DataValueLength$))))
break
default
Value$=DataValue$(1,dec(swp(DataValueLength$)))
break
end switch
success=1
}
result=dll(DllHnd,"RegCloseKey",KeyHnd%)
result=dll(drop DllHnd,err=*proceed)
return success
```

Note: DLLs not placed into memory, are considered single use only. Attempting the above operation without an **ADDR**, would result in most of the DLL calls failing.

Passing Values in a DLL Call

As with any function call, you must define and pass the correct parameters in order for the DLL to work. While some DLL functions are designed to take arguments that can handle more than one data type, they normally would expect one of the following:

- Number without decimals (sent as integer).
- Number with decimals (converted to integer).
- String value (sent as pointer to the string).

In a 32 bit operating system, integers and pointers must be 4 bytes long.

DLL( ) *Parameters*

The arguments / parameters you use when you call the **DLL( )** function are passed to the function in the following ways:

Example	*Type*	*32-Bit Data Format Passed*
X$	Strings	Address of string
X	Numeric Variables	Double word value (32-bit)
X%	Integer Variables	16-bit value passed as 32-bit
INT(X+1)	**INT( )** Function	Standard 32 bit
X+Y	Numeric Expression	32-bit value

All parameters being passed and returned should be defined in your code as outlined in the documentation provided with the API / DLL you wish to implement. Many DLLs expect a predefined format in the form of a data structure (see below).

Data Structures

To pass a data structure in a DLL call, format a string to the required size then pass the string. In effect, this sends a pointer to the structure. Following is the MS Window's API reference to a `WindowPlacement` structure:

```
typedef struct _WINDOWPLACEMENT {
UINT length;
UINT flags;
UINT showCmd;
POINT ptMinPosition;
POINT ptMaxPosition;
RECT rcNormalPosition;
} WINDOWPLACEMENT
```

The above structure translates into ProvideX as follows:

```
DIM WindowPlacement$(4 + 4 + 4 + (4 + 4) + (4 + 4) + ((4 + 4) + (4 + 4)), $00$)
! Or DIM WindowPlacement$(4 + 4 + 4 + 8 + 8 + 16, $00$)
! Or DIM WindowPlacement$(44, $00$)
```

`UINT` values are un-signed integers of a 32-bit value (4 bytes). The `POINT` is a structure of two `LONG` values (each LONG is a 32-bit value); therefore, 8 bytes total for a `POINT` structure. The `RECT` is a structure of 4 `LONG` values (each `LONG` is a 32-bit value); therefore, 16 bytes total for a `RECT` structure. If the structure contained pointers, then you would leave room for a 4-byte value to hold the pointer.

Converting Data To/From Local Representation

Different CPUs and operating systems have different criteria for how numbers are kept in memory. This is known as ***byte ordering***. The order of the bytes in memory are critical when using the **DLL( )** function. Because memory is being directly accessed, you must ensure the data you are putting in memory will be interpreted correctly by the DLL; and conversely, when data is returned from the DLL, it must be converted back into something usable.

When you pass a string or a number on the **DLL( )** function line, ProvideX will automatically convert the data into the form required. Since strings are passed as pointers, ProvideX knows to convert the pointer itself into the local byte ordering format. However, data within the string is your responsibility. ProvideX does not know if it should convert the values within the string, or what those values represent; therefore, it cannot convert data within a string to the local byte order.

Since a structure is a simple pointer to a memory location, and a specific number of bytes in memory starting at that location, you need to convert items within the structure to the local values.

Numbers within a string need to be converted, whether they are integers or pointers. These functions can used to manipulate values prior to using them in the DLL call:

SWP( ) Converts numbers to / from the local OS / CPU format.

BIN( ) Converts numbers into byte values; e.g., `BIN(1234567,4)` becomes the 4 byte integer of 1234567.

MEM( ) Returns memory locations (pointers) for data and allows for direct access to data in memory.

DEC( ) Converts byte values into numbers. Signed values are produced by default, but for unsigned values use `DEC($00$+$..$)`.

If you are defining a string that is a structure and that structure needs a pointer to a piece of data, you would have to allocate space and convert the location of that data in memory; e.g.,

```
Struct {
   UINT DataLength;
   char *pData;
} MyStruct
DIM Data$(255,$00$) ! Create a string in memory of the required length
      and initialize it with NULLs.
pData = MEM(Data$) ! Get a memory pointer to the string
DataLength = LEN(Data$) ! Set the length of our data string
DIM MyStruct$(4 + 4, $00$) ! Dimension our structure
```

The values must be put into the structure in order to pass the structure in a DLL function call; e.g.,

```
MyStruct$(1,4)=SWP(BIN(DataLength,4)) ! Convert the length to a local
      values and put it into the MyStruct$ structure
MyStruct$(5,4)=SWP(BIN(pData,4)) ! Convert the pointer to a local value
      and put it into the MyStruct$ structure
Result = DLL("SomeDLL","SomeFunction",MyStruct$) ! Passes the memory
      pointer of MyStruct$
```

If the DLL returns an updated structure, then the values will need to be converted back into something usable as follows:

```
UpdatedLength = DEC(SWP(MyStruct$(1,4)))
```

In this case, `Data$(1,UpdatedLength)` would show the new value. Since the pointer is passed to the `Data$` variable, the DLL simply updates X number of bytes at the location in memory of the `Data$` string.

However, if the DLL function call modified the pointer to `Data$`, then the following would be required to get the new pointer value, and read memory to get the new string:

```
pData = DEC(SWP(MyStruct$(5,4))
Data$ = MEM(pData, UpdatedLength)
```

This reads memory at the location given in the `MyStruct$` for the data, and reads for the length given.

By using the **SWP( )** function, ProvideX automatically swaps the number to or from the local byte ordering format. Whether it is for an Intel or Power PC CPU, **SWP( )** defaults to the correct format for the processor. It is possible to override the byte ordering format with an option to the **SWP( )** function.

Examples

The following examples illustrate use of the functions described in the earlier sections for working with Windows DLLs in ProvideX. For examples of UNIX/Linux function calls, see **Working with DLL Calls in UNIX/Linux**, *p.259*.

Example 1. The following statement swaps the left and right mouse buttons:

```
0010 A=DLL("USER32.DLL","SwapMouseButton",1) ! Passes 32-bit value
```

An integer value of zero swaps the mouse buttons back.

Example 2. Pointers are commonly used to pass string values, usually terminating with a null byte. This passes a pointer to a DLL to return the handle of a window with the title `Notepad`:

```
0030 A=DLL("USER32.DLL","FindWindowA",0,"Notepad"+$00$)
```

Example 3. Sometimes a pointer refers to a region of memory or a structure to receive information from a DLL. You must pre-allocate a string variable to receive the data. The example below returns the window title, given the handle. The returned string terminates with a null byte (`$00$`):

```
DIM X$(256)
0030 A=DLL("USER32.DLL","GetWindowTextA",Hndl,X$,LEN(X$))
```

Example 4. To pass a pointer to a number, define a two- or four-byte string and read the value after swapping the bytes, as follows:

```
DIM X$(256)
A=DLL("SOME.DLL","GetSomething",Hndl,X$)
X=DEC($00$+SWP(X$)
```

Example 5. Program `DLLS1` in this example starts `Notepad`, locates the handle and closes the window handle:

```
0010 !dlls1 Start Notepad and Close It
0020 INVOKE "NOTEPAD"
0030 INPUT "Press enter after Notepad has started ",X$
```

```
0040 LET HWND%=DLL("USER32.DLL","FindWindowA","Notepad"+$00$,0)
0050 IF HWND%<=0 THEN PRINT "Notepad not found!"; STOP
0060 LET WM_CLOSE=DEC($0010$),WPARAM%=0,LPARAM=0
0070 LET X=DLL("USER32.DLL","SendMessageA",HWND%,
0070:INT(WM_CLOSE),WPARAM%,LPARAM)
```

Example 6. The following example illustrates inter-task communication. The program DLLS3 starts a second PVXWIN32 session, finds its Window handle, sends CTL values 101 through 103, defines an atom (pointing to a string variable in a global string table) and passes the atom reference to the second session. Then it waits for the second session to terminate.

```
1340 X=DLL("USER32.DLL","SendMessageA",HWND%,INT(WM_USER),WPARAM%,LPARAM)

00010 ! DLLS3 - Inter-task communication - Send CTLs & a string
00020  MAX_COL=MXC(0)+1,MAX_ROW=MXL(0)+1
00030  PRINT 'SHOW'(-1),'DIALOGUE'(0,0,MAX_COL,INT(MAX_ROW/2)-1, \
             "Inter-task com0030:munication",'MODE'($000F$)+'CS'),
00040  IF ARG(1,ERR=*NEXT)="Receiver" \
        THEN GOTO RECEIVER
01000 ! !^1000
01010  LOOP=0
01100 ! !^100 - Find Receiver & start if necessary
01110  IF LOOP++>1 \
        THEN PRINT "Cannot start/find the Receiver!";
             STOP
01120  HWND%=DLL("USER32.DLL","FindWindowA",0,"PVXWIN32 - Receiver"+$00$)
01130  IF HWND%>0 \
        THEN GOTO 1200
01140  INVOKE ARG(0)+" "+PGN+" -ARG Receiver"
01150  WAIT 1
01160  GOTO 1100
01200 ! !^100 - Send CTL 101 to 103 to Receiver
01210  WM_USER=DEC($0400$),WPARAM%=0,LPARAM=0
01220  FOR WPARAM%=101 TO 103
01230  PRINT "Sending CTL value of",WPARAM%," to Receiver"
01240  X=DLL("USER32.DLL","SendMessageA",HWND%,INT(WM_USER),WPARAM%,LPARAM)
01250  NEXT
01300 ! !^100 - Send CTL 104 & the address of a String
01310  WM_USER=DEC($0400$),WPARAM%=104,LPARAM=0
01320  ATOM$="Message to
      Send",LPARAM=DLL("KERNEL32.DLL","GlobalAddAtomA", \
             ATOM$+$00$)
01330  PRINT " Sending CTL Value ",WPARAM%,"to Receiver with message:", \
             'BR',ATOM$,'ER'
01340  X=DLL("USER32.DLL","SendMessageA",HWND%,INT(WM_USER),WPARAM%,LPARAM)
02000 ! !^1000
02010  LOOP=0;
       PRINT "Waiting for Receiver to shutdown",
02020  HWND%=DLL("USER32.DLL","FindWindowA",0,"PVXWIN32 - Receiver"+$00$)
```

```
02030  IF HWND%<=0 \
        THEN GOTO 2100
02040  PRINT ".",;
       IF LOOP++<30 \
        THEN WAIT 1;
             GOTO 2020
02100 ! !^100
02110  X=DLL("KERNEL32.DLL","GlobalDeleteAtom")
02120  PRINT 'DROP'(-1),'SHOW'(2),
02130  STOP
03000 ! !^1000 - Receive CTL values from Sender
03010 RECEIVER:
03020  PRINT 'CAPTION'("PVXWIN32 - Receiver"),'MOVE'(0,INT(MAX_ROW/2)+1),
03030  INPUT "Press F4 when finished ",*;
       PRINT @(0),'BR',"Received CTL=",CTL,'ER','CL'
03040  IF CTL=4 \
        THEN GOTO 3200
03050  IF CTL<>104 \
        THEN GOTO 3030
03060  ATOM=TCB(81) ! Atom identifier
03070  DIM ATOM$(512)
03080  X=DLL("KERNEL32.DLL","GlobalGetAtomNameA",ATOM,ATOM$,LEN(ATOM$))
03090  X=DLL("KERNEL32.DLL","GlobalDeleteAtom") ! Release Atom
03100  X=POS($00$=ATOM$);
       IF X \
        THEN ATOM$=ATOM$(1,X-1)
03110  PRINT "Received: ",'BR',ATOM$,'ER'
03120  GOTO 3030
03200 ! !^100
03210  STOP
```

DLL Calls via WindX

Be aware of the following issues when accessing DLLs from within WindX:

- Know where to find the data – *is it on the client or the server?*
- Do not pass pointers to data if they exist on a server to a DLL call that is being handled by a WindX workstation.
- DLLs on a WindX workstation cannot access a memory location on the server.
- It is often best to encapsulate DLL work into a single program that can be called from the server via the **[WDX]** command tag; e.g., `CALL "[WDX]...`

More information on this subject can be found in the *ProvideX Client Server Reference*.

Working with DLL Calls in UNIX/Linux

As mentioned earlier, the function **TCB(196)** will return 1 to indicate UNIX/Linux support. One of the primary conditions for this feature is that the libraries accessed must be shared. This is demonstrated by the suffix "`.so`".

By typing "`info libc`" at a shell prompt on a Linux machine, the `info` command will return all the information held in the library. Choose the feature within the library you wish to use, and then access the library.

In the following example, the `getpwnam` command is used from the `lib.so.6` on a Red Hat system. When the man pages for `getpwnam` are examined, the format is as follows: `Struct passwd *getpwnam(const char *name);`

The result is a pointer to a structure containing the fields of the record in the password database. The structure is as follows:

```
Char    *pw_name    user name
Char    *pw_passwd  User password
Uid_t    pw_uid     User ID number
Gid_t    pw_gid     User Group number
Char    *pw_gecos   Real Name
Char    *pw_dir     Home Directory
Char    *pw_shell    Shell Command
```

The ***** *asterisk* marks the field as a pointer.

```
0010 ! ACCESS_LIBRARY
0015 INPUT "Enter user name ",NAME$
0017 PW_GID=0,PW_UID=0
0020 LIBRARY$="libc.so.6" ! Shared object library on Red Hat Linux
0030 LET SOME_VARIABLE=DLL(ADDR LIBRARY$,ERR=*NEXT);GOTO 50
The purpose of the ADDR  is hold the library in memory.
0040 ! Could not open the library.
0045 GOTO WRAP_UP
0050 LET STS_OF_CALL=DLL(FIND SOME_VARIABLE,"getpwnam",ERR=*NEXT);GOTO 70
```

This statement finds the "C" library call in the library opened with the handle of `SOME_VARIABLE`.

```
0060 GOTO WRAP_UP
0070 DIM X_NAME$(64,$00$);LET MID(X_NAME$,1,LEN(NAME$)=NAME$
0080 LET X=DLL(*,STS_OF_CALL,X_NAME$)
0090 IF X=0 THEN LET NAME$=NAME$+"  INVALID";GOTO WRAP_UP
```

After looking at the structure of the return value the memory will need to be assigned.

```
0100 LET X$=MEM(X,(5*TCB(301))+(2*TCB(306)))
```

This returns 5 pointers and two integers.

```
110 PW_UID=DEC($00$+SWP(X$(TCB(301)+TCB(301)+1,TCB(306))))
```

The decimal value of the swapped memory location represents the user ID number. The **TCB(301)** is the pointer size and the **TCB(306)** is the integer size made available because of 64 bit OS.

```
120 PW_GID=DEC($00$+SWP(X$(TCB(301)+TCB(301)+TCB(306)+1,TCB(306))))
150 WRAP_UP: ! WRAP UP
160 PRINT NAME$,STR(PW_UID:"###"),STR(PW_GID:"###")
```

ProvideX COM Support

The **COM** interface in ProvideX allows developers to integrate external components produced by third-party vendors into their ProvideX applications in MS Windows. It provides convenient syntax for obtaining information about, as well as access to, the internal properties and methods of a *Component Object Model* (COM) object. This functionality is carried out via the COM ***automation*** standard, which is an implementation of the IDispatch interface in Windows. (This was commonly known as **OLE** *automation* in earlier Windows versions.)

When an application or library supports automation, the objects exposed by the application can be accessed through the ProvideX COM interface and manipulated to invoke their methods and get or set their properties. For example, a spreadsheet application might expose a worksheet, chart, cell, or range of cells, each as a different type of object; and a word processor might expose objects such as applications, documents, paragraphs, bookmarks, or sentences.

Topics

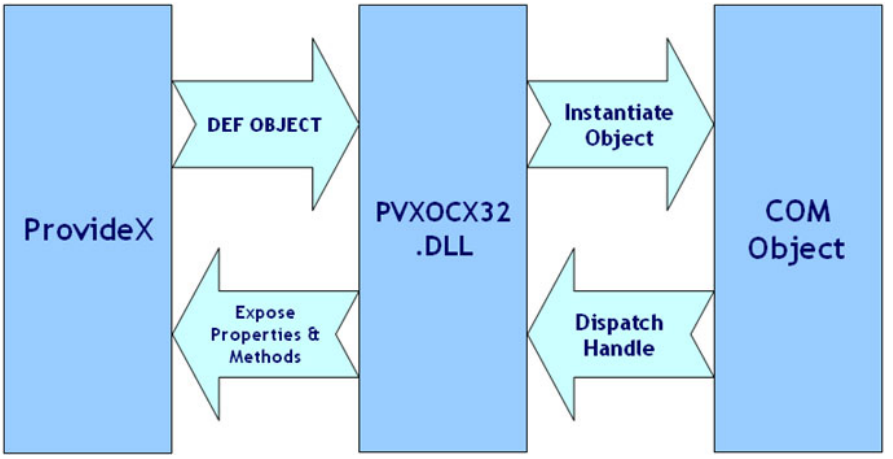
COM Concepts

The following terms / definitions apply to automation and the ProvideX COM interface:

Control	An object that exposes a user interface; e.g., a dialogue with OK and Cancel buttons can be implemented as a control. These are now typically based on **ActiveX** vs. the older **OLE** control technology (**OCX**).
Object	Any item that can be programmed, manipulated or controlled. Interfacing with an object is done through property setting and getting, and calling of methods.
Property	A property is a characteristic of an object (an adjective). For example, properties of a `Textbox` object might include: `Name`, `Visible`, `Forecolor` etc.
Method	A function that performs an action on an object (a verb). For example, an `Application` object might expose a `Close` method.
Instantiation	To create an instance of (instantiate, define, name) an object.
Binding	The process of connecting property and method calls to an object.
Late Binding	Obtaining a reference to an object without any prior information about the object. Property and method names are resolved at run time. This is the binding style used by ProvideX.

To program against an object, a reference to that object must first be obtained. This process is commonly referred to as ***binding***. Unlike other languages, ProvideX simplifies this process by providing one statement that can be used to create new objects, reference running objects, and connect to remote objects.

The following illustration outlines the processes involved in the access and manipulation of COM objects in ProvideX.



Communication between an application and a COM object is performed either by reading or writing the object's properties or by invoking methods within the object. The ProvideX **Apostrophe Operator** (a.k.a. *tick*) is used to access properties within a COM object and invoke its methods. See **Accessing an Object's Properties and Methods**, *p.266*.

The ProvideX commands used in the handling of COM objects are as follows:

DEF OBJECT Directive used to create a new instance of a COM object. See **Referencing a COM Object**, below.

DELETE OBJECT Directive used to disconnect from a COM object. See **Releasing an Object Reference**, *p.266*.

DROP OBJECT Alternative to **DELETE OBJECT** directive. The two directives may be used interchangeably in ProvideX applications.

ON EVENT Directive used to process COM control events in a ProvideX application. See **Event-Driven COM**, *p.294*.

FUNCTION Directive supports the **FOR EVENT** keywords for setting a method to be processed for a particular incoming event. See **Event-Driven COM**, *p.294*.

Referencing a COM Object

The **DEF OBJECT** directive is used to create a new instance of a specified object.

DEF OBJECT *obj_id*,[@(*col,ln,wth,ht*)]{, | **=** } *obj_name*$ [;**LICENSE**=*key*] [;**FINALIZE**=*method*] [,**ERR=***stmtref*]

Where:

@(*col,ln, wth,ht***)**	Numeric expressions. Column and line coordinates for top left corner, width in number of columns and height in number of lines.
FINALIZE=*method*	Optional method name of the object instance to be run upon release of the object. See **Finalize Method**, *p.265*.
LICENSE=*key*	Optional license key that should be applied when attempting to bind to an object. See **Using Licensed Objects**, *p.265*.
obj_id	Numeric variable that will be used to save the object reference.
obj_name$	String expression identifying the object to be referenced, as well as any object-specific parameters. See **Object Name Contents**, *p.263*.
ERR=*stmtref*	Optional program line number or statement label to which to transfer control in case of error. If an error occurs during the **DEF OBJECT** statement, the error code will always equal 12. Use the **MSG(-1)** function to obtain further details on the failure.

Upon successful execution of **DEF OBJECT**, a reference to the object *obj_name$* will be placed into the supplied numeric variable *obj_id*. Leave out the column and line coordinates for non visible objects. Use "*****" to list all registered COM controls.

Object Name Contents

The options below may be used in the **DEF OBJECT** *obj_name$* string. Square brackets are part of the statement's syntax (see **Example Statements**, *p.265*).

*****	Use an * ***asterisk*** to display a pop-up window listing all 32-bit COM controls installed on the system.
CLSID	Class identifier (GUID) for the object in the format {*hhhhhhhh-hhhh-hhhh-hhhhhhhhhhhh*}, where the *h* indicates a hexadecimal character.
progID	Programmatic identifier name for the object. An example of this is `Word.Document`.
[DESIGN]*filename*`.XML`	Indicates that a previously saved **OLE** / **ActiveX** control definition should be loaded from the specified `.XML` file. For more information, see **PVXSAVE**, *p.272*.

[DCOM]*server;name* Indicates that the object is located on a remote system. *server* parameter is optional, and can be specified either by name, or by IP address. If not supplied then the object is considered local. *name* parameter is the *CLSID* or *progID* for the object.

[FILE]*x:\filename* Indicates that the object should be created using the specified file name. An example of this would be a Microsoft Word document file.

[GETOBJECT]*name* Indicates that ProvideX should bind to a running instance of the named object. *name* parameter is the *CLSID* or *progID* for the object or a file-based moniker. File monikers are commonly used when dealing with WMI, LDAP, and related services in Windows.

[GLOBAL]*name* Indicates that a reference to an object exposed by the use of `PvxMakeGlobal` should be obtained. The *name* parameter is the name used to expose the object.

[REGISTER]*x:\filename;name*

Ensures that the object information is properly registered before attempting to create an instance of the object. *x:\filename* parameter is the name of the executable file or library that exposes the automation object. *name* parameter is the *CLSID* or *progID* for the object.

[RUNNING]*name* Indicates that ProvideX should bind to a running instance of the named object, where *name* is given as *CLSID* or *progID*. An error occurs if the object is not currently running.

[RUNNING OR NEW]*name*

Indicates the same functionality as **[RUNNING]** syntax; however, if the object is not currently running, ProvideX attempts to create a new instance of the named object.

[PICTURE]* Indicates creation of an empty `IPicture` object.

[PICTURE]*filename* Indicates that an `IPicture` object should be created and the specified image file should be loaded by the object.

[PICTURE]*[#]*name*;{**BMP**|**CUR**|**ICO**}

Indicates that an `IPicture` object should be created and specified resource contents loaded by the object. For numeric resources, *name* should be prefixed with a **#**; e.g., `#101`. The image type is specified as the second parameter, and indicates the resource group that contains the desired resource.

Using Licensed Objects

Redistribution of a third party COM control may sometimes require the use of a ***license file*** (usually identified by a `.lic` extension). The license file usually permits developer-level access to the control and is not for redistribution. In some cases, a ***license key*** must be extracted from the license file in order for the control to function in run-time mode.

The following steps outline how to extract the license key from the license file, and how to make it available in a run-time environment:

1. On the system where the COM object and license file have been installed, obtain a reference to the object without specifying the license information.
2. Query the `PvxLicense$` property of the object for the license key. If the object is licensed, the key data is returned as a string of hex characters.
3. Add the **LICENSE**=*key* data to the **DEF OBJECT** statement.

Once the new **DEF OBJECT** statement has been generated, the object reference can then be obtained on systems that do not have the license file installed.

Finalize Method

A method of the object instance can be specified to run when the object is released. This method should not require any parameters to be passed to it. This is intended to simplify the handling of automation "servers" (such as Word or Excel) that require the `Quit()` method to be executed in order to shut down the server. By specifying a *finalize* method, the `Quit()` method (or similar) of an object does not need to be called in order to free resources. See **Releasing an Object Reference**, *p.266*.

Example Statements

The following are examples of **DEF OBJECT** statements:

```
DEF OBJECT X, "*"
DEF OBJECT X, "Word.Application;Finalize=Quit", ERR=*NEXT
DEF OBJECT X, @(1,1, 70, 20)="Word.Document"
DEF OBJECT X, "[DCOM]MyServer;Shell.Explorer"
DEF OBJECT X, @(10, 2, 20, 10)="[File]c:\my documents\test.doc"
DEF OBJECT X, "VCF1.VCF1Ctrl.1;License=1234567890ABCDEF..."
DEF OBJECT X, "[Running or New]Excel.Application"
DEF OBJECT X, "[Picture]*PVXHAPPY;cur"
DEF OBJECT X, @(40, 1, 40, 15) = "[Design]chartfx.xml"
DEF OBJECT X, "[GetObject]winmgmts:\\.\servername\root\cimv2"
```

The **DEF OBJECT** statement can also be used to bind child objects which are returned as the result of either a property access or method call. The syntax is as follows:

```
DEF OBJECT X
```

Releasing an Object Reference

In order to manage the life time of an object, the ProvideX developer has been provided with the **DELETE OBJECT** statement.

DELETE OBJECT *obj_ID*[,**ERR=***stmtref*]

Where:

obj_ID Numeric variable name of object reference.

stmtref Program line number or statement label to which to transfer control.

The **DELETE OBJECT** statement takes one parameter, which is the ProvideX variable that has been bound to an object reference. After execution of this statement, the reference to the object is terminated, and the object is released from memory.

When dealing with automation "servers" (multi-client applications such as MS Word or Excel), it may be necessary to execute a method of the object in order to close the application; e.g., for MS Office applications, this method is `Quit()`. If a **FINALIZE=** *method* was specified in the **DEF OBJECT** statement, the method name will be executed automatically before the object is released.

Note: For child objects, it is an error to perform a **DELETE OBJECT** if a **DEF OBJECT** has not been performed first. All child objects will be automatically released from memory when the parent object is released.

Accessing an Object's Properties and Methods

Once a reference to an object has been obtained, the next step would be to manipulate or control the object. Use the ***tick-star*** (**'***) internal property to find out which properties and methods are available:

PRINT *obj_ID*'*

This statement returns a comma separated list of all exposed properties and methods. Methods are indicated by having trailing parentheses "( )" appended to their name.

This list is merely an overview of the object, and does not include data type or parameter information.

Note: Some objects return a listing that only contains the ProvideX extended properties and methods (explained later). This occurs when the object does not expose run-time type information.

Proper object documentation is very important. Without proper documentation, it will be impossible to tell:

- What parameters are to be passed to methods
- What return values can be expected from properties or methods
- What parameters are considered by reference, vs. those that are by value

- What method parameters are optional
- What properties are indexed, and what data types are used for the indexes.

Sage only provides support for the ProvideX object ***interface***. Specific interface information for an object must be obtained from the respective vendor.

Retrieving Property Values

To retrieve a property value, place the object variable and property name on the right hand side of the assignment:

variable **=** *object'property* **[$]**

Automation is case insensitive; therefore, a property name can be written in upper, lower, mixed, or proper case. If the property is to return a string data type, then a **$** symbol should be placed at the end of the property name. Properties may also be collections or arrays, which would require a slightly different syntax when retrieving values:

variable **=** *object'property* **.get[$](***index* **[$], ...)**

The **.get** that is appended to the property name indicates to ProvideX that this is a property, and not a method call. For string type properties, the **$** symbol should be placed at the end of the **.get** and before the open parentheses. The *index* parameter indicates the property element to retrieve. Unlike ProvideX arrays, the index for the property might not be a numeric data type (check the object documentation).

Example:

```
STYLE=DOCUMENT'Styles.get("Normal")
```

Where `Styles` is a property and `"Normal"` indicates the indexed value to retrieve.

Note: Some objects allow indexed property access without specifying **.get** notation.

Assigning Property Values

To assign a property value, place the object variable and property name on the left hand side of the equation:

object'property **[$] =** *variable*

Automation is case insensitive. Therefore, a property name can be written in upper, lower, or mixed case. If the property is to return a string data type, then a **$** symbol should be placed at the end of the property name. Properties may also be collections or arrays, which would require a slightly different syntax when assigning values:

result **=** *object'property* **.put**(*index*[**$**], …, *data*[**$**])

This syntax is identical to a method call, but with a few exceptions. The *result* of the property assignment is always zero, and it can be disregarded. The **.put** indicates to ProvideX that this is a property, and not a method call. And finally, the *data* to be assigned to the property is passed in as the last parameter between the parentheses.

This syntax style is also required when assigning an object to a non-index/non-array property. The reason for this is due to a syntax conflict in ProvideX, where the following would be invalid:

*object'property=*other_object*

The correct syntax for the above example would be:

result=*object'property*.**put(***other_object***)**

Note: When assigning or passing an object, it is required that an asterisk (*) appear before the object reference. This allows ProvideX to differentiate between a variable holding a numeric value, and one that holds an object reference.

There may also be cases where the property assignment is expecting to be set by reference. A common example occurs when an object property is changed to refer to a new object. For most properties that are object types, the **.put** syntax will work correctly. If an error does occur, then the following syntax should be tried:

result=*object'property*.**putref(***otherobject***)**

Calling Methods

To call a method, place the object variable, method name, and parameter list on the right hand side of the equation:

result=*object'method***[$] (***param1***,** *param2* **[, ...])**

Automation is case insensitive; therefore, a method name can be written in upper, lower, mixed, or proper case. If the method is to return a string data type, then a **$** symbol should be placed at the end of the method name. Some methods are written to accept ***optional*** parameters. In ProvideX, these would be passed using an * ***asterisk***. For further details, see **Passing Optional Parameters**, *p.287*.

Invocation Hints

Due to ProvideX syntax restrictions, desired control of how a method is invoked and how data is returned, a set of invocation hints have been developed that can be used to direct the control of the COM interop layer. The valid invocation hints are listed below:

`.GET` Indicates that the call should be performed as a property "get". This is normally required when dealing with indexed properties, as the syntax of the statement is translated as a call; e.g.,

```
S$ = OBJ'CELLS.GET$(1, 2)
```

`.PUT` Indicates that the call should be performed as a property assignment. This is normally required when dealing with indexed properties, as the syntax of the statement is translated as a call; e.g.,

```
OBJ'CELLS.PUT$(1, 2, S$)
```

This syntax is also required when assigning an object to a property; e.g.,

```
OBJ'SELRANGE.PUT$(*X)
```

.PUTREF	Indicates that the call should be performed as a property reference assignment. This is normally required when assigning an object reference to an object's property.
.CALL	Indicates the call should be performed as a method only. Normally, the COM interop layer will attempt (through trial and error), to resolve the call as either a method or property.
.GETB$	Indicates that the returned property data, if a string type, should not be converted from double byte to ANSI string.
.CALLB$	Indicates that the returned method data, if a string type, should not be converted from double byte to ANSI string.
.GETV	Indicates that the returned property data should be returned as variant vs. its base data type. Do not use on a property that returns an internal object; e.g., ***CONSTANT**, ***PROXY**, ***ERROR**, ***MASTER**, etc. are internal objects that cannot be expressed in a variant.
.CALLV	Indicates that the returned method data should be returned as variant vs. its base data type. Do not use on a method that returns an internal object; e.g., ***CONSTANT**, ***PROXY**, ***ERROR**, ***MASTER**, etc. are internal objects that cannot be expressed in a variant.

These hints are added to the end of a method / property name so that the call reads as *object'method***{.***hint***}(***parameters***)**. If you attempt to get or set array properties without using an "invoke hint", the DLL will attempt to resolve the calling type; however, it can only do this by trial and error (up to 4 separate attempts).

The following is an example of a grid object that exposes a CELL property that is accessed using a row and column indicator:

```
10 DATA=GRID'CELL.GET(ROW, COL)        ! Get the cell value
20 DATA= DATA+10                       ! Add 10 to the value
30 NULL=GRID'CELL.PUT(ROW, COL, DATA)  ! Set the cell value
```

For speed reasons, as well as clarity, the developer should always specify a hint when calling a property using the syntax of a method call. In the previous example, line 30 should be changed to:

```
30 NULL=X'VAL.PUT(*Y) ! Assign object Y to X'VAL
```

Extended Properties and Methods

During the development of the ProvideX COM interface, it was realized that developers may require information and helper functions that are not exposed by all COM objects, such as the container window handle, or the internal COM object class name, etc. To accommodate this, a set of properties and methods were created for access by all COM objects instantiated in ProvideX. The DLL handles the execution

of these internally, but to the developer, they function identically to the other COM properties and methods of the object. Following is a list of additional COM members available for use with ProvideX.

PVXALIAS(*member$***,** *user$***)**

Method Call. This method takes two string parameters. The first, *member$*, indicates the actual COM member name to alias. The second, *user$*, is the user-defined alias name to use. Upon success, the object can be accessed using the alias name in place of the actual name. For example, if an object had a property called DAY, an Error 20 would occur when attempting to access it. To correct this:

```
Z = OBJECT'PVXALIAS("DAY", "_DAY")
A$ = OBJECT'_DAY$
```

PVXALLOCRECORD(*RecordName$***)**

Method Call. This method allocates an instance of a record specified by the *RecordName$* parameter. For a list of available record types, see **PVXRECORDS$.**

PVXCONSTANTS ***Read Only Property.*** This property is used to return a ***CONSTANTS** object that is built from the current object's type library enumerated constants. The returned ***CONSTANTS** object will expose all constant values as read only properties. The ***CONSTANTS** object is described in more detail under **Extended Objects**, *p.272*.

PVXDESCRIBE$(***member$*)**

Method Call. This method accepts one string parameter, which is the actual COM member name to "describe" (requires type information to be available). If successful, returns a multi line string of information describing the member, its return type, as well as parameters and their types. This can also be used to display field information for record type objects by passing the field name in the *member$* parameter.

PVXDOVERB(*verb***)** ***Method Call.*** This method takes one numeric parameter which is the verb to perform on the COM control. If the object does not expose a control, then no action is performed. Common verbs are as follows: `Primary` (0), `Show` (-1), `Open` (-2), `Hide` (-3) , `UIActivate` (-4), `InPlaceActivate` (-5), `DiscardUndoState` (-6).

PVXERROR[$] ***Read Only Property.*** Returns the last error code, or message (depending on $ suffix) that occurred when accessing a property or method of the object.

PVXEVENTS[$] ***Read Only Property.*** Returns the ProvideX class object that is handling events, or it provides a list of events names (depending on $ suffix), for the object. See **Event-Driven COM**, *p.294*.

PVXEXTDATA[$] ***Read Only Property.*** This property provides buffering for returned string data that exceeds 32K. If a result returns more than 32K of data, the first 32K bytes will be returned, and the remaining data can be accessed via this property. The numeric value is the amount of data remaining in the buffer, the string value is the next 32000 bytes, or whatever is remaining.

PVXFOREACH ***Read Only Property.*** This property is only valid for objects that expose a collection (`IEnumVariant`), or ***VARARRAY** objects that contain a single array dimension. Attempting to read this property for any other object type will return an error. On success, a ***FOREACH** iterator object will be returned that allows traversal of the object's data. The ***FOREACH** object is described under **Extended Objects**, *p.272*.

PVXFREE(["*children***"])**

Method Call. This method will release the instance of the object. This is useful for sub objects, when **DEF OBJECT** has not been called. It also accepts one optional string parameter, which if passed, should be set to "*children*". This will cause all the children of the object to be released, but will not release the object itself. Any other setting will cause the object to be released.

PVXHANDLE ***Read Only Property.*** Returns the window handle for the container window that is hosting the control. If the object is not a control, a zero will be returned.

PVXHEIGHT ***Read / Write Property.*** Used to set the height, in pixels, of a control. If the object is not a control, then setting this property has no effect, and reading this value will return a value of (-1).

PVXID ***Read Only Property.*** This property returns the interop handle to the object. It can be assigned to another integer variable, which can then be used in a **DEF OBJECT** statement. This is useful for situations in WindX where the ProvideX local variable goes out of scope.

PVXISA$ ***Read Only Property.*** Returns the internal COM class interface name from the associated type library, if available. If no type library is available, a blank string is returned.

PVXLEFT ***Read / Write Property.*** Used to set the left border, in pixels, of a control. If the object is not a control, then setting this property has no effect and reading this value will return a value of (-1).

PVXLICENSE$ ***Read Only Property.*** Returns a hexadecimal string for objects that utilize license keys (providing the key is available). This string can then be used in a **DEF OBJECT** statement, which will allow the code to create an instance of the object without the key being available (runtime client sites).

PVXMAKEGLOBAL(*global$***)**

Method Call. This method accepts one string parameter, which is the name to make "global". If successful, another ProvideX session can access this object though a **DEF OBJECT** statement using the **[GLOBAL]** parameter. This allows a ProvideX session to expose objects to other sessions in a server like fashion. This is only valid for COM based objects.

PVXMODE ***Read / Write Property.*** For controls that differentiate between development and run time mode, this is used to set the "user mode" state. Setting this to zero will place the control in development mode, any other value will place it into run time mode. If the object is not a COM control, then setting this property has no effect.

PVXNAME ***Read Only Property.*** Returns the string used to create the instance of the object. For sub objects, the string also includes the property / method names.

PVXPARENT ***Read Only Property.*** Returns the parent handle (`interop id`, also see **PVXID**) for the object, or zero if the object is top level.

PVXRECORDS$ ***Read Only Property.*** Returns a comma delimited list of record type names that are available from the type library associated with the calling object.

PVXSAVE(*FileName$***)**

Method Call. Used for saving a control's design time property set. On success (return not zero), the property set will have been saved as XML content within the specified file. This allows a developer to modify a control during design time, and to then reload it using the **[DESIGN]** option later.

PVXTOP ***Read / Write Property.*** Used to set the top border, in pixels, of a control. If the object is not a control, then setting this property has no effect and reading this value will return a value of (-1).

PVXTYPELIB$ ***Read Only Property.*** Returns the file name that exposes type information for the object. (type information must be available).

PVXWIDTH ***Read / Write Property.*** Used to set the width, in pixels, of a control. If the object is not a control, then setting this property has no effect and reading this value will return a value of (-1).

Extended Objects

Extended objects complete the COM functionality and provide a mechanism for future enhancements. These pseudo objects are instantiated in the same way as all other COM objects within ProvideX, except for the fact that an asterisk must preface the object name; i.e.,

DEF OBJECT "*{*extended name***}"**

It should be noted that while these objects appear identical to other COM objects, they are not COM based in nature. This means that extended objects: do not expose events, do not expose controls, cannot have member information "described", and cannot be made global to other processes.

Also note that the ***FOREACH**, ***RECORD**, and ***CONSTANTS** object types are not able to be directly created. These object types are returned (respectively) as results from **PVXFOREACH**, **PVXALLOCRECORD** and **PVXCONSTANTS** (see **Extended Properties and Methods**).

The currently supported (internal) extended objects are listed below:

***CONSTANTS**	Provides a mechanism for exposing all enumerated constant values from an object's type library.
***ERROR**	Wrapper export for last error information.
***FOREACH**	Wrapper around an `IEnumVariant` interface, which is more commonly referred to as a "collection". ProvideX allows this wrapper to be created on both collection objects as well as single dimension COM SafeArrays. Provides a means of iterating data without regards to the lower or upper bounds of the container.
***GLOBAL**	Provides an enumerator for the system wide list of objects exposed using `PvxMakeGlobal`.
***MASTER**	Serves as a system object for the interop layer. Provides object iteration capabilities, version information, etc.
***PROXY**	Allows the developer to expose instantiated ProvideX objects directly to the COM world.
***RECORD**	Wrapper around an `IRecordInfo` interface, which is also referred to as a COM user defined type (record structure).
***VARIANT**	Wrapper around an OLE variant data type. Provides a means for passing data by reference, as well as converting data from a ProvideX type to Automation types.
***VARARRAY**	Wrapper around an OLE safe array of variants. Provides a mechanism for COM array data handling.

These extended objects are described in detail in the sections that follow.

*CONSTANTS

The goal of the constants object is to expose all the enumerated constant values from an object's type library. For example, Microsoft Word exposes over 1100 constants from its type library. Attempting to use any of these values would require the developer to research the type library to locate the constant name, in order to find its

associated value. The developer would then be forced to code these constants into his/her program in order to use them. Using the constants object provides the following benefits:

- Coding effort is reduced, as a single constants object contains all constant values from a type library.
- The developer can refer to the constant values by name, which makes transposing code (e.g. from VB, etc.) a much simpler task.

The following belong to the ***CONSTANTS** object:

{CONSTANTNAME} ***Read Only Property.*** All constant names from a type library are exposed as read only properties. The return type for each constant value will be an integer.

The following example demonstrates how to acquire an instance of a constants object and then utilize a constant value in code.

```
0010 DEF OBJECT WORD, "Word.Application;Finalize=Quit"
0020 LET WORD'VISIBLE = 1
0030 LET WORDCONSTS = WORD'PVXCONSTANTS
0040 LET KEYCODE = WORD'BUILDKEYCODE(WORDCONSTS'WDKEYALT,
      WORDCONSTS'WDKEYF1)
0050 WORDCONSTS'PVXFREE()
0060 ESCAPE
0070 DELETE OBJECT WORD
```

*ERROR

This static table is exposed as another internal object. It can be **DEF**'ed after an error has occurred and will still contain the last error information. One note in regards to this class: while multiple instances can be created, they all point to the same data block; i.e., clearing one ***ERROR** object will in effect clear all other instances. The following members belong to the ***ERROR** object:

CLEAR( ) ***Method Call.*** Clears the last error information.

CODE(*Number***)** ***Method Call.*** Returns the OLE code for the passed number, which is treated as an HRESULT to evaluate. If no parameter is passed, the code for the **OLEERROR** property is returned.

DESCRIPTION$ ***Read Only Property.*** Contents depends on if the last error was DISP_E_EXCEPTION or not. If so, it returns the object defined message. If not, it returns the string representation of the OLE HRESULT based on the **OLEERROR** property.

FACILITY(*Number***)** ***Method Call.*** Returns the OLE facility for the passed number, which is treated as an HRESULT to evaluate. If no parameter is passed, the facility for the **OLEERROR** property is returned.

HELPCONTEXT	***Read Only Property.*** If last error code was DISP_E_EXCEPTION, then this property will return the associated help context number, if available.
HELPFILE$	***Read Only Property.*** If the last error code was DISP_E_EXCEPTION, then this property will return the associated help file name, if available.
NUMBER	***Read Only / Default Property.*** Returns last object code set for a dispatch call that failed with DISP_E_EXCEPTION.
OLEERROR	***Read Only Property.*** Returns the last OLE HRESULT code that was raised.
SEVERITY(*Number***)**	***Method Call.*** Returns the OLE severity for the passed number, which is treated as an HRESULT to evaluate. If no parameter is passed, the severity for the **OLEERROR** property is returned.
SOURCE$	***Read Only Property.*** If the last error code was DISP_E_EXCEPTION, this property will return a textual, human-readable name of the source of the exception, if available.

*FOREACH

When dealing with collections, one of the biggest pitfalls is determining if the collection index is zero or one based. To aid the developer, an enumerator object of ***FOREACH** is made available from all collection objects (that expose the IEnumVariant interface). To access this object, the **PVXFOREACH** property must be queried; e.g.,

```
FOREACHOBJ = OBJECT'PVXFOREACH
```

Once returned, the ***FOREACH** object can be used to access each element in the collection. In addition to collections, the interop layer also allows the developer to obtain a ***FOREACH** object from an instance of a ***VARARRAY** object, providing the underlying COM SafeArray only contains a single dimension.

Note: A ***FOREACH** object obtained from an array becomes a snapshot of the original array. Any changes to the original array will not be reflected during enumeration.

The following members belong to the ***FOREACH** object:

DATA[$]	***Read Only / Default Property.*** Returns the item data that was returned from a call to **NEXT( )**. The actual data type depends on the collection or array item being enumerated.

NEXT( ) ***Method Call.*** This method reads the next available collection or array item into the **DATA** property and returns *true* on success. If no further items exist, *false* will be returned. This method must be called (and return success) before accessing the **DATA** property.

RESET( ) ***Method Call.*** Resets the "for each" enumerator and clears the **DATA** property. This allows item enumeration to start from the first item. Only required if a second (or more) iteration is desired.

The following example demonstrates how to acquire an instance of a ***FOREACH** object and then enumerate all items in a collection:

```
0010 DEF OBJECT WMI, "[GetObject]winmgmts:\\.\root\cimv2"
0020 LET ITEMS = WMI'EXECQUERY("Select * from
     WIN32_NetworkAdapterConfiguration where IPEnabled = True")
0030 LET ITER = ITEMS'PVXFOREACH
0040 WHILE ITER'NEXT()
0050 PRINT ITER'DATA'PROPERTIES_("MACAddress")'VALUE$
0060 LET IPITER = ITER'DATA'PROPERTIES_("IPAddress")'VALUE'PVXFOREACH
0070 WHILE IPITER'NEXT()
0080 PRINT IPITER'DATA$
0090 WEND
0100 IPITER'PVXFREE()
0110 WEND
0120 ITER'PVXFREE()
0130 ESCAPE
0140 DELETE OBJECT WMI
```

*GLOBAL

The purpose of the ***GLOBAL** object is to provide an enumerator for the system wide list of objects exposed using PvxMakeGlobal. With this enumerator, a developer can quickly determine the available global objects, and can bind to any object within this list. The following members belong to the ***MASTER** object:

BINDTO(*index/name$*) ***Method Call.*** Returns a reference to the global object specified by either name or index of item in the global list.

COUNT ***Read Only Property.*** Returns the count of objects in the system wide list.

ITEM$(*index*) ***Method Call.*** Returns the name of the global object at the specified index. Please note that **ITEM$** should be treated as a zero-based array.

REFRESH( ) ***Method Call.*** Refreshes the system wide list of global objects.

*MASTER

The master object provides direct access to the object list maintained by the interop layer, as well as any future global settings that may be introduced. With this access, a developer can quickly determine the number of objects in use, which is essential when tracking down code that may not be handling sub object references properly. The following members belong to the ***MASTER** object:

CALLCOUNT — ***Read Only Property.*** Returns the number of command call executions since the interop library was initialized.

COUNT — ***Read Only Property.*** Returns the number of objects being managed by the interop layer (includes the master object in this count).

ITEM(*index***)** — ***Method Call.*** Returns the interop handle for the object at the specified *index* (see **PVXID**). Please note that **ITEM** should be treated as a zero-based array.

LISTPVXMETHODS — ***Read / Write Property.*** Determines if the internal PVX methods should be listed when the **'*** property of an object is queried. Defaults to true (-1).

VERSION$ — ***Read Only Property.*** Returns the file version number for the interop library PVXOCX32.

*PROXY

The proxy object allows the developer to expose instantiated ProvideX objects directly to the COM world. A good example of this would be to a hosted scripting language. In order to use the proxy, a binding to a ProvideX object must first be established. This is done using the **ON EVENT FROM** syntax; e.g.,

```
0010 oCol = NEW("*obj/Collection")
0020 DEF OBJECT colProxy, "*PROXY"
0030 ON EVENT FROM colProxy PROCESS oCol
```

Note: The ***PROXY** object is not intended to be used directly from ProvideX as it is only a translation layer for the COM to ProvideX method calls. Because of this, issuing a **'*** against the proxy will only return the PVX prefixed internal method names. Once the binding has been done, the proxy wrapper may be passed to a COM object using the standard ***** notation for objects.

The following members belong to the ***PROXY** object:

ASOBJECT(*ObjID***)** — ***Method Call.*** Allows the COM client to wrap a (ProvideX) object handle. If the *ObjID* represents a valid ProvideX object handle, an internal proxy wrapper will be created and the dispatch interface will be returned.

INVOKE(*name, flag* [, *parameters*…]**)**

Method Call. Executes the property or method name specified by the *name* parameter. The *flag* value must be one of the following: **1** (property get), **2** (property set), or **3** (method call). Any *parameters* that should be passed to the ProvideX object will then follow.

This method is provided to handle overload situations where it can be unclear which property or method should be called, e.g.,

```
PROPERTY TEST
PROPERTY TEST$
FUNCTION TEST()
FUNCTION TEST$()
```

Would translate to the following Invoke calls.

```
.Invoke("TEST", 1)
.Invoke("TEST$", 1)
.Invoke("TEST", 3)
.Invoke("TEST$", 3)
```

EVALUATE(*statement***)** ***Method Call.*** Passes the statement string parameter to the ProvideX interpreter for evaluation.

EXECUTE(*statement***)** ***Method Call.*** Passes the statement string parameter to the ProvideX interpreter for execution.

*RECORD

COM user defined types (UDT) or records are handled using this object type. A record object may be returned as a result of a method or property call, or can be directly created using the **PVXALLOCRECORD** method. Each field in the record will be handled as a read / write property, and will be displayed when the **'*** listing is queried. The following is a predefined member in the record object:

PASSBYREF ***Read / Write / Default Property.*** Determines if the `IRecordInfo` interface will be passed by reference. Setting the property to zero indicates false, any other value indicates true. The default setting for this property is true.

*VARIANT

Being a wrapper around an OLE variant, this object can store any data type, including other objects. It also provides a means for converting data in place, and handling data types that are not directly supported by ProvideX. Its primary purpose is for method calls that expect data to be passed ***by reference***, but it is also useful when a COM object will only handle data of a specific type.

The following members belong to the ***VARIANT** object:

ADD(*data*[$]**)** ***Method Call.*** Adds the value of the data parameter to the currently held variant data. Logical use is for building strings greater than 32K in length, but can be used to add numbers, etc. Calling this method will perform an implicit conversion to the data type of the passed parameter.

ADDB(*data*[$]**)** ***Method Call.*** Adds the string data parameter to the currently held variant string data. No conversion to Unicode is performed. Requires the passed data parameter to be a either a string or a variant object that contains a string. Also requires the current data to be a string type.

CLEAR( ) ***Method Call.*** Clears the contents of the variant. After the **CLEAR** finishes, the data type for the variant will be set to E(*mpty*). If the variant contained an object or array reference, the object will be released.

EXPLICITBYREF ***Read / Write Property.*** Determines how the variant will be passed if **PASSBYREF** is set to true. When this property is set to false (zero), a pointer to the variant will be passed to the COM method. When set to true (non zero), a pointer to the actual data will be passed. The default setting for this property is false.

LEN ***Read Only Property.*** Returns the length of the data held by the variant. Logical use is primarily for string data.

LENB ***Read Only Property.*** Returns the length of the data held by the variant. This is identical to the **LEN** property except when the variant contains string data. For strings, the actual byte count is returned, which is not the same as the character count when dealing with Unicode strings.

PASSBYREF ***Read / Write Property.*** Determines if the variant will be passed by reference. Setting the property to zero indicates false, any other value indicates true. The default setting for this property is true. Also see **EXPLICITBYREF.**

TYPE$ ***Read / Write Property.*** Returns the data type for the data being held in the variant. When set, will attempt to convert the data to the requested data type. An error will occur if the conversion fails.Valid types:

A	Array data type (not settable)
B	Boolean data type
C	Byte data type
D	Date data type
E	Empty data type
F	Single data type
I	32 Bit Integer data type
L	Decimal data type
M	Currency data type
N	16 Bit Integer data type
O	Object data type
R	Double data type
S	String data type
X	Record data type
Z	Null data type (value)

VAL[$] ***Read / Write Property.*** Used to get/set the data held by the variant. It should be noted that setting the data may cause an implicit conversion. For example, if the current data type is B, setting the **VAL** property to 0 will cause a conversion to data type I. When assigning an object to the **VAL** property, the following notation must be used:

object'**VAL.PUT(*** *otherobject*)

VALB[$] ***Read / Write Property.*** Used to get/set the string data held by the variant. The difference between this property and the **VAL** property is that the data is assigned "as is". No conversion from Unicode to ANSI is performed on the data.

Examples:

The following example helps to illustrate the difference between the default string handling vs. byte handling properties and methods:

```
0010 DEF OBJECT V, "*VARIANT"
0020 LET V'VAL$ = "Hello World"
0030 PRINT V'LEN, ",", V'LENB$
0040 PRINT V'VAL$
0050 PRINT V'VALB$
0060 ESCAPE
0070 DELETE OBJECT V
```

If an object's documentation indicates that a parameter for a method call is "by reference", then ***VARIANT** must be used. The ***VARIANT** object is always passed "by reference" to the object, thus allowing the developer to retrieve the new value after method execution.

```
0010 DEF OBJECT V, "*VARIANT"
0020 DEF OBJECT X, "MSScriptControl.ScriptControl"
0030 V'VAL=10
0040 Z=X'RUN("MySub", *V)
0050 ! Pretend that the object X has set the data for V to "Hello World"
0060 ?V'VAL$ ! Will print "Hello World"
```

The data for V was set to 10 and then passed as a parameter to the object's method call. Because it was passed "by reference", the method call can change the data to anything or any type it wishes. If unsure of the data type returned in the ***VARIANT** object, then check the **'TYPE$** property.

*VARARRAY

Instances of this object type are used to facilitate methods that either accept or return COM arrays. What distinguishes this object from ProvideX arrays is the fact that each element can accept data of any type. For example, ***element 1*** may hold a string value, ***element 2*** an object, ***element 3*** an integer, etc. It should also be noted that this object is always passed by reference. The ***VARARRAY** wrapper is also able to return a ***FOREACH** enumerator when the COM array contains a single dimension.

When an array is assigned to a ***VARIANT**, it is important to remember that the ***VARIANT** will end up with a copy of the array, not the actual array itself. Any changes, even releasing the array, will not affect the array held by the ***VARIANT**.

Following is a list of members in the variant array object:

ARRAYTYPE$ ***Read Only Property.*** Returns the array type.Valid types:

B	Boolean data type
C	Byte data type
D	Date data type
F	Single data type
L	Decimal data type
I	32 Bit Integer data type
M	Currency data type
N	16 Bit Integer data type
O	Object data type
R	Double data type
S	String data type
V	Variant data type.

CLEAR(*index***[,** *index***])** ***Method Call.*** Clears the data being held in the variant array element. It is legal to call this method without passing in the element index, in which case the index will default to zero. After the clear completes, the element data type is set to **E**(*mpty*).

COPY ***Read Only Property.*** Returns a variant array object that contains an exact copy of the contents of the current variant array.

CREATE(*elements***[,** *elements***])**

Method Call. Initializes the variant array to the dimension count, and element count for each dimension. This method must be passed at least one element count (to create a single dimensioned array), and can accept a maximum of 20 element counts.

Examples:

```
10 ! Create 1 dim array with 10 elements
20 OBJECT'CREATE(10)
30 ! Create 2 dim array
40 OBJECT'CREATE(10, 100)
```

Either this method, or **CREATEVECTOR**, must first be called before attempting to access data. Calling **CREATE** multiple times is also allowed, but will clear any existing data.

CREATEBYTEARRAY(*data*[*$*])

Method Call. Requires the passed data parameter to be a either a string or a variant object that contains a string. Initializes the array as a single dimension array of byte (ArrayType = "C"), with the element count of the array equal to the length of the data string. For string data, a conversion to Unicode is performed, resulting in an element count that is 2x the length of the string. If byte handling is desired then, a ***VARIANT** object should be used.

Examples:

```
10 DEF OBJECT V, "*VARIANT"
20 DEF OBJECT A, "*VARARRAY"
30 V'VALB$="Hello world"
40 A'CREATEBYTEARRAY(*V)
```

CREATEEX(*type$, elements***[, *elements*])**

Method Call. Initializes an array of the type specified by *type$*, where *type$* is one of the valid characters from **ARRAYTYPE$**. The number of dimensions is determined by the count of passed elements. Each element parameter determines the element count for the dimension. This method must be passed at least one element count (to create a single dimensioned array), and can accept a maximum of 19 element counts. Also see **ARRAYTYPE$**.

CREATEVECTOR(*data***[, *data*])**

Method Call. Initializes the array as a single dimension variant array with the element count equal to the number of parameters passed in. Each element in the array will be assigned the contents of the corresponding *data* parameter.

Example:

```
OBJECT'CREATEVECTOR("John", "Smith", 31, 150.3)
```

Creates a single dimension array with an **LBOUND** of zero, a **UBOUND** of 3, and element data types of:

```
[0] = "S", [1] = "S", [2] = "I", [3] = "R"
```

Calling **CREATEVECTOR** multiple times is allowed, but will clear any existing data.

DIMENSIONS ***Read Only Property.*** Returns the number of dimensions in the variant array.

GETDATA[$](*index***[, *index*])**

Method Call. Returns the data being held in the variant array element. It is legal to call this method without passing in the element index, in which case the index will default to zero.

GETDATAEX(*index***[, *index*])**

Method Call. Returns the data being held in the variant array element as a variant object. It is legal to call this method without passing in the element index, in which case the index will default to zero.

LBOUND(*dimension***)** ***Method Call.*** Returns the lower boundary for the dimension in the array. Please note that this will always be zero for a valid dimension. When passing *dimension*, the value should be an integer between 1 and the number of valid dimensions; otherwise, an error occurs.

OLEARRAY ***Read Only Property.*** Returns a variant array copy of the current array. If the current **ARRAYTYPE$** property is "V", then no conversion is required.

PASSBYREF ***Read/Write Property.*** Determines if the array will be passed by reference. Setting the property to zero indicates false, any other value indicates true. The default setting for this property is true.

SETDATA(*index***[,** *index***],** *data***[$])**

Method Call. Assigns the contents of data to the variant array element. It is legal to call this method without passing in the element *index*, in which case the *index* will default to zero. For variant arrays, the *data* parameter can be any valid data type, including variant and variant array objects. For all other arrays, the *data* parameter must be convertible to the array type.

TYPE$(*index***[,** *index***])** ***Method Call.*** Returns the data type for the data being held in the variant array element. It is legal to call this method without passing in the element index, in which case the index will default to zero. For example, the following is identical for a two dimension array:

```
T$=OBJECT'TYPE$()
T$=OBJECT'TYPE$(0, 0)
```

Unlike the variant object though, the **TYPE** property is read only. If conversion of a data type is required, then the use of a variant object is mandatory. See ***VARIANT**.

UBOUND(*dimension***)** ***Method Call.*** Returns the upper boundary for the dimension in the array. This will be one number less than the total count of elements in the dimension; i.e., **0 ...** *Element_Count***-1**.

Example:

The following is an example of initializing an array using all the valid array types:

```
0010 ! Set array types to create
0020 DEF OBJECT A, "*VARARRAY"
0030 LET TYPES$ = "VBCMDRNIOFS"
0040 FOR I = 1 TO LEN(TYPES$)
0050 A'CREATEEX(TYPES$(I,1),10)
0060 PRINT "ARRAY TYPE:", A'ARRAYTYPE$
0070 PRINT "ARRAY DIMENSIONS:", A'DIMENSIONS
0080 PRINT "ARRAY LBOUND:", A'LBOUND(1)
0090 PRINT "ARRAY UBOUND:", A'UBOUND(1)
0100 FOR X = A'LBOUND(1) TO A'UBOUND(1)
0110 PRINT A'GETDATA$(X)
0120 NEXT X
0130 ESCAPE
0140 NEXT I
```

COM Error Handling

Lack of proper documentation, misbehaved COM objects, and incorrect data types are just a few of the numerous reasons that errors will occur. When a COM error does occur, it normally appears in the form of an `Error #88: Invalid/unknown property name`. This should only be taken as an indicator that a COM method (property) call ***failed***.

In order to further identify the problem, the following steps should be taken:

1. **Break multi tick (') statement lines into single tick statements.** Multi tick statement lines only specify one object variable. The other objects created to resolve the expression are temporary. If an error is reported on one of the temporary objects, it cannot be evaluated after the statement executes, thus the context of the error is lost.
2. **Check the result of MSG(-1).** When a COM error occurs, the interop layer will set the textual error message for **MSG(-1)**.
3. **Create an instance of *ERROR and evaluate the information exposed through its properties.** The ***ERROR** object is a wrapper around the COM interop's error information table, which is used to track the last error. This allows a ***ERROR** object to be created after the fact.
4. **Check the PVXERROR[$] for the object**. The error code returned by **PVXERROR** is also known as the HRESULT by developers in other languages. When working with third party developers, this information may be required. The string representation of this error is identical to what is displayed by **MSG(-1)**.
5. **Verify (with documentation or type library viewer) that the information you are passing is correct.** One of the most common errors is passing incorrect or invalid data to COM methods. The second most common error is passing too many, or too few, parameters.

In most cases, the steps above will be enough to resolve the COM errors that occur. Should this not be the case, then proper documentation will be critical in resolving the issue. If documentation is not available for the COM object, then the use of **PVXDESCRIBE** can be used as a last resort. This extended method takes one string parameter, which is the name of the object's property or method to generate information for.

If the object does not expose type information, then the interop layer will not be able to provide any detailed information.

Note: An object does not require type library information to be programmable. However, without proper documentation, it will be impossible to determine what member names are available, and how they should be called.

Advanced Usage

This section describes some of the more advanced aspects of COM usage within the ProvideX environment.

String Handling in COM

When string data is passed from ProvideX to the COM interop layer, a translation is performed that converts the data to a double byte character string. For example, the string "HELLO WORLD" becomes the following in memory:

H	\0	E	\0	L	\0	L	\0	O	\0		\0	W	\0	O	\0	R	\0	L	\0	D	\0

When string data is passed back to ProvideX, the double byte character string is converted back to a single byte (ANSI) string. While this is the expected behavior in most situations, it can cause problems when the string data type is used to transport binary data between a client and object. For example, if an object used the string data type to return data that represented a picture image, attempting to convert the data to a single byte string would be incorrect. The following describes the mechanisms that should be used for correctly dealing with this situation.

- When passing binary data to an object (using the string data type), a ***VARIANT** object must be used. The variant exposes the **VALB** property which allows data to be set in binary mode, bypassing the normal double byte conversion. The **LENB** property can be used to determine the byte length, and the **ADDB** method can used to build a stream of byte data.
- When receiving a byte string from a property or method call, the respective **Invocation Hints** `.GetB$` and `.CallB$` must be used. The invocation hints inform the COM interop layer that the return string data should not be converted from a double byte to single byte string.

The following pseudo code demonstrates the mechanics for both passing and receiving binary data from a fictitious object.

```
0010 DEF OBJECT X, "SOME OBJECT"
0020 DEF OBJECT V, "*VARIANT"
0030 LET V'VALB$ = $000102030405060708AB1215$
0040 X'PASSTHEDATA(*V)
0050 LET S$ = X'GETTHEDATA.CALLB$()
0060 DELETE OBJECT V
0070 DELETE OBJECT X
```

Passing Optional Parameters

Many objects have methods where the parameters are defined as optional. This is an indication to the developer that the parameter can be excluded. But how is this accomplished in ProvideX? Below is an example of an ADO Recordset Open method in VB as translated to ProvideX:

Visual Basic: `RS.OPEN "GL1_ACCOUNTS", CONN, , , 2`

ProvideX: `RS'OPEN("GL1_Accounts", *CONN, *, *, 2)`

In ProvideX, the optional parameters are passed using ***** ***asterisk***. The one exception is that ProvideX will not allow ***** to be passed as the last parameter. If these parameters are not required for the method call, then they must be omitted, and the method call should be terminated with a closing parenthesis at the last actual parameter:

Incorrect: `RS'OPEN("GL1_Accounts", *CONN, *, *, *)`

Correct: `RS'OPEN("GL1_Accounts", *CONN)`

Note: Refer to the method's documentation or use a type library viewer to determine if a method uses optional parameters.

Sub-Object Handling

In ProvideX, a sub-object is defined as any object that is the return value of another object's property or method call. Since a sub-object is owned by the base object, it will be destroyed when the base object is destroyed.

```
10 DEF OBJECT WORD, "Word.Application;Finalize=Quit"
20 DOCS = WORD'DOCUMENTS
30 DELETE OBJECT WORD
```

After line 30 executes, the DOCS object will no longer be valid, as the owning object has been destroyed. To determine if an object is owned by another (parent) object, the **PVXPARENT** property can be examined (see **Extended Properties and Methods**, *p.269*). For top level objects, the return value will be zero, otherwise the parent object handle will be returned. It is important to be aware of this, as the lifetime of an object is directly controlled by the lifetime of its parent.

Reserved Word Conflicts

Occasionally there are cases where an object's property or method is identical to a ProvideX reserved word. When this happens, ProvideX will generate a syntax error on any attempt to use the object's property or method name. For example, if you queried a calendar control for a property called Day, the following would occur.

```
10 DEF OBJECT CAL, "MSCAL.Calendar.7"
20 PRINT CAL'DAY
Error #20: Syntax error ...Day
```

An `Error #20` is raised after line 20 executes. To handle these situations, the COM interop layer provides a mechanism called "aliasing" (see **PVXALIAS** under **Extended Properties and Methods**, *p.269*). This allows a developer to call a property or method by creating user-defined names. For example, the following could be done to correct the previous example.

```
0010 DEF OBJECT CAL, "MSCAL.Calendar.7"
0020 CAL'PVXALIAS("DAY", "CALENDARDAY")
0030 PRINT CAL'CALENDARDAY
```

Comparisons with Visual Basic

Many third party vendors tailor their code examples for Visual Basic. While translating these examples to ProvideX is relatively straight forward, there are some nuances in the Visual Basic language that can cause problems. This section lists some of the more common translation errors, and describes how they should be handled.

Default Member Access

What is a default member? A default member is a property or method of an object that is invoked when a client does not specify a property or method name. Take the following Visual Basic example:

```
Dim rs As ADODB.Recordset
rs("CompanyName") = "SomeCompany"
rs!CompanyName = "SomeCompany"
```

The code above is actually a shortcut for:

```
Dim rs As ADODB.Recordset
rs.Fields("CompanyName").Value = "SomeCompany"
rs.Fields!CompanyName.Value = "SomeCompany"
```

The problem with this coding style is that the code is no longer self documenting. A programmer must have knowledge of the `ADODB.Recordset` object in order to determine what the code is actually doing.

Determining when a shortcut has been used is a little more difficult. If the code returns an object, and the assignment is performed using a string, number, or object (with out a `Set` statement), then a shortcut is most likely in use. To access the default member in ProvideX, an `_` underscore must be used. See the corresponding ProvideX code:

```
10 DEF OBJECT RS, "ADODB.Recordset"
20 RS'_("CompanyName")'Value$ = "SomeCompany"
30 RS'_("CompanyName")'_$ = "SomeCompany"
```

Note: While this coding style is supported in ProvideX, its general use is discouraged.

For Each

The `For Each` statement is used in Visual Basic when an object exposes a collection interface (using `IEnumVariant`); e.g., `Excel.Application` exposes a collection called `WorkBooks`:

```
Dim ExcelApp As Object
Set ExcelApp = CreateObject("Excel.Application")
For Each WorkBook in ExcelApp.WorkBooks
  '... do something with WorkBook
Next
```

ProvideX also has the capability to utilize the collection enumerator through the use of the ***FOREACH** object. The following is the ProvideX equivalent for the Visual Basic code above:

```
0010 DEF OBJECT EXCELAPP,"Excel.Application;Finalize=Quit"
0020 LET WORKBOOK=EXCELAPP'WORKBOOKS'PVXFOREACH
0030 WHILE WORKBOOK'NEXT()
0040 ! do something with WORKBOOK'DATA
0050 WEND
```

Note: The benefit of using a "for each" enumerator is that it does not matter if the collection is zero or one based; the enumeration code remains the same.

Named Arguments

Some objects allow method arguments to be passed in using name positioning rather than index positioning. For example, the `Open` method of the Microsoft Excel Workbooks object (for opening a workbook) takes 13 arguments. All arguments are optional in the `Open` method, and could be written in Visual Basic as:

```
Workbooks.Open "book2.xls", , , , , , , , , , , , True
```

Given that Microsoft Excel will accept named arguments, the preceding code could also have been written as:

```
Workbooks.Open FileName:="book2.xls", AddToMru:=True
```

The use of a named argument is very easy to spot when converting Visual Basic code, given the **`:=`** syntax. The difficult part is converting the above sample to ProvideX, which must pass arguments by index. This is where documentation, or a good type library viewer, is necessary.

Once the index location of `FileName` and `AddToMru` are found, coding the statement in ProvideX is simple:

```
Workbooks'Open("book2.xls", *, *, *, *, *, *, *, *, *, *, *, True)
```

This is very close (syntactically) to the original Visual Basic example, except for the use of asterisks as optional arguments.

Calling Conventions

This section discusses how to determine when you should, or should not, expect a result from a method call. If a Visual Basic statement passes parameters, but does not contain open and close parentheses, then the statement is performing a call to a procedure, and no result is returned. Using the previous Excel example:

```
Workbooks.Open "book2.xls", , , , , , , , , , , , True
-
10 Z = Workbooks'Open("book2.xls", *, *, *, *, *, *, *, *, *, *, *, True)
```

In this example, no data is returned from the Visual Basic call; but, in ProvideX, a zero would be returned to Z. The zero in ProvideX, for this situation, represents a null return value. Using another example:

```
Dim I as Integer
I = RecordSet.Fields(0)
-
10 I = RecordSet'Fields(0)'Value
```

The data returned for the field value might be zero, but it does have meaning in this context (it is the data value for the Fields). Finally, if the statement you are converting starts with a Set, then your code should be written to expect an object return value.

```
Dim Fld as Object
Set Fld = RecordSet.Fields(0)
-
10 Fld = RecordSet'Fields(0)
20 DEF OBJECT Fld
```

Examples

This section contains code samples that indicate how to perform a number of actions using the ProvideX COM Interface. These are not complete programs and are intended only to demonstrate some practical non-event COM usage in ProvideX applications. For event-driven program examples of COM, see **Event-Driven COM**, *p.294*.

Example 1:

This example demonstrates how to embed a shell browser window within ProvideX, and to display either a web page or a folder view as the contents.

```
0010 DEF OBJECT IE, @(2, 2, 70, 16) = "Shell.Explorer"
0020 IE'Navigate2("http://www.pvx.com")
0030 ESCAPE
0040 IE'Navigate2("file://c:\")
0050 ESCAPE
0060 DELETE OBJECT IE
```

Example 2:

The folling invokes Adobe PDF printing hidden from the user.

```
0010 DEF OBJECT PDF,"PDF.PdfCtrl.6"
0020 PDF'LoadFile("C:\tmp\test.pdf")
0030 PDF'Print()
0040 DROP OBJECT PDF
```

Example 3:

This example demonstrates how to use the tool window dialog to select a control.

```
0010 PRINT 'CS'
0020 DEF OBJECT AXCTL, @(40,1,30,10)="*", ERR=0100
0030 PROG$ = AXCTL'PVXNAME$
0040 ESCAPE
0050 END
0100 PRINT MSG(-1)
```

Example 4:

This example demonstrates how to embed a Word document from a file reference. Requires Microsoft Word to be installed.

```
0010 GET_FILE_BOX READ WORDDOC$, LWD, "Word Document", "Word
      Document|*.doc,"
0020 IF WORDDOC$="" THEN END
0030 DEF OBJECT WD, @(0,0,80,20)="[File]"+WORDDOC$
0040 INPUT *
0050 DELETE OBJECT WD
0060 END
```

Example 5

This example demonstrates how to open an Access database and to read the table schema. Requires Microsoft Access to be installed.

```
0010 GET_FILE_BOX READ MDB$, LWD, "Access Database"," Access Database
      |*.mdb,"
0020 IF MDB$="" THEN END
0030 DEF OBJECT CONN, "ADODB.Connection"
0040 LET CONNSTR$ = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source="+MDB$
0050 CONN'OPEN(CONNSTR$)
0060 DEF OBJECT VARDATA,"*VARARRAY"
0070 VARDATA'CREATE(4)
0080 VARDATA'SETDATA(3,"Table")
0090 LET R = CONN'OPENSCHEMA(20,*VARDATA)
0100 LET N$ = ""; FOR I=0 TO R'FIELDS'COUNT-1; LET
      N$=N$+R'FIELDS(I)'NAME$+" | "; NEXT I; PRINT N$
0110 LET RA = R'GETROWS(20)
0120 FOR I=0 TO RA'UBOUND(2)
0130 LET TXT$ = ""
0140 FOR II=0 TO RA'UBOUND(1)
0150 LET TXT$ = TXT$+RA'GETDATA$(II,I)+" | "
0160 NEXT II
```

```
0170 PRINT TXT$
0180 NEXT I
```

Example 6:

This example demonstrates how to use WMI to display the MAC and IP address for the local computer. It also demonstrates the "for each" enumerator usage using a collection and an array.

```
0010 DEF OBJECT WMI,"[GetObject]winmgmts:\\.\root\cimv2"
0020 LET COLITEMS = WMI'EXECQUERY("Select * from
      WIN32_NetworkAdapterConfiguration where IPEnabled = True")
0030 LET ITEMSFOREACH=COLITEMS'PVXFOREACH
0040 WHILE ITEMSFOREACH'NEXT()
0050 PRINT ITEMSFOREACH'DATA'PROPERTIES_("MACAddress")'VALUE$
0060 LET IPARRAY=ITEMSFOREACH'DATA'PROPERTIES_("IPAddress")'VALUE
0100 LET IPFOREACH=IPARRAY'PVXFOREACH
0110 WHILE IPFOREACH'NEXT()
0120 PRINT IPFOREACH'DATA$
0130 WEND
0140 IPARRAY'PVXFREE()
0150 WEND
0160 ITEMSFOREACH'PVXFREE()
0170 ESCAPE
0180 DELETE OBJECT WMI
```

Example 7:

This example demonstrates how to use the Scripting File System object to list the available drive letters on the local computer.

```
0010 DEF OBJECT FSO, "Scripting.FileSystemObject"
0020 LET DRIVE_ITER = FSO'DRIVES'PVXFOREACH
0030 WHILE DRIVE_ITER'NEXT()
0040 LET DRIVE = DRIVE_ITER'DATA
0050 PRINT DRIVE'DRIVELETTER$
0060 WEND
0070 ESCAPE
0080 DELETE OBJECT FSO
```

Example 8:

This example demonstrates how to create a picture interface, and then load the picture into a control that displays picture data.

```
0020 INPUT "Enter Picture Value:",PIC$
0025 IF LEN(PIC$)=0 THEN GOTO 0100
0026 IF (X>0) THEN X'PVXFREE()
0028 DEF OBJECT X,@(50,1,30,15)="Forms.Image.1"
0030 DEF OBJECT Y,"[picture]"+PIC$
0040 X'PICTURE.PUT(*Y)
0050 GOTO 0020
```

Example 9:

The following code sample is used for controlling a Shockwave Flash animation.

```
0010 print 'dialogue'(0,0,60,20,"Flash",'CS')
0020 Places=10
0030 !
0040 def object Flash,@(0,0,45,15.5)="Shockwaveflash.shockwaveflash.1"
0050 R$=lwd+dlm+"earth.swf"
0060 Flash'LOADMOVIE(0,R$)
0070 Flash'SETVARIABLE("n","Earth")
0080 !
0090 drop_box Places,@(47,5,10,10)
0100 print 'text'(@x(47),@y(4.25),"Visit:"),
0110 drop_box load Places,"<Just Spin>,North and South America,Africa,
      Hawaii,Asia,Austriala,Russia,Europe,"
0120 drop_box write Places,"<Just Spin>"
0130 !
0140 obtain X
0150 if ctl=4 then print 'pop'; end
0160 if ctl=Places then gosub MOVE
0170 goto 0140
0180 !
0190 MOVE:
0200 if Places'CURRENTITEM=1 then Flash'PLAY();
      Flash'SETVARIABLE("n","Earth"); goto 0140
0210 if Places'CURRENTITEM=2 then EARTHP=39; goto SETP
0220 if Places'CURRENTITEM=3 then EARTHP=62; goto SETP
0230 if Places'CURRENTITEM=4 then EARTHP=44; goto SETP
0240 if Places'CURRENTITEM=5 then EARTHP=52; goto SETP
0250 if Places'CURRENTITEM=6 then EARTHP=58; goto SETP
0260 if Places'CURRENTITEM=7 then EARTHP=61; goto SETP
0270 !
0280 SETP:
0290 Flash'STOP()
0300 C=Flash'CURRENTFRAME()
0310 if C=EARTHP then goto 0140
0320 if C>EARTHP then INC=-1 else INC=1
0330 R$=Places'VALUE$
0340 Flash'SETVARIABLE("n","Moving To "+R$)
0350 !
0360 for I=C to EARTHP step INC
0370 Flash'GOTOFRAME(I)
0380 wait .05
0390 next I
0400 Flash'SETVARIABLE("n",R$)
0410 goto 0140
```

Event-Driven COM

In COM automation, when a control wants to notify its client applications that an action has occured, it sends out a message called an event. The process of sending these types of messages is referred to as ***event firing***. As with any event-driven program in ProvideX, the client application that is interacting with a control must be listening for its events. However, in the COM case, it is imperative for the client application to advise the COM object that it is ready and expecting to receive events. For more on handling COM objects, see **ProvideX COM Support**, *p.261*.

Response Required

External events must be responded to immediately. This is a standard requirement for COM. While access to properties and methods is typically controlled from the client application's side, the timing for when events can fire is determined by the COM object itself. An event can happen whenever the application is reading or updating files, processing data, printing reports, or waiting for user input. An event can even fire while the application is in the process of servicing another event. The client application has to be ready to recieve an event at any time.

Working With Events

There are a couple of ways to access, receive and process COM control events in a ProvideX application:

- **Linking to an OOP Object** – Events from a COM object may be linked to an OOP object in ProvideX using the syntax **ON EVENT FROM** .. **PROCESS**. An example of this funtionality, is provided in the section **Calendar Object, COM Event Example**, *p.297*.
- **Accessing a Specific Event via an OOP Object** – Individual events can be activated via an associated ProvideX OOP object using the syntax **FUNCTION** .. **FOR EVENT**. An example of this funtionality, is provided in the section **Calendar Object, COM Event Example**, *p.297*.
- **Generating a CTL Event for a COM Event** – A CTL value can be placed into the input queue when the COM event occurs using **ON EVENT** .. **FROM** .. **PREINPUT**. An example of this funtionality, is provided in the section **Automated Timer, COM Event Example**, *p.300*.

Various descriptions and examples of ProvideX COM event processing are provided in the sections that follow.

Topics

Linking to an OOP Object

Linkage to a COM object can be managed/controlled through the use of a ProvideX object using **Data Integration** functionality.

ON EVENT FROM *com_id* **PROCESS** *oop_id*

Where:

com_id Numeric CTL value of a Windows COM object.

oop_id Numeric identifier of an OOP object.

Support for events from a COM object are limited to a single ProvideX OOP object, whereas, a single ProvideX OOP object is capable of supporting events from multiple COM objects. In other words, the relationship from a COM object to an OOP object is "one-to-one", while the relationship from OOP object to COM object(s) can be "one-to-many".

If ProvideX receives an event while processing another event, it will suspend the first event, process the second entirely, then resume the first event before returning to the main task. With this in mind, it is possible for a ProvideX task to become overloaded with event requests; therefore, a limit of 64 outstanding event requests has been imposed. ProvideX ignores subsequent event requests when there are 64 active requests being processed. Regular processing continues once the number of outstanding requests drops below 64.

Generally speaking, files can be shared between currently running tasks and event handlers. However, it is possible for an event to interrupt a task when it is part way through the execution of a file I/O directive or function. Under these circumstances, the file will not be available to the code associated with the event and the (new message) `Error #89: File access denied -- I/O operation` pending will be reported.

Usage Notes

- An invalid *com_id* generates an `Error #65: Window element does not exist or already exists`.
- An invalid *oop_id* generates an `Error #95: Bad Object Identifier`.
- Unexpected errors, such as when a COM object does not support events, will generate an `Error #88: Invalid/unknown property name` and, if available, place a description of what caused the error in the **PvxError$** property for the COM object, as well as in **MSG(-1)**.
- An *oop_id* of 0 (zero) deactivates event processing for the current COM object.
- Dropping a ProvideX OOP object deactivates event processing for all COM objects associated with the OOP object.
- The numeric OOP object identifier is stored in a read-only property called **PvxEvents**.

- A comma-separated list of the events that are supported by the COM object is available by querying the **PvxEvents$** property. Supported events are prefixed with a plus sign (+) while unmanaged events are prefixed with a leading minus sign (-).
- While the **PvxEvents** and **PvxEvents$** properties will appear in the property list for a COM object, they will not contain an available list of events until an **ON EVENT** directive has been executed.
- Issuing a subsequent ON EVENT directive for a COM control will discontinue event processing for the first ProvideX OOP object, then activate it for the new OOP object (provided the new oop_id contains a non-zero value).

Accessing a Specific Event via an OOP Object

In order to receive events, a ProvideX object must first be designed with a method written for each event to be handled. The following syntax sets a method to be processed for a particular incoming event:

FUNCTION *method*(*args*) *logic* **FOR EVENT** {*event$*|**SAME**}

Where:

method	Name of the *method/function* that the object can perform.
(*args*)	Optional list of arguments to be used in the logic when the method is actually invoked. Parentheses are required with / without arguments.
logic	Procedure associated with *method*. This should return a value; if not, the system forces 0 or " ".
event$	Name of the corresponding COM event.
SAME	Keyword to use if the *method* name matches *event$* name.

For example, when writing an event class for a COM object that exposes the event

ONCLICK(X *as Integer*, Y *as Integer*, Button *as Integer*, Shift *as Integer*)

.. the following code would be required:

```
10 DEF CLASS "FOO"
20 FUNCTION ONCLICK(X, Y, B, S) ONCLICK FOR EVENT "ONCLICK"
30 END DEF
40 STOP
50 ONCLICK:
60 ENTER X, Y, B, S
70 PRINT "OnClick was fired"
80 RETURN
```

The method and label name are not required to match the name of the event. It is the string after the **FOR EVENT** portion of the statement that determines the event name that will be handled. The event name is not case sensitive, and argument names are

not required to match the COM object's event declaration. When the OOP method name matches the COM event name, then **SAME** can be used to describe the name of the event.

Once the class is complete, an instance of the class must be instantiated in order to bind the COM object to the event handler. An example of the binding is listed on line 30:

```
10 DEF OBJECT FOO, "{Your Com Object Name}"
20 FOOEVENTS = NEW("FOO")
30 ON EVENT FROM FOO PROCESS FOOEVENTS
```

Once bound, the ProvideX event class function will be called when the corresponding event occurs.

Calendar Object, COM Event Example

The following event-driven program uses Microsoft's built-in *Calendar* object. It encapsulates the control of the calendar object completely inside a ProvideX OOP object and then processes the events from the calendar.

```
0010 def class "Calendar" create OnCreate required delete OnDelete required
0020 local Calendar,LastEvent$
0030 property GetDay
0040 property GetMonth
0050 property GetYear
0060 property GetDate$
0070 function Click()";Click" for event same
0080 function DoubleClick()";DoubleClick" for event "DblClick"
0090 function Change()";Change" for event "SelChange"
0100 function DateDoubleClick()";DateDoubleClick" for event "DateDblClick"
0110 end def
0120 !
0130 OnCreate:
0140 enter X,Y,W,H
0150 def object Calendar,@(X,Y,W,H),"MSComCtl2.MonthView.2"
0160 on event from Calendar process _obj
0170 return
0180 !
0190 OnDelete:
0200 if Calendar
        then drop object Calendar;
        Calendar=0
0210 return
0220 !
0230 Click:
0240 LastEvent$="Click"
0250 goto DoneEvent
0260 !
0270 DoubleClick:
0280 LastEvent$="DoubleClick"
```

```
0290 goto DoneEvent
0300 !
0310 Change:
0320 LastEvent$="Change"
0330 goto DoneEvent
0340 !
0350 DateDoubleClick:
0360 LastEvent$="DateDoubleClick"
0370 goto DoneEvent
0380 !
0390 DoneEvent:
0400 GetDay=Calendar'DayOfWeek
0410 GetMonth=Calendar'Month
0420 GetYear=Calendar'Year
0430 GetDate$=Calendar'Value$
0440 if LastEvent$="DoubleClick" or LastEvent$="DateDoubleClick"
       then msgbox "You selected a date of: "+GetDate$
0450 return
```

Handling Events

It is highly recommended that event functions be kept relatively short and simple. One reason for this is that normal code execution will be suspended when an event is handled and will not resume until the event function is finished. It is also possible to introduce code (or use commands such as **MSGBOX**) to create a re-entrancy issue in the event function.

Re-entrancy is allowed; however, a limit has been imposed to prevent runaway objects. If the event call stack reaches a depth of 64, all further events will be discarded while waiting for the current events to finish. The following related TCB values have been added to ProvideX to expose this information:

TCB(120) Returns the interop handle (see **PVXID**) for the COM object that fired this event. Similar to the `_obj` when in a class function.

TCB(121) Returns the number of events that have been ***dispatched*** to ProvideX.

TCB(122) Returns the current depth of the event call stack.

TCB(123) Returns the number of events that have been ***discarded*** by ProvideX due to excessive outstanding event calls.

It is important that the function declaration in the ProvideX class matches that of the COM event. Otherwise, your function may not get called, or may cause an error when the event is fired. If a situation occurs where an event argument is defined as variant (and may receive either string or numeric data) then a second overloaded event function should be defined, e.g.,

OnData(*data***)**

The following would be required in your ProvideX class to properly handle both cases:

```
010 DEF CLASS "FOO"
020   FUNCTION ON_DATA(N) ON_NDATA FOR EVENT "ONDATA"
030   FUNCTION ON_DATA(S$) ON_SDATA FOR EVENT "ONDATA"
040 END DEF
050 STOP
060 ON_NDATA:
070 ENTER N
080 PRINT "Numeric data ", N
090 RETURN
100 ON_SDATA:
110 ENTER S$
120 PRINT "String data ", S$
130 RETURN
```

Another aspect of event handling deals with the scope of passed arguments. All arguments passed into an event are local in scope, and should not be considered to exist when the event is finished. It is common for many event routines to pass COM objects in as parameters. These can be programmed against during the event, but should not be persisted (assigned for later use) when the event is finished.

Event Templates

The Type Library Browser (PVXtlb.exe) was developed to provide extended type information for Windows COM objects, and to help simplify the process of creating event class objects. This utility is freely downloadable from www.pvx.com. For further information on this utility refer to the ProvideX *Type Library Browser* documentation. The following steps briefly outline the event template generation feature of the TLB:

1. Use the ***Type Library Browser*** to open the desired type library. The COM object's **PVXTYPELIB$** and **PVXISA$** properties are useful for determining this information.
2. Locate the CoClass (COM Class) that supports the COM object, and locate the Default Event Interface in the Entity Documentation section.
3. Browse to the event interface by clicking the link in the Entity Documentation, or by double clicking the interface in the Members list.
4. In the Entity Documentation section of the utility, a link is available to generate an event template. Click this link to open the Save As dialog.
5. Save the template to the desired file name. The **DEF CLASS** statement in the template is automatically updated to reflect the filename assigned.

The event template (in text form) can then be edited in any text processor. All event functions, type casting, and def object statements will have been automatically created. The only thing that is required is to fill in the event function bodies with the necessary ProvideX code. Each function will have a commented section identified with the tag "< Insert code here >", which is where the custom code should be placed.

Errors in Events

In a typical ProvideX program that uses class objects, an error raised by an object will be cascaded back to the calling program. Events, on the other hand, are called by COM objects and not ProvideX code. If an error occurs during the event handling, the COM interop layer will notify the COM object that the event failed. However, nothing is reported on the ProvideX side; i.e., there is no program to report to. For this reason, all error handling should be performed within the class object.

Also, it is not wise to release the calling COM object, or its event handler object, during the execution of the event. Releasing the COM object during the event can lead to unpredictable results. Objects that were passed in as event parameters may be safely released.

Generating a CTL Event for a COM Event

ProvideX can also process external events by generating a ProvideX CTL whenever an identified COM event occurs. The following **ON EVENT** syntax places a CTL value into the input queue:

ON EVENT *evtname$* **FROM** *com_id* **PREINPUT** *ctl_id*

Where:

evtname$	Event name, maximum 255 characters.
com_id	Numeric CTL value of a Windows COM object.
ctl_id	Numeric CTL signal to generate (preinput) when a given event occurs.

The **PREINPUT** format simplifies the event interface by eliminating the need to create an OOP object to manage events. This also works across WindX. The event process does not have access to any event parameters as it will not be running in-line when the event occurs. If event parameters are required, then refer to the section **Linking to an OOP Object**, *p.295*.

Automated Timer, COM Event Example

The following event-driven program sets an automated timer by generating a ProvideX CTL to process the associated COM event.

```
0010 def object Timer,"*system"
0020 on event "Timeout" from Timer preinput 500
0030 Timer'SetTimer(5)
0040 input a$,;
     if ctl=500
     then print "Got a Timer Event";
     goto *same
0050 Timer'SetTimer(0)
0060 drop object Timer
```

JavX COM Support

A version of **ProvideX COM Support** is also available in JavX to enable access to applications in the Java Runtime Environment (JRE). If a ProvideX application (running under JavX) needs to interact with third party Java classes, then the JavX version of this interface can be implemented for this purpose.

Accessing Java classes in JavX is very similar to accessing COM objects under WindX. Java events are handled using ProvideX OOP objects. As with ProvideX and COM, the **DEF OBJECT** directive can be used to interface directly with the Java run-time while in a JavX session. For example, to create an instance of a Java *AWT Button* class, execute the following:

```
DEF OBJECT BTN,@(5,5,5,5),"[WDX]java.awt.Button"
```

This functionality allows for the use of any Java class as well as Java Add-On classes. It can also be used to create entire GUI screens in Java code that can be used in devices such as PDAs. For more details on the handling of Java classes (controls / objects) by ProvideX applications running under JavX, refer to the ***ProvideX Client-Server Reference***. Implementation of the COM interface in ProvideX is documented in the sections **ProvideX COM Support**, *p.261* and **Event-Driven COM**, *p.294*.

JavX COM support is outlined in the sections below.

Array Support

JavX supports the ProvideX extended object ***VARIANT**, but does not currently support the extended objects ***VARARRAY**, ***MASTER**, ***ERROR**. However, ***VARARRAY** may be supported in a future release of JavX. Currently, JavX supports arrays through a JavX-specific extended object ***ARRAY**. Refer to the ***ProvideX Client-Server Reference*** for documentation on ***ARRAY.**

"No Argument" Constructor

In Java, a class's constructor may may be private (typically when implementing a Singleton design pattern), or public (but requiring arguments). To emulate the fact that Window's COM objects always have a public ***no argument*** constructor, the JavX version of the COM interface includes the ability to load a class and then instantiate it. This means that a handle to a class can be retrieved, and then a static method in the class that returns an instance of the class can be called. Refer to the ***ProvideX Client-Server Reference*** for more details on this functionality.

Event Support

The handling of COM events in JavX is also similar to that of ProvideX (documented in the section **Event-Driven COM**, *p.294*). However, there are a few minor differences; e.g., Java event listeners provide notification when an event has occurred on an object.

The following example creates a button and ActionListener (an object that responds to action events):

```
10 DEF OBJECT JBUTTON,@(24.5,5,10,10)="[wdx]java.awt.Button"
20 EXECUTE "[wdx]on event java.awt.event.ActionListener from
      "+STR(JBUTTON)+" preinput 100"
30 OBTAIN A$
40 PRINT A$
```

This example adds `ActionListener` to the button and when an action event occurs, a CTL value of 100 is sent to the ProvideX host program. The following example adds mouse event support only:

```
10 DEF OBJECT JBUTTON,@(24.5,5,10,10)="[wdx]java.awt.Button"
20 EXECUTE "[wdx]on event java.awt.event.MouseListener from
      "+STR(JBUTTON)+" preinput 10"
30 OBTAIN A$
40 PRINT A$
```

In this example, the CTL value 10 will be returned to the host program when the user's mouse enters or exits the button, or the mouse button is pressed / released on the button. Essentially every possible mouse event causes the CTL value 10 to be returned to the ProvideX host.

Using COM in JavX - Example

The flexibility of the COM interface in JavX is demonstrated in the following example, which creates an AWT FRAME and button and places them on a frame.

```
0070 def object AWTWINDOW,"[wdx]java.awt.Frame"
0080 def object AWTBUTTON,"[wdx]java.awt.Button"
0090 def object EXITBUTTON,"[wdx]java.awt.Button"
0100 def object AWTMULTI_LINE,"[wdx]java.awt.TextArea"
0110 !
0120 ! PRINT awtWindow'*
0130 AWTWINDOW'SETSIZE(260,320)
0140 AWTWINDOW'SHOW()
0150 AWTBUTTON'SETBOUNDS(30,246,80,40)
0160 EXITBUTTON'SETBOUNDS(140,246,80,40)
0170 AWTBUTTON'SETLABEL("Hit Me!")
0180 EXITBUTTON'SETLABEL("Exit")
0190 AWTMULTI_LINE'SETBOUNDS(25,45,200,200)
0200 ! add events
0210 execute "[wdx]on event java.awt.event.ActionListener from
      "+str(AWTBUTTON)+" preinput 10"
0220 execute "[wdx]on event java.awt.event.ActionListener from
      "+str(EXITBUTTON)+" preinput 20"
0230 !
0240 AWTWINDOW'SETLAYOUT(*-1)
```

```
0250 AWTWINDOW'ADD(*AWTBUTTON)
0260 AWTWINDOW'ADD(*EXITBUTTON)
0270 !
0280 EXITBUTTON'LINE=17.5
0290 EXITBUTTON'COL=18
0300 EXITBUTTON'LINES=3
0310 EXITBUTTON'COLS=9
0320 !
0330 AWTWINDOW'ADD(*AWTMULTI_LINE)
0340 AWTWINDOW'VALIDATE()
0350 AWTWINDOW'REPAINT()
0360 obtain A$
0370 EVT=ctl
0380 if EVT=10 then {
0390 !
0400 LOADTEXT$="Ctl :"+str(EVT)
0410 CURTEXT$=AWTMULTI_LINE'GETTEXT$()
0420 AWTMULTI_LINE'SETTEXT(CURTEXT$+" "+LOADTEXT$)
0430 goto 0360
0440 }
0450 if EVT=20 then {
0460 goto CLEAN_UP
0470 } else {
0480 goto 0360
0490 }
0500 !
0510 !
0520 ! clean up
0530 CLEAN_UP:
0540 AWTWINDOW'REMOVE(*AWTBUTTON)
0550 AWTWINDOW'REMOVE(*AWTMULTI_LINE)
0560 AWTWINDOW'HIDE()
0570 AWTWINDOW'DISPOSE()
0580 delete object AWTWINDOW
0590 delete object AWTBUTTON
0600 delete object AWTMULTI_LINE
0610 end
```

ProvideX Type Library Browser

The ***ProvideX Type Library Browser*** (TLB) is designed to provide extended type information for Windows COM objects and to help simplify the process of creating event class objects. This utility (`pvxtlb.exe`) is freely downloadable from www.pvx.com.

The TLB can be used to display any COM object's type library, showing the object's CoClass and GUID information along with the properties, methods, and events supported by the control (see below for an explanation of each).

It can also be used to create ProvideX OOP objects that will help simplify the handling of events generated by COM controls.

Note: A GUID (Globally Unique IDentifier) is a 128-bit number maintained in the Windows Registry for uniquely identifying COM objects, DLLs, etc. This number may be used to obtain details on any COM object in the Registry (type library, physical location, etc.).

Using the TLB

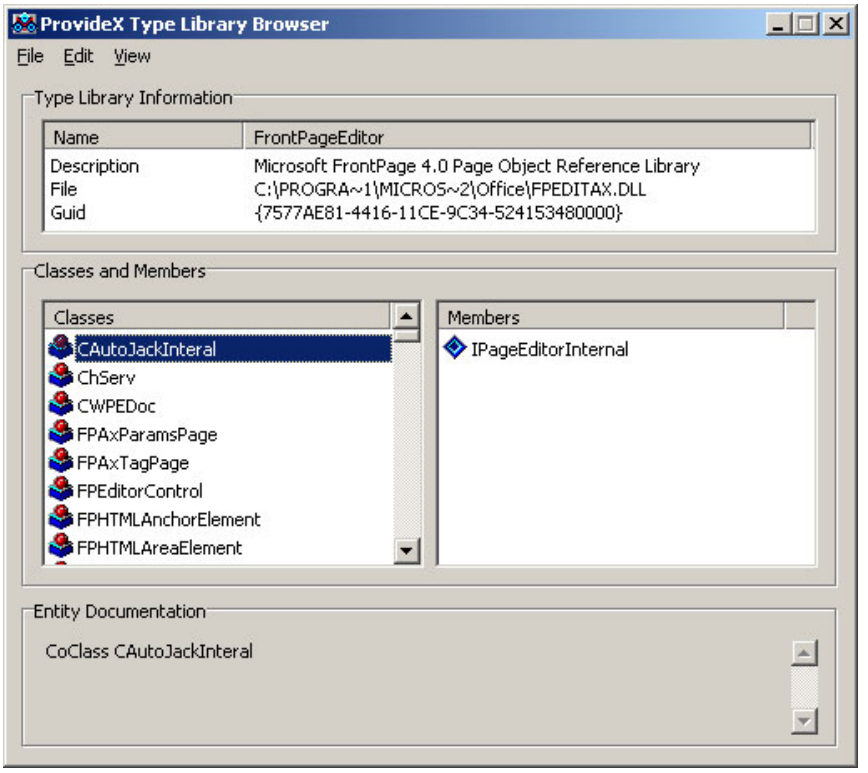
The following steps illustrate how to start the *Type Library Browser* and display extended type information for a selected COM object (in this case, the Microsoft *Calendar Control*):

1. Start the TLB by launching `pvxtlb.exe` from Windows Explorer, or via **Start > Run**..
2. Select **Open** from the **File** menu to display all registered OLE/COM objects. The Registered Type Libraries dialogue window appears.
3. Scroll down and select Microsoft Calendar Control.

   The type library information for the Microsoft Calendar Control is loaded into the TLB.

   Refer to the illustration on the following page.

   The Type Library Information section includes a description of the object, the location and name of the OCX file, and the GUID. (This can be stored internally in this OCX file or in a separate file with a `.TLB` extension).



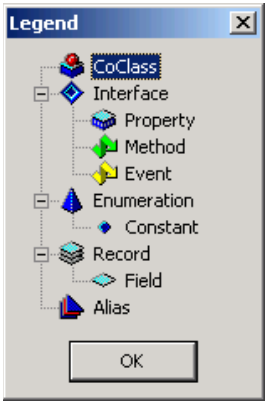
The Classes and Members lists contain all the components of this control.

The Entity Documentation section provides more detail for the selected item including type information and parameter lists.

4. Click on the Calendar object in the Classes list. This shows that the Calendar object has two members: *DCalendarEvents* and *ICalendar.*

5. The icon to the left of each item denotes the class type. To see a complete list of the icons used to identify class types in the TLB, select **Legend** from the **View** menu.

Refer to the illustration on the following page.

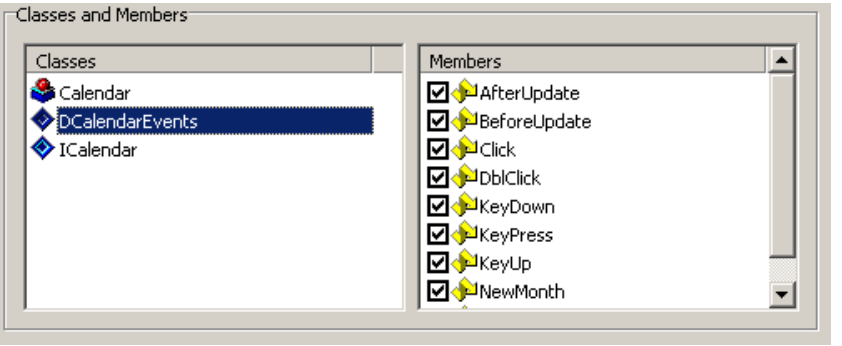


The class types are described as follows:

CoClass	A CoClass is a COM class which contains a collection of interfaces defined by the COM object. The Calendar class is considered a CoClass object.
Interface	An interface is a class which exposes methods, properties and events for use by other objects. The DCalendarEvents and ICalendar classes are interfaces.
Property	Properties are the class's member variables that represent the class's current state. Typically properties should only be set or read by class methods. (Refer to the ProvideX OOP documentation for more information on class functions and properties).
Method	Essentially, methods are similar to ProvideX class functions. They direct the control's behavior.
Event	COM objects can generate events to let the host object know something has changed; e.g., when a user clicks a button changing the current month, the Calendar object may fire a NewMonth event. This event will be dispatched to any host object that has registered to listen for NewMonth events.
Enumeration	An Enumeration is a collection of related integer constants. COM object's Constants are only accessible through Enumerations.
Constant	Constants are attributes that cannot be changed, usually numbers; e.g.,an OCX that controls a multiline object has three constants representing the possible alignment of text: *left*, *right*, and *center*. These constants would be exposed through an Enumeration.
Record	Records represent collections of elements, called fields.
Field	A field is an element variable contained in a record (much like a single field in an IOList for a file).
Alias	An alias is a data type that is a reference to another data type.

6. Close the Legend window and select the ICalendar class. The Members list will now display the methods and properties of the ICalendar class. This class contains about a dozen methods with names such as NextWeek() and NextYear(), which can be accessed using the ProvideX OCX / COM Interface.

7. Select the DCalendarEvents class from the Classes list. The Members list will display all the methods, properties, and events supported by the DCalendarEvents class.



Check boxes identify which members are to be used by the TLB for generating an *event template* (described below). Initially, they are all selected by default. This defines all the events the Calendar control can fire.

Creating an Event Template

The following steps describe the event template generation feature of the TLB:

1. To create an event template for the DCalendarEvents interface, choose events (select / deselect check boxes) from the Members list, then click the link **Create an EventTemplate based on this Interface** located in the Entity Documentation section. The Save As file box appears.

2. Save the generated file as `DCalendarEvents.pvc`. The TLB will generate a text file that contains the source code for a ProvideX OOP object. This text file can be opened with an editor, such as *Notepad*.

   The ProvideX event handler class `DCalendarEvents.pvc` has a function declared for each member event selected in the Members list. If the NewMonth event check box is selected then an Event_NewMonth function will be declared.

3. Each function contains a comment "`Insert code here`". Replace this comment with the necessary ProvideX code; e.g.,

```
Print "NewMonth Event has been fired"
```

Retrieving a Loaded COM Object's Type Library Information

The TLB accepts optional arguments that enable the retrieval of the type library information of a given COM object. To retrieve the type library Information of the Calendar control, pass the name and path of the Calendar OCX. To do this from a ProvideX console, perform the following

```
invoke "pvxtlb.exe "+QUO+C:\Program Files\Microsoft
     Office\Office\MSCAL.OCX"+QUO
```

ProvideX objects contain two properties: *PvxTypeLib* and *PvxISA*. The PvxTypeLib property returns the type library file from an object. The PvxISA property contains the name of the instantiated class. Invoke the TLB from a ProvideX console and pass the value of these two properties to display the type library information of a loaded COM object; e.g.,

1. Instantiate the Calendar COM Object:
   `Def object x, "MSCAL.Calendar.7"`
2. Invoke the TLB, passing the ProvideX object's `PvxTypeLib$` and `PvxISA$` properties:
   `Invoke "pvxtlb.exe "+QUO+x'PvxTypeLib$+QUO+" "+x'PvxISA$`

The TLB window displays the Calendar controls type library with the *Icalendar* class highlighted.

An alternative method for invoking the Type Library Browser is to use the ProvideX command line shortcut program called **TLB**; e.g.,

```
Def object x, "MSCAL.Calendar"
TLB x
```

ProvideX OLE Server

The ProvideX OLE Server (`ProvideX.Script`) delivers the *reverse functionality* to that of **ProvideX COM Support** – it allows external applications to directly invoke and interact with ProvideX and ProvideX objects.

The purpose of this interface is to expose the ProvideX environment to virtually any *outside* COM-compliant application. With the OLE Server running, it is possible to create a ProvideX object, then invoke it from other languages such as VB, VBScript, Delphi, and C++. You can also use DCOM to invoke the ProvideX object remotely.

General descriptions of Microsoft OLE and COM are provided under **Concepts and Terminology**, *p.250*.

Registration of the OLE Server

Access to `ProvideX.Script` requires running `pvxcom.exe`, which is included with the installation of ProvideX for Windows. Use of the interface may require a separately-purchased activation key apart from your initial ProvideX for Windows activation. Further to this, `pvxcom.exe` may need to be registered manually on your Windows OS (if registration did not occur with your installation of ProvideX).

To add `pvxcom.exe` to the Windows registry, enter the following via Start > Run:

path\pvxcom.exe /regserver

path is the directory of the ProvideX version you are running. This can be *unregistered* by entering the following via Start > Run:

path\pvxcom.exe /unregserver

One way to verify that the executable `pvxcom.exe` is registered correctly, is by running the following VB script:

```
Set pvx = CreateObject("ProvideX.Script")
pvx.Init ""
pvx.execute("v=tcb(29)")
nm=pvx.Evaluate("v")
MsgBox "Version =" & nm
set pvx=nothing
```

Just copy the above code into a notepad session and save it as `tcb.vbs`. Double click on this file to run the script, which will show you the **TCB(29)** value reporting the current build level of ProvideX.

Using `ProvideX.Script`

The interface is invoked in an external language via `ProvideX.Script`; e.g.,

```
VB:      Set pvx = CreateObject("ProvideX.Script")
Delphi:  pvx:=CreateOleObject("ProvideX.Script")
Java:    var oPvx = new ActiveXObject("ProvideX.Script"); // Java
```

There are six ***methods*** available for use with `ProvideX.Script`.

Init (*path*) Called before accessing any other method in the script engine in order to set the working directory for the server and perform binding to the ProvideX dll layer. The *path* should be set to the desired working directory, or blank to indicate the current directory. (In most cases, the current directory will be the system directory).

Execute (*stmt*) Invoked to execute any ProvideX command statement.

Evaluate (*expr*) Evaluates and returns the expression provided and returns its value as a variant. (It can be used as either a numeric or string).

Run (*prog*) Runs the specified program. This method does not return until the program ends.

Reset ( ) Closes all local files and clears variables (executes a **BEGIN**).

NewObject(*name, params ...***)**
Creates a new object of the specified name and returns an object reference (see **Additional Properties**, below).

There are also three ***properties*** associated with the interface:

Instance Returns a unique string to identify the server during its lifetime.

State Returns the state of the server. 0 ***zero*** - un-initialized, 1 - initialized.

Parameter Read / write property used to access the specified ProvideX parameter.

Additional Properties

To create an object within the OLE Server, the method **NewObject** is used. It returns an object identifier to the ProvideX object that was created. Along with the ProvideX defined properties and methods, the OLE server adds the following properties:

Instance Returns a unique identifier of the `ProvideX.Script` object that was used to create the automation object. This is important because the automation object cannot be passed to another `ProvideX.Script` server.

ScriptObject Returns the parent `ProvideX.Script` object.

CmdHandle Returns the ProvideX handle that the automation object is tied to.

Naming Conventions

Be aware that COM is generally data-type insensitive; i.e., it does not support the **$** or **%** suffix. This can cause a problem when dealing with ProvideX via the OLE Server, since in ProvideX you can have `Cust_no$` and `Cust_no` (as two unique data variables).

To avoid this issue, the OLE Server mandates that all property references indicate the property type in the first character of the property name:

s - for string variables.
n - for numeric variables.
i - for integer variables.
o - for object variables (required).

For example, `Cat$` becomes `sCat`, `Dog` becomes `nDog`, and `Pig%` becomes `iPig`. These prefixes are only for use by programs using the OLE Server, not by the ProvideX application itself. This means that the property `Cust_no$` in ProvideX would be `sCust_no` when referenced by the OLE Server and the property `Cust_no` in ProvideX would be `nCust_no`, etc.

Note: Do not use these prefixes in the property definition.

Use of the **o** prefix allows the specification of properties and/or methods within ProvideX objects to be declared themselves as objects.

ProvideX as Windows Script

Through the use of the OLE server, ProvideX itself can become a script language for any Windows system. ProvideX files saved with a `.pvs` suffix may be passed to the ProvideX script engine to be executed as a script in Windows applications.

Using PvxScript

The ProvideX-based script files are exposed through `pvxscript.dll`, a specially-built interface that utilizes IActiveScript and IActiveScriptParse for access to MS scripting (hosts such as IIS, WScript, MS Script Control, etc). Code execution is passed to `pvxcom.exe`, which sends commands to `pvxwin32.dll`. The whole process looks like the following:

Windows Application --> PvxScript --> OLE Server --> ProvideX

The `pvxscript.dll` must be installed in the same directory as `pvxcom.exe` and `pvxwin32.dll`. As with `pvxcom.exe`, the `pvxscript.dll` may need to be registered manually on your Windows OS (if registration did not occur with your installation of ProvideX). Adding PvxScript to the registry ensures the creation of:

1. Default COM registry entries for the ActiveX library.
2. `WScript.exe` and `CScript.exe` associations in the registry. This allows you to create text-based scripts that can be saved with a `.pvs` extension, and then be invoked automatically in WScript/CScript (the scripting engine uses the associations to determine which ActiveScript engine will be invoked).

Usage Notes

Script definitions vary slightly from regular ProvideX language syntax in that the text is basically passed to ProvideX as a series of execute commands. However, there is an exception: any code that starts with a statement label and ends with **END** will be considered a program and built as if it were typed in. After the **END** directive, you can then issue a **GOTO** *label*; **RUN** to begin execution of the code (The short cut is **#***xxxxx* where *xxxxx* is the *label*).

Script-based programs do not recognize line numbers in the same manner as ProvideX. When an error occurs in a script, the line number reported will reflect the (sequential) line number from the original script input file / document.

Host applications can usually expose what are referred to as "*global named items*". These items are COM objects that the host exposes to the script interface to allow the script to interact with the hosting application. For example: WScript exposes `wscript`, `wsh`; IE exposes `window`; IIS exposes `response`, `request`, etc. It is the developer's responsibility to understand the syntax related to the environment he / she will be developing scripts for. Named items are based on the context of the host, so the same code may not be valid in all environments.

Sample Script

Following is a `.pvs` file that may be executed from Internet Explorer:

```
10  ! preinput -1300;preinput 0;input *
15  ! Uncomment line 10 to get the trace window
20  objArgs = %WScript'Arguments
30  MSGBOX str(%WScript'*)
40  MSGBOX str(objArgs'Count)
50  i=objArgs'Count-1
60  WHILE (i >= 0)
70  MSGBOX Str(i); MSGBOX objArgs'_$(i)
80  WEND
90  MSGBOX %WSH'*
100 STOP
#10 ! GOTO line 10 and EXECUTE
MSGBOX "Done"
```

This logic executes as follows:

1. Add lines 10 to 100 to the engine.
2. Execute a `GOTO 10.`
3. Execute a `RUN.` The code will stop running at line 100 (and the script interface gets control back).
4. Execute `MSGBOX` directly.

10
Data Integration

As mentioned in *Chapter 5.* **File Handling**, ProvideX supports several options for storing and retrieving data. ProvideX data can be utilized in different ways with a wide variety of third-party technologies, depending on your system design and how it fits with your overall system design and user requirements.

This chapter covers ProvideX data integration from different sides of the processing equation: facilities that enable external applications to access and read ProvideX-based data — facilities that enable ProvideX applications to store and use data maintained in an external format.

Topics

Overview

While most aspects of a business application can be served within the ProvideX family of products, today's end-users are often required to work with data that resides in completely different software worlds. Businesses may need to integrate popular "off-the-shelf" software with their legacy systems, applications and databases. ProvideX adheres to all the generally-accepted software standards, and this level of adaptability allows for the integration of ProvideX applications into several non-ProvideX environments.

ProvideX components are highly optimized to work together and developers have the tools necessary to build a sizable database entirely within ProvideX itself. But if there is a need, ProvideX data can be made readable to external applications via **ODBC** or using **PVKIO**. Conversely, ProvideX applications can be designed to work directly with data that is stored and maintained in a number of **External Databases** or as **XML Content**, over networks and on completely different operating systems.

In any case, ProvideX has many solutions to help businesses integrate their data, regardless of the software or platforms involved.

Introduction to SQL

Following is a brief discussion on Structured Query Language (SQL) and how SQL syntax works with ProvideX. It is recommended that you consult specific database manuals, industry publications, tutorials, and other online resources for more in-depth SQL documentation.

Topics

SQL is an English-like database access language created specifically for end-users to view, retrieve, and manipulate information from relational databases. All databases and / or file systems that supply generic interfaces, such as ODBC, must provide an SQL interface. ProvideX is no exception to this rule. SQL plays an important role in some of data processing functionality and interfaces described in this chapter.

Over the years, the language has been standardized by ANSI and adopted by a large number of database manufacturers. SQL's original intent was to provide ad-hoc access to data — not as a development language or as a database interface tool. With the advent of ODBC and other generic interfaces, SQL has become the de-facto industry standard for manipulating databases.

Because the SQL language is English-like in its structure, it is easy to learn and understand. Following is a typical SQL statement:

```
SELECT cst_id, cst_name FROM Customer
```

This retrieves customer numbers and names from the `Customer` table. For other SQL syntax examples, see **Data Access Using SQL**, below.

SQL Terminology

SQL has its own terminology (as stated earlier) which is intended to be better understood by the end-user. It is similar to the terms used to describe a spreadsheet. Following are the most commonly-used SQL terms:

Database

In SQL, a database is a collection of tables. For example, if you had a Sales Order application, you would consider all the related tables within the application (such as the Customer table, the Order table, the Product table, etc.) as a database. A system can have one or more databases depending on the needs of the application. The database may consist of one or more physical files.

Table

Normally, this term is used to describe a *logical* rather than *physical* file. For example, our database might have a Customer table, Product table, etc. Each tables is referred to by a meaningful logical name such as Customer, rather than by its physical name. Depending on the system, each table may be associated with a different physical file, or several tables may reside within a larger common physical file.

Row

This term applies to each of the logical records within a ***table*** (or file).

Column

This term applies to each of the logical fields within each ***row*** (or record). For instance, a Customer file / table contains records / rows which contains fields / columns such as Customer Number, Name, Address, Amount Owed, etc.

Data Access Using SQL

The basic SQL commands for accessing data are **SELECT** (to read and return data), **UPDATE** (to alter existing data records), **INSERT** (to add records) and **DELETE**, (to remove data records). These are outlined in the sections below. For a complete overview of the SQL syntax that is supported by ProvideX, refer to *SQL Syntax Table* in the ProvideX ***ODBC Reference*** documentation.

Reading Data

The **SELECT** statement is used to **READ** data from tables in a SQL database.

SELECT *columns* **FROM** *table*
WHERE *condition* **ORDER BY** *columns*

SELECT returns all rows that match the condition specified; e.g.,

```
SELECT * FROM "table"
```

This returns all columns from all rows in the specified table. The **WHERE** clause is used to specify which rows that the application is interested in retrieving; e.g.,

```
SELECT * FROM "Customer" WHERE SALESMAN = "MFK"
```

This would retrieve only those rows which have the column SALESMAN equal to the value MFK. Ideally, the columns specified on the **WHERE** clause will be a key / index field within the database, thus avoiding a linear scan of the table in order to return only those records that match the criteria.

Also, you can also specify that only certain columns are to be retrieved by stating their name in the SQL statement; e.g.,

```
SELECT CUSTOMER_NO, NAME, SALESMAN FROM "Customer"
```

Writing and Altering Data

The **INSERT** statement is used to **WRITE** new data to a SQL table.

INSERT INTO *table* (*columns*)
VALUES (*values*)

The **INSERT** directive is used to add new rows to an SQL database; e.g.,

```
INSERT INTO "Customer" ( CUSTOMER_NO, NAME, SALESMAN) VALUES
("123456", "Acme Plumbing", "MFK")
```

This would add a row / record to the table and update all the required key indices. If not all the columns for the table were specified in the **INSERT** command, then the columns omitted would be set to null. Depending on the database definition, each field may have validation rules such as ranges, reference tables, etc. If any of these validation rules are violated, the **INSERT** reports an error and the row would not be added.

The **UPDATE** directive is used to alter the contents of all rows in the database that satisfy the condition.

UPDATE *table* **SET** *column=value*, …
WHERE *condition*

If no condition (**WHERE** clause) is specified, all records in the table are altered; e.g.,

```
UPDATE "Customer" SET NAME = "Acme Plumbing"
```

This statement would set the column `NAME` to `Acme Plumbing` for all rows / records in the `Customer` table. In almost all cases, there will be a **WHERE** clause when using **UPDATE**.

Removing Data

The **DELETE** directive is used to remove rows from a table.

DELETE FROM *table*
WHERE *condition*

Like the **UPDATE** directive, there is almost always a WHERE clause to identify the rows to be removed; e.g.,

```
DELETE FROM "Customer"
```

The above SQL statement will result in the deletion of all rows in the `Customer` table, whereas the following will result in the removal of only those records that have `CUSTOMER_NO = "1234"`.

```
DELETE FROM "Customer" WHERE CUSTOMER_NO = "1234"
```

How ProvideX Translates to SQL

When a ProvideX application accesses an external database and table, file I / O directives and functions are mapped automatically to the equivalent SQL statements described previously. This process is described below.

For documentation on supported interfaces, see **External Databases**, *p.320*. For more information on the ProvideX input and output operations discussed in this section, see *Chapter 5.* **File Handling**.

To open an external database table for access from within ProvideX, the *filename* must consist of a **[***tag***]** representing the supported interface, followed by the *database* and *table* name; e.g.,

```
OPEN (1) "[ODB]MYDB;CUSTOMER"
```

The example above establishes a connection to the ODBC datasource name (DSN) "MYDB" and table "Customer". Current database interface options in ProvideX include **[ODB]**, **[OCI]**, **[DB2]**, and **[MYSQL]**. Specific syntax for these interfaces can be found in the *Language Reference,* Chapter 9.

Once the channel is defined as an external database file, ProvideX will automatically generate SQL statements in place of the input and output operations to that channel. Each **READ**, **WRITE**, and **REMOVE** is translated into SQL commands specifically for the selected table; e.g.,

```
0010 ! SQL001 - Basic SQL access
0020 OPEN (1)"[ODB]MYDB;CUSTOMER"
0030 READ RECORD (1)R$
0040 PRINT R$
0050 END
```

A simple **READ** directive to the database and table from the ProvideX application would issue the following SQL statement:

```
SELECT * FROM CUSTOMER
```

Each row of data returned by the SQL statement results in a logical record consisting of columns with each column separated by the field separator ($8A$). ProvideX has a built-in SQL debugging facility that allows you to see the generated SQL directive. To view the actual SQL, set the **'!Q='** system parameter.

Keyed Access to an SQL Database

To emulate **KEYED** file access, the open file name can include the name of the field (or fields) to be used as the record key; e.g.,

```
0010 ! SQL002 - Basic SQL Keyed access
0020 OPEN (1)"[ODB]MYDB;CUSTOMER;KEY=CST_ID"
0030 LET K$=KEY(1)
0040 READ RECORD (1)R$
0050 TRANSLATE R$,",",SEP ! Change Sep to ',' for readability
0060 PRINT "Key=<",K$,"> Rec=<",R$,">"
0070 END
```

The database can then be accessed as if it was a keyed file with the key field being the CST_ID column; e.g.,

```
READ (1,KEY="SAGE")
```

This converts into the following SQL

```
SELECT * FROM CUSTOMER WHERE CST_ID = 'SAGE'
```

A subsequent **READ** translates to the following:

```
SELECT * FROM CUSTOMER WHERE CST_ID > 'SAGE'
ORDER BY CST_ID
```

Multiple **KEY=** clauses may be specified. The first **KEY=** clause defines the Primary key, which must be unique. All subsequent **KEY=** clauses define the alternate keys and can be referenced using the **KNO=** option in the **READ** and **WRITE** directives.

The **KEY( )**, **KEP( )**, **KEL( )**, **KEF( )** and **RNO( )** functions can be used against an external database as well as the **RNO=** option. **KEN( )** does not return key information, but it does return information about the database. The **REFILE** directive is not supported.

Building an IOList from an SQL Database

If the **OPEN** directive includes the **IOL=*** option, then ProvideX automatically creates an **IOList** based information in the external database; e.g.,

```
0010 ! SQL003 - Creating IOList
0020 BEGIN
0030 OPEN (1,IOL=*)"[ODB]MYDB;CUSTOMER;KEY=CST_ID"
0040 PRINT LST(IOL(1))
0050 ESCAPE
0060 READ (1)R$
0070 DUMP
0080 END
```

ProvideX accesses the internal data dictionary to get fields for the table. Column names are used to generate field names within the generated IOList. Invalid characters are replaced with the _ *underscore* character. Numeric data are converted to numeric variables, while all other data types are converted to strings.

Defining a Logical Record Layout

If desired, a **REC=** option can be included in the **OPEN** pathname to define the exact format of the record that is to be returned to the ProvideX application; e.g.,

```
0010 ! SQL004 - Manually defining record
0020 BEGIN
0030 OPEN (1)"[ODB]MYDB;CUSTOMER;REC=CST_ID,CST_NAME+CST_ADDR"
0040 READ RECORD (1)R$
0050 DUMP
0060 END
```

The **REC=** option consists of a series of column names separated by either commas or **+** plus signs. If column names are separated by a comma, then a field delimiter character is inserted between the column. If the column names are separated by a plus sign, then the first column will be padded to its maximum length followed immediately by the second column.

In the example, `REC=CST_ID,CST_NAME+CST_ADDR` yields a record consisting of the contents of the `CST_ID` column, a field separator ($8A$), and the contents of `CST_NAME` padded to its maximum length, which is immediately followed by the contents of `CST_ADDR`.

The record format specified is used as well to parse record data written to the file by ProvideX programs.

Using SQL Directly Within ProvideX

The generic SQL statements described in the previous sections can also be passed "as is" from ProvideX to the target database. There are two ways that this can be accomplished. In the syntax examples below, **[***tag***]** represents one of the available database options, and **<***SQL***>** represents an actual SQL statement embedded within a ProvideX statement.

The following syntax sends the SQL statement through a channel tied to a table (limited to the size of the **KNO** used):

OPEN (*chan***,IOL=*)"[***tag***]***database***;***table*[;*fileopt*]"
READ (*chan***,KEY="!<***SQL***>")**

SQL statments can be passed directly using a straight connection to the database without specifying the table name, as follows:

OPEN (*chan***,IOL=*)"[***tag***]***database***;**[;*fileopt*]"

To issue an SQL statement and retrieve the first result value:

READ [RECORD] (*chan***,KEY="!<***SQL***>")** *iolist*

To simply issue an SQL statement:

WRITE RECORD (*chan***)"<***SQL***>"**

To retrieve the results:

READ [RECORD] (*chan***)** *iolist*

Note: Ensure that you are using the correct SQL syntax for the target database; i.e., refer specific product documentation for the database system being accessed.

External Databases

There are a few ways for ProvideX to interface with non-ProvideX data files. This material is intended for those who need to maintain their data in an external database.

Topics

The facilities described in this documentation are designed to emulate a collection of ProvideX files in an external database. The primary advantage of this approach is that the same ProvideX program will work regardless of the database being used. However, you should be aware that interfacing your application to an external database can result in lower performance. We recommend that you consult with an expert in the external database product to help optimize the conversion process.

Note: Developers should have a good working knowledge of SQL if they plan to use these facilities. For a brief discussion on SQL, see **Introduction to SQL**, *p.314*. A basic *SQL Syntax Table* can also be found in the ProvideX ***ODBC Reference*** documentation.

The *ODBC* interface is supported in ProvideX for Windows (plus some UNIX/Linux releases) and it allows access to any database system that publishes an ODBC driver. Use the **[ODB]** tag as a prefix in an **OPEN** statement to denote that ProvideX is to route all file I/O requests to an external ODBC database. UNIX/Linux servers also support ODBC through the use of WindX with an open ODBC connection on the WindX workstation.

The remaining conversion facilities are *call-level interfaces* to specific products: Oracle, IBM DB2, and MySQL. These types of databases are supported on Microsoft Windows and various UNIX/Linux platforms. Access to these databases is denoted using the **[OCI]**, **[DB2]**, or **[MYSQL]** tag as a prefix in an **OPEN** statement. These may require a separately-purchased activation key apart from your initial ProvideX activation, so contact your local ProvideX dealer/distributor or visit www.pvx.com for product information and licensing.

Implementation involves the selection of a target database, or perhaps databases, and the writing of the routines that will allow the re-creation of the ProvideX database on the target database. This process is described in the sections that follow.

Selecting a Database Type

As mentioned in the previous section, ProvideX supports different interfaces. The one that most people are familiar with is Open Database Connectivity (ODBC). This interface is a set of calls that provide standardized access to a variety of data. The

data can be stored in a flat file, a proprietary format such as ProvideX, or a Microsoft SQL Server. ProvideX can read and write that data as long as an ODBC driver exists for the data store.

The main problem with this approach is the overhead imposed by the ODBC manager. The call-level interfaces were created for ProvideX to overcome these limitations. Direct access to Oracle, DB2, and MySQL is available for selected platforms without the overhead of the ODBC administrator. Another advantage to using one of the direct interfaces is that functionality that is specific to the target database can be added to the ProvideX interface.

The decision of which database product to use can be difficult. It depends on the target market and the customer's in-house knowledge and skill set. If the target market is Windows, then it might make more sense to go with MS SQL Server. If the target market must support clients on UNIX/Linux, then a Microsoft-only approach may not be feasible.

Each database will need to be evaluated for usability, and each has its own learning curve – some steeper than others.

Interface Options

Use one of the following **Special Command Tags** as a prefix in an **OPEN** statement to denote that ProvideX is to route all file I/O requests to the selected external database.

[ODB] Built-in ProvideX functionality. *You do not need the ProvideX ODBC driver to use this tag*. ProvideX supports ODBC under Windows as well as two open source versions of ODBC for UNIX/Linux (iODBC and unixODBC). Use **TCB(197)** to determine if ODBC support is enabled for a given UNIX/Linux system.

[DB2] Requires activation of ProvideX DB2 support. Use **TCB(198)** to check if DB2 is supported on a platform.

[MYSQL] Requires activation of ProvideX MySQL support. Use **TCB(194)** to check if MySQL is supported on a platform.

[OCI] Requires activation of ProvideX Oracle Call Interface (OCI) support. Use **TCB(200)** to check if OCI is supported on a platform.

These options are documented in *Language Reference*, Chapter 9. For additional information, see **OPEN** in the *Language Reference, p.231*.

OLE DB

Object linking and embedding for databases (OLE DB) is a set of Component Object Model (COM)-based interfaces that provide applications with uniform access to data stored in diverse information sources, or data stores. OLE DB accesses disparate data

in a standard, object oriented way when using a Microsoft Operating System. ProvideX does not have an OLE DB consumer; however, developers may write their own interface via **ProvideX COM Support**, *p.261*.

Connections

When working with an external database, ProvideX does not interact directly with the data in the file. The data is accessed through an interface. That interface may result in local calls to a DLL that reads the data from disk, or it might involve opening a TCP/IP connection to a server on the other side of the planet. This is the first place where a failure may result in an error not handled by existing code. If the **OPEN** fails, an `Error #15` will typically occur, with the text provided by the data provider available via **MSG(-1)**.

Each connection to the database may require an access licence. When using a Professional or eCommerce bundle, ProvideX will access multiple tables using a single connection. This results in only a single licence being used. The developer should calculate the resource cost of a base system with database access licences against the cost of a Professional or eCommerce ProvideX bundle

Creating the Database

The step subsequent to installing a chosen database system is to create a database, or databases, for the application. When doing so, be aware of case sensitivity and the sort (collation) sequence of the database. These issues are discussed below.

Case Insensitivity

Case insensitivity most often affects key values, but it may also affect table names and column names. While having the table names and column names case insensitive makes using the database far easier to access and manipulate for the user, having case insensitive keys throughout the system will likely have major implications on your application.

If possible, when creating the database, specify a case sensitivity and collation sequence that matches ProvideX. If you cannot do this then consider that there may be an impact on the following areas:

- Output sort sequences
- Files that have upper/lowercase keys that are identical
- Usage of extended ASCII characters and hex data in applications.

With the following effects:

- Report sequences will differ, if case is involved.
- Program logic may have to be revisited.
- Cannot rely on the order of the characters.
- Special considerations need to be made when handling mixed case data.
- It is possible that file layout and contents may need reconsideration.

Sort Sequence

By default, Microsoft SQL Server ***does not*** use the standard ASCII sort sequence when maintaining sorted lists. The sort sequence treats upper and lowercase characters identically and places special characters at a different place within the sequence. This can cause problems, not only in the order of the data returned, but also in the program logic.

Generally, while the output of data in reports and screen displays is not a major problem, users will need to get used to the new sort sequence when reviewing output. Major problems can arise when the application makes assumptions about the sort sequence and the order of special characters, numeric characters, uppercase data, and lowercase data.

An inadvertent lowercase key in a data file can cause problems; e.g., assume the keys,

```
CAR010, CAR020, Car025, CAR100, CAR150
```

If you were to write logic such as:

```
P$ = "CAR"
READ (1, KEY=P$,DOM=*NEXT)
LOOP:
K$ = KEY(1,END=Done); IF K$(1,3)<>P$ THEN GOTO Done
READ (1); PRINT K$; GOTO LOOP
Done:
```

This would not work correctly (as in ProvideX files) since, as soon as `"CAR"` is not the first three characters of `"Car025"`, the loop would exit without processing `CAR100` or `CAR150`.

Also, consider the impact of upper and lowercase keys during an *update*. Key fields with different cases are considered the same; e.g.,

```
WRITE (1,KEY="Mike")
WRITE (1,KEY="mike")
WRITE (1,KEY="MIKE")
```

All of the above update the same record as opposed to updating/creating three unique records in a ProvideX file. The impact of this is that a record key of `"Jeans Unlimited"` is the same as `"JEANS UNLIMITED"`. While in many places this is acceptable, it can cause numerous problems in applications that use upper/lower case keys as a means to segregate records. Another issue is that the database may insert accented characters in the sort sequence directly following the non-accented characters. While this may seem somewhat harmless, it has a major problem if you attempt to use `$FF$` as a partial key; e.g.,

```
READ (1,KEY = K$+$FF$, DOM=*NEXT)
```

`$FF$` just happens to be **ÿ**, which means that the above **READ** positions the file somewhere between `Y` and `Z` – not *after* `Z`. We recommend switching to `$FE$`, which does not correspond to any accented characters in most character sets.

Binary Fields

Case insensitivity also applies to accented characters as well; therefore, any use of high order ASCII data may result in missing data, etc. One solution is to use Binary fields to resolve the issue. This would result in:

- Sort sequence becomes normal ASCII
- Data becomes case sensitive
- Accented characters return to binary sequence

However, the data can not be used directly in SQL statements, as the following is required:

```
SELECT CONVERT( CHAR , column_name )
```

Products, such as Crystal Reports, may need the data to be converted before it can be used in a report. In most cases, the key fields usually consist of data that is contained within the actual record, so that no conversion is needed. Only if the key fields are required, would they need to used with the **CONVERT** function.

When using Binary fields in the **REC=** clause, a **:B** following the column name indicates to the ODBC system that the field is Binary. Failure to do this can result in invalid data being stored in the system.

Spaces and Null Character

Within Microsoft SQL, trailing spaces are considered redundant. This means that records whose key values differ by trailing spaces only would be considered equivalent. For example, "ABC" is treated same as "ABC ". In most cases, this does not cause any application problems, but if the application is sensitive to it, then the key fields should be defined as Binary.

Another issue is that the null character (HEX 00) cannot be stored in standard text columns. Therefore, if null bytes are significant to your application, then the column should be declared as Binary. A null character is used internally to indicate the end of a SQL statement; therefore, not having the field declared as Binary, will result in the SQL statement being truncated.

Creating Tables

The first requirement for conversion to an external database is a *data dictionary* of some sort. This can be an existing ProvideX **data dictionary** or a custom solution. The database must be accurate, as relational databases will not allow strings to be stored in numeric fields.

To ease the conversion, there should be one table per physical file. If there are non-normalized (multi-record type) tables this will still be true. Changing this relationship will require much more extensive testing.

If you are working with ProvideX databases, where an identifier (i.e., company code) is part of the physical file name, create multiple databases as opposed to multiple tables. For example, a system with a two-character prefix for the company might

have physical tables "10customer" and "11customer". Rather than creating two tables in a single database, create two databases, "C10" and "C11", each with its own "customer" table. This simplifies integration with third party tools such as Crystal Reports.

All ProvideX file types are supported, including Indexed and Sort; however, for special conditions that must be met, refer to **Creating the Prefix File**, *p.325*.

When creating tables, be aware of limits such as the length of names, the maximum size for a particular data type, and possibly the total number of columns allowed per table. Reserved words (e.g., DATE) in column or table names are not allowed. Also be aware of what characters are allowed in names.

Date fields require special handling when creating tables. ProvideX only supports a limited number of translations between the standard database format of YYYY-MM-DD and the multitude of date formats used in Business Basic. If the application uses a date format that is not supported by ProvideX, then date fields must be stored as a simple string in the database. The alternative would be to request that the format be added to ProvideX, and then wait for a release that includes that format.

ProvideX Data Dictionary

If you are using an **Embedded Data Dictionary**, then the process of creating the database tables is vastly simplified. ProvideX ships with a utility program, ***DICT/GENSQL**, that generates the data definition language (DDL) statement from the data dictionary entries. This utility was written with the intended target of Microsoft SQL Server, so the syntax of the **CREATE TABLE** statement may need to be modified for other databases. For example, Microsoft SQL Server uses the data types Varchar and Decimal, whereas Oracle uses the keywords Varchar2 and Number.

Creating the Prefix File

The **OPEN** directive is used to establish a logical connection between ProvideX and the database. The path name and option string defines the database, the name of table to be accessed, and a variety of data formatting and processing options.

Use the **PREFIX FILE** directive to specify the special Keyed file that contains information to be used for dynamic translations of files when they are opened in your applications.

The prefix file key is the old file name; i.e., CSTFILE. If the company code is part of the physical file name, then two separate records are required – one for each unique file name. For example, if all the files are prefixed by the company code (i.e., 10 and 11), then a 10CSTFILE and 11CSTFILE must exist within the file.

If the same file is opened in different ways (sometimes with an absolute path, sometimes with a relative path) then each case must be added to the prefix file.

For example,

```
OPEN(1)"SimpleName"
OPEN(2)"./SimpleName"
OPEN(3)"/usr/data/SimpleName"
OPEN(4)"SIMPLENAME"
```

Each of the above **OPEN**'s represent the same file, but require a unique name in the prefix file.

Normally, the pathname contains a variety of file access and formatting options as well. These options are used to define information regarding how to access the table to ProvideX. Typically, this includes the definition of the key fields, record formatting characteristics, and the details regarding variant record processing for non-normalized files.

Additional options can provide information to the database connection; e.g., user ID and password, database qualifier name (database name within server), or record locking characteristics.

Because the size of the string needed to pass this information to ProvideX can become quite long and, due to the fact that ProvideX limits file path names to a maximum of 511 characters, options can be specified in the **OPT=***string* clause for the **OPEN**. Options provided in the **OPT=** string are not treated any differently than those options passed in the pathname.

The prefix file data records consist of the actual pathname to use in the first field and the option list in the second field. For example, in MAS200 (pre-Version 4.x) the ARF terms code file is described as follows:

```
File:       ARFABC.SOA
Fields:     TermsCode              2char     (concatenated with)
            Description           30char    (concatenated with)
            DaysBeforeDue          3char    (concatenated with)
            DueDateADayOfMonth     1char    (concatenated with)
            DaysBeforeDiscDue      3char    (concatenated with)
            DiscDateADayOfTheMonth 1char   (concatenated with)
            MinDaysAllowedInvDue   3char   (concatenated with)
            MinDaysAllowedDiscDue  3char   (concatenated with)
            DiscountCalcMethod     1 char
            DiscountPercentage     numeric 19.7
Key:        TermsCode
```

To record the definition to the prefix file:

```
KEYED "PFXFILE",12
OPEN (1) "PFXFILE"
WRITE(1,KEY="ARFABC.SOA")"[odb]MAS90MFK;ARF_TermsCodeMasterfile;DB=MAS_AB
      C;","KEY=TermsCode;REC=TermsCode:2+Description:30+DaysBeforeDue:3+D
      ueDateADayOfMonth:1+DaysBeforeDiscDue:3+DiscDateADayOfTheMonth:1+Mi
      nDaysAllowedInvDue:3+MinDaysAllowedDiscDue:3+DiscountCalcMethod:1,D
      iscountPercentage:19.7"
```

```
CLOSE (1)
```

Generally, we suggest that the first field contains the DSN / Table declaration and the second field contains the record layout.

Once this record is created, the application can enable the **PREFIX FILE** and access the database table by opening the file ARFABC.SOA.

```
->PREFIX FILE "PFXFILE"
->OPEN (1) "ARFABC.SOA"
->PRINT PTH(1)
[odb]MAS90MFK;ARF_TermsCodeMasterfile;DB=MAS_ABC;
->
```

The file will need to be created with a record size large enough to accommodate the largest file definition string that the application requires. A prefix file with a record size of 4000 bytes is not uncommon.

Conversion Process

The conversion process moves the data from the ProvideX database to the new tables in the target database. It may also involve some program modifications for the application to function correctly and efficiently with the new database.

Loading the Data

Once the prefix file has been defined, the load is as easy as *read record – write record*. There are two approaches to opening the existing tables and the database. One approach is to set the prefix file (using the **PREFIX FILE** directive), open the data directory, then (for each file in the data directory), to open the physical file plus the database table; e.g.,

```
10 begin
20 prefix file "db2_prefix.dat"
30 open(1)"data/"
40 read (1,end=eod)f$
50  open(2)"data/"+f$ ! include the directory so it doesn't match prefix
      file key
60 open(3)f$ ! prefix file will redirect to database
70 read record (2,end=eoi)r$
80 write record(3)r$
90 goto 70
100 eoi:
110 close(2); close(3)
120 goto 40
130 eod:
140 print "done"
```

This approach works well with a few data directories, but it can be clumsy when there are complex prefix rules. An alternative approach is to open the prefix file and use it in conjunction with the prefix rules to open the physical data files.

Considerations When Loading Data

The approach taken for opening tables can vary depending on the conversion situation. If it is a *one-off* conversion, then hard-coded paths may be acceptable. However, if the conversion may have to be run on hundreds of sites, each with a unique directory structure, then a more flexible procedure should be considered.

Be sure to account for reloads. It is unlikely that the data load will only be run once. It is more likely there will be multiple loads, as the process is adjusted to achieve correct data conversion. A table may need to be reloaded because, either the database table itself changed, or the rules applied by the prefix file have changed. It may be that the entire conversion will need to be re-run, or a single table may need to be reloaded.

One of the more difficult situations to handle is when the database table has been *altered*. It is best to manually adjust the input database rather than write a routine that must check for differences between the database table and the original definition. This is not impossible, and it can be accomplished using the ***dict/sql** and related objects.

Test Your Application

This is the area that will take the most time. Every line of code that affects what is read or written to file needs to be ***tested***. Before and after versions of reports need to be ***compared***, and every insert, update, and delete needs to be tested. Timing tests should be run to see which areas suffer significant performance degradation due to the overhead associated with the external database system.

Other areas that need to be thoroughly examined are those that involve file creation, deletion, or renames. One example would be the creation of a new company. This is an area that has to be re-written if the logic creates new keyed files, as the **KEYED** statement is not translated into a **CREATE** statement. Not only would the table creation have to be addressed, but the tables would also need to be added to the prefix file.

Also check areas that depend on the **FIN( )** and **FIB( )** functions. These functions do not contain all the information that is available for a Keyed file; e.g., the key definitions and current record count are not available.

Dynamically-named temporary tables can also be difficult to emulate. Often these files are left as ProvideX files due to their *temporary* nature.

If a version of ProvideX prior to Version 7 is used, the **ERASE** and **PURGE** directives are not supported on external database tables. Support for these directives was added in Version 7; however, they may still fail if there is insufficient information to open the database.

Performance Tuning

Generally speaking, external databases are slower than native ProvideX files. This is for two reasons usually: 1- indirect access to the data, 2- different data models (result set versus row set orientation). However, by adjusting a number of system options, you may be able to improve system performance (substantially in some cases). The following approaches can be taken to improve the performance of an application:

- Shared connections.
- Eliminate data dictionary lookup (define record layout).
- Using the **TOP** option in the **SELECT** statement.
- Stored procedure for **RNO**.
- Using the TSQL option.
- Prepared statements.

Note: Some of these settings may not be ideal for all applications.

Shared Connections

Using shared connections reduces time to **OPEN** files and uses less resources and users on the server. This is enabled by default for ProvideX ***Professional*** or ***eCommerce*** activations, unless the INI file contains `UNIQUE=Y`, or `y`, or `1`.

It shares connections in same sessions that:

- Have same DSN / database name
- Have same database / qualifier

System performance is improved by sharing the connection, since all the logic involved with user authentication, security, and establishing the connection, is avoided. This leads to quicker file **OPEN**s in the application, as well as reduced system resources by sharing connections.

Another advantage of using shared connections is that database servers often charge based on the number of concurrent connections used. Sharing a single connection per process can reduce costs.

The criteria to share files is that both must have the same DSN and Database / Qualifier name (case is not important).

Eliminate Data Dictionary Lookups

By default, whenever ProvideX opens an external database, it reads the data dictionary for the table in order to determine column names, data types, and format (size / scale). This can account for a *significant* amount of processing and data transfer *overhead*. It can be particularly critical if a file is being opened only to access a couple of records.

In order to avoid this, include the field format on the **REC=** clause. If all the columns specified in the **REC=** clause have their size and type defined, then ProvideX will not attempt to read the data dictionary for the table and will assume that the definition supplied is correct.

There is one drawback to defining the field formats on the **REC=** clause. There is no run-time validation to check that the columns that have been specified, actually exist within the database. Therefore, it is possible to cause ProvideX to create SQL directives that cannot execute.

Defining the Record Layout

The **REC=** phrase is used to control the formatting of the data record as viewed by the ProvideX application. The format consists of a series of field descriptors and/or literals, each separated by either a comma or a plus sign. (Record type indicators can be present within the **REC=** phrase as well, but these will be discussed later.)

The simple format is:

REC= *fieldspec* **{ ,** *or* **+ }** *fieldspec* **...**

Where:
The *fieldspec* contains the name of the field and optional format, length, and scale. Fields are separated by either a comma or plus sign. When *comma*-separated, then a field delimiter is inserted. When *plus*-separated, then the field is padded to full size and no separator is inserted. Literals may be included if enclosed in apostrophes.

Example:

```
REC=CST_ID, NAME, ADDRESS
```

This results in a record with three fields, each separated by a field separator.

```
REC=CST_ID + NAME + ADDRESS
```

This results in a record consisting of the fields with each one padded to its full length and no intervening field separator. For example, if `CST_ID` is 6 characters long and `NAME` and `ADDRESS` are both 30, then the record would be 67 characters long, including the record terminator.

Any column name can be followed by an optional colon and format specification. This format specification consists of a data type (if not numeric or string) followed by the field length. If the field is numeric, the length includes a decimal point followed by the number of decimal positions.

The possible data types are:

`P`	Packed (BIN) data
`H`	Data is stored in HEX
`B`	Data is a Binary field
`D`	Field is a Date

Some examples would be:

`CST_ID:7`	(*string, seven characters long*)
`OWING:8.2`	(*8 digits with 2 decimal places*)

AMOUNT:P4.2 (*4 bytes containing BIN value scaled by 100*)
NAME:B30 (*30 byte binary field*)

It is a good idea to include the field descriptions for all fields since this prevents ProvideX from having to read the table's data dictionary to determine field sizes and types. If ProvideX has to use the database dictionary, the REC= values may be overwritten.

Hex and Binary values can be used to store non-printable and/or binary data that would cause problems otherwise when passed in a SQL statement. Binary fields (type P) can be used to define numeric data that has been packed into a string using the **BIN( )** and **DEC( )** functions. If specified, the scale indicates the number of implied decimal places that the value contains.

Literals may be inserted within the record layout in order to insert padding where a field or column is not presently used, but space has been reserved for it. Literals should be enclosed with apostrophes and separated by a comma or plus sign.

Defining a Keyed File

The following is required:

- DSN or database name
- Table name
- Record format optional, if each field matches the database definition
- Keys using the **KEY=** option
- First **KEY=** is **KNO #0**, Second is **KNO #1**, …

The most common file format within ProvideX applications is a Keyed file where the key for the record is contained in the record data. These types of files can be defined directly as tables with each data field defined as a single column in the database.

The **KEY=** clause can be used to define the field(s) that comprise the record key. If more than one key is required then multiple **KEY=** clauses can be included for each alternate key required. The following is an example of the definition for a Keyed file:

File: ICALPXF

Fields:

Company	2 char
Alpha_sort	10 char
Item_num	20 char
Stocked	1 char
Update_Inventory	1 char

Key 1: Company, Item_num
Key 2: Company, Alpha_sort, Item_num
Key 3: Company, Stocked, Alpha_sort, Item_num
Key 4: Company, Update_Inventory, Alpha_sort, Item_num
Key 5: Company, Stocked, Update_inventory, Alpha_sort, Item_num

Definition:

```
[DB2]MYDB;IC_ITEM_ALPHA;
REC=COMPANY, ALPHA_SORT, ITEM_NUM, STOCKED, UPDATE_INVENTORY;
KEY=COMPANY, ITEM_NUM; KEY=COMPANY, ALPHA_SORT, ITEM_NUM;
KEY=COMPANY, STOCKED, ALPHA_SORT, ITEM_NUM; KEY=COMPANY,
UPDATE_INVENTORY, ALPHA_SORT, ITEM_NUM; KEY=COMPANY, STOCKED,
UPDATE_INVENTORY, ALPHA_SORT, ITEM_NUM
```

This would declare a Keyed type of file with one primary and four alternate keys.

Defining an Indexed File

The following is required:

- DSN name
- Table name
- Record format
- Record Index field
- Must be a numeric field in database
- Should be 8 digits in length - no decimal points

While databases do not typically support an Indexed-style table structure, ProvideX can emulate this file structure using a numeric field within the table as the record index. This numeric field should contain a minimum of 8 digits with no decimal points.

The following is an example of the definition for an Indexed file:

File:	`PODABC.SOA`		
Index field:	`IndexSql`	Numeric 8	
Fields Rec#0:	`RecordsActivelyUsed`	Numeric 19.7	
	`MaximumRecordsInFile`	Numeric 19.7	
	`NextNewIndexAvailable`	Numeric 19.7	
	`LastRemovedIndexRecord`	Numeric 19.7	
Fields other:	`StdMessageLine1`	50 char	(concatenated with)
	`StdMessageLine2`	50 char	

Definition:

```
[MYSQL]MyDB;POD_POMessage;
IND=IndexSql:8; TYP=IndexSql,1,8; REC=?"00000000"+
RecordsActivelyUsed:19.7, MaximumRecordsInFile:19.7,
NextNewIndexAvailable:19.7, LastRemovedIndexRecord:19.7+
?"........"+ StdMessageLine1:50+ StdMessageLine2:50+ '              '
```

Defining a Direct File

Basically, this is the same as a Keyed file. The following is required:

- If the External Key does not exist as field(s) within the record data, then the **KEYDATA=** clause must be specified.
- **KEYDATA=** must specify a single field
- This field is the primary key of the record

• Field must not appear within the **REC=** fields

Most Direct files can be defined using the same format as a Keyed file. However, if the key for a Direct file is not contained within the data columns, then a special **OPEN** syntax is required. This is due to the fact that ProvideX normally attempts to generate the record key from the contents of the record data columns.

The **KEYDATA=** option is used to define a single column name for the data that contains the record key. In many cases, it will be used when the key is not derived directly from the data in any logical manner or when the file is non-normalized and the key components vary from record type to record type. The column specified in the **KEYDATA=** clause must not exist in the **REC=** clause.

The following is an example of the definition for a Direct file whose key is not made up of column data and contains two record types:

File:	ARWABC.SOA		
Fields "F" rec:	RecordNo	1char	(concatenated with..)
	FromDivision	2 char	(concatenated with..)
	FromCustomerNumber	7 char	(concatenated with..)
	ToDivision	2 char	(concatenated with..)
	ToCustomerNumber	7 char	
Fields "T" rec:	RecordNo	1char	(concatenated with..)
	ToDivision	2 char	(concatenated with..)
	ToCustomerNumber	7 char	(concatenated with..)
	FromDivision	2 char	(concatenated with..)
	FromCustomerNumber	7 char	

Definition:

```
[ODB]MyDB;ARW_GlobalCustRenEntry
KEYDATA=KeySql:B10; TYP=RecordNo,1,1; REC=?"F"+RecordNo:1+
FromDivision:2+ FromCustomerNumber:7+ ToDivision:2+
ToCustomerNumber:7+ ?"T"+RecordNo:1+ ToDivision:2+
ToCustomerNumber:7+ FromDivision:2+ FromCustomerNumber:7
```

Basically, the key for this file is the record as defined, thus the actual columns used to create the key, vary between the "T" and "F" records.

Defining a Sort file

Defining a Sort file requires the following:

• DSN and table name.
• Record format.
• Fields must be concatenated with the '+' operator.
• Specify **KEY=*.**
• This indicates that the key is the record.

• No record data will exist logically.

Sort files pose a different problem to the ProvideX interface, as there is no data record but just a key field.

To define a Sort file, the table definition should contain the columns that are required to construct the key with a **REC=** clause which defines the layout of the key. This requires the use of the **+** plus sign between all the fields. Use a **KEY=*** option to indicate that the record itself is the key and that no record actually exists.

An example of an **OPEN** definition for this type of file follows:

File: ARCALX

Fields:

Company	2 char
Alpha_Sort_Key	10 char
Customer_Num	10 char

Definition:

```
[ODB]MyDB;AR_Cust_Alpha ;
REC=Company+Alpha_Sort_key+Customer_Num ; KEY=*
```

TOP option in the SELECT statement

The **TOP=** option is a method to limit the number of rows being retrieved for a result set. The syntax varies between databases but the effect is the same. It reduces data transfer.

Specifying the **TOP=***nnn* option, either in the appropriate database section of the ProvideX INI file or on the **OPEN** options, indicates to ProvideX that the database accepts the "**SELECT TOP** *nnn ...*" SQL command. If this option is not present, then ProvideX assumes that the database does not support this capability. The DB2, MySQL, and Oracle interfaces also support **TOP=**; however, they use the database specific syntax.

If **TOP=** is provided using the value -1, then the **KEF( )**, **KEL( )**, and **KEP( )** functions will generate code that requests only a single record from the database. Without the option, these functions generate SQL statements that will result in a dataset the size of the table being generated.

In addition, if this option is specified with a positive non-zero value, then the standard read next logic will limit its reads to a series of **SELECT** statements, each limited to the number specified. This may reduce unneeded data selections from occurring. For example, consider a file with 1,000 records that are grouped by individual prefixes so that never more than 20 records are retrieved. By adding **TOP=20**, the database will stop generating the result set once it has found 20 records.

If **TOP=** is greater than zero, it is possible to 'lose' records. When a non-unique key is read sequentially with a **TOP=** clause, once the result set is read, ProvideX will start with the next group of records. For example, five non-unique alternate keys:

```
ID,  Job
0001 MANAGER,
0002 MANAGER,
0003 SALESREP,
```

```
0004 SALESREP,
0005 SALESREP
```

If reading the above using Job as the sort sequence and a **TOP=** value of 1, then only one MANAGER record, and one SALESREP record would be retrieved; i.e.,

```
0001, MANAGER
0003, SALESREP
```

This is because the result set is exhausted after one **READ**, so ProvideX would generate a **WHERE** clause with the value greater than the value just read. The SQL statements are:

```
SELECT TOP 1 * FROM Employee WHERE Job > ''
```

A **READ** retrieves ID 0001. The next **READ** returns no data, so the next SQL statement is:

```
SELECT TOP 1 * FROM Employee WHERE Job > 'MANAGER'
```

A **READ** retrieves ID 0003. The next **READ** returns no data, so the next SQL statement is:

```
SELECT TOP 1 * FROM Employee WHERE Job > 'SALESREP'
```

This returns end-of-file.

Stored Procedure for RNO

This applies to applications that use the **RNO( )** function in ProvideX.

The **RNO=** file access option requires that the system is able to position itself to a record specified by its logical sequence within a file/table. Unfortunately, there is no equivalent functionality within most external databases.

The standard ProvideX implementation of access by **RNO** involves **SELECT**s of the complete data set, then skipping forward to the specified record. This is a slow process that can have a major impact on the system performance. If possible, consider re-engineering code using **RNO**.

In order to improve the performance when running against an external database, use the **EXEC_SPRNO** option on the database **OPEN** and execute a stored procedure to duplicate this functionality. The **RNO( )** search is performed on the host, which minimizes data transfer and provides much faster access times. The disadvantage to this approach is that database-specific knowledge is required to create the stored procedure. The following stored procedure has been used with MS SQL Server:

```
1st field:
     [odb]MAS200SQL;APB_CheckHistory;DB=MAS_PVX;
2nd field:
   KEY=BankCode,CheckNumber,SeqNumber;KEY=Division,VendorNumbe
   r,BankCode,CheckNumber,SeqNumber;REC=BankCode:1+CheckNumber
   :6+SeqNumber:1+ CheckType:1+CheckDate:D+ ClearedBank:1+
   Division:2+VendorNumber:7+Source:2+PayeeName:30,CheckAmount
   :19.7;EXEC_SPRNO
```

You must create a stored procedure for each table and key that an **RNO** is need for. *Format:* **spRNO***tablename_keynumber.*

The **RNO** stored procedure (written for MS SQL Server):

```
CREATE PROCEDURE [spRNOAPB_CheckHistory_0]
(@rno   int)
AS BEGIN
SET NOCOUNT ON
DECLARE @BankCode VARCHAR (1)
DECLARE @CheckNumber VARCHAR (6)
DECLARE @SeqNumber VARCHAR (1)
DECLARE tmpCur CURSOR DYNAMIC FOR
SELECT BankCode,CheckNumber,SeqNumber FROM APB_CheckHistory ORDER BY
      BankCode,CheckNumber,SeqNumber
OPEN tmpCur

Fetch Relative @rno from tmpCur INTO @BankCode,@CheckNumber,@SeqNumber

CLOSE tmpCur
DEALLOCATE tmpCur
SELECT * from APB_CheckHistory WHERE BankCode = @BankCode AND CheckNumber
      = @CheckNumber AND SeqNumber = @SeqNumber
END
```

The above stored procedure creates a cursor that represents the keys of all of the records in the table sorted in the order as defined by the key. Then, we issue a relative fetch of the desired record to obtain the key for the record, and then use this key to actually read and return the record.

Using the TSQL option

Many applications read data files sequentially looking for selected records, such as un-printed invoices, past due accounts, accounts with current transactions, etc. Application execution time can be improved by having SQL server pre-process the data and return only those records that the application requires.

In order to accomplish this task, the **TSQL=** option was added for external databases. The TSQL option allows a manually defined **SQL SELECT** directive that is going to be used to retrieve the file data for the application.

When using a TSQL directive, simple **READ NEXT** statements (or **KEY( )** functions) will execute the SQL command provided in order to obtain the data. The SQL statement must be constructed manually – ProvideX won't do it.

Use of the TSQL function requires application changes, as well as a fair amount of SQL expertise. It is not an easy change to implement, but the results can be very impressive in the areas implemented.

The following example shows a simplified use of the **TSQL=** option. Note the use of the **REC=** clause. Failure to correctly define the resulting record can cause memory corruption. The SQL statement in the example returns only two columns of a larger table where the employee ID is less than 7600, sorted by name descending.

```
0030 OPEN (1)"[oci]robertf.pvx;emp;user=scott;pswd=demo;tsql=select
      empno, ename from emp where empno < 7600 order by ename
      desc;rec=Empno:N22,Ename:C20"
0051 READ RECORD (1,END=*NEXT)r$; PRINT CVS(CVS(SUB(r$,SEP," *
      "),16),32);
    INPUT 'CI',*; IF CTL<>4 THEN GOTO *SAME
```

Prepared Statements

Basically, an SQL *prepared statement* is a parameterized SQL statement that can be pre-compiled, then re-executed repeatedly while simply changing the parameter values. A typical use of this would be to access a file by a key where the SQL statement rarely changes other than in the key value itself. The advantage to this approach is that the database server does not need to re-compile the statement on every execution. This technique may improve performance significantly; e.g.,

```
SELECT BankCode, CheckNumber, SeqNumber, CheckType,
CheckDate, ClearedBank, Division, VendorNumber, Source,
Source, PayeeName, CheckAmount
FROM "APB_CheckHistory"
WHERE BankCode = ? AND CheckNumber = ? AND SeqNumber = ?
```

ProvideX will generate the above SQL statement and pre-compile it. Then, when accessing the file by key, it simply re-executes the statement after changing parameters 1, 2 and 3.

Multiple Record Types

Non-normalized files are a critical aspect of a number of applications. These are files that use multiple record formats to contain the data used by the application. Applications developed by many Business Basic developers have a number of these types of files. Unfortunately, this type of data structure is not supported in a normalized database, such as IBM DB2, or MS SQL Server.

In order to migrate these files to a SQL-based database, the table contents have to be normalized. The ProvideX database interfaces have a mechanism built into it to convert data from a SQL normalized file to appear as a non-normalized file. The approach used is to define one column for each potential field which can exist in all the different record layouts within the table that contains the ProvideX data file contents. Then, we define to ProvideX the record layout to be used based upon the contents of one or more fields.

In order to accomplish this, ProvideX must know which fields/columns contain the data needed to determine the record format to be used. The column(s) are declared in the **TYP=** option. If multiple columns are required, each column must be separated by a plus sign as in:

```
TYP=CST_ID+CST_TYPE
```

This requires the definition of values to be returned by the contents of the fields defined in the **TYP=** clause within the **REC=** clause. These values are identified by question marks followed by the contents to match against the fields.

The contents of the value to match can contain a number of special characters that are designed for simplified matching:

.	Period - matches any character.
[abc]	Matches any of the characters a, b, or c.
[0-9]	Matches any of the characters between 0 and 9.
[ ]	Matches end-of-string / no data.
^	If the first character is a '^', then records that don't match the string are selected; e.g., `'?^PROD'` selects anything that doesn't match PROD.

The column names specified in the **TYP=** clause can be followed by a comma and a length specifier, if you only want the first *nnn* columns used in the match. For example, the clause `TYP=CST_ZIP,3` would only check the first three positions of the field CST_ZIP.

The following is a typical example of a non-normalized file, where each record has a three-character prefix on the key field definition:

File: GCOMP (two record types "G/L" and "DPT")

All record:	Prefix Key	3 char	
	Company Key	2 char	
	Id Key	10 char	(Dept or Acct #)
	Name	35 char	(Dept or G/L name)
"DPT" rec:	Restriction_flag	1 char	
"G/L" rec:	Control_acct	1 char	
	Stmt_A_Line	4 numeric	
	Stmt_B_Line	4 numeric	
	Subaccount_flag	1 char	
	Inventory_acct	1 char	
	Print_zero	1 char	

Definition:

```
[db2]MyDB;GL_Dept_master;
```

```
KEY=Prefix,Company,Id; TYP=Prefix;
REC=?"DPT",Prefix,Company,Id,Name,Restriction_flag,?"G/L",
Prefix,Company,Id,Name,Control_acct,Stmt_A_Line,Stmt_B_Line,
Subaccount_flag,Inventory_acct,Print_zero
```

Other Considerations

In a perfect world, the application works flawlessly with the new database. The reality is that some things may not work correctly, and the application will need to be adjusted.

Views

Views programs may fail when trying to access index information that may not have been included in the conversion.

Complex Variant Records

Occasionally the rules required to identify the correct variant record are so complex, the syntax does not support the record definitions. These types of files require the use of the **RECDATA=** option. **RECDATA=** specifies the column name that represents the data. The data type of this column is typically Binary, as no processing of any sort is performed upon it. No attempt is made to parse the data or apply indices. This option requires the use of **KEYDATA=**. Alternate keys are not supported.

File Information

Programs that depend on **FIB( )** or **FIN( )** will typically fail as not all the information about the file exists.

ProvideX ODBC

The ProvideX ODBC driver enables any ODBC-compliant application on any Windows platform to communicate with your ProvideX database from any location, including over a network. For more information, see **About ODBC** below. For complete information on this subject, refer to the ProvideX ***ODBC Reference*** documentation.

Currently, two ProvideX ODBC configurations are available for download:

- **Local ODBC**, *p.341*
- **ODBC Client-Server**, *p.342*

ProvideX ODBC installations are available with or without Microsoft Data Access Components. If you choose not to install MDAC, the installation automatically verifies if your current version of MDAC (if any) is compatible with ProvideX ODBC.

These products are available separately from the base ProvideX installation and require separate licenses, installation files, and activation procedures.

About ODBC

ODBC is the acronym for ***O****pen* ***D****ata****B****ase* ***C****onnectivity*, an interface standard that maintains a common access method for DBMS (***D****ata****B****ase* ***M****anagement* ***S****ystems*). The ODBC interface provides a standard set of functions or APIs (***A****pplication* ***P****rogram* ***I****nterfaces*) that allow applications to access a variety of ODBC-compliant databases. It also administers the database names and drivers associated with the data files. ODBC access is based on **SQL**, *p.314*.

ODBC Architecture

Typically, the standard ODBC architecture consists of four major components:

Application	Responsible for interacting with the user and for calling ODBC functions to submit SQL statements to, and retrieve results from, one or more data sources.
Driver	Processes the ODBC function calls, submits SQL requests to a specific data source, and returns results to applications. Also, the driver is responsible for interacting with the software needed to access a specific data source.
Driver Manager	Loads / calls drivers on behalf of an application. The driver manager processes ODBC function calls or passes them to the driver.
Data Source	Represents the data to be accessed. It can be a flat-file, or a particular database in a DBMS. It also refers to the actual location of the data as well as any technical information needed to access the data (driver name, network address, user ID, password, etc.)

This architecture enables an application to access different ODBC data sources, in different locations, using the same function calls available in the ODBC API. Components interact in the following chain of events:

1. ODBC-compliant application uses API calls to submit SQL directives to the data source.
2. Communication between the application and ODBC driver is handled by the driver manager, which loads the driver and passes along the API requests.
3. The ODBC driver implements ODBC API functions for the selected DBMS data source.
4. Requests are processed by the data source, and the results are sent back up the chain to be retrieved by the application.

Implementation

The ProvideX ODBC driver itself is easy to implement. It installs automatically from the setup program. Open the Microsoft ODBC Administrator's control panel applet and create a new ProvideX Data Source Name (DSN). Once a DSN is established, other applications will be able to use SQL requests to access the ProvideX native database. The driver also supports a "DSN-less" connection to ProvideX data using a connection string supplied by the calling application.

ODBC allows your ProvideX data to be accessed by the most popular database managers, query applications, and report writers: MS SQL Server, Excel or MS Word with MSQUERY, Crystal Reports, just to name a few. Most programming languages have an ODBC access facility to allow files to be read or updated as well.

ODBC and SQL allows standardized access to ProvideX data via:

- Standardized data formats (text strings, numerics, dates).
- Logical relationships (relates files with common data elements).
- Data sorting, grouping and filtering.
- Simple Data computations (*Sum*, *Max*, *Min*, *Count*, *Avg*).

The ProvideX ODBC driver supports three basic types of data: strings, numerics, and dates. The `SELECT` statement is used to establish logical relationships between data files (usually referred to as *joining* files). A typical *join* would be:

```
SELECT cst_id, cst_name, smn_name FROM Customer, Salesman
WHERE smn_id = cst_smn
```

The statement reads the entire `Customer` file and for each customer, reads the `Salesman` file for any records whose *smn_id* matches *cst_smn*. If the field *smn_id* is a Key field for the file, then the ProvideX ODBC driver reads the file directly by key, otherwise the file is read in its entirety. The `WHERE` clause can be used to selectively filter out any unwanted data.

The ODBC driver can sort the data on any field using the `ORDER BY` clause of the `SELECT` statement. If the `ORDER BY` fields match any of the key fields of the primary file, then the primary file is accessed by this key. In addition, you can `GROUP` data `BY` common fields.

`SUM`, `COUNT`, `AVG`, `MAX`, `MIN` functions can be used to provide statistical information on the data fields.

The complete list of supported SQL keywords can be found in the ProvideX ***ODBC Reference*** documentation.

ODBC Client-Server

For greater performance and security over the network (without the need for additional software) consider the client-server version of the ProvideX ODBC driver. This interface performs optimization of query processing on the server side to ensure safe high-speed access to your data, particularly for implementing distributed multi-user applications.

The client side is freely distributable. However, to be operational, it must be connected to a fully installed and activated ProvideX File Server (included with the ***Professional*** and ***eCommerce*** product bundles). For further details, refer to the ProvideX *ODBC Reference* documentation.

PVKIO - ProvideX I/O Library

The ProvideX IO Library (PVKIO) consists of a set of functions that can be used with programs written in C, C++, and other programming languages. These functions enable direct access to ProvideX keyed and indexed data files from applications that are external to ProvideX and have been pre-compiled for use in MS Windows, AIX, RedHat, and SCO UNIX.

Note: PVKIO does not support EFF files, **Enhanced File Format**, *p.106*.

The ProvideX IO Library includes functions for performing a variety actions; i.e.,

- Allocate / de-allocate environment
- Get / Set environment variables
- Extend file open
- Close file
- Read a record from a file
- Position within keyed / indexed file
- Write / rewrite a record
- Write a new record
- Update an existing record
- Remove a record
- Get / Set address / position within file
- Return last error status, last error message
- Read dictionary
- Point to internal structure block
- Describe file
- Get tables in catalog.

For a list of all the available functions, and for syntax details, refer to the ProvideX *PVKIO* documentation.

Use and distribution of PVKIO requires a separately-purchased activation key apart from your initial ProvideX activation. A warning message to this effect is presented whenever a file is opened unless the application first invokes the **PVK_register( )** function with a valid registration string and registration number. Contact your local ProvideX dealer / distributor or visit www.pvx.com for complete product information and licensing.

XML Content

The ProvideX ***XML** Interface can be used to access, parse and create documents based on XML DOM (Document Object Model). It provides a standardized method for sharing data, making it easier to store, transmit, and display information across different applications and platforms.

This product may require a separately-purchased activation key apart from your initial ProvideX activation. Contact your local ProvideX dealer / distributor or visit www.pvx.com for complete product information and licensing.

About XML

XML (Extensible Markup Language) was developed by the World Wide Web Consortium (W3C). It is a simplified version of SGML (Standard Generalized Markup Language) that allows designers to create their own customized tags, and enables structured definition, transmission, validation, and interpretation of data.

Like HTML (Hypertext Markup Language) XML was created to specifically address the issue of writing documents for the Web. However, the two languages are really intended for two different purposes. HTML focuses on the presentation of information to users, while XML deals with the data itself. HTML is able to change the look of a web page, while XML does nothing but describe the data and must be used in conjunction with other languages.

While it was originally designed for Web publishing, XML has a variety of other uses as a portable document structure. It is now widely used in content management systems, eCommerce transactions, direct-to-consumer implementations, bibliographic referencing, mobile / handheld devices, and many other commercial applications.

Further information on the advantages and capabilities of XML can be found on the Internet, in public forums and various industry publications.

ProvideX XML Interface

The XML Interface can be used to parse and serialize XML documents for use with your ProvideX applications. This facility is built on the Xerces XML Parser, which is an open-source C++ library for parsing, generating, manipulating, and validating XML documents based on the W3C DOM specification.

DOM (Document Object Model) is the language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of XML documents. It is used to define the objects and properties of all document elements, and the methods (interface) for accessing them.

Implementation of the XML Interface with your applications requires that the Xerces XML Library as well as libraries `pvxxml.dll` (for Windows) and `pvxxml.so` (on UNIX/Linux) are installed on your system. **TCB(193)** can be used to determine the availability of XML support.

Defining the XML Interface

The **DEF OBJECT** directive is used to create a new instance of ***XML**; e.g.,

DEF OBJECT *obj_id*,**"*XML"**

The **DELETE OBJECT** is used to remove/disconnect the object:

DELETE OBJECT *obj_id*

Syntax Options

Once defined, the interface supports various options for accessing and manipulating XML documents and content.

'CREATE	Creates and opens an XML document with a given file name and document root.
'OPEN	Opens an XML document.
'SET_ELEMENT	Sets the current element for reading and writing.
'READ_ELEMENT$	Sets the current element for reading and writing.
'NEXT_SIBLING	Sets the current element to the next sibling.
'PREVIOUS_SIBLING	Sets the current element to the previous sibling.
'READ_CHILDELEMENT$	Reads the value of the child element.
'BUILD	Initializes output buffer for a new data block with the specified parent tag as a child element of the current element.
'ADD_CHILDELEMENT	Add a new child element to the end of current element block set in the output buffer.
'ADD_ATTRUBUTE	Adds an attribute to the latest node, either created by **'BUILD** or **'ADD_ CHILDELEMENT**.
'COMMIT	Commits the output buffer to the XML document file/string.
'CLOSE	Closes the XML document, cleans-up workspace and releases any resources that are used.

For syntax details and examples, refer to the ***XML** interface in the *Language Reference, p.760*. See also **DEF OBJECT** in the *Language Reference, p.70*.

11
Object-Oriented ProvideX

Object-Oriented Programming (OOP) is a development approach where applications are composed entirely of reusable components that can act on each other. Unlike in traditional structured programming, where functionality and information is kept apart, object orientation merges both into a single indivisible entity called an *object*. This concept provides greater flexibility and easier maintenance across large systems and can make understanding and analyzing complex procedures much easier, especially in a collaborative development environment.

ProvideX is equipped with all the language apparatus for designing, developing, and implementing true object-oriented programs. Several ProvideX-supplied classes are shipped with the ProvideX installation for use in your applications (each is documented on the ProvideX website, www.pvx.com). The **OLE Server** can be used to invoke ProvideX objects from external products. This chapter leads you through the object-oriented programming elements in ProvideX.

Topics

Why use Object Oriented Programming?

There are a number of excellent reasons for going *object oriented*. Objects make large development projects simpler by breaking them down into smaller manageable chunks. They may be re-used in other applications, which saves development time. They can hide and protect critical data. They enforce modularity, which improves code maintainability and promotes a true collaborative development environment.

Consistency in Design and Code

In object orientation, each object is treated like a "black box". Developers don't need to know about what goes on inside an object to be able to use it. All they need is access to the object via its object ID, which provides access to what the object does but keeps the details hidden from outside. Once created, the object behaviour is ***never modified***. This is what maintains ***consistency*** throughout a project. Developers should be confident that the object will always work as originally designed.

For example, if there is a `Delete` method for all objects that relate to data information throughout the application design (`Client`, `Vendor`, `Product`, etc.), then the programmer only ever has to remember to use `Delete` to remove these items. The object itself determines if all conditions are met to allow for the removal.

Also, a common property (e.g., `InActive`) can be provided for consistent testing of the different conditions of a `Client` versus a `Vendor` versus a `Product`. This allows programmers to write generic object-related routines such as:

```
Function PurgeHistory( ObjId )
   ObjId'Start()
   while ObjId'GetNext()
      if ObjId'Inactive then ObjId'Delete()
   wend
return 1
```

This code would work with virtually all objects to purge inactive information from the system — provided the object has `Start`, `GetNext`, and `Delete` functions and an `InActive` property.

Data Protection

Certain data elements can be created that will be visible only while running within the object. These are considered local or static elements — they are particularly useful for maintaining critical information (internal reference pointers, etc.) that is controlled solely by the object and is not directly accessible to outside code.

An example of this would be a field in a file that contains a linked list pointer. To avoid the issue of programs accidentally corrupting the pointer, you could make this data internal to the object only and declare it as local or static.

You may want some data elements (properties) to be accessible from outside the object yet be able to control all access to them. Access to properties can be controlled within the object. The object definition and logic decides which properties the user can read and update. For example, a field that displays *Salary* could be restricted to only those users who rightfully have access to the data.

This capability can also be used to make sure that data content is correct. For example, a *Date* field could have update logic applied to it so that only valid dates are placed into a file.

Control and Separation of Code

Another underlying advantage that objects give us is the ability to re-use common application code. Object orientation provides this advantage in two ways:

First, common object-related functionality can be developed — methods that take advantage of the fact that an object is self-contained and need not concern themselves with the object's particulars. For example, the `PurgeHistory` method can be used on any object, provided the object has `Start`, `GetNext`, `InActive`, and `Delete` properties / methods. This allows the code to be used in multiple places within the application, reduces duplication, and minimizes the chance of introducing errors.

Second, *inheritance* can be applied — objects that utilize code and concepts from other objects. This further reduces code duplication, reduces errors, and makes for a smaller and tighter application.

Objects themselves can be used as function libraries, which enables code to be written once and used in numerous places within the system.

Access from Outside the Application

If you design objects that are fundamentally self-contained, they can also be used by outside applications (via the ProvideX **OLE Server**). It is possible to create a ProvideX object, then invoke it from other languages such as VB, VBScript, Delphi, and C++. The ProvideX OLE Server allows virtually any object to be invoked by, and directly interact with, outside application development environments. You can use **DCOM** to invoke ProvideX objects remotely.

General Concepts and Terminology

To understand OOP is to understand its vocabulary. This section provides a quick reference of all the standard object-oriented programming concepts and terminology.

Topics

Objects

OOP systems are modeled after real world things, or *objects*. In object-oriented programming, objects are entities that consist of data, and the functionality that operates on that data.

Analogy: A car is represented in a computer system as a `Car` object.

Classes

A *class* defines an object. Each object belongs to only one class. Similar objects are grouped by class. Because an object represents an *instance* of a class, the action of creating the object is often called *instantiating*.

Analogy: A `BlueFord` has properties and methods defined in the `Car` class of objects. The `BlueFord` object is an instance of `Car` created at runtime.

Distinguishing Between an Object and a Class

It is essential to understand the difference between an object and a class. An object is a unique instance of a particular type, whereas the class is that type; e.g.,

Object	Class
Bob's Bank Account	**Bank Account**
MIT	**School**
Out of control car	**Car**
Bank of Nova Scotia	**Bank**

Question: Can any of the above objects be a class?
Tip: Try adding an "S" to the objects. Content is important.

Object Identifiers

There can be many instances of a class running in the system; therefore, ProvideX uses a numeric value to identify each object. An *object identifier* allows us to send a message to a particular object. Applications cannot directly access any of the data, but must go through the object identifier.

Analogy: `BlueFord` is represented by a numeric variable that points to an instance of a `Car` object.

Properties

Properties are the data held inside an object. The same properties will appear in every object of a particular class, but the value of each property may be different.

Analogy: The object identified as `BlueFord` has a property called `Fuel_Level`. Every instance of `Car` will have a `Fuel_Level` property, but the value of `Fuel_Level` may be unique to each instance.

Methods

Methods are procedures held within an object. They define what actions each object is able to perform.

Analogy: The `Car` class has methods such as `getFuelLevel` or `fillTank` that affect the state of the `Fuel_Level` property. It also has a method called `Start` that contains all of the logic responsible for starting the engine.

Note: A class should only have properties / methods related to the intended objects; e.g., a `Car` class would not have a `Rudder` property or `DropAnchor` method.

Encapsulation

The properties of an object are not addressable from outside of the object. Only the object's methods should be able to change its properties. The enforcement of this rule is known as ***encapsulation***. Encapsulation means a message can only access an object's properties via the object's methods; the object's methods will validate all incoming messages.

Warning: Do not break this rule.

Inheritance

This term describes how one class inherits elements from another class.

Analogy: Both the `Car` class and the `MotorCycle` class share common elements (e.g., `GasTank`) which reside in a more general class called `Vehicle`. The specific structure and behavior defined in a `Car` and `MotorCycle` are based on the elements of a `Vehicle`.

`Car` and `MotorCycle` are called *derived* classes (a.k.a sub-classes or children). The `Vehicle` class is referred to as a *base* class (a.k.a super class or parent). Each derived class has properties and associations of its parent. Both `Car` and `MotorCycle` are a "kind-of" `Vehicle`. The "kind-of" relationship is important. It means that, with inheritance, restraint is placed on the *type* the object is an instance of.

Substitutability Principle

A derived class must be usable through the methods declared in the base class. At anytime the derived class can be substituted for the base class. For example, if a block of code expects to receive a reference to a `Vehicle`, and instead it receives a reference to a `Car`, because `Car` is based on the elements of a `Vehicle`, the logic will still execute as expected.

Tip: Delegation is an alternative to inheritance. Misuse of inheritance can reduce reusability and complicate maintenance.

Aggregation

An *aggregate* object is an object comprising several other objects, to which it delegates responsibilities. An aggregate forwards messages to properties that are objects known as *delegates*. Client objects sending messages to the aggregate are unaware that multiple objects are working behind the scenes. An aggregate delegates responsibilities like inheritance but without the "kind-of" restraints of inheritance.

Collections

A collection is an object that groups multiple objects into a single unit. Collections are used to store, retrieve, and transmit objects from one method to another. They don't manipulate the objects, but simply store and retrieve their object identifiers. Collections typically hold objects that form an expected group.

Analogy: A `DealerShip` object has a collection of `Car` objects, whereas a `ParkingLot` has a collection of `Vehicle` objects.

ProvideX collection objects are `*OBJ/COLLECTION` and `*OBJ/HASHCOLLECTION`.

Polymorphism

Polymorphism is based on a Greek word meaning "many forms". In object-oriented modeling, it refers to the ability of objects to respond to a particular message in a manner appropriate to the object's class. Polymorphism is common among classes that are derived from a common base class.

Analogy: All objects that are derived from the `Vehicle` class have a method called `Start`. If a `ParkingLot` object were to cycle through its collection of `Vehicle` objects to call each `Start` method, the `Car` objects will execute logic specific to cars, and `Motorcycle` objects will execute logic specific to motorcycles, and so on. Although the same message was sent to all `Vehicle` objects, each object responds with a different `Start` method.

ProvideX OOP Interface

Specific OOP-related directives and functions have been added to the language for the definition, creation, and deletion of classes and objects. The ProvideX environment takes care of all of the housekeeping chores required for their implementation.

Applications only have to request that a new instance of an object be created and ProvideX will locate and load its definition. When the object is no longer needed, the object and all its properties are released. Class definitions are handled similarly. Unless specifically defined, ProvideX will dynamically load the definition of an object class and maintain it in memory until there are no references to it (either by an object or another class inheriting it).

ProvideX enlists the use of several OOP-related syntax elements.

Topics

Overview

Following is a brief discussion of ProvideX OOP mechanisms. Specific syntax is provided later. For a collection of sample objects (illustrating the various mechanisms available) refer to the section **Putting It All Together**, *p.364*.

Classes and Objects

Object-oriented programming in ProvideX begins with the definition of *classes*. Each class provides the name given to an object definition (such as **Company**, **Customer**, or **Supplier**) as well as access to the object's properties and methods. Classes are saved in the form of a *class*.**pvc** program file and can include:

- Property definitions (both hidden and exposed)
- Method definitions
- References to other objects whose characteristics are to be inherited
- References to programs and related logic that supports the object.

A class definition begins with a **DEF CLASS** directive followed immediately by the object description (comprising various ProvideX OOP-related directives). An **END DEF** directive is used to mark the end of the definition. The **DEF CLASS** statement must appear at the beginning of the .**pvc** program file.

Several OOP-related directives are available for use in a ***class definition construct***:

```
0010 DEF CLASS "class$" ...
0020 PROPERTY prop1, prop2, ...
0030 LOCAL prop1, prop2, ...
0040 FUNCTION method (param) "
0050 LIKE "otherclass", ...
0060 PROGRAM "interface_prog"
0070 PRECISION nnn FOR OBJECT
0080 END DEF
```

Creating/Accessing an Object

ProvideX has a built-in feature that simplifies the loading and unloading of classes. Once an object class has been defined, it can then be used to create objects via the **NEW( )** function. Whenever a class is referenced by **NEW( )**, and the class is not already defined in the system, the system will automatically load the class definition from the .**pvc** file. The **LOAD CLASS** directive can also be used to auto-load a class definition from a file. This definition remains in memory until no references (handles) to it exist - at which point the system automatically deletes the class definition. Definitions are also deleted using the **DROP CLASS** directive, or when a **START** directive is issued.

NEW( ) loads the class definition (if necessary), instantiates the object, and creates a logical handle to the object. Because the system automatically tracks the number of open handles to each class, an application can have multiple references to the same class; and, as long as they always have a corresponding **DROP OBJECT** for each **NEW( )**, the system will keep track of the loading and unloading of class definitions. The **REF( )** function can also be used to monitor and control access to objects by controlling the reference count. All objects are destroyed automatically when a **START** directive is issued.

Using Methods and Properties

The properties and methods defined in an object are accessible to a program via the apostrophe operator (**'** *tick*), using the format:

obj_id'*method*[$](*args*)
obj_id'*property*[$]

The syntax **'*** (*tick asterisk*) can be used to return a comma-separated list of all the methods and properties within an object.

When referencing a method, a **$** *dollar sign* means that the method will return a string – otherwise, a numeric value would be returned. Methods that end in **%** are assumed to return an integer. Methods usually return values, but sometimes they are used to perform logic that returns a simple indication of success or failure; e.g., **1**or **0**.

Properties are referenced via the apostrophe operator in the same manner as for methods. However, when executing ***within*** an object, all object properties become simple ProvideX variables (as far as the application logic is concerned). An object's properties and their values are preserved for as long as the object exists (similar to the way global variables are preserved throughout the life of a ProvideX session).

To reference properties and methods within the object that you are currently executing (e.g., `PayRate`) use the **_Obj** prefix (`_Obj'PayRate`). This prefix is a system-supplied variable containing the current object identifier.

During the execution of program logic, direct property manipulation will not invoke any **PROPERTY GET** / **SET** logic (e.g., `PayRate=n`); however, if you refer to a property using its object identifier (`_Obj'PayRate=n`), the system will invoke the **GET** / **SET** logic.

Note: A significant number of sample objects are provided in the section **Putting It All Together**, *p.364*.

OOP Syntax Elements

Following is a descriptive list of all the directives and functions used in the implementation of ProvideX object-oriented mechanisms:

DEF CLASS	Defines the object class.
PROPERTY	Declares data / properties for the object.
LOCAL	Declares internal data / properties for the object.
FUNCTION	Declares functions or methods for the object.
LIKE	Specifies other objects that this object inherits from.
PROGRAM	Defines the default program that contains the object logic.
PRECISION	Sets default program precision for use within the object.
LOAD CLASS	Pre-loads a class definition into memory from a .**pvc** file.
DROP CLASS	Deletes class definition and all related information.
RENAME CLASS	Change name of an existing class.
STATIC	Dynamically declares **LOCAL** variables.
NEW( )	Creates an object instance.
REF( )	Controls reference counts.
DROP OBJECT	Deletes an object.
OPEN OBJECT	Opens a file for exclusive use in an object.
END DEF	Marks conclusion of object class definition.

System-supplied variables available while executing code on behalf of an object:

_Obj	Identifies internal object identifier of current object
_Class$	Contains the name of the class for current object
_Refcnt	Contains the reference count for this class of object.

DEF CLASS Directive

DEF CLASS *class$* [**UNIQUE**][**CREATE** *label* [**REQUIRED**]][**DELETE** *label*[**REQUIRED**]]

The ProvideX **DEF CLASS** directive is used to declare the start of a **Class Definition Construct**, *p.354*. It provides the name of the object (*class$*) and it can be used to override object creation and deletion logic. Class names are case insensitive and forward / backward slashes are considered equivalent. Duplicate names are not allowed within the system. A definition closes with the **END DEF** directive.

On Create/On Delete Logic

Sometimes it may be necessary to perform initialization (or cleanup) routines when the object is created or destroyed. Initialization logic can be used to establish default values for properties, open files that the object may need, validate security, etc. Cleanup logic should be used to close files and release system resources (such as other objects) whenever an object is dropped.

By default, ProvideX executes a call to the label `ON_CREATE` within an object definition whenever an object is created. This logic can open files, initialize variables, and process additional parameters passed in the **NEW( )** function call. When an object is deleted from the system, ProvideX executes a call to the label `ON_DELETE`.

Default label names can be overridden via the **DEF CLASS** options **CREATE** *label* and **DELETE** *label*. If there is no **CREATE** or **DELETE** declaration, ProvideX executes the default `ON_CREATE` / `ON_DELETE` in the primary class definition file only. The creation / delete entries of inherited classes are not executed unless the **CREATE/DELETE** options specify the **REQUIRED** clause. If an object inherits another object that has creation logic, the creation logic for the inherited object is performed first. Deletion logic is performed in the opposite order.

Note: To avoid syntax / logic errors, it is recommended that your create / on delete logic be placed *outside* of the **DEF CLASS .. END DEF** block.

UNIQUE *Option*

For many applications it may be desirable to limit an object to a single instance in memory. For example, an object that is essentially a subroutine library — only one instance of this library is necessary.

The **UNIQUE** option for the **DEF CLASS** directive forces a single instance of an object in memory. Any object declared as **UNIQUE** will have a single instance created, and any subsequent attempt to create an instance will simply return the same object identifier and increment the object reference count by one; e.g.,

```
DEF CLASS "WindowsAPI" UNIQUE ON_CREATE REQUIRED ,,,
```

This allows programs to logically create the object then drop the object when done. The **UNIQUE** clause guarantees only a single instance of the object will ever exist.

Another use of a unique object could be a session or application control object that could have information such as the company codes, user information, operating date, etc. By declaring this object as unique, only one instance will ever be created and any application can access it.

PROPERTY Directive

PROPERTY *prop1* [**OBJECT**] [**GET** *label*|**ERR**] [**SET** *label*|**ERR**]**,** *prop2 ..*

The **PROPERTY** directive is used within a **DEF CLASS .. END DEF** block to declare an object's properties (*data*). These properties can be treated like any other variable and are accessible using the **Apostrophe** operator. To define properties for an object class that are not to be exposed to external applications, refer to the **LOCAL** directive.

If the property contains an object identifier to another object, specify the keyword **OBJECT** after the property *name*. When the object is deleted, ProvideX will use the **REF( )** function against the object identifier to remove it (as long as it has no other references); e.g.,

```
DEF CLASS "Customer"
PROPERTY File OBJECT
```

When you delete an object whose class is `Customer`, then the system reduces the reference count of the object whose identifier is in `File` and, if it is no longer being referenced, deletes it as well.

You can also specify logic to be called automatically whenever a property is read or written. This capability allows the application designer to control how the underlying application will view and update any of the object properties.

On Read Logic

A property name *prop* can be followed by a **GET** *label* to define the location of the logic to call whenever the property is read in the application. With a **GET** in place, the specified logic issues a **RETURN** value which returns the actual value of the property to the application; e.g.,

```
PROPERTY ExtendedAmount GET Extension
...
Extension:
RETURN Quantity * Price
```

If the logic resides outside of the defining program, then the name of the program and entry point is provided in quotes instead. If the logic required to do a read is simply a formula, then it can be inserted directly into the **PROPERTY** definition clause using an ***equal sign*** instead; e.g,

```
PROPERTY ExtendedAmount = Quantity * Price
```

On Write Logic

To intercept all of the property updates, you can specify **SET** label. With a **SET** in place, the system calls the logic whenever the property is being updated and passes it the value being set. Like the **GET** option, an external program / entry point can be provided; e.g.,

```
PROPERTY Quantity SET "Invline;ChgQty"
```

Where `Invline` contains:

```
2000 ChgQty:\
2010 ENTER NewQty
2020 IF NewQty = Quantity then RETURN
2030 ! .. Update inventory...then RETURN
```

If the logic is within the defining program, then you would simply specify the statement label; e.g.,

```
PROPERTY Quantity SET ChgQty
```

Blocking **GET** / **SET**

To prevent a user from being able to **GET** or **SET** a property, specify the keyword **ERR** following the **GET/SET** declaration; e.g.,

```
PROPERTY ExtendedAmount GET Extension SET ERR
```

The **SET ERR** after this property makes it read-only. An error exit within the **GET/SET** causes an error to be returned to the application.

LOCAL Directive

LOCAL *prop1* [**OBJECT**]**,** *prop2* [**OBJECT**] **...**

To create a *private* property, use the **LOCAL** directive within the class definition instead of the **PROPERTY** directive. Properties that are declared **LOCAL** are not visible to applications outside of the execution of the object itself, nor are they available to be read or updated.

Other typical uses of **LOCAL** properties would include values such as file numbers, status flags, handles to subordinate object libraries, confidential information (such as security codes / passwords), as well as other types of information that you do not want applications outside of the object to access.

FUNCTION Directive

FUNCTION [**PERFORM**] [**LOCAL**] *method***(***args***)** *logic* [**FUNCTION END**]

A method (*function*) is declared via the **FUNCTION** directive within the **DEF CLASS .. END DEF** block. Every *method* declaration needs to have associated *logic* that will be called when it is invoked. If arguments (*args*) are used, then the type and number must match parameters that the application code provides. Parentheses are part of the method name (whether or not arguments are used).

PERFORM indicates that the *logic* is to be loaded and executed (as in a **PERFORM** directive); therefore, all variables will be shared with the calling program. While this does have some uses, it clearly violates the rules of **Encapsulation**, *p.351*. **LOCAL** indicates that the method is only to be called internally from within the object. It cannot be called externally.

Statement Label or In-Line Logic

A statement label is normally used to access *logic* defined as a procedure outside the **DEF CLASS .. END DEF** block; e.g.,

```
0060 FUNCTION Find(X$) LookupCust
...
0120 END DEF
...
0210 LookupCust:
0220 ENTER Cst_id$
...                 ! Logic to find the client
0240 RETURN sts     ! Return value indicates success
```

However, the *logic* can also appear directly following the **FUNCTION** directive; e.g.,

```
0060 FUNCTION Find(X$)
0070 ENTER Cst_id$
...                 ! Logic to find the client
0090 RETURN sts     ! Return value indicates success
0100 FUNCTION END
...
0170 END DEF
```

In which case, the method declaration should close with a **FUNCTION END** clause. However, the logic ends automatically with the start of the next method declaration or **END DEF** (whichever comes first).

Return Value

Each method should return a value. The value can take the form of a string or numeric value depending on the name associated to the method (strings must end with **$**). If no **RETURN** value is specified, then the system will return a value of zero for numeric methods and " " (*null*) for string methods.

Naming Conventions

Multiple definitions of the same method name can be specified, as long as each method has different *args*. In order to determine which method to actually use, ProvideX attempts to match up the arguments specified with the lists provided in the application; e.g.,

In the class definition:

```
FUNCTION Find(X$) LookupByName
FUNCTION Find(X) LookupByNumber
```

```
... ... ...
LookupByName:
ENTER Cst_id$
... ... ... ! Logic to find the client by name
RETURN ...
LookupByNumber:
ENTER Cst_id
... ... ... ! Logic to find the client by number
RETURN ...
```

In the application:

```
Cst'Find("ABCD")     This calls LookupByName
Cst'Find(1234)       This calls LookupByNumber
```

LIKE Directive

LIKE *"otherclass", "otherclass", ...*

Multiple classes can be included in the definition of a single class. This can be accomplished using the **LIKE** directive within a **DEF CLASS .. END DEF** block to inherit the properties from one or more other classes.

For example, you might want to create a generic class that includes the contents of a number of different independent smaller classes (one for security, one to handle dates, etc.). Rather than merge the definitions, **LIKE** allows you to keep them separate. Not only does this simplify maintenance, but the independent classes may then be shared with other applications.

Example:

```
DEF CLASS "MyAppl"
LIKE ""DateUtil","Security"
END DEF
```

When a handle to `MyAppl` is created, it can be used to access all the methods contained within `DateUtil` and `Security`.

When multiple occurrences of the same property / method are found within the inheritance, the first class declared in the **LIKE** directive takes precedence. This can be overridden by specifying the class using a **FROM** clause in the method arguments; e.g., `x'AddItem(FROM "myClass",Item$)`.

PROGRAM Directive

PROGRAM *"interface_prog"*

The **PROGRAM** directive is used within a **DEF CLASS .. END DEF** block to define the default program name that is going to service a class of object. Entry points can process methods and read / write properties.

The following optional entry point labels are supported:

ON_CREATE called when the object is created.

ON_DELETE called when the object is deleted.

No error will be reported if the label does not exist. Any references to program logic in a property read / write or a method definition can contain a leading semi-colon. For example, the following class definitions are effectively the same:

```
PROGRAM "Cust"
FUNCTION Find(X$) ";LookupByName"
or
FUNCTION Find(X$) "Cust;LookupByName"
```

PRECISION Directive

PRECISION *num* **FOR OBJECT**

By default, an object will inherit the default system precision (normally 2, but based on the **'PD'** system parameter). By indicating **PRECISION** *num* **FOR OBJECT** within a **DEF CLASS .. END DEF** block, you can set pre-defined precision for all subsequent method invocations within the current object instance. However, an object's precision may be overridden within method logic where it specifically declares a **PRECISION** to use (preserving encapsulation).

LOAD CLASS Directive

LOAD CLASS *class$*

The **LOAD CLASS** directive may be used to pre-load a class definition into memory from a **.pvc** file; e.g., `LOAD CLASS "aaa"` loads from `aaa.pvc`. The first code in the **.pvc** file must be a **DEF CLASS .. END DEF** block assigned the same name (*class$*) as specified in the **LOAD CLASS** directive.

DROP CLASS Directive

DROP CLASS *class$*

Once a class definition is established, it may not be changed but it can be deleted using the **DROP CLASS** directive. This would allow the class to be redefined. Deletion is only possible if no objects exist for that class. Any attempt to delete a class that has a reference to it will return an `Error #50: "Class in use or already defined."` The **DELETE CLASS** and **DROP CLASS** directives may be used interchangeably for this purpose.

RENAME CLASS Directive

RENAME CLASS *old_name$* **TO** *new_name*

The **RENAME CLASS** directive is used to alter the name of a previously-defined class. This functionality allows the application designer to alter an existing object class easily, without having to change programs; e.g., `RENAME CLASS "xxx" TO "yyy"` would auto-load the new class name `yyy` from the file `xxx.pvc` (if the definition does not already exist).

STATIC Directive

STATIC *varlist*

The **STATIC** directive is used to declare that all the variables specified (*varlist*) are to be added to the list of variables maintained within the object — in effect, adding the variables to the **LOCAL** list at runtime. This allows the object to read a record from the file and have the data elements remain available for subsequent calls to the object's methods. All elements of an arrary are defined as static by specifying the simple array name in the statement; e.g., `STATIC A$` sets all elements of the A$ array as static. Static variables only take effect on references that follow the **STATIC** declaration.

NEW( ) Function

NEW(*class$*)

The **NEW( )** function is used to create a reference to, or *instance* of, an object based on a specified class name (*class$*). If the class name already exists, then its definition is used. If the class has not been defined previously, the system attempts to load the program *class$***.pvc** and execute / define the **DEF CLASS** within it.

For example, `Comp=NEW("Company")` would create a new instance of a `Company` class. If the class definition for `Company` does not already exist in memory, then the system attempts to load the program `Company.pvc`. If this is successful, the **NEW( )** function will return the object identifier assigned to the object.

If the label `ON_CREATE` exists in the class definition, it will be called to perform initialization logic. You can pass parameters to the **NEW( )** function following the class name as well. These parameters will be passed to the `ON_CREATE` entry point; e.g.,

```
C = NEW("Customer", File_number)
```

In `Customer.pvx`:

```
0010 ON_CREATE: ENTER File_no
```

For more information, see **On Create/On Delete Logic**, *p.356*.

Reference Count

Internally, each instance of an object sets a hidden reference count property, which is used for handling multiple access to the same object by different aspects of an application. Logically, when an object instance is first created, its reference count is

set to 1 (*one*). When an object is dropped, the system reduces the reference count by 1 (*one*). Once the count goes to 0 (*zero*), the object and all its contents are discarded. This reference count (number of instances) may be increased / decreased using the **REF( )** function (see below).

REF( ) Function

REF({**ADD**|**DROP**} *obj_id***)**

The **REF( )** function can be used to monitor and control multiple instances of a specified object via its internal **Reference Count**, *p.362*:

REF(*obj_id*)	Returns the current reference count for the specified *obj_id*.
REF(**ADD** *obj_id*)	Increments the reference count for the specified *obj_id*.
REF(**DROP** *obj_id*)	Decrements the reference count for the specified *obj_id*.

Basically, when accessing an object, use **REF**(**ADD** *obj_id*) to indicate that the application is to use the object (and to increment the reference count). When finished, use **REF**(**DROP** *obj_id*) (to decrement the reference count). When no other references exist, the object is automatically deleted.

Only when an object reference count goes to 0 does the system actually delete the object and all its properties. `ON_DELETE` logic is only executed at the point when the object reference count goes to zero (see **On Create/On Delete Logic**, *p.356*). The **DROP OBJECT** may be used as another method for deleting an object.

All objects are destroyed automatically when the application issues a **START** directive or if the **END** directive is entered at command mode.

DROP OBJECT Directive

DROP OBJECT *obj_id*

The **DROP OBJECT** directive is used to delete an object and its properties (if the reference count is 1). This directive has the same effect as using **REF**(**DROP** *obj_id*) to decrement the reference count from 1 to 0. The **DELETE OBJECT** and **DROP OBJECT** directives may be used interchangeably for this purpose.

OPEN OBJECT Directive

OPEN OBJECT (*chan*[,*fileopt*])*string$*

Use the **OBJECT** keyword in an **OPEN** statement to open a file for exclusive use within an object. This ensures that only logic executing inside the object that executed the **OPEN** can change the status of the specified file. This denies any **READ**, **WRITE**, **REMOVE**, **CLOSE**, or other directive that the application attempts to make from outside the object. External attempts to alter the state of the specified file returns `Error #13: File access mode invalid`. The file will be closed automatically when the object is deleted, or if there is an explicit **CLOSE (***chan***)** from within the object.

Putting It All Together

This section describes the creation and implementation of three objects in ProvideX (***Company, Customer, Supplier***) to illustrate the OOP syntax described earlier under **ProvideX OOP Interface**, *p.353*.

Topics

Company Class

The contents of the `company.pvc` class definition file appears as follows:

```
0010 DEF CLASS "Company"
0020 PROPERTY Name$,Address$,City$,PhoneNumber$,FaxNumber$
0030 FUNCTION Add()Add
0040 FUNCTION Delete()Delete
0050 FUNCTION Fax()Fax
0060 FUNCTION ShowSpecial()";ShowSpecial"
0070 END DEF
0100 ! ^100
0110 On_Create:; GOSUB Where_are_we; RETURN 1
0120 On_Delete:; GOSUB Where_are_we; RETURN 1
0200 ! ^100
0210 Add:; GOSUB Where_are_we; RETURN 1
0220 Delete:; GOSUB Where_are_we; RETURN 1
0230 Edit:; GOSUB Where_are_we; RETURN 1
0240 Find:; GOSUB Where_are_we; RETURN 1
0250 Fax:; GOSUB Where_are_we; RETURN 1
0300 ! ^100
0310 ShowSpecial:
0320 X$="Special Variables that exist in every Object:"+SEP
0330 X$+=$09$+"_OBJ = "+STR(_OBJ)+SEP+$09$
0340 X$+="_CLASS$ = "+_CLASS$+SEP+$09$
0350 X$+="_REFCNT = "+STR(_REFCNT)+SEP
0360 MSGBOX X$,"In the Company "+FNLabelName$+" logic"
0370 RETURN 1
0400 ! ^100
0410 Where_are_we:
0420 MSGBOX "In the Company "+FNLabelName$+" logic","FYI"
0430 RETURN
8000 ! 8000
8010 def fnLabelName$=mid(pgm(-3),pos(";"=pgm(-3)+";")+1)
```

The definition of this class starts with the **DEF CLASS** directive on statement 0010 and concludes with the **END DEF** directive on statement 0070. All of the fields (**Properties**) of the object are declared by the **PROPERTY** directive on statement 0020. The **Methods** available are defined using the **FUNCTION** directive on statements 0030 through 0060.

Creating a reference to, or *instantiating* the class in an application is accomplished using the **NEW( )** function; i.e., `Comp=NEW("Company")`. The numeric variable `Comp` contains a reference to the object known as an **Object Identifier**. All interaction with the object takes place using this object identifier**.**

Creating a new object automatically executes the logic that follows the statement label called `On_Create` in the program associated with the object. This label is optional and will not generate an error if it does not exist.

Determining what properties and methods are available for a given object is accomplished using the **Apostrophe** operator (' *tick*) followed by an ***** *asterisk*.

Example:

```
print Comp'* ! Show all Properties & Methods
! Produces the following results:
FaxNumber$,PhoneNumber$,City$,Address$,Name$,ShowSpecial(),Fax(),
Delete(),Add(),
```

The properties are listed as simple string or numeric variable names while the available methods end with a pair of parentheses "( )".

Methods are called by specifying the object identifier and the **Apostrophe** operator followed by the method name. As methods are designed to return a status value, they must appear on the right side of a **LET** assignment or be used with a directive such as a **PRINT**.

Example:

```
print Comp'ShowSpecial()
! Displays the following status value
 1
```

This produces a message box that lists all of the special variables that are available while executing code on behalf of an object; i.e., `_Obj`, `_Class$`, and `_Refcnt`.

After acknowledging the message box, a value of 1 (*one*) appears on the next line. This value represents the status returned by the method when it executes the `RETURN 1` on statement 0370. A non-zero status value usually indicates that a method is successful, while a value of zero signifies that it is not. (Although, this is at the discretion of the programmer.)

Objects must be dropped or deleted when the application is finished with them. This is accomplished using **DROP OBJECT** or **REF( )** techniques; e.g.,

DROP OBJECT *obj_id*
VARIABLE = REF(DROP *obj_id***)**

The following example removes the **Company** object just created:

```
Drop Object Comp
```

Removing an object results in the On_Delete statement label being executed (if it exists). In the **Company** object example, a message box will appear that indicates "In the Company On_Delete logic".

Customer Class

The contents of the customer.pvc class definition file appears as follows:

```
0010 DEF CLASS "Customer"
0020 LIKE "Company"
0030 PROPERTY Limit,LastInvDate$
0040 FUNCTION Invoice()";Invoice"
0050 FUNCTION Edit()";Edit"
0060 END DEF
0100 ! ^100
0110 On_Create:; GOSUB Where_are_we; RETURN 1
0120 On_Delete:; GOSUB Where_are_we; RETURN 1
0200 ! ^100
0210 Invoice:; GOSUB Where_are_we; RETURN 1
0220 Edit:; GOSUB Where_are_we; RETURN 1
0300 ! ^100
0310 Where_are_we:
0320 MSGBOX "In the Customer "+FNLabelName$+" logic","FYI"
0330 RETURN
8000 ! 8000
8010 def fnlabelname$=mid(pgm(-3),pos(";"=pgm(-3)+";")+1)
```

This class definition introduces the **Inheritance** aspect of object oriented programming by its use of the **LIKE** directive on line 0020. **LIKE** is used to include all of the characteristics from a different object into the current object. Creating an object using this class definition and viewing all of its properties and methods best illustrates this.

Example:

```
Cust=NEW("Customer")
print Cust'*
! Produces the following results:
LastInvDate$,Limit,Edit(),Invoice(),FaxNumber$,PhoneNumber$,
City$,Address$,Name$,ShowSpecial(),Fax(),Delete(),Add(),
```

The **Customer** class explicitly defines the LastInvDate$ and Limit properties and the Invoice() and Edit() methods. The remaining properties (City$, Address$, etc.) and methods (ShowSpecial(), Fax(), etc.) are inherited from the **Company** class. This allows classes that are related by inheritance to share

common properties and methods without having to duplicate any of the programming code normally required. Having the code in a single location helps to reduce the amount of time spent on maintenance and enhancements.

The `Where_are_we` subroutine used in both the **Company** and **Customer** classes could be converted to a method, which would allow both to share the same code. To simplify the examples, this approach is not taken.

From an application standpoint, there is no difference between accessing a method defined in the base class or one that is inherited.

Example:

```
Status = Cust'Fax()
```

Although the **`Customer`** class does not have a `Fax()` method, it is available since it is inherited from the **`Company`** class. Remember to delete the instance of **`Customer`** (`Cust` object) when finished:

```
Drop Object Cust
```

Supplier Class

The `supplier.pvc` class definition file appears as follows:

```
0010 def class "Supplier"
0020 like "Company"
0030 property MinOrder,LeadTime
0040 function Order()";Order"
0050 function Edit()";Edit"
0060 end def
0100 ! ^100
0110 On_Create:; gosub Where_are_we; return 1
0120 On_Delete:; gosub Where_are_we; return 1
0200 ! ^100
0210 Order:; gosub Where_are_we; return 1
0220 Edit:; gosub Where_are_we; return 1
0300 ! ^100
0310 Where_are_we:
0320 msgbox "In the Supplier "+fnLabelName$+" logic","FYI"
0330 return
8000 ! 8000
8010 def fnLabelName$=mid(pgm(-3),pos(";"=pgm(-3)+";")+1)
```

The basic concepts illustrated with the **Customer** class also apply to **Supplier**.

Additional Classes

This section makes a few changes to the **Customer** class discussed earlier. Each of these samples will introduce some additional features of ProvideX OOP syntax: **Cust2**, **Cust3**, **Cust4**, **Cust5**, and **Cust6**.

A simple data file called `cstfile` is used in these classes. The following tables outline the required information to create this file.

Field definitions:

Field Name	*Description*	*Type*	*Len*	*Delimited?*
`Cust_No$`	Customer ID	String	6	Yes
`Name$`	Customer Name	String	30	Yes
`Addr$`	Address	String	30	Yes
`City$`	City	String	30	Yes
`Salesrep$`	Sales Rep	String	3	Yes
`Amt_Owing`	Amount Owing	Numeric	10.2	Yes

Key definitions:

Key #	*Description*	*Fields*
0	Primary Key	`Cust_No$`
1	Sort by Name	`Cust_Name$ + Cust_No$`
2	Sort by Sales Rep	`Salesrep$ + Cust_No$`

For the final two sample objects, a NOMADS *File Maintenance* panel for the `Customer` file (Cstupd) is assumed.

Cust2 Class

The contents of the `Cust2.pvc` class definition file illustrates the ability to associate program logic with a property when an attempt is made to alter its contents.

```
00010  DEF CLASS "Cust2"
00020  PROPERTY Cust_No$ SET Change_Cust
00030  PROPERTY Name$,Addr$,City$,Salesrep$
00040  PROPERTY Amt_Owing SET ERR
00050  FUNCTION Find(X$)Get_Customer
00060  FUNCTION Next()Read_Next
00070  FUNCTION Update()Update_Customer
00080  END DEF
00100 ! !^100
00110 On_Create:
```

```
00120  IF %Customer_File=0 \
        THEN OPEN (GFN,IOL=*)"cstfile";
             %Customer_File=LFO
00130  %Customer_Object++
00140  RETURN
00200 ! !^100
00210 Get_Customer: \
       ENTER C$
00220  READ (%Customer_File,KEY=C$) ! Loads all the variables
00230  RETURN 1
00300 ! !^100
00310 Change_Cust: \
       ENTER C$
00320  READ DATA FROM "" TO IOL=IOL(%Customer_File)
00330  Cust_No$=C$
00340  READ (%Customer_File,KEY=C$,DOM=*END)
00350  RETURN
00400 ! !^100
00410 Read_Next:
00420  READ (%Customer_File,END=*NEXT);
       RETURN 1
00430  RETURN 0
00500 ! !^100
00510 Update_Customer:
00520  WRITE (%Customer_File)
00530  RETURN 1
00600 ! !^100
00610 On_Delete:
00620  IF --%Customer_Object<>0 \
        THEN RETURN
00630  CLOSE (%Customer_File);
       %Customer_File=0
00640  RETURN
```

A **PROPERTY** definition can include an optional keyword **SET** followed by either a line label or the keyword **ERR**. In the case of a line label, the program logic starting at that label is executed when an attempt is made to alter the contents of the property. The keyword **ERR** is used to prevent the contents of a property from being changed and an error is returned to the program attempting to do so. By default, an `Error #88: Invalid/unknown property name` is returned. However, this can be overridden as shown in a subsequent sample class.

These options are used on the following property declarations:

```
0020 property Cust_No$ set Change_Cust
0040 property Amt_Owing set err
```

To illustrate use of this class, create a new instance of **Cust2**:

```
->Cust2=NEW("Cust2")
```

Auto-execute the `Change_Cust` method by setting the `Cust_No$` variable. Use `Settrace/Endtrace` to show the code executing:

```
->SETTRACE
->Cust2'Cust_No$="000011"
0310 Change_Cust: ENTER C$
0320 READ DATA FROM "" TO IOL=IOL(%Customer_File)
0330 LET Cust_No$=C$
0340 READ (%Customer_File,KEY=C$,DOM=*END)
0350 RETURN
->ENDTRACE
```

If you try to change the `Amt_Owing` property, this results in an `Error` #88 due to the **SET ERR** clause on the property definition:

```
->Cust2'Amt_Owing=100
Error #88: Invalid/unknown property name
```

Drop the object when finished:

```
->DROP OBJECT Cust2
```

Note: One key issue regarding **Cust2** is the use of *global* variables, which results in a violation of the principle of **Encapsulation**, *p.351*. ProvideX itself has no mechanism to enforce OOP "rules". *Cust3* (below) shows a way to avoid this problem.

Cust3 Class

This class illustrates a better approach for handling **Encapsulation** – by replacing the *global* variables with *local* variables. It also introduces the ability to associate program logic when reading the contents of a property. The contents of the `cust3.pvc` class definition file appears as follows:

```
00010  DEF CLASS "Cust3"
00020  LOCAL Customer_File,Customer_Object
00030  PROPERTY Cust_No$ SET Change_Cust
00040  PROPERTY Name$,Addr$,City$,Salesrep$
00050  PROPERTY Amt_Owing GET ";CheckSecurity" SET ERR
00060  FUNCTION Find(X$)Get_Customer
00070  FUNCTION Next()Read_Next
00080  FUNCTION Update()Update_Customer
00090  END DEF
00100 !
00200 ! ^100
00210 On_Create:
00220  IF Customer_File=0 \
        THEN OPEN (GFN,IOL=*)"cstfile";
             Customer_File=LFO
00230  Customer_Object++
00240  RETURN
```

```
00300 ! ^100
00310 Get_Customer: \
       ENTER C$
00320  READ (Customer_File,KEY=C$) ! Loads all the variables
00330  RETURN 1
00500 ! ^100
00510 Change_Cust: \
       ENTER C$
00520  READ DATA FROM "" TO IOL=IOL(Customer_File)
00530  Cust_No$=C$
00540  READ (Customer_File,KEY=C$,DOM=*END)
00550  RETURN
00600 ! ^100
00610 Read_Next:
00620  READ (Customer_File,END=*NEXT);
       RETURN 1
00630  RETURN 0
00700 ! ^100
00710 Update_Customer:
00720  WRITE (Customer_File)
00730  RETURN 1
00800 ! ^100
00810 CheckSecurity:
00820  IF %Security_OK \
        THEN RETURN Amt_Owing \
        ELSE EXIT 52
00900 ! ^100
00910 On_Delete:
00920  IF --Customer_Object<>0 \
        THEN RETURN
00930  CLOSE (Customer_File);
       Customer_File=0
00940  RETURN
```

Cust3 replaces the *global* variables `%Customer_File` and `%Customer_Object` with properties that are *local* to the object. This is accomplished with the use of the **LOCAL** directive; i.e.,

```
0020 LOCAL Customer_File,Customer_Object
```

Properties declared **LOCAL** are available when executing logic on behalf of an object, yet they are invisible to the outside world.

While **Cust2** demonstrated the ability to intercept any attempt made to change a value, **Cust3** introduces the syntax required to intercept any attempts made to read a property. This is accomplished by adding the keyword **GET** to the **PROPERTY** definition; i.e.,

```
0050 property Amt_Owing get ";CheckSecurity" set err
```

The `CheckSecurity()` method in **`Cust3`** is designed to return the value of the `Amt_Owing` property if a global variable, `%Security_OK` is set to a non-zero value. Otherwise, it returns an `Error #52: Program is password protected`; e.g.,

To illustrate use of this class, create a new instance of **Cust3**:

```
->Cust3=NEW("Cust3")
```

Get the next customer on file with the `Next()` method:

```
->x=Cust3'Next()
```

Disable the security override flag:

```
->%Security_OK=0
```

An attempt made to query the `Amt_Owing` property will result in an `Error #52` due to the `EXIT 52` on statement 0820:

```
->PRINT Cust3'Amt_Owing
Error #52: Program is password protected
```

Enable the security override flag:

```
->%Security_OK=1
```

Display the `Amt_Owing` property after enabling **SETTRACE** and **ENDTRACE** to show the code executing:

```
->SETTRACE
->PRINT Cust3'Amt_Owing
0810 CheckSecurity:
0820 IF %Security_OK THEN RETURN Amt_Owing ELSE EXIT 52
 0
->ENDTRACE
```

Drop the object when finished:

```
DROP OBJECT Cust3
```

Cust4 Class

This class introduces the concept of *method overloading*, which is the ability to reference a method with a varying number and/or type of parameters. This technique simplifies the external interface to the object by having a single method name. One example of this would be the ability to find a customer using either a string or numeric value. The contents of the `cust4.pvc` class definition file appears as follows:

```
00010  DEF CLASS "Cust4"
00020  LOCAL Customer_File,Customer_Object
00030  PROPERTY Cust_No$ SET Change_Cust
00040  PROPERTY Name$,Addr$,City$,Salesrep$
00050  PROPERTY Amt_Owing GET ";CheckSecurity" SET ERR
```

```
00060  FUNCTION Find(X$)Get_Customer
00070  FUNCTION Find(X)Get_Customer_Numeric
00080  FUNCTION Next()Read_Next
00090  FUNCTION Update()Update_Customer
00100  END DEF
00200 ! ^100
00210 On_Create:
00220  IF Customer_File=0 \
        THEN OPEN (GFN,IOL=*)"cstfile";
             Customer_File=LFO
00230  Customer_Object++
00240  RETURN
00300 ! ^100
00310 Get_Customer: \
       ENTER C$
00320  READ (Customer_File,KEY=C$) ! Loads all the variables
00330  RETURN 1
00400 ! ^100
00410 Get_Customer_Numeric: \
       ENTER C
00420  C$=STR(C:"000000",ERR=*NEXT)
00430  C$=PAD(UCS(C$),6)
00440  READ (Customer_File,KEY=C$) ! Loads all the variables
00450  RETURN 1
00500 ! ^100
00510 Change_Cust: \
       ENTER C$
00520  READ DATA FROM "" TO IOL=IOL(Customer_File)
00530  Cust_No$=C$
00540  READ (Customer_File,KEY=C$,DOM=*END)
00550  RETURN
00600 ! ^100
00610 Read_Next:
00620  READ (Customer_File,END=*NEXT);
       RETURN 1
00630  RETURN 0
00700 ! ^100
00710 Update_Customer:
00720  WRITE (Customer_File)
00730  RETURN 1
00800 ! ^100
00810 CheckSecurity:
00820  IF %Security_OK \
        THEN RETURN 1 \
        ELSE EXIT 52
00900 ! ^100
00910 On_Delete:
```

```
00920  IF --Customer_Object<>0 \
        THEN RETURN
00930  CLOSE (Customer_File);
       Customer_File=0
00940  RETURN
```

Notice that **FUNCTION** statements for the `Find()` method appear twice; i.e.,

```
0060 FUNCTION Find(X$)Get_Customer
0070 FUNCTION Find(X)Get_Customer_Numeric
```

The significant difference between these two definitions is the type of parameter specified in the parentheses. The first accepts a *string* value while the second requires a *numeric* entry.

Multiple definitions of the same method can be specified provided each has a different parameter list. Knowing which method definition to use is determined by matching the parameters based on their type, either *string* or *numeric*, and the number of parameters specified.

To illustrate use of this class, create a new instance of **Cust4**:

```
Cust4=NEW("Cust4")
```

Locate the first customer using a string argument:

```
->x=Cust4'Find("000011")
->PRINT Cust4'Cust_No$," - ",Cust4'Name$
000011 - Flagship Importers
```

Locate the second customer using a numeric argument:

```
->x=Cust4'Find(27)
->PRINT Cust4'Cust_No$," - ",Cust4'Name$
000027 - Yorktown Wood Products
```

Drop the object when finished:

```
DROP OBJECT Cust4
```

Panel Class

This class will be used with the **Cust5** example. It assumes a NOMADS *File Maintenance* panel called Cstupd has been created for the `Customer` file. The contents of the `panel.pvc` class definition file appears as follows:

```
00010 ! Panel Definition
00020  DEF CLASS "Panel"
00030  LOCAL Panel_Name$,Panel_Library$
00040  FUNCTION Edit()Do_Panel
00050  END DEF
00060 Do_Panel:
00070  IF NUL(Panel_Name$) \
        THEN Panel_Name$=_OBJ'_CLASS$
```

```
00080  IF NUL(Panel_Library$) \
        THEN Panel_Library$="ProvideX.en"
00090  Sv_prc=PRC;
       RESET ;
       PRECISION Sv_prc
00100  Scrn_ID$=Panel_Name$,Scrn_Lib$=Panel_Library$
00110  PERFORM "*winproc;Post_Enter"
00120  RETURN 1
```

File Class

This class will be used with the **Cust5** example. The contents of the `file.pvc` class definition file appears as follows:

```
00010 ! "File" class definition
00020  DEF CLASS "File"
00030  LOCAL FileNo
00040  FUNCTION Open(FILENAME$)Open_File
00050  FUNCTION Close()Close_File
00060  FUNCTION Find(K$)Read_Via_Key
00070  FUNCTION Find(K$,KeyNo)Read_Via_Alt_Key
00080  FUNCTION Find(K$,KeyNo$)Read_Via_Alt_Key2
00090  FUNCTION Find(I)Read_Via_Ind
00100  FUNCTION Next()Read_Next
00110  FUNCTION Next(KeyNo)Read_Next_Alt_Key
00120  FUNCTION Next(KeyNo$)Read_Next_Alt_Key2
00130  FUNCTION Update()Write_Record
00140  FUNCTION Update(K)Write_Record_With_Key
00150  FUNCTION Insert()Insert_Record
00160  FUNCTION Insert(K$)Insert_Record_With_Key
00170  FUNCTION Valid()=1 ! Always true unless overridden
00180  END DEF
00200 ! ^100 Open
00210 Open_File:
00220  ENTER F$
00230  IF FileNo<>0 \
        THEN EXIT 13
00240  X=HFN;
       OPEN (X,IOL=*)F$
00250  FileNo=X;
       RETURN 1
00300 ! ^100 Close
00310 Close_File:
00320  IF FileNo=0 \
        THEN EXIT 13
00330  CLOSE (FileNo);
       FileNo=0
00340  RETURN 1
```

```
00400 ! ^100 Read via Index
00410 Read_Via_Ind:
00420  ENTER I
00430  READ DATA FROM "" TO IOL=IOL(FileNo)
00440  READ (FileNo,IND=I,DOM=*NEXT,END=*NEXT);
       RETURN 1
00450  RETURN 0
00500 ! ^100 Read via primary key
00510 Read_Via_Key:
00520  ENTER K$
00530  READ DATA FROM "" TO IOL=IOL(FileNo)
00540  READ (FileNo,KEY=K$,KNO=0,DOM=*NEXT);
       RETURN 1
00550  RETURN 0
00600 ! ^100 Read via an alternate key
00610 Read_Via_Alt_Key:
00620  ENTER K$,KeyNo
00630  READ DATA FROM "" TO IOL=IOL(FileNo)
00640  READ (FileNo,KEY=K$,KNO=KeyNo,DOM=*NEXT);
       RETURN 1
00650  RETURN 0
00700 ! ^100 Read via an alternate key
00710 Read_Via_Alt_Key2:
00720  ENTER K$,KeyNo$
00730  READ DATA FROM "" TO IOL=IOL(FileNo)
00740  READ (FileNo,KEY=K$,KNO=KeyNo$,DOM=*NEXT);
       RETURN 1
00750  RETURN 0
00800 ! ^100 Read Next on primary key
00810 Read_Next:
00820  READ DATA FROM "" TO IOL=IOL(FileNo)
00830  READ (FileNo,KNO=0,END=*NEXT);
       RETURN 1
00840  RETURN 0
00900 ! ^100 Read Next via an alternate key
00910 Read_Next_Alt_Key:
00920  ENTER KeyNo
00930  READ DATA FROM "" TO IOL=IOL(FileNo)
00940  READ (FileNo,KNO=KeyNo,END=*NEXT);
       RETURN 1
00950  RETURN 0
01000 ! ^100 Read Next via an alternate key
01010 Read_Next_Alt_Key2:
01020  ENTER KeyNo$
01030  READ DATA FROM "" TO IOL=IOL(FileNo)
01040  READ (FileNo,KNO=KeyNo$,END=*NEXT);
       RETURN 1
```

```
01050  RETURN 0
01100 ! ^100 - Update
01110 Write_Record:
01120  IF NOT(_OBJ'Valid()) \
        THEN EXIT 17
01130  WRITE (FileNo);
       RETURN 1
01140 !
01150 Write_Record_With_Key:
01160  ENTER K$
01170  IF NOT(_OBJ'Valid()) \
        THEN EXIT 17
01180  WRITE (FileNo,KEY=K$);
       RETURN 1
01200 ! ^100 - Insert record
01210 Insert_Record:
01220  IF NOT(_OBJ'Valid()) \
        THEN EXIT 17
01230  WRITE (FileNo,DOM=*NEXT);
       RETURN 1
01240  RETURN 0
01250 !
01260 Insert_Record_With_Key:
01270  ENTER K$
01280  IF NOT(_OBJ'Valid()) \
        THEN EXIT 17
01290  WRITE (FileNo,KEY=K$,DOM=*NEXT);
       RETURN 1
01300  RETURN 0
```

Cust5 Class

This class illustrates the ability to inherit the characteristics of additional classes. The contents of the `cust5.pvc` class definition file appears as follows:

```
00010  DEF CLASS "Cust5"
00020  LIKE "File","Panel"
00030  PROPERTY Cust_No$ WRITE Change_Cust
00040  PROPERTY Name$,Addr$,City$,Salesrep$,Amt_Owing
00050  END DEF
00060 !
00070 On_Create:
00080  Panel_Library$="cstfile.en",Panel_Name$="cstupd"
00090  RETURN _Obj'Open("cstfile")
00100 !
00110 On_Delete:
00120  RETURN _Obj'Close()
00130 !
```

```
00140 Change_Cust: \
       ENTER C$
00150  IF NOT(_Obj'Find(C$)) \
        THEN Cust_No$=C$
00160  RETURN 1
```

While the definition is relatively small, this class is more powerful than it looks due to the inclusion of **File** and **Panel** classes provided earlier. Referencing methods and properties from inherited classes is accomplished by prefixing the method name with **_Obj**, as illustrated in the following statements:

```
0090 RETURN _Obj'Open("cstfile")
0120 RETURN _Obj'Close()
0150 IF NOT(_Obj'Find(C$)) THEN Cust_No$=C$
```

In each case, prefixing the method name with **_Obj** (in place of the object identifier) indicates that the method exists somewhere in the current class. Knowing its exact location is not necessary. See **Using Methods and Properties**, *p.354*. The example below uses the `Edit()` method from the **Panel** object to process a NOMADS panel.

To illustrate use of this class, create a new instance of **Cust5**:

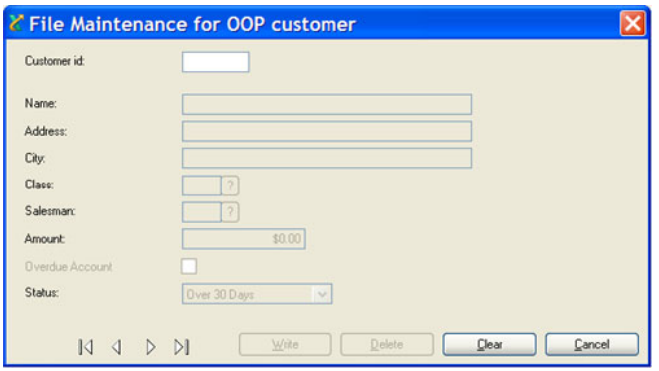
```
Cust5=NEW("Cust5")
```

Display the available properties and methods:

```
->PRINT Cust5'*
Addr$,Amt_Owing,City$,Close(),Cust_No$,Edit(),Find(),Insert(),
Name$,Next(),Open(),Salesrep$,Update(),Valid(),
```

Invoke a NOMADS panel:

```
->x=Cust5'Edit()
```



Drop the object when finished:

```
Drop Object Cust5
```

Cust6 Class

This class demonstrates the **Polymorphism** aspect of object orientation. It also introduces the concept of *method overriding*, which is the ability for a child class to override methods (same name and argument list) with different functionality. The contents of the `cust6.pvc` class definition file appears as follows:

```
00010  DEF CLASS "Cust6"
00020  LIKE "File","Panel"
00030  PROPERTY Cust_No$ WRITE Change_Cust
00040  PROPERTY Name$,Addr$,City$,Salesrep$,Amt_Owing
00050  FUNCTION Find(X$)Do_MyFind
00060  END DEF
00070 !
00080 On_Create:
00090  Panel_Library$="cstfile.en",Panel_Name$="cstupd"
00100  RETURN _Obj'Open("cstfile")
00110 !
00120 On_Delete:
00130  RETURN _Obj'Close()
00140 !
00150 Change_Cust: \
       ENTER C$
00160  IF NOT(_Obj'Find(C$)) \
        THEN Cust_No$=C$
00170  RETURN 1
00180 !
00190 Do_MyFind:
00200  ENTER K$
00210  K$=STR(NUM(K$,ERR=*NEXT):"000000")
00220  K$=PAD(K$,6)
00230  RETURN _Obj'Find(FROM "File",K$)
```

The `Find()` method declared in **Cust6** is designed to augment the functionality of the `Find()` method from the **File** class. The `Find()` method in this class intercepts the value, re-formats it to match the key value for the data file, and then passes the resulting value to the `Find()` method in the **File** class.

To illustrate use of this class, create a new instance of **Cust6**:

```
Cust6=NEW("Cust6")
```

Pass what appears to be an invalid Customer ID to the `Find( )` method:

```
->x=Cust6'Find("67")
->PRINT "Customer: ",Cust6'Cust_No$," - ",Cust6'Name$
Customer: 000011 - Flagship Importers
```

Drop the object when finished:

```
Drop Object Cust6
```

A
Appendix

Overview

This appendix covers miscellaneous topics and includes information that is supplementary to the other chapters in this manual. Section headings, page numbers, and outlines of these topics are listed below:

Security Features

Most operating systems allow different users to have access to the same machine, each with their own files, desktop settings etc. Business applications are often designed so that multiple users can be logged onto the same system, to access the same data over various types of networks and/or the internet.

Of course, with multi-user capabilities comes issues of security. Operating systems and networks provide their own level of protection, but ultimately, it is the developer who is responsible for securing sensitive data and for managing all the activities that are to be carried out within their applications.

ProvideX security measures can include various combinations of encryption, authentication, authorization, and or access control functionality. This section discusses the various network-level, program-level, and data security features available to you when designing and building your applications.

Topics

Restricting Access to Command Mode

Using two directives, **SETESC** and **ERROR_HANDLER**, you can totally eliminate the possibility of the user accessing ProvideX command mode from your application. Use of the **ERROR_HANDLER** directive to handle abnormal error conditions is described in the *Chapter 3.* **Development Tools**.

The **SETESC** directive allows the user to disable the **ESC**ape (or **Break**) key recognition within ProvideX. When **SETESC OFF** is executed, all user escape requests are ignored.

This mode of operation continues for the duration of the session or until **SETESC ON** is executed. An **ESC**ape/**Break** will be recognized in those programs that specifically have a **SETESC** *nnnn* directive regardless of the **ON**/**OFF** status. This allows programs that have escape handling logic to continue to function properly.

To prevent the **ESC**ape/**Break** key from allowing the user to interrupt the execution of an application you should include the following line in your START_UP initialization program:

```
0000 SETESC OFF
```

During the testing and development stages of an application, it may be desirable not to execute this directive or to make the execution of these directives optional based on the userID (**UID** or **WHO**).

Password Protection

There are several places in ProvideX to assign passwords for restricting access to programs and data files, at the language level and in the development environment.

Program Security

The **PASSWORD** directive may be used to place a user-specified password on the current *program code*. Simply load the program, enter the **PASSWORD** directive followed by the desired *password*, and then save the program. Once saved with a password, the program may not be listed or modified in any way without first removing the password by re-entering the **PASSWORD** directive with the correct password string.

If the password string is preceded by an ***** *asterisk*, the **PASSWORD** directive defines a common password which will be automatically applied to all passworded programs as they are loaded and assigned to all new programs.

Passwords are maintained within the actual object code of the program and are encrypted. If a password is placed on a program and subsequently forgotten there is no simple way to remove it. If you need to remove a password, contact *Sage ProvideX Support* for assistance.

The **'EL'=** system parameter can be used to check the current encryption setting, or set encryption to a new level. For further information on this feature, see **PASSWORD** in the *Language Reference, p.238*.

Data Passwording and Encryption

The **PASSWORD** directive can also be used to assign passwords to *files* with the option to encrypt data. The **KEY=***pswd$* option is required to **OPEN** files that have been passworded. Encryption is only available for VLR and EFF files. In order to define/change a password, you must have exclusive access to the file and it must be empty. The different syntax formats for passwording data via the **PASSWORD** directive are listed and described in the ProvideX *Language Reference, p.238*.

The following table shows the usage, access level, and encryption associated with each syntax format used to assign a password to a data file:

PASSWORD *Format*	*Access Level*	*Without Password*			*With Correct Password*			*Encrypted*
		Open	*Read*	*Write*	*Open*	*Read*	*Write*	
OPEN	0	No	No	No	Yes	Yes	Yes	No
WRITE	1	Yes	Yes	No	Yes	Yes	Yes	No
OPEN AND ON DATA	2	No	No	No	Yes	Yes	Yes	Yes
WRITE AND ON DATA	3	Yes	Yes	No	Yes	Yes	Yes	Yes

Another level of data security may be added by introducing the hash function into the process. See **Hash Function**, *p.384*.

Other Passwording in ProvideX

Following are some of the other password-related mechanisms used in the ProvideX development suite:

Data Dictionary. The Data Dictionary Maintenance interface in the NOMADS toolset includes a Data File Password Utility that allows you to add, change or remove passwords in ProvideX data dictionary files.

Views System. The ProvideX Views system in ProvideX allows you to lock View definitions with a password to control modification.

Application Server. The administrator is able to set passwording for server-side access as well on remote user access to the application server.

Multi-Lines. Creating a multi-line control in NOMADS and at the language level includes the option to cover password entries using the **$** *dollar sign* symbol as substitute for each character entered. The **AutoComplete** feature is disabled when the multi-line used for a password field.

Hash Function

The ProvideX **HSH( )** function introduces another technique for data protection. It is not used to encrypt data directly, but to check authenticity. The hashing mechanism takes a block of data as input, and produces a hash key value as output. The key value can then be used to check the integrity of the original data without having to reveal its contents. Hashes are most often used to obscure sensitive data that should not be made available to anyone but the owner; e.g., credit card numbers, bank IDs, passwords, etc.).

HSH( ) supports the most widely used and trusted hash algorithms in the industry, including MD5 and SHA-1; however, only versions of ProvideX that support OpenSSL and have OpenSSL installed properly will be able to access these types of hashes. It also has the ability to access ciphers in OpenSSL libraries to encrypt and decrypt information.

For further information on the use of ProvideX hash functionality, see **HSH( )** in the *Language Reference, p.387*.

Software Registration and Activation

ProvideX includes tools and facilities to protect applications from software piracy through the use of package registration and activation. All instances of ProvideX require a base system or primary-level activation that enables use of the language

itself, program editing, listing, saving, and access to console mode. Activation keys are based on specific hardware configurations thus preventing applications from being moved from one computer to another without re-activation.

Note: Complete information on ProvideX product licensing can be found in the ProvideX *Installation Guide*.

Registration System

Developers can also have activation keys assigned specifically to their applications where, without the correct key, the programs cannot be viewed, modified or run. In fact, the programs are encrypted with a key that only the original developer will have access to. The registration system is also useful for releasing software that has a built-in expiry date/time based on an assigned activation key.

Each development shop using this feature would be required to maintain their own key generation system for creating and distributing keys. They will be given a unique ***owner code***, where each generated key is specific to that code only. One owner code will not provide access to another development shop's programs. Each is independent of the other.

A registered program may be assigned from 0 to 20 flags. When a key is generated, it may be done in such a way that one or more flags will be valid. This gives the developer the ability to create keys that allow or deny users access to specific modules within their application. Also, flags may be used to control whether or not the key allows the person running the program the ability to modify or list programs. To obtain a registration number and the tools required to implement this facility, contact your local distributor.

Creating Registered Programs

To create a program which is to be secured via a registered activation key the programmer uses the following command:

SAVE "*progname*",**OWN=***package_no* [,**FLG=***n*:*n*:*n*]

The package number must be a registered package number that has been assigned to the developer and for which he/she must have been activated as a developer. The **FLG=***n*:*n*:*n* option allows the programmer to define which registration flags must be active in order to **LOAD**/**RUN** this program. If any flags are specified, then at least one flag must match that of the users registration.

Note: Full details on the use of this facility are beyond the scope of this manual but can be provided to developers upon request.

Other Considerations

This type of registration is not for everyone. It requires that each development shop maintain their own system to implement, track, and distribute their unique keys. Due to commitment involved in this type of system, the majority of ProvideX developers prefer to use other distribution controls within their applications; i.e., **Password Protection**, *p.383*.

Also, Sage may be able to recover a passworded program when the password is lost; however, it is impossible to recover a program that is encoded with an owner code if the mechanisms used to encrypt it are lost. Because of this danger, you can arrange for Sage to maintain a copy of your activation key generation parameters which provides a method of recovery in case you lose your key generation components.

NOMADS Security Manager

The Security Manager system in NOMADS allows you to protect your GUI-based ProvideX applications from unauthorized access. Individual screen components can be set up to allow full access, view-only access, or no access for specific users. Each user's authorization level can be based on User ID and / or security classifications.

This system is maintained via the Security drop-down menu in the NOMADS Session Manager menu bar.

By default, if a NOMADS panel has no existing object classifications, there is no security and all users are granted full access. If you assign one or more security classifications, then only those users who are *registered* will have access. NOMADS does not display the control to users who are not registered in an assigned classification.

More information on NOMADS is provided in *Chapter 6.* **Graphical User Interfaces**. Complete documentation on this feature is provided in the ProvideX NOMADS manual.

Secure Socket Layer (SSL)

The Secure Socket Layer (SSL) is an industry-standard protocol for managing the security of message transmissions over the internet. It is used by millions of web applications around the world for the protection of online customer transactions. Using SSL protocol in a website instills confidence that the user can expect a secure link, and that the source belongs to a valid, legitimate organization.

SSL encodes the data rendering it unreadable to anyone who may try to intercept the transmission. When an SSL session is started, the server sends its public key to the client, which the client uses to send a randomly-generated private key back to the server in order to establish a secret key "exchange" for that session.

For information on how SSL can be used with ProvideX applications, see **ProvideX SSL Support**, below.

Certificates and Encryption Keys

In order to be able to generate an SSL link, a server requires an SSL certificate. While it is possible to generate your own certificate using tools available on the internet, you should only use these internally or for testing your site. With self-generated certificates, users will likely receive warning messages on their browsers stating that the certificate has not been authenticated and may not be from a "*trusted authority*".

Of course, "*certificate not trusted*" errors in your commercial applications would be undesirable and could drive away customers. If you wish to avoid warnings like these, you should obtain your SSL certificates from a trusted third party Internet Certification Authority. Most major operating systems and browsers maintain lists of trusted Certification Authorities and will establish secure links using their certificates transparently. The padlock symbol is often used to signify when a encrypted link is established using a *trusted* SSL certificate.

Usually, the encryption key obtained from a Certificate Authority controls the level of encryption, such as 40-Bit, 56-Bit, 128-Bit, or 1024-Bit encryption.

Note: Because ProvideX uses OpenSSL libraries to provide SSL functionality, you will need an X509 certificate created for use with OpenSSL or Apache. (Apache also uses OpenSSL libraries.)

Internet communication, client-to-server or server-to-client, must be encrypted at both ends or un-encrypted on both ends. You cannot mix requests as it is not possible for an un-encrypted browser to communicate with an encrypted server and vice versa. By convention, browser requests addressed `http://` will make requests from an un-encrypted web server; use `https://` to make requests to an SSL encrypted web server.

ProvideX SSL Support

Support for TCP/IP-level SSL-encryption is available as part of the base ProvideX license. The SSL option is available with all ProvideX thin-clients when configured for use in the ProvideX *Application Server*. (SSL is not fully supported under *NTHost.) Instructions for using SSL to secure client-server applications can be found in the ProvideX *Client-Server* reference manual. See also **Hosting Facilities**, *p.242*.

The ProvideX *WebServer* interface supports SSL encryption for transactions between your web server and the user's browser. For details, refer to the ProvideX WebServer documentation.

FIN( ) *Function for SSL*

If SSL encryption is being used with your ProvideX application, the **FIN( )** function will obtain SSL details regarding a particular connection; e.g., `FIN(10,"SSL_CIPHER")`. The following keywords pertain specifically to SSL:

X509_ISSUER	Who issued the certificate.
X509_SUBJECT	Who the certificate was issued to.
X509_NOT_BEFORE	Earliest date the certificate is valid for.
X509_NOT_AFTER	Latest date the certificate is valid for.
X509_KEYTYPE	Type of key in the certificate (DSA / RAS and number of bits)
SSL_CIPHER	SSL connection cipher used information as per the OPENSSL SSL_cipher_description specs.

All X509 keywords return information about the remote certificate; i.e., from either the server certificate (if a client requests it) or the client certificate (if the server requests it). Any of the X509 requests can be prefixed with `MY_` to return the local station's certificate information; e.g., `MY_X509_KEYTYPE`, `MY_X509_SUBJECT`.

These requests can only be made once a connection is established. When made on a server, the X509 / SSL information pertains to the last / current socket connected.

Minimizing Client-Server Risks

Controlling the TCP / IP services available on a particular machine, and plugging security holes at the OS level (if any) will help keep unauthorized people from accessing sensitive data in a client-server implementation.

Your hosting facility should be able to manage what your users are capable of running via WindX, JavX, and UltraFX. Remember that the *NTHost server supplied with ProvideX does not have built-in security and is not recommended for use outside of a closed local area network. The ProvideX Application Server add-on is a much better choice. It includes administrative services that allow you to protect your applications and data over an unsecured network, such as the internet. See *Chapter 8.* **Client-Server** for more information.

It is the security holes in your applications that are the greatest concern. While the OS itself may be tightly secured, programming mistakes in the applications which run on the server are far more prevalent security risks, particularly if they grant access to console mode or give the ability to run OS commands. For more on this topic, see **Restricting Access to Command Mode**, *p.382*

Servers that are protected behind firewalls do provide some security. However, each port on a firewall that allows inbound access, or is redirected to an internal server, will have the potential to show cracks. Security depends on the application that is servicing the particular port. Whether or not you are opening (or redirecting) one

port or one thousand ports on a firewall, does not really make a difference. Once a single port is open for an application to service, the security is only as good as that provided within the application.

Virtual Private Networks (VPN) that allow remote users to become a part of the internal LAN provide good security if the authentication methods are sufficiently stringent. The best security VPN servers provide for security measures is SecureID, a rotating authentication code. Only those users with the correct access codes and one time SecureID codes may get in.

VPNs do require a large amount of computing power at both the server side and the client end. Since all packets transmitted by each client are encrypted and then placed inside a packet which will be routed across the Internet, the client must be fairly powerful to keep up with encrypting outgoing packets and decrypting incoming packets. The server, on the other hand, must be able to recognize which packets are destined for which VPN-connected clients, capture them out of the data stream, encrypt them, and transmit them to the appropriate client.

Remote access solutions such as Citrix, MS Remote Desktop, and VNC provide security at the login level. Once a client has a login, then they typically have access levels that are the same as any local PC on a network and can do as much damage as any given PC / User Login may do.

Secure Socket Layer (SSL) encryption only deals with data that is in transit from the server to the client and vice versa. Encrypting the data does not provide application level security; it only prevents unauthorized people from capturing the data packets between machines and using the data within those packets. See also **Secure Socket Layer (SSL)**, *p.386*.

Device Drivers

One of the more powerful features of the ProvideX development environment is the ability for developers to modify *device drivers* for use with their applications. Device drivers are critical software components used to define control sequences and special processing commands for handling system hardware (primarily terminals and printers).

No special device drivers are required to run applications under Windows versions of ProvideX. For UNIX/LinuX installations, ProvideX ships with a variety of pre-built device drivers (which should meet most user requirements). However, developers may need to make modifications in order to accommodate functionality that is not implemented in an existing driver, or a new driver may be required so that an application can take advantage of a device that is not currently listed.

Topics

In simple terms, device drivers are analogous to ProvideX subprograms (see **Called Procedures**, *p.81*); only, in this case, the program (device driver) is used exclusively to perform the following:

1. Define the type of device.

2. Define the command sequences needed to process mnemonics.

3. Define the input sequences for the various CTL values associated with the device.

Because device drivers are programs, a variety of functions can be performed within them. They can even be used to redirect the file being opened to another device or file, ask questions of the user, or issue operating system commands. In general, it can do anything that a normal ProvideX program can do, so long as nothing happens to effect the current state of the program which originally opened the device; e.g., a **BEGIN** statement would close all files.

When ProvideX first accesses a device (and there are no problems opening the device) it automatically calls the associated device driver, if specified. For terminals, the device driver is run at the startup of ProvideX. For printers, the device driver is run when data is sent to the printer. During execution of the device driver, the system variable **LFO** has the file number for the device.

Defining Devices

Depending on the associated device, device driver code will begin with one of the following directives:

DEFPRT (*chan*)*col*,*ln* ... *for a printer device*
or
DEFTTY (*chan*)*col*,*ln* ... *for a terminal device*

DEFTTY and **DEFPRT** tell Providex that the channel opened is either a terminal or a printer and will give you the **FIN( )** information back in either terminal format or printer format. It uses the column and line information to give you some starting values to use, if you query the **FIN( )**, **MXC( )** and **MXL( )** functions. For more information, see **DEFTTY** and **DEFPRT** in the *Language Reference, p.80*. Printer and terminal device drivers are further explained in the sections that follow.

*DEV and *UDEV Directories

The associated device driver file must be located in the ProvideX ***DEV** directory (e.g., `/pvx/lib/_dev`) and is referenced via ***DEV/***xxxxxxxx*, where *xxxxxxxx* is the name of the driver specified. Driver names are limited to 12 characters in length, lowercase only. If the driver cannot be located, this may result in an `Error #100 - No driver for terminal type or library missing.`

If you plan to customize one of the drivers supplied with ProvideX, it is not recommended that you re-save the driver using the same file name in ***DEV**. Unfortunately, the ProvideX installation process will automatically over-write these files and you will lose all changes.

For terminal drivers, you have the option of using the ***UDEV** directory (e.g., `/pvx/lib/_udev`) to store customized drivers. This option is not available for printer device drivers. For more information, see **Creating a Supplemental Device Driver**, *p.393*

Printer Devices

ProvideX device drivers for printers are used to consolidate mnemonics and (legacy) control sequences into a single file to be called from the main program whenever the target printer is opened.

A printer device can also be defined to the system by a logical device descriptor, or *link file*. This type of file is used in ProvideX to define the actual device path and device type. The use of a link file allows you to maintain a common printer name (***alias***) that will represent all possible print destinations from your application. Link files can be edited directly or maintained using the ProvideX utility ***UCL**. Printer device drivers are fully documented in *Chapter 7.* **Printing**.

Terminal Devices

Historically, the main user interface hardware of a computer (keyboard and video display) was called a terminal. However, this definition is not as common today with the advent of personal computers, GUI environments, and the variety of operating systems and interface devices available. While the term is still used on occasion, the "terminal" is usually associated with a system's command line shell or console-style environment; e.g., text terminal or terminal window.

Applications running on some older operating systems may require a terminal device driver to be configured for user access in this environment. As mentioned earlier, terminal device drivers are opened during the ProvideX startup. ProvideX uses the value assigned in the TERM environment variable to determine which device driver is used for the current terminal. Generally this is already set to an appropriate value by the operating system.

Driver Content

The following is an example of a terminal device driver. In general, terminal drivers are much like printer drivers in that they define the command sequences for the mnemonics, as well as the input sequences that the terminal generates for function and edit command keys.

```
0010 ! WYSE 50 -- Terminal driver
0020 DEFTTY (LFO)80,24
1000 ! 1000 - Mnemonics
1010 MNEMONIC (LFO)'@@'=ESC+"=\LB\CB"
1020 MNEMONIC (LFO)'BS'=$08$
1030 MNEMONIC (LFO)'CE'=ESC+"Y"
1040 MNEMONIC (LFO)'CH'=ESC+"= "
1050 MNEMONIC (LFO)'CL'=ESC+"T"
1060 MNEMONIC (LFO)'CS'=ESC+"+"
1070 MNEMONIC (LFO)'CF'=ESC+"&"+ESC+";"+ESC+"'"
1080 MNEMONIC (LFO)'DC'=ESC+"W"
1090 MNEMONIC (LFO)'IC'=ESC+"Q"
1100 MNEMONIC (LFO)'LD'=ESC+"R"
1110 MNEMONIC (LFO)'LI'=ESC+"E"
1120 MNEMONIC (LFO)'RB'=$07$
1130 MNEMONIC (LFO)'RM'=ESC+"`"+$0000$+ESC+"("+
1130:ESC+"H"+$03$+ESC+"G0"
1140 MNEMONIC (LFO)'SB'=ESC+")"
1150 MNEMONIC (LFO)'SF'=ESC+"("
1160 ! Define Attribute output table
1170 MNEMONIC (LFO)'AT'=ESC+"G\A"+"&"+$0F$+"T"+
1170:$10$+"0p4t2r6v8x<|:z>~"
1180 MNEMONIC (LFO)'GS'=ESC+"H"+$02$
1190 MNEMONIC (LFO)'GE'=ESC+"H"+$03$
1200 MNEMONIC (LFO)'GD'="2315:6849=0"
9000 ! 9000 - Load terminal control keys
9010 RUN "*TTY"
```

A terminal device driver is typically much larger than a printer device driver. This has to do with the fact that more mnemonics are required for a terminal than for a printer. While it is not necessary to define matches within a terminal driver for all ProvideX mnemonics, it is better to define as many as possible. For example if no **'CL'**

mnemonic is specified, ProvideX will output spaces in order to clear a line (this is not very efficient). The only mnemonic absolutely necessary is the internal mnemonic '@@' which is used to define how to position the cursor.

The **DEFTTY** command is also mandatory. As mentioned before, this directive indicates to ProvideX that the device is a terminal (not a printer). **DEFTTY** must contain two additional parameters: the default number of columns and the default number of lines.

About ***TTY**

The last line of any terminal device driver should be `RUN "*TTY"`. This utility performs additional terminal initialization for function keys, etc. It also checks the `lib/_udev` directory automatically to locate and **CALL** the supplemental terminal device driver (if found).

Creating a Supplemental Device Driver

As mentioned earlier, terminal device drivers supplied with ProvideX should not be modified directly. Instead, create a supplementary driver and store it in a directory called ***UDEV** (e.g., `/pvx/lib/_udev`). Follow the steps below:

1. If it does not already exist, create the ***UDEV** directory (e.g., `/pvx/lib/_udev`).
2. Create a new program. Remember, don't modify / over-write the original driver. Place only modified or additional content in the supplemental driver. This program should terminate using an **END** directive (do not use `RUN "*TTY"`).
3. Save the new program in ***UDEV** using the same file name as the original driver.

At runtime, ProvideX will start by calling the original driver. The ***TTY** program will automatically locate and **CALL** the supplemental version of the device driver from ***UDEV**. For example, if you need to make modifications to ***DEV/VT100**, you would create second file as ***UDEV/VT100** to include the changes – ***TTY** checks the ***UDEV** directory for the supplemental **VT100** driver immediately after it runs the original ***DEV/VT100** driver.

Creating a New Terminal Device Driver

If you need a device driver for a terminal type that is not already installed with ProvideX, a new driver may be created for this purpose. You can take a chance and save the file in the ***DEV** directory but, if ProvideX decides to support this terminal type in a future release, your new device driver may get over-written.

This risk can be avoided using a technique similar to creating a supplemental driver in the ***UDEV** directory. In this case, the "supplemental" file contains almost all of the driver content. Follow the steps below:

1. Create a new program that contains only two commands; i.e.,

```
DEFTTY 80,25
RUN "*TTY"
```

This will be the starting device driver. The maximum lines and columns must be set here, and cannot be changed in the supplemental program. Save it in ***DEV** (e.g., /pvx/lib/_dev).

2. If it does not already exist, create the ***UDEV** directory (e.g., /pvx/lib/_udev).
3. Create a new program to map the mnemonics and control sequences required for running the new device. This program should terminate using an **END** directive (do not use RUN "*TTY"). Save this file in ***UDEV** using the same name as the starting device driver (created in step 1).

At runtime, ProvideX will **CALL** the first driver, then immediately **CALL** the "supplemental" device driver from ***UDEV** to run the true driver content.

Note: Terminal display mnemonics are outlined in the sections that follow. A general discussion of mnemonics is provided in *Chapter 2.* **Language Elements**. For the complete list, see **Mnemonics** in the *Language Reference, p.575*.

Variable Mnemonic Data

Some mnemonics require dynamic information (variables) to generate specific output sequences (e.g., the line and column number defined in the **'@@'** mnemonic). There are 11 variables that may be included in the output sequence. Identifying these variables is done by specifying a \ *backslash* followed by a one character variable identifier and output formatting information.

Define Variable: **=ESC+"[***id_char***{***modifier***}***fmt_code***"**

Where:

id_char Identifier for the variable. Include the backslash in the syntax. Valid identifiers include:

\b	Bottom margin of window / scroll region
\h	Height of window / scroll region
\l	(lowercase L). Left margin of window / scroll region
\r	Right margin of window / scroll region
\t	Top margin of window / scroll region
\w	Width of window / scroll region
\A	Current attributes in binary
\B	Background colour index
\C	Current column
\F	Foreground colour index
\L	Current line

fmt_code The variables above must be followed by one of the following output format character codes:

2	Output is two-byte ASCII
3	Output is three-byte ASCII
a	Output is ASCII plain text number (base 0)
b	Output is single byte binary (base 0)
A	Output is ASCII plain text number (base 1)
B	Output is single-byte binary (base 0x20)
T	Table output. Following T, the next byte must contain the number of bytes in the table. The table must follow that in the output sequence. If the number of bytes exceeds table length, no output is generated.

modifier Between the output format code and variable you can optionally specify one or more modifiers of the variable value. The list below shows valid modifiers and their associated functions:

+	(Plus sign)	Add the value of the next byte
-	(Minus sign)	Subtract the value of the byte
&	(Ampersand)	AND the value of the next byte
^	(Caret)	XOR the value of the next byte
|	(Pipe)	OR the value of the next byte

'@@' *Mnemonic - Cursor Position*

This mnemonic must be defined for a terminal. It is used by ProvideX to position the cursor within the screen. Typically the command sequence for this mnemonic will reference the mnemonic variables \L and \C.

If the terminal supports the ability to establish scroll region, you may specify a **'WX'** mnemonic. This mnemonic will be output by ProvideX whenever a **'SCROLL'** or **'WINDOW'** related mnemonic is executed. The **'WX'** mnemonic command sequence must establish a new top/bottom line on the screen and restrict all displays within this region. If the terminal does not support a scroll region (as is usually the case) ProvideX will emulate this functionality.

'RM' - *Reset Mode*

This mnemonic should define the required command sequence to restore the terminal to foreground mode, line wrap enabled, no blinking, no underscore, no reverse, scroll mode enabled, and white on black text.

'F*n***'** *and* **'B***n***'** - *Colour Attributes*

The mnemonics **'F0'** through **'F7'** and **'B0'** through **'B7'** are used to define the command sequences to set terminal foreground and background colours respectively.

'AT' - *Visual Attributes*

Since many terminals vary in the command sequences required to enable and disable visual attributes (underscore, blink, etc.), ProvideX provides special processing for attribute mnemonic output.

Generally when the user program issues a mnemonic to enable a visual attribute, ProvideX will output the command sequence associated with the specified mnemonic. When the program issues the mnemonic to disable the attribute, its command sequence is output. Unfortunately not all terminals have command sequences to disable specific attributes. In this case ProvideX will issue a **'RM'** mnemonic, then re-send all other necessary command sequences needed to set the terminal to the proper mode.

Some terminals require that visual attributes be merged into a single attribute character. To handle these types of terminals, ProvideX uses the pseudo mnemonic **'AT'**. If the **'AT'** mnemonic is defined, ProvideX will output it in order to satisfy all visual attribute changes. The command sequence for the **'AT'** mnemonic can use the parameter '\A' (current attributes in binary) in order to determine the desired attribute.

The attribute values (\A) is comprised of a binary value representing the various attribute options. These are: 1 - *Bold/Foreground*, 2 - *Inverse video*, 4 - *Blink*, 8 - *Underscore*, 16 - *Graphics character.*

'C*n*' - *Cursor Display Mnemonics*

Three mnemonics are used to control the display of the terminal cursor: **'C0'** - *Hide cursor command sequence*, **'C1'** - *Regular cursor display sequence*, **'C2'** - *Insert mode cursor display sequence.*

'GD' - *Graphic Characters*

The mnemonic **'GD'** is used to define the 11 characters that can be used for text mode line-drawing operations. The standard line-drawing characters are A-K (and for compatibility 0-9 and colon). See **'GD'** in the *Language Reference, p.609*.

'CP' *and* 'SP' - *Screen Size*

When specifying a mnemonic that will change the size of the screen (such as **'CP'** for condensed print) you must provide new screen dimensions with the **MNEMONIC** directive. For example a WYSE 60 supports the following command sequence:

```
MNEMONIC (LF0)'CP'=ESC+"`;":132,25
```

The parameters 132,25 tell ProvideX what the new dimensions of the screen are.

Flexible Text Fonts. Applications can change the text font used by the screen in Windows by simply defining the font and the logical screen size as a mnemonic; e.g.,

```
MNEMONIC (0) 'CP' = "Courier New,-8":120,40
MNEMONIC (0) 'SP' = "*":80,25
```

Once the above mnemonics have been set up, issuing a `PRINT 'CP'` will change the screen font to *Courier New* with a size of 8 points. `PRINT 'SP'` will restore the screen font to the default.

Note: The text plane font must always be a fixed font.

Specifying Conversion Tables

Two special mnemonics are also supported for character translation:

'*I' defines a 256 byte table of characters. As ProvideX receives each character of input, it is used as an offset into this table. The byte found at the offset is the actual character value returned to the program.

'*O' defines a 256 byte table which is used during output. If defined, each character output is used to offset into the table to get the true character to send to the device.

Defining Control Key Values

Within the device driver the **DEFCTL** directive can be used to define the terminal input sequences for the various function and editing keys. To simplify the process of defining these keys, the ProvideX utility ***TTY** is called. It reads the system file `"*KYBRD.CFG"` which contains the key configuration information and will automatically load them.

The utility ***UCK** is provided to allow the user to define and change the keyboard map based on the terminal type and user ID.

Handling Images and Icons

As described in earlier sections of the documentation, the following techniques, allow you can retrieve and incorporate images and icons for a variety of purposes in your GUI-based applications:

- Display images via the **'PICTURE'** mnemonic (**Display Objects**, *p.201*).
- Associate images for GUI controls (**Control Objects**, *p.150*).
- Specify icons to customize the upper left corner of dialogue windows via the **'OPTION'** mnemonic or INI file setting, `ICON=` .

Note: For more information on handling images in NOMADS, refer to the *ProvideX NOMADS Reference*.

This section discusses some the general concepts and ProvideX functionality involving the use of images in ProvideX applications. For more information on creating graphical controls and other graphical objects, see *Chapter 6.* **Graphical User Interfaces**.

Internal vs External Images

Images are recognized as ***internal*** in ProvideX if they have a leading exclamation mark (!) in their filenames; e.g.,

```
PRINT 'PICTURE'(220,210,600,500,"!Binoculars",2)
```

Internal images can be accessed if they are embedded within the ProvideX executable itself, supplied in an associated resource library, or exist in a file located in the `*BMP` directory. Use the **'PICTURE'** mnemonic to return a list of available internal images (including the set of standard OS icons); e.g.,

```
X$='PICTURE'(*)
PRINT X$
```

To access images that are ***external*** to ProvideX, specify the path and filename instead of the exclamation mark; e.g.,

```
PRINT 'PICTURE'(1,1,100,100,"C:\WINDOWS\CLOUDS.BMP,T",0)
```

Note: The names of image files placed in the `*BMP` directory should be in lowercase.

Recognized File Types

By default, only `bmp` or `ico` image formats are supported automatically in ProvideX. By installing the ***Multiple Image Support*** add-on package, you can extend the list of supported graphic images to include several raster and vector graphic file formats, including `jpg, ico, gif, tiff, png, pcx, pax, wmf, emf, apm`, and `tga`.

This product may require a separately-purchased activation key apart from your initial ProvideX activation. Contact your local ProvideX dealer / distributor or visit www.pvx.com for complete product information and licensing.

Sizing and Placement

For some controls (i.e., **LIST_BOX**, **MENU_BAR**, and **POPUP_MENU**) the dimensions of the first bitmap / icon defined in a list of elements is assumed to be the default size and placement for all images in the control. See **Control Objects**, *p.150*.

Enhanced Icons

ProvideX is able to use `ico` files that contain multiple icons, different sizes, and different colour formats. Icons can also be retrieved from other file types, including `.exe`, `.dll`, `.ocx`, `.drv`, `.cpl`, `.scr` and `.icl` (icon libraries). The file types are described below.

The following optional syntax items may be applied in the **'PICTURE'** and **'OPTION'** mnemonics, as well as in control object directives to define the location and / or attributes of an icon to be retrieved from an enhanced icon source:

[*filename*] [*@resourcename* | *@resourcenum*] [**%***Size*] [**,T** | **,G**]

Where:

filename	File containing icon(s). If no filename is given, then the currently loaded resource library is searched.
@resourcename	String name of the specific resource identifying the icon. If no resource is specified, the first icon in the file will be loaded.
@resourcenum	Specific resource number identifying the icon. If no resource is specified, the first icon in the file will be loaded.
%size	1 to 3-digit number of pixels representing the X or Y size of the icon displayed. This size is used for both X and Y axes; e.g., `%16` displays the icon in 16 x 16 pixels. If no *%size* is given, then the first format available for the icon is used. If `%0` is specified, then the default OS icon size is used.
T *or* **G**	Transparency substitution indicator where **T** means use upper left most pixel colour or **G** means use colour RGB: 192,192,192.

Note: Syntax options (if applied) must appear in the above defined order.

Standard ProvideX search rules apply to the filenames. A leading exclamation will also search the `*BMP` directory. Colour depth selection (16, 256 or 24-bit) is chosen automatically by the OS based on the user's current video card colour depth and the colour depth of the icon available within the file.

Examples:

```
C:\Pvx\Pvx.ico%16,T
C:\Windows\System32\Shell32.dll@137%32,T
C:\Windows\System32\Shell32.dll@137%32,T
!myico.ico%48
!myico.ico%48,G
@ProvideX,T
<path>\pvxwin32.exe@ProvideX%16,T
```

Icon File Types

ProvideX accepts icons from the following file extensions:

`.ico` Does not support loading by resource name / resource number. You must specify the filename (with or without path), ending in .ICO.

`.exe`, `.dll`, `.ocx`, `.drv`, `.cpl`, `.scr` (and any file type that the MS Windows API allows a `LoadImage` from). You must specify a resource name or number.

`.icl` Icon Library (commonly used by Icon editing tools). You must specify a resource number only (0 based). It ignores icon size specifications. If you intend to use icons from a .ICL, we recommend that you convert it to a DLL so that you may take advantage of any size specifications.

Enhanced Icons in INI Files

The `[Config]` section of your INI file, allows for an `ICON=` using any of the above syntax for specifying the icon. If no filename is given or there is no leading @ sign, then the name given is assumed to be a resource name from either the currently loaded resource library (if any) or from the ProvideX executable; e.g.,

```
[Config]
Icon=myname ! Would be the icon "myname" in a resource library
Icon=@myname ! Same as above
Icon=mydll.dll@myapp
```

Note: Specifying an icon in a INI file can impact your user license count for ProvideX, as the icon name will be used for the Window Class Name, and ProvideX sessions with different Class Names do not share their user licenses.

Enhanced Icons in Objects

When using enhanced icons in objects, the filename and syntax must be enclosed in curly braces; e.g.,

```
BUTTON 10,@(40,2,8,3)="{pvxwin32.exe@ProvideX%32,T}"
BUTTON 10,@(40,2,10,2)="{@90w%32,T}"
```

Note: Transparency effects are currently not supported in menus or popup menus.

Index

F

G

S