
PEG

Portable Embedded GUI

**Programming
And
Reference Manual**

**Eighth Printing
November 2002**

© Copyright 1998,1999,2000,2001,2002 Swell Software Inc., all rights reserved.

© Copyright 1998,1999,2000,2001,2002

Swell Software, Inc.
2321 Water Street Suite E
Port Huron, MI 48060
PH: (810) 982-5955
FAX: (810) 982-5949

info@swellsoftware.com

All rights reserved.

PEG is a registered trademark of Swell Software, Inc.

Table Of Contents

<i>Forward</i>	<i>1</i>
<i>Introduction</i>	<i>4</i>
<i>Chapter 1: Building the PEG Library</i>	<i>10</i>
Build Options	12
<i>Chapter 2: Common Terms and Concepts</i>	<i>43</i>
Notes to users new to C++	44
Windowing Interface Terminology	49
<i>Chapter 3: PEG Execution Model</i>	<i>53</i>
Overview	54
Program Startup	56
PegPresentationManager	64
<i>Chapter 4: PegMessageQueue</i>	<i>73</i>
Signals	84
<i>Chapter 5: PegScreen</i>	<i>89</i>
Video Controllers	91
<i>Chapter 6: Fundamental Data Types</i>	<i>118</i>
PegPoint	119
PegRect	120
PegColor	123
PegMessage	126
PegTimer	127
PegFont	130
PegCapture	136

Chapter 7: The Mighty Thing	137
PegThing Members	139
Public Inline Functions	158
Public Data Members	160
Using PegThing Member Functions	161
Chapter 8: Programming with PEG	178
Rules Of Memory Ownership	182
Creating PegThings	184
Deleting/Removing PEG Things	186
Object Boundries:	195
Viewports	197
Programming Examples	199
Example 1- Getting Started	201
Example2- Using PegTimer	203
Example 3: More Message Handling and Signals	207
Example 4- Deriving a Custom Window	215
Chapter 9: PEG Multitasking	228
Why Different Models?	230
Graphics Server Model	234
MULTITHREAD Model	241
Window Execution Under Secondary Thread	245
Multithread Execution on the Win32 Development Platform	248
OS Porting Advanced Topics	261
Chapter 10: PegFontCapture	267
Chapter 11: PEG Image Convert	280

<i>Chapter 12 WindowBuilder</i>	298
The Project Window	303
Project Window Menu Commands	304
The Target Window	321
The String Table Editor	330
Source Code Generation	336
Example 1: Creating a simple PegDialog window	341
<i>Appendix A: PEG Directory Structure</i>	364

Forward

We at Swell Software thank you for choosing PEG!

The authors of PEG are first and foremost embedded systems programmers like yourself. With extensive experience developing software for closed-loop servo robotics, industrial control systems, measurement and monitoring equipment, and consumer electronics, we most likely share many common experiences with you. We believe that this kinship will allow us to anticipate your requirements and to provide you with the tools and support you need as you develop your next product.

In addition to the PEG development package, Swell Software provides consulting and contract programming services to clients in a wide diversity of industries. These services range from one-day on-site evaluations and tutorials to complete screen prototyping and development. We encourage you to take advantage of these services as early as possible in your project cycle. If you have purchased or are evaluating the PEG library, you can of course contact us at any time via phone or email to answer your technical questions

PEG is currently being used in projects around the globe, yet PEG also continues to grow and improve. Minor updates are often published every few weeks as we incorporate suggestions and requests from PEG users. We encourage you to provide us with feedback as you begin using PEG in your application development. If you find something that just isn't working for you, or we are missing something that you feel is a requirement, please do not hesitate to let us know. We will make every effort to satisfy your request and provide you with an updated release in as short a time frame as possible. We are committed to making your embedded development effort an overwhelming success!

Introduction

How this manual is organized

This manual is organized such that the first section explains the configuration and build procedures you will need to know in order to begin using the PEG library. This allows you to be up and running and experimenting with the library very quickly.

The middle section provides an ‘under the hood’ view of PEG library internals and introduces basic concepts that are needed to fully understand how PEG works. You will need to read and understand this material before you begin serious development of your application level software. This is followed by descriptions of the fundamental PEG classes. These descriptions contain many working examples that will prove valuable to you as you begin writing your own system software.

Next, our supporting cast of utility programs is described. These include PegFontCapture, PegImageConvert, and PegWindowBuilder. The appendices describe the PEG installation directories and the example programs.

While this manual provides extensive information about the fundamental PEG classes, the full class reference provided in HTML format. We believe that the HTML format class reference is more convenient to use on a day to day basis than the printed manual. This approach also works well in that as you read this manual you are not overloaded (no pun intended!) with member function names and descriptions. Instead, we encourage you to first concentrate on obtaining a high-level understanding of how PEG works. Later, as you begin working on your system software, you will probably want to keep the HTML class reference open at all times. The class reference begins with the file `\peg\manual\index.htm`. Select the “Reference Manual” link on this page to open the PEG Class Reference.

This programming manual is also provided in online format, and the online format often contains last minute changes or additions that you will not find here. These additions are found in the “Manual Supplement” section of the online manual. The

Introduction

online manual is found by selecting the “Programming Manual” link on the above mentioned index page.

Introduction

Historically speaking, graphical user interfaces have almost exclusively been the domain of desktop personal computers. This has been the result of two main factors: the cost of graphical display hardware and the lack of GUI software suitable for use in real-time systems.

In the area of industrial control systems, there have been attempts at providing graphical presentations, but these have been cumbersome at best and terribly expensive as well. These types of systems have typically avoided the use of mainstream video output devices, and opted instead for very expensive and functionally limited industrial display terminals.

Today, this attitude has changed to the point where it is very common for an embedded system to contain many of the very same hardware components found in a desktop computer system. This makes sense strategically because it allows the inventor of an embedded product to leverage the sales volume and pricing of the components sold primarily for desktop computer use. The result is that the cost of including graphical display hardware in an embedded product has declined significantly over the past few years. A wide variety of LCD display panels, VGA display panels, video controller chips and high-performance CPUs capable of driving a graphical interface are now available.

Unfortunately, the software side of the equation has not advanced nearly as quickly. Until now there has been no graphical interface solution that is small enough and portable enough for an embedded system while at the same time providing a modern and professional appearance. There have been previous attempts at meeting this need, but so far these attempts have missed the mark.

The alternative solutions that provide a modern, full-featured interface have all been derived from desktop computing environments, and carry along with them

years of acquired baggage. These solutions impose very high hardware costs on your system, and even higher costs in terms of the man-hours required to successfully integrate these large software packages with your real-time software. This of course assumes that you have the time and expertise required to actually build a working system with one of these products. We have seen more than one project descend into a never-ending abyss of delays, technical setbacks, and finally failure caused by trying to force-fit software that was not intended for real-time systems.

We believe that you deserve a better solution!

What PEG IS

PEG is an acronym for Portable Embedded GUI. We chose this name because we believe it accurately reflects the design and motivation that went into the creation of our development package.

PEG is Portable

We have designed our software to be portable to any target hardware that is capable of graphical output. PEG does not expect or require any underlying software components in order to do its job. If you have a C++ compiler and hardware capable of pixel-addressed graphical output, you can run PEG.

PEG is Embedded

This statement is rather vague, because it means so many different things to different people. The bottom line is that ***PEG is, and will always be, targeted only at real-time embedded systems***. This distinction is so important that we felt it should be included in the name of our library.

Introduction

PEG is GUI

The PEG class library provides the building blocks for a powerful and extensible graphical user interface. Users of PEG will find that they can create a graphical presentation rivaling anything on the market today. Extensive thought and research have gone into the design of our product to insure that you are receiving a library that is fully capable of supporting all of the advanced GUI features you need today, while also accommodating future enhancements. Advanced clipping techniques, font support, graphic image support, and smart object methodologies are incorporated in our library. We are confident that the internal design of the library is such that PEG can grow and advance for years to come while building on the existing foundation.

In addition to the class library itself, PEG provides all of the other tools, documentation, and support you will need to construct a custom graphical interface for your project. This includes utilities for generating graphical fonts (***PegFontCapture***), processing, optimizing, and compressing graphical images (***PegImageConvert***), and ***PegWindowBuilder***, a RAD prototyping tool for use with PEG. With the class library and related tools, PEG without question provides the most powerful, professional, and complete GUI solution available to real-time embedded system developers.

What PEG is NOT

The large software companies are today providing software that is intended to be a 'one size fits all' solution. This has led to some confusion among many developers concerning what a GUI library should do, what it should not do, and what components are required to build a working system. We believe it is worthwhile addressing these questions up front to insure that we are all working from the same starting point.

PEG is not an operating system. PEG provides no code for task switching, memory management, resource management, or inter-task communication. Contrary to popular belief, the desktop windowing environments are not part of the operating system either. This should be obvious from the fact that the graphical environment can be significantly updated without improving or otherwise changing the underlying OS. In order to build a real-time multitasking system using PEG, you will also have to incorporate an operating system kernel. PEG is already fully integrated with most of the leading real-time operating systems available today. PEG can easily be integrated with other operating systems as well, and PEG can run standalone if multitasking is not required for your application.

PEG is not an application program. The PEG library, by itself, will provide an end user with absolutely zero in terms of useful interaction or information display. It is your job to create the windows, dialogs, and other objects that will be used to retrieve input from and display information to the end user. Of course, the whole point to using PEG is that our library provides the tools and components that make creating your application level interface a manageable task.

Finally, PEG is not a PC-library. While PEG does support common PC development environments as a matter of convenience and productivity, the goal of PEG is to provide a full graphical interface solution to real-time embedded systems developers. This solution includes the software, utilities, documentation and support required to make your embedded development effort a success, regardless of the final hardware implementation.

Where PEG is going

Over the near term, the core PEG library will continue to grow and improve in terms of the native control types that are available and the flexibility of those controls. Over the longer term, we plan to add entirely new groups of object types that we feel would be useful for embedded systems. Of course, the long-range development list also depends on input we receive from you, the developer using

PEG. In any event, the basic library will retain the ability of allowing you to remove unwanted components in order to build a system that exactly fits your size and performance requirements.

While PEG is can be ported to nearly any hardware configuration capable of graphical output, this effort can seem confusing to a new user. For this reason, we have undertaken to add reference platforms for many common hardware configurations. This allows PEG users to begin running PEG immediately on hardware which is similar if not identical to the final target. Currently reference platforms have been completed or are in development for x86, ARM7, StrongARM, MPC823, ColdFire, DragonBall, C167, and MC68332 platforms. Additional reference platforms will be added as evaluation hardware platforms become available for other CPU types that are popular in the embedded community.

WindowBuilder, our newest and most powerful support utility, is continually being enhanced to meet the needs of PEG users. Fully integrated support for Unicode and efficient code generation through the use of container classes are new features as of this manual printing.

Library Updates

Library updates are posted on the Swell Software www site roughly every 90 days. If you are a PEG customer, you are entitled to a minimum of six months of technical support and library updates. The User Support page on the Swell Software website is password protected. If you do not know the password please email support@swellsoftware.com and request the current password.

The user support page lists the most recent changes or library enhancements, and also allows you do download the latest release of the PEG library source code, supporting utilities, and documentation. This page is undergoing enhancement to include a Frequently Asked Questions page and useful contributions from PEG users.

Chapter 1: Building the PEG Library

Building the library is very simple, although you may have to create a make file specific to your build tools. All of the required PEG source files are in the directory `\peg\source`. Likewise, the header files are in the directory `\peg\include`. Building the library is not quite as simple as “Compile all the source files and link ‘em together”, but this statement is not far from the mark.

This chapter begins by describing the library configuration flags, which are flags used to include or exclude various components and features of PEG. You may need to modify at least a few of these flags before you build the library. The library source files are then listed, along with lists of source files required for a minimal, typical, and full build of the library. The last section of this chapter describes the pre-configured library make files that are included in your PEG distribution.

Library Code Size

The code size of the library can vary dramatically depending on which resources your application is using and how you have configured the library. A full build of the PEG class library currently generates approximately 175K of code when all options and modules are included. The size of the generated code also varies slightly depending on the compiler, optimization levels, and CPU being used. The total code size of the PEG library continues to grow over time as various new classes and control types are introduced. This does NOT mean that your system code size will increase when /if you update to a newer version of the PEG library, since only those classes which you are actually using are linked into your system software.

Since not all PEG classes and features are used by a typical application, we find it more useful to refer to the typical footprint of the library on an embedded target. The typical footprint is a measure of the ROM or FLASH space actually used by the PEG library when linked with your application modules. Our measure of the

Building the PEG Library

typical footprint is derived from several real-world embedded systems using the PEG library. The typical footprint of the library is between 80K and 140K bytes, and can be as small as 64K Bytes, depending on which features of the library your application is using.

There are developers who prefer to include in the library only what is actually used, and others who place everything in the library and depend on the linker to extract only what is needed. The second approach is taken by the pre-configured make files, however the following information will allow you to build the library any way you prefer.

File Naming Conventions

Every effort has been made to insure that all PEG source files and documentation files are named in a consistent manner to avoid problem encountered when moving between Windows, X11, or DOS development environments. All PEG source files use the standard DOS 8.3 file name format, i.e. long file names are NOT used. Likewise, all source file names use lower-case letters exclusively to avoid case differences when running on Unix or Linux systems.

Finally, the PEG installation utility converts CR+LF sequences to the standard Unix CR line end when installing on Unix/Linux/X11 development hosts.

Build Options

Before building the library, it is necessary to set certain compiler directives contained in the PEG configuration file named `pconfig.hpp`. These directives determine the target environment for the library, along with other options such as input device drivers, screen drivers, screen resolution, clipping algorithm, and tasking model.

Throughout the PEG programming and reference manuals, we will refer to the PEG library configuration flags and how they affect library operation. These configuration flags are simply `#defines`, usually in the file `\peg\include\pconfig.hpp`. We use the terms “**turned on**” or “**turned off**” when referring to these flags. The term “turned on” means that the `#define` is active, meaning that it has not been commented out of the header file. The term “turned off” means that the configuration flag has been commented out of the header file. We suggest that you comment out configuration flags that you want to turn off, rather than deleting them entirely, to make it easier to go back and change your build configuration at a later date.

It is possible that additional configuration flags have been added to PEG that are not contained in this manual. Please read the notes at the top of `\peg\include\pconfig.hpp` for a full description of each flag and explanations of how these flags affect the resulting library. As delivered, the header files are usually configured to build PEG as a Win32 standalone static library. This configuration allows you to quickly begin using and experimenting with PEG, without being concerned about target specific configuration or multitasking complexities.

It is **not** necessary to fully tune your PEG configuration before building the library for the first time. If you want to begin working as quickly as possible, simply select one of the pre-configured target platforms and leave all other configuration flags at their default settings. You can return and modify the PEG configuration flags at any time and re-build the library.

Target Platform

PEG as delivered supports standalone operation or running under Win32, DOS, or any one of several leading real-time operating systems. Using PEG with other a different CPU or RTOS than those supported “out of the box”, or target hardware requires minor modification to the I/O drivers and RTOS interface functions. There are comments at the top of the file `\peg\include\pconfig.hpp` that explain which options to turn on and off depending on which development environment you want to use.

PEG is usually configured for stand-alone Win32 operation upon delivery. We suggest that you initially build PEG for either the DOS, Win32, or X11 standalone development environments as these environments allow you to quickly begin using PEG and creating your user interface software. The DOS configuration is also generally very easy to port to a custom target environment, and should be used as a starting point if you will be using PEG stand-alone or with an RTOS that is not yet supported.

To build the library for a DOS real-mode development, turn on the definition “PEGDOS”.

To build the library for Win32 development, turn on the definition “PEGWIN32” (default).

To build the library for an integrated RTOS, find the definition that corresponds to your RTOS and turn on that definition.

If you are using an RTOS or target which is not yet directly supported by PEG, it is usually easiest to simply build PEG for standalone operation and run PEG as a single task. This can be enhanced at any time to support true multi-tasking operation as described in later chapters. To build PEG for standalone operation, you should turn on the definition “`#define PEGDOS`” and turn off all other targets

Building the PEG Library

before building the library. This is a convenient starting point when using PEG with an RTOS that is not yet specifically supported by PEG.

The target environment is the only setting that is required initially. The remainder of the configuration flags are used to ‘fine tune’ your library configuration, and can you can skip the remainder of this section if you simply want to get things up and running quickly. After you have PEG running, you can return and adjust the remaining configuration flags at any time.

Screen Driver

There are several screen drivers provided with the PEG library, and there are additional options for some drivers. If you are building PEG for Win32, the correct screen driver options are automatically determined as soon as you select the “PEGWIN32” target. The same is true if you are building PEG for use under DOS.

PEG provides screen driver interface classes for Win32, generic VGA, VESA SuperVGA, 1-bpp (monochrome) LCD of any resolution, 2-bpp (4-color) LCD of any resolution, 4-bpp (16 color) LCD of any resolution, and 8-bpp (256 color) of any resolution, 16-bpp (65535 colors) of any resolution, 24-bpp (TrueColor) of any resolution, rotated drivers for each color depth and resolution, and fully customized drivers for a variety of popular chipsets.

When you first build PEG, you should probably use either the Win32, X11, or generic VGA screen drivers. While these drivers do not offer the best performance, they allow you to get PEG up and running quickly.

If you are building for Win32 development, you need to include the file `w32scrn.cpp` in your make file. This is the PEG screen driver for use under Windows. If you are building for x86 standalone development, you need to include the file `vgascrn.cpp` in your makefile. This is the generic VGA screen driver. If you are building for development under the X11 development environment, you should use the `x11scrn.cpp` screen driver.

The generic VGA screen driver can be configured to use the PC BIOS (if available) for video configuration, or to configure the VGA registers directly. This option is turned on or off via the “USE_BIOS” definition, which is contained in the screen driver header file “vgasrn.hpp”. If you are initially building PEG for DOS real-mode development, the USE_BIOS definition should be turned on. For all other targets or protected mode operation, USE_BIOS must be turned off. This forces the generic VGA screen driver to directly configure the controller register set without using the PC BIOS.

The pconfig.hpp header file contains the definition **#define PSCREEN**, which must be set to equal the header file corresponding to the screen driver you are using. The value of PSCREEN defaults to “w32scrn.hpp” if building for the Win32 environment, “x11scrn.hpp” if building for the X11 environment. For all other targets PSCREEN defaults to “vgasrn.hpp”, which is the header file for the generic VGA screen driver. Full descriptions of each of the provided PEG screen interface classes are contained in chapter 5, *PegScreen*.

Keyboard or Keypad Input

Keyboard input drivers for Win32 and DOS are automatically configured by selecting one of these target platforms. Keyboard input drivers for the supported RTOS environments are also configured automatically by selecting correct RTOS target.

The definition “PEG_KEYBOARD_SUPPORT”, in the file `\peg\include\pconfig.hpp`, determines whether or not the library will internally include the support and message processing functions required for standard keyboard input. The library code size can be slightly reduced by turning off this definition for targets that do not require support for a standard keyboard. If your target will be using a keyboard or keypad input device of some type, you should turn on PEG_KEYBOARD_SUPPORT. If your final target will be using a touch screen or softkeys exclusively, you should turn off this flag.

Building the PEG Library

Further information regarding keyboard or keypad input handling methods in PEG can be found in chapter 3 of this manual. Contact Swell Software if you need assistance with configuring keyboard input for your target hardware platform.

Focus Indicators

When operating with a keyboard or keypad input device, it is usually a requirement that various PEG control types draw themselves differently when they have keyboard input focus. This provides a visual queue to the end user as to which control is currently active. When operating with a touch screen or mouse, drawing focus indicators is generally not required and therefore it reduces CPU overhead and library code size to eliminate this function. The configuration flag `PEG_DRAW_FOCUS` determines whether or not PEG controls draw focus indications as they are selected. `PEG_DRAW_FOCUS` is turned on by default when `PEG_KEYBOARD_SUPPORT` is defined, otherwise `PEG_DRAW_FOCUS` defaults to being turned off. However, this configuration flag is completely independent and can be modified to preference regardless of the input devices your target is actually using.

Mouse Input

PEG is delivered with mouse input drivers for the supported development environments. If you have selected the `PEGDOS`, `PEGWIN32` or `PEGX11` targets, mouse support routines are automatically configured.

For x86 protected mode RTOS environments, PEG provides direct hardware mouse interface drivers for PC compatible development stations. If you are using an RTOS in x86 protected mode, you will need to select the correct mouse configuration. Refer to the notes provided with your RTOS integration package for more information about configuring the mouse input support routines.

On all platforms, the definition `PEG_MOUSE_SUPPORT` (contained in the file `\peg\include\pconfig.hpp`) can be used to enable or disable internal support for

mouse input. If your target system is not using mouse input, turning off `PEG_MOUSE_SUPPORT` will slightly reduce library code size. Turning off `PEG_MOUSE_SUPPORT` also removes several system bitmaps used to draw the mouse pointer on the screen, further reducing library size.

Further information regarding Mouse input can be found in chapter 3 of this manual.

Touch Screen Input

The definition `PEG_TOUCH_SUPPORT` configures the library to support touch-screen input. This is very similar to mouse input, although a few controls work slightly differently when `PEG_TOUCH_SUPPORT` is defined. Specifically, the library does not require "pointer-move" messages to be received in order to run correctly when `PEG_TOUCH_SUPPORT` is turned on. In most cases, you would either turn on `PEG_MOUSE_SUPPORT` or `PEG_TOUCH_SUPPORT`, but not both.

UNICODE

The PEG library can be built to support 16-bit character encoding, also known as Unicode. To enable support for UNICODE, the definition `PEG_UNICODE` should be turned on before building the library.

Note that some compilers provide intrinsic support for 16-bit string manipulation in their respective run-time libraries, and others do not. For this reason, the definition `PEG_STRLIB` (below) is often also required when UNICODE support is enabled.

Please refer to chapters 8, 10, and 12 for more information on how to create multi-lingual applications using PEG.

The default setting is to turn off `PEG_UNICODE`.

Building the PEG Library

String Library

The PEG library requires a small set of string manipulation functions. These are almost always provided by the run-time libraries that are included with your compiler. However, many compilers and associated run-time libraries do not support 16-bit character encoding. In addition, a few run-time libraries force you to link in or work around unneeded functions when you use the run-time library provided with the compiler.

For these reasons, PEG provides a limited sub-set of the standard string manipulation functions (such as `strcmp`, `strcpy`, `strlen`, etc.). The provided functions include all functions required by PEG, thus eliminating the need to link in the compiler run-time library in order to run PEG.

If you are using your compiler's run-time library for other reasons, you probably want to disable the PEG string manipulation functions as most compilers provide highly optimized versions of these functions.

The PEG string functions are included by turning on the definition `PEG_STRLIB` in the header file `\peg\include\peg.hpp`. When this definition is turned on, the header file `\peg\include\pegtypes.hpp` defines replacements for each of the supported string functions. The available string functions are listed near the top of this header file.

The default setting of `PEG_STRLIB` is turned on if `PEG_UNICODE` is turned on; otherwise it is turned off.

Clipping Style

PEG supports multiple levels of object clipping. This is described in later chapters. When you are initially using PEG, always include the line in `peg.hpp` that reads:


```
#define PEG_FULL_CLIPPING
```

On your final target, you can disable `PEG_FULL_CLIPPING` if your application screens are very simple and for performance reasons you are concerned about improving software overhead. When `PEG_FULL_CLIPPING` is disabled, only the window that has focus is allowed to update the screen. This algorithm works well for small screens which do not ordinarily have multiple overlapping windows.

Tasking Model

PEG supports three *tasking models*, introduced in chapter 4, “PEG Execution and Internals” and described in detail in chapter 10, “PEG Multitasking”. The term *tasking model* refers to the mechanisms PEG has in place to allow multiple GUI tasks to operate in parallel, or possibly even providing support for multiple processors and remote processes.

The default tasking model is `SINGLE THREADED`. This is the model used for standalone operation such as running under DOS, and can also be used in a multi-tasking environment where only one task or process has access to the GUI interface. The `SINGLE THREADED` model provides the lowest overhead and is the simplest supported tasking model. This model can also be used as a starting point when running with an unsupported RTOS.

The next level of tasking support is the `PEG_MULTITHREAD` model. Under this model any number of tasks or processes may directly create, display, and modify user interface elements. This model is selected by turning on the definition “`PEG_MULTITHREAD`”. The `PEG_MULTITHREAD` execution model is the default model when PEG is configured for a supported real-time operating system. If you are using an RTOS for which PEG has been integrated, you should find the the definition `PEG_MULTITHREAD` is already turned on when you build the PEG class library for your RTOS.

The final tasking model is the *PRESS* model. *PRESS* stands for **PEG Remote Screen Server**. This model was first introduced to support protected, virtual address

Building the PEG Library

space environments such as multiple processes running under Unix or Linux. This model has also been used to support remote processes or multi-processor environment. Under this model, all screen drawing is performed by the Screen Server. Application processes do not directly draw to the display buffer, but instead send all drawing commands to the remote drawing server. The PRESS tasking model is selected by turning on the definition `PEG_BUILD_PRESS`.

Color Depth

PEG runs most efficiently and produces the smallest footprint when the output color depth is determined at compile time. This is done by defining `PEG_NUM_COLORS` to equal the number of colors supported on the target system. The alternative is to determine the number of output colors at run-time.

If the output color depth is to be determined at run time, the definition `PEG_NUM_COLORS` should not be defined, and the definition `PEG_RUNTIME_COLOR_CHECK` should be defined. This instructs the PEG classes to compile versions of the object drawing functions that adapt 'on the fly' to the active color depth.

If `PEG_RUNTIME_COLOR_CHECK` is defined, PEG includes drawing routines for all color depths. This produces a slightly larger library size, and introduces a few run-time function calls that are not required if `PEG_NUM_COLORS` is defined. If you know at compile time which screen driver you will be using on your target, define `PEG_NUM_COLORS`. If you will be including multiple screen drivers in your system software, and loading the correct one based on a power-up hardware check, define `PEG_RUNTIME_COLOR_CHECK`.

As delivered PEG is configured for either 16 color output screens (generic VGA) or 256 output colors (Win32). This can be modified by adjusting the `#define PEG_NUM_COLORS` contained in the file `\peg\include\peg.hpp` and using a different screen driver. The appearance of several PEG control types, along with

the default object colors, are modified depending on the number of colors supported on the target system.

PEG as delivered supports color depths of 2, 4, 16, 256, 65535, and 24-bit RGB color. If you are running in 2-color mode (i.e. monochrome), there are two additional flags for controlling the default appearance of the PEG controls. These include:

MONO_BUTTONSTYLE_3D- simulates 3D-button appearance when running in monochrome mode. This flag has no effect when the number of available colors or grayshades is greater than two.

MONO_INDICATE_TITLE_FOCUS- causes PegTitle objects to draw a dithered background to indicate the window that has focus. This operation can be disabled by commenting out this flag. This flag has no effect when the number of available colors or grayshades is greater than two.

Packed-Pixel modes

When running in 256 color mode, there are two common 256 color video output schemes. The most common is the mode in which the video controller contains a set of 256 palette registers. Color values are used as an index into this set of palette registers. The second scheme is commonly referred to as “packed-pixel” mode. In this mode, each byte in video memory is interpreted directly as a 3:3:2 bit R:G:B color value, i.e. 3 bits for red, three bits for green, and 2 bits for blue.

Either mode is supported by PEG, and there is no performance difference between the two modes. The default mode when PEG_NUM_COLORS is set to 256 is the palette mode. To run in packed pixel mode, turn on the definition **EIGHT_BIT_PACKED_PIXEL**.

Building the PEG Library

Default Colors

The default colors for each peg object type are defined in the file `\peg\include\pegtypes.hpp`. You can freely modify these definitions to preference.

The library always uses color definitions found in the file `\peg\include\pegtypes.hpp` when drawing PEG objects. These color definitions have names like “PCLR_DIALOG” and “PCLR_CLIENT”, which stand for ‘PEG Color Dialog’ and ‘PEG Color Client’, respectively. Default color definitions are provided corresponding to each of the supported color depths. You do not need to make any modifications to these default color definitions to run the PEG library for the first time. However, you should make a mental note of these definitions in the event that you want to modify them later.

Scrolling Method

Several PEG objects are able to scroll their client areas using two distinct algorithms. The first algorithm simply re-positions the objects children and re-draws the client area of the object when scrolling is performed. The second algorithm uses a direct video memory move (called a ‘**bitblit**’) operation to scroll the majority of the client area, and only re-draws the newly exposed portion of the client area.

If your video controller supports hardware memory-move or bitblit operations, directly moving a rectangular area of video memory is generally much faster than completely re-drawing the scrolled window area. In real terms, scrolling a large window using the re-paint algorithm can require several hundred milliseconds of CPU time, while scrolling the same area with the use of hardware bitblit can require less than 1 millisecond of CPU time.

The scrolling method is selected via the `#define` **FAST_BLIT**. PEG objects do *not* interrogate the screen driver to determine which scrolling method to use, as this would cause unnecessary run-time overhead. Instead, PEG objects are compiled differently based on the **FAST_BLIT** definition. When this definition is turned on

the **FAST_BLIT** scrolling method is used. You should initially leave the **FAST_BLIT** definition disabled, and on your final target this method should only be used if your video controller and screen driver support hardware accelerated bitblit.

The provided screen drivers emulate **FAST_BLIT** operation by capturing and restoring areas of video memory, which generally does not provide much performance improvement over the default scrolling method, and can even be slower for VGA resolutions.

If on your final target you are using a hardware accelerated video controller with a customized PegScreen driver, the library should be built with the **FAST_BLIT** definition enabled. This tells the PEG window and control classes that you are using a hardware-accelerated driver.

Default Fonts

Several PEG objects display textual information. These objects include title bars, prompts, text buttons, etc.. The font used by any individual object can be easily modified at run time, and we will provide examples of this later in this manual. The **default** fonts used by each of the PEG objects are defined in the header file `\peg\include\pfonts.hpp`. When you first begin working with the PEG library, you can leave each of these at the default settings. Later, if you generate and use custom fonts, you can modify these settings to apply your new fonts by default to each of the different PEG classes. Full documentation of PEG fonts is provided in a later section of this manual.

Vector Fonts

Most embedded applications use bitmapped fonts exclusively. This is the default operation of PEG. However a vector font format is also supported, and can be enabled by turning on the definition **PEG_VECTOR_FONTS**. For more information refer to chapter 6, section PegFont.

Building the PEG Library

The default setting is to turn off vector font support.

Graphics Primitives

PEG does not internally use all available graphics primitives. If your application level software does not require extended graphics primitives, the code size of the library can be reduced by turning off the definition **PEG_FULL_GRAPHICS**. The exact primitives enabled by this definition are listed in the comments above the definition.

In addition, a few advanced graphics primitives require floating point math support. Since many embedded targets do not have any support for floating point math, it is common to want to exclude these functions from the library. These functions are excluded by commenting out the define **PEG_FP_GRAPHICS**. Disabling **PEG_FP_GRAPHICS** does not affect the appearance of the stock PEG windows and controls, it only means that your application level software will not be able to use those graphics functions which require floating point math support.

Run-time Image Conversion

PEG provides a set of classes for converting BMP, GIF, PNG, and JPG files into the PEG bitmap format during program execution. While these classes are available, **most embedded applications do not require run-time image conversion**, and opt instead to use the PegImageConvert utility to convert images prior to compile time.

If you do need support for run-time image conversion, you should turn on the definition for the required images types. These definitions are:

PEG_BMP_CONVERT for runtime OS2/Windows bitmap support.

PEG_GIF_CONVERT for runtime GIF support.

PEG_JPG_CONVERT for runtime JPEG image support.

PEG_PNG_CONVERT for runtime PNG file decoder.

Building the PEG Library

If you enable any of the run-time image conversions, there are a few additional configuration settings for the image conversion classes in the file `\peg\include\pimgconv.hpp`. For more information, refer to the notes in this header file and the PEG class reference.

The default setting is to turn off all run-time image conversions since images are usually pre-converted using the PegImageConvert utility program.

ZIP-UNZIP Support

The PEG class library optionally includes a general purpose data compression/decompression capability based on the Lempel-Ziv data compression algorithm. This data decompression capability is required when `PEG_PNG_SUPPORT` is enabled because this is the data compression format used by PNG files. You can also enable `PEG_ZIP` and/or `PEG_UNZIP` for use by your own application software if desired. This provides a very convenient and easy to use general purpose data compression library for your system software to use as required.

Run-time Image Scaling

PEG provides functions to resize, at run time, bitmap images in the PegBitmap format. These functions can be removed from the library by turning off the `#define PEG_IMAGE_SCALING`.

The default setting is to run off run-time image scaling.

Bitmap Writer

This definition includes an OS2/Windows bitmap write functionality within the PEG screen driver. This is very often used during product development to capture

Building the PEG Library

PEG “screen shots” as bitmaps for use in advertising, training, or other literature. The PEG bitmap writer produces a bitmap in memory, and you must save the bitmap to a suitable storage device using whatever means are available on your target system. The ability to save screen captures to standard bitmap files is included by turning on the definition `PEG_BITMAP_WRITER`.

LTOA

Two PEG classes use the non-ANSI function `LTOA` to convert integer data to string format. Many compiler run-time libraries provide this function, while others do not. For this reason, PEG optionally provides its own version of the `LTOA` function. If your compiler run-time library does not provide the `LTOA` function, turn on the definition `USE_PEG_LTOA`, and include the file `\peg\source\ltoa.cpp` in your PEG library make file.

The default setting is to enable the PEG `ltoa` function only if `PEG_UNICODE` is defined, else the PEG version is disabled.

PEG_CHARTING

PEG includes a set of charting classes that can be used to draw various types of chart displays. If charting is not required by your application, these charting classes can be removed from the library code by turning off the definition `PEG_CHARTING`.

PEG_FILE_DIALOG

The PEG class library includes a class for presenting a “file chooser” dialog the the user. This dialog is similar in purpose to the standard Windows or X11 file open dialogs. This class does include dependencies on the underlying file system, and since PEG is designed not to require any underlying file system this class is only included in the class library when the `PEG_FILE_DIALOG` option is turned on.

PEG_HMI_GADGETS

A further collection of user-interface gadgets, named ‘HMI Gadgets’ (HMI stands for Human Machine Interface), can be included or removed from the library by the `#define PEG_HMI_GADGETS`. This collection currently includes all `PegDial`, `PegLight`, and `PegScale` derived classes. These classes are not needed for basic GUI operation, and if not used by the application they can be removed from the library to reduce your library build times.

The default setting is to enable `PEG_HMI_GADGETS`.

PEGFAR

A few of the library classes (most notably the generic VGA screen driver) need to use “far” pointers when running on an x86 target in real mode under the large memory model. Since these classes must also compile correctly when used on targets that do not support the non-ANSI concept of “far” pointers, PEG uses the definition `PEGFAR` to define these pointers. On all but x86 real mode targets, `PEGFAR` is defined as nothing. On x86 real mode targets only, `PEGFAR` should be defined as “far”.

Library Source Files

This section describes each of the PEG source files and details which files are needed for building the library for different target environments. If you are building the library for DOS or Win32, you can skip to the next section and use one of the pre-configured project files provided with PEG to build the library.

Since PEG is normally built as a library, your linker should automatically ignore the PEG classes that are not used by your application. However we have found that embedded programmers are a distrusting lot, and because many linkers seem to include code that is not referenced, you may prefer to only include the PEG source

Building the PEG Library

files in your make file for the PEG objects you are actually using to insure you are using the least possible code space.

The PEG library is distributed with all of the source and header files required to build the library. You do not need to include in your make file the source files for classes you will not be using, and in addition most linkers are effective at removing class member functions which are not referenced.

It is not even really necessary to build PEG as a library. If you prefer, you can simply include the PEG source files in with your application modules as part of your make file. Most people prefer to build PEG as a library for the following reasons:

- You will not often (if ever!) be required to change any of the PEG source files, and so you will have faster build times if PEG is included as a library.
- A good linker will not include PEG components that are not used by your application. Typical run-time usage of PEG requires roughly 50K-70K bytes of code space, since not all of the PEG classes are required for a typical application.
- Linkers generally search library files last when resolving references, which allows you to replace individual functions without actually removing them from the library.
- If you include the PEG object files individually in your linker command line, they will generally all be included in your system software whether they are actually referenced or not.

The following sections list each PEG source and header file along with a brief description of the contents of each file. This is followed by a list of source files which are always required (the minimum build), the source files which are typically

Building the PEG Library

included (the typical build), and the optional files which are only included when required by the application software (the full build).

There are also a few files that are specific to each of the development environments, and those files are more fully described to insure that you understand which files to include for your target. There are a few additional source files associated with the RTOS integration packages. These source files are not described here, but are instead described in the appendix specific to each integration.

Source File Overview

The following source files are included in your PEG distribution:

Source File	Contents
dospeg.cpp	PegTask for DOS standalone operation
Elotouch.cpp	Touch Screen driver for ELO controller
L2scrn.cpp	Linear 4 color screen driver template
L4scrn.cpp	Linear 16 color screen driver template
L8scrn.cpp	Linear 256 color screen driver template
L16scrn.cpp	Linear 65535 color screen driver template
L24scrn.cpp	Linear 24-bit screen driver template
LinuxPeg.cpp	Integration for running PEG with Linux
Ltoa.cpp	Implementation of ltoa() function
Monoscrn.cpp	Monochrome screen driver template
panimwin.cpp	PegAnimationWindow
pbutton.cpp	PegButton, PegTextButton, PegBitmapButton, PegRadioButton, PegCheckBox
Pbitmaps.cpp	System bitmaps
Pblight.cpp	PegBitmapLight class
Pbmpconv.cpp	Run-time bitmap conversion
Pcbdial.cpp	PegCircularBitmapDial
Pcdial.cpp	PegCircularDial
Pchart.cpp	Charting base class
Pclight.cpp	PegColorLight class
Pcombo.cpp	PegComboBox

Building the PEG Library

Pdecwin.cpp	PegDecoratedWindow
Pdial.cpp	PegDial class
Pdialog.cpp	PegDialog class
Peditbox.cpp	Multi-line edit control
Pfbdial.cpp	PegFiniteBitmapDial class
Pfdial.cpp	PegFiniteDial class
Pfdialog.cpp	PegFileDialog class
Pfonts.cpp	PegFont and PegTextThing
Pgifconv.cpp	PegGifConverter class
Pgroup.cpp	PegGroup class
Picon.cpp	PegIcon class
Pimgconv.cpp	PegImageConvert run-time image converter
Pjpgconv.cpp	PegJpgConverter class
Plbscale.cpp	PegLinearBitmapScale class
Plight.cpp	PegLight class
Plist.cpp	PegList, PegHorzList, PegVertList
Pliteral.cpp	Global String literals and string library
Plnchart.cpp	PegLineChart class
Plscale.cpp	PegLinearScale class
Pmenfont.cpp	Menu Font
Pmenu.cpp	PegMenuBar, PegMenu, PegMenuButton
Pmsgwin.cpp	PegMessageWindow
Pmessage.cpp	PegMessageQueue
Pmlchart.cpp	Multi-line chart
Pmltbtn.cpp	PegMLTextButton (multi-line text button)
Pnotebk.cpp	PegNotebook class
Ppresent.cpp	PegPresentationManager class
Pprogbar.cpp	PegProgressBar class
Pprogwin.cpp	PegProgessWindow class
Pprompt.cpp	PegPrompt
Pquant.cpp	PegQuant run-time color quantizer
Prect.cpp	PegRect class
Proscrn1.cpp	Profile mode monochrome screen driver
Proscrn2.cpp	Profile mode 4-color screen driver
Proscrn4.cpp	Profile mode 16-color screen driver
Proscrn8.cpp	Profile mode 256-color screen driver
Prect.cpp	PegRect and PegCapture
Pscale	PegScale class
Pscreen.cpp	PegScreen

Building the PEG Library

Pscroll.cpp	PegHScroll and PVScroll
Psincos.cpp	Fixed-point sin/cos implementation
Pslider.cpp	PegSlider
Pspin.cpp	PegSpinButton
Pspread.cpp	PegSpreadSheet
Pstatbar.cpp	PegStatusBar
Pstchart.cpp	PegStripChart
Pstring.cpp	PegString
Psysfont.cpp	System Font
Ptable.cpp	PegTable
Ptextbox.cpp	PegTextBox
Pthing.cpp	PegThing
Ptitle.cpp	PegTitle
Ptree.cpp	PegTreeView
Pvprompt.cpp	PegVPrompt
Pwindow.cpp	PegWindow
Svgascrn.cpp	Generic VESA Super-VGA screen driver
Vgascrn.cpp	Screen driver for generic VGA
w32scrn.cpp	Screen driver for Win32 development
Winpeg.cpp	PegTask for Win32
X11scrn.cpp	Screen driver for X11 development

Header File Overview

The following header files are included in your PEG distribution:

Header File	Contents
L2scrn.hpp	Linear 4 color screen driver template
L4scrn.hpp	Linear 16 color screen driver template
L8scrn.hpp	Linear 256 color screen driver template
L16scrn.hpp	Linear 65535 color screen driver template
L24scrn.hpp	Linear 24-bit color screen driver template
LinuxPeg.hpp	Header for running PEG with Linux
Monoscrn.cpp	Monochrome screen driver template
Panimwin.hpp	PegAnimationWindow
Pblight.hpp	PegBitmapLight class definition
Pbmpconv.hpp	PegBitmapConvert run-time image converter
pbutton.hpp	PegButton, PegTextButton, PegBitmapButton,

Building the PEG Library

	PegRadioButton, PegCheckBox definitions
Pcdial.hpp	PegCircularDial
Pchart.hpp	PegChart base class
Pclight.hpp	PegCircurlarLight
pcombo.hpp	PegComboBox definition
Pconfig.hpp	PEG library configuration file
Pdecbtn.hpp	PegDecoratedButton class definition
Pdecwin.hpp	PegDecoratedWindow definition
Pdial.hpp	PegDial class definition
pdialog.hpp	PegDialog definition
Peditbox.hpp	Multi-line edit control
peg.hpp	PEG configuration file and global include
Pegkeys.hpp	PEG control key defintions and scan code translation table.
Pegtypes.hpp	PEG data types and basic definitions
Pfbdial.hpp	PegFiniteBitmapDial class definition
Pfdial.hpp	PegFiniteDial class definition
Pfdialog.hpp	PegFileDialog class definition
pfonts.hpp	PegFont and PegTextThing
Pgifconv.hpp	PegGifConverter run-time decoder
pgroup.hpp	PegGroup
picon.hpp	PegIcon
Pimgconv.hpp	PegImageConvert class definition
Pjpgconv.hpp	PegJpgConverter run-time decoder
Plbscale.hpp	PegLinearBitmapScale class definition
Plight.hpp	PegLight class definition
plist.hpp	PegList, PegHorzList, PegVertList
Plnchart.hpp	PegLineChart
Plscale.hpp	PegLinearScale class definition
pmenu.hpp	PegMenuBar, PegMenu, PegMenuButton
pmesgwin.hpp	PegMessageWindow
pmessage.hpp	PegMessageQueue
Pmlchart.hpp	Multi-line chart
Pmltbtn.hpp	PegMLTextButton
pnotebk.hpp	PegNotebook
ppresent.hpp	PegPresentationManager
Pprogbar.hpp	PegProgressBar
Pprogwin.hpp	PegProgessWindow
pprompt.hpp	PegPrompt

Building the PEG Library

Pquant.hpp	PegQuant run-time color quantizer
prect.hpp	PegRect and PegCapture
Proscrn1.hpp	Profile mode monochrome screen driver
Proscrn2.hpp	Profile mode 4-color screen driver
Proscrn4.hpp	Profile mode 16-color screen driver
Proscrn8.hpp	Profile mode 256-color screen driver
Pscale.hpp	PegScale class definition
pscreen.hpp	PegScreen
pscroll.hpp	PegHScroll and PVScroll
Psincos.hpp	Fixed-point Sin()/Cos() Implementation
pslider.hpp	PegSlider
pspin.hpp	PegSpinButton
pspread.hpp	PegSpreadSheet
pstatbar.hpp	PegStatusBar
Pstchart.hpp	PegStripChart
pstring.hpp	PegString
ptable.hpp	PegTable
ptextbox.hpp	PegTextBox
ptthing.hpp	PegThing
ptitle.hpp	PegTitle
Ptoolbar.hpp	PegToolBar
ptree.hpp	PegTreeView
pvecfont.hpp	Vector font data file
Pvprompt.hpp	PegVPrompt
pwindow.hpp	PegWindow
svgascrn.hpp	Generic VESA SVGA screen driver
vgascrn.hpp	Screen driver for generic VGA
w32scrn.hpp	Screen driver for Win32
Winpeg.h	PegTask for Win32
Winpeg.rh	Resource file Win32
X11scrn.hpp	Screen driver for X11 environment

Minimum Build

The following files are the minimum set of source files required to run PEG. These files must be included in every library build, regardless of the intended target platform:

Building the PEG Library

Pbutton.cpp	Pbitmaps.cpp
Pfonts.cpp	picon.cpp
Pliteral.cpp	Pmessage.cpp
Pmenfont.cpp*	
Ppresent.cpp	Pprompt.cpp
Prect.cpp	Pscreen.cpp
Pscroll.cpp	Psysfont.cpp*
Pthing.cpp	Pwindow.cpp

*These files may be replaced with custom font files if desired.

Typical Build

The following files are added in addition to the minimum build files to make a typical full-featured version of the PEG library. These files are all included in the pre-configured make files which are part of your PEG distribution.

Pblight.cpp	
Pcbdial.cpp	Pclight.cpp
Pchar.cpp	Pcombo.cpp
Pcdial.cpp	Pdecwin.cpp
Pdialog.cpp	Pdial.cpp
Peditbox.cpp	Pfbdialog.cpp
	Pgroup.cpp
Plist.cpp	Plight.cpp
Plbscale.cpp	Plscale.cpp
Pmesgwin.cpp	pmenu.cpp
Pmltbtn.cpp	Pnotebk.cpp
Pprogbar.cpp	Pprogwin.cpp
Pslider.cpp	pspin.cpp
Pstatbar.cpp	Pstring.cpp
Pscale.cpp	Ptree.cpp
Ptextbox.cpp	Ptitle.cpp
Ptoolbar.cpp	Pvprompt.cpp

Optional Classes

The following files are optional and only need to be included if you will be making use of these specific classes. These files are also included in the pre-configured PEG library make files.

Panimwin.cpp	Pbmpconv.cpp	Pchart.cpp
Peditbox.cpp	Pgifconv.cpp	Pfdialog.cpp
Plnchart.cpp	Pjpgconv.cpp	Pimgconv.cpp
Pquant.cpp	Pspread.cpp	Pmlchart.cpp
Pmlchart.cpp	Ppngconv.cpp	Plnchart.cpp
Ptable.cpp	Pzip.zpp	Pstchart.cpp

Target Specific Files

There are a few files that are specific to each target environment. These files contain the ***PegTask*** and ***PegScreen*** implementations for each environment. ***PegTask*** and ***PegScreen*** are described fully in subsequent chapters, however a quick summary will allow you to quickly build a working version of PEG.

There are many derived versions of the ***PegScreen*** class, not all of which are included in the standard source code distribution. Many versions *PegScreen* are available for specific video controllers used in embedded systems and CPUs with built-in video controller functionality. If you are using a video controller not specifically supported by one of the derived *PegScreen* classes in your source code distribution, contact Swell Software regarding the availability of a version for your specific controller.

You should always include the *PegScreen* base class in your make file, and ONE of the derived target specific screen interface class source files in your make file, depending on which platform you are building the library for. The versions of *PegScreen* available as of this printing include:

```
pscreen.cpp          // the base PegScreen class, which must always be included.
```

Building the PEG Library

vgascrn.cpp	// 640x480 generic VGA standalone screen class.
svgascrn.cpp	// 1024x768x256 VESA standalone screen class.
w32scrn.cpp	// screen class for use when running under Win32.
Monoscrn.cpp	// monochrome screen driver, all resolutions
L2scrn.cpp	// 4-color screen driver, all resolutions
L4scrn.cpp	// 16-color linear screen driver, all resolutions
L8scrn.cpp	// 256-color linear screen driver, all resolutions
L16scrn.cpp	// 65535-color linear screen driver, all resolutions
Proscrn1.cpp	// profile-mode monochrome template
Proscrn2.cpp	// profile-mode 4-color template
Proscrn4.cpp	// profile-mode 16-color template
Proscrn8.cpp	// profile-mode 256-color template
C154xpci.cpp	// Cirrus Logic PCI screen driver*
CT65scrn.cpp	// Chips & Tech 655xx screen driver*
690008.cpp	// Chips & Tech 69000 screen driver*
ct545_4.hpp	// C&T 65545 16 color/gray
ct545_8.hpp	// C&T 65545 256 color
ct550_8.hpp	// C&T 65550 256 color
ct690008.hpp	// C&T 69000 256 color
ct69_24.hpp	// C&T 69000 24bpp color
1330scrn.cpp	// SED 1330 screen driver*
1353scrn.cpp	// SED 1353 screen driver*
1354scrn.cpp	// SED 1353 screen driver*
1355scrn.cpp	// SED 1353 screen driver*
1356scrn.cpp	// SED 1356 screen driver*
1374scr4.hpp	// SED1374 eval card, 4-bpp all res
1375scr4.hpp	// SED1375 eval card, 4-bpp all res
1375scr8.hpp	// SED1375 eval card, 8-bpp all res
1376scr8.hpp	// SED1376 eval card, 8-bpp all res
1386scrn.hpp	// SED1386 eval card, 8-bpp all res

Building the PEG Library

13a0scr8.hpp	// S1D13A04 eval card, 8-bpp all res
sc400_1.hpp	// 1-bpp, sc400 controller
sc400_2.hpp	// 2-bpp, sc400 controller
8106scrn.cpp	// SED 8106 screen driver*
lh77mono.cpp	// ARM monochrome driver*
lh77gray.cpp	// ARM grayscale driver*
lh531_4.hpp	// Sharp LH79531- 4 bpp
lh531_8.hpp	// Sharp LH79531- 8 bpp
7111gray.hpp	// gray-scale for Cirrus Logic 7111 ARM
7111mono.hpp	// monochrome for Cirrus Logic 7111 ARM
7212gray.hpp	// gray-scale for Cirrus Logic 7212 ARM
7212mono.hpp	// monochrome for Cirrus Logic 7212
7312scr8.hpp	// 8bpp for Cirrus Logic 7312 ARM
ppc823_8.cpp	// Power PC 823 screen driver*
mq200_8.cpp	// 256 color for MediaQ 200 controller*
mq200_16.cpp	// 65K colors for MediaQ 200 controller*
permedia2.cpp	// 256 colors for Permedia2 controller
sascrn8.hpp	// StrongARM SA1100- 256 colors
smscreen.hpp	// Silicon Motion video card
a400_16.hpp	// Sharp 7A400- 16-bpp
omapscrn.hpp	// TI OMAP platform with Sharp TFT

*These screen drivers have been developed and tested using specific hardware platforms. Slight modification may be required to use one of these controller-specific PegScreen classes with your system. In some cases variations of these drivers are also available for running in profile mode LCD orientation.

Building the PEG Library

In addition, there are three '*PegTask*' replacement modules provided for running under DOS, Windows, or X11. You should include ONE of the following in your makefile or project file, depending on your environment:

```
dospeg.cpp          // include this file to build for DOS
winpeg.cpp          // include this file to build for Windows
x11peg.cpp          // include this file to build for X11 development environment
```

One final note: The RTOS integration packages contain custom versions of *PegTask* specific to each RTOS. Information regarding these implementations may be found in the integration notes provided in your integration package. Likewise, the integration packages are most often provided ready to run on a reference platform, which usually requires a screen driver specific to that platform. In this case, the RTOS integration notes include descriptions of the reference platform and screen interface class that should be used.

Pre-configured Project Files

Building PEG for WIN32 using Microsoft VC++ 5.0 - 6.0

Microsoft project files for MS VC++ 5.0 are provided for building the Win32 version of the PEG library and each of the example applications. These project files are generated using MSVC++ 5.0, however they can also be used with MSVC 6.0 as this version of the Microsoft compiler will import and translate MSVC 5.0 project files. The project file for building the Win32 version of the PEG library is contained in the directory \peg\build\win32. Likewise, Win32 project files for each of the example application programs are included in the directories containing the example source code.

Before attempting to build this version of PEG, you should open the file peg.hpp and comment out the line that reads "#define PEGDOS". Just below this line, you will see a line that reads "#define PEGWIN32". Un-comment this line to configure PEG for Win32 execution.

Building the PEG Library

To build a new version of the PEG library for Win32 using MS VC++, start VC++ and create a new workspace in any directory you prefer. Choose the command “Insert Project Into Workspace”, and use the dialog browser window to select `\peg\build\win32\ms50\peg.dsp`. This project file will build the PEG library. Before you can build the library, you will need to choose the command “Tools|Options|Directories” and add the directory “`\peg\include`” to the include directories, and the directory “`\peg\source`” to the source file directories.

If you want to build an executable application program to test a new version of the library, you will also need to add to your workspace one of the example applications contained in the `\peg\examples` directory. The standard PEG demo application is contained in the directory `\peg\examples\pegdemo`. Again choose the command “Insert Project Into Workspace” and use the dialog browser to select `\peg\examples\pegdemo\pegdemo.dsp`. After you have inserted this project into your workspace, select “Project|Dependencies”, and click on the check box next to “peg”. This will cause the build command to build and link the PEG library along with the example application program. Once you have done this, you should be able to select the ‘build’ command and generate the example executable program.

Building PEG for DOS using Borland C++ Command Line

A Borland command line makefile and the necessary configuration files for Borland C++ version 4.52 are provided in the directory `\peg\build\dos`. The Borland make file as delivered will build the DOS development version of PEG. ***Note that the makefile provided assumes that the Borland tools have been installed to `d:\bc45\bin`.*** You will need to modify these directory names in the makefile to match your system’s installation.

The batch file “`makpeg.bat`” builds the PEG library for DOS, using the makefile `dospeg.mak`. The batch file “`maklib.bat`” creates a library file from the PEG object files that can then be linked with your application level software.

Building the PEG Library

The Borland DOS make file actually builds all of the required PEG source files, along with the \peg\examples\pegdemo demo application. A separate batch file is provided for assembling the resulting object files into a standard library. This batch file is called “maklib.bat”, and is also provided in the directory \peg\build. If you simply want to build the PEG library, remove the demo application files from the dospeg.mak makefile. The demo files are readily identified by the fact that they are found in the directory \peg\examples\pegdemo.

Building PEG for WIN32 using Borland IDE

PEG can be built for WIN32 development using the Borland IDE environment. Borland desktop and project files are provided in the directory \peg\build\win32\bc5 or \peg\build\win32\bc452 depending on which compiler version you would like to use.

Building PEG for X11 Development Environment

PEG can be built to run under the X11 windowing environment under Linux. To build for this environment switch to the peg/build/x11/linux directory and type “make”.

Building for Other Integrated RTOS

Build procedures for other RTOS products are provided in a separate document for other RTOS integrations at this time. Please refer to the document entitled **Integration Notes** included in your PEG distribution. If you purchased PEG for use with an RTOS supported by Swell Software, you should have complete instructions for configuring and building the library for use with your RTOS, in addition to an example program pre-configured for a specific compiler and target.

Building PEG for other targets

If you are using a toolset other than the pre-configured environments, you should be able to follow the examples provided to create a make file specific to your build tools. Since you should not be required to re-build the PEG library very often, you may find it faster to create a simple batch file to compile the PEG source files and link the resulting object files into a library, rather than creating an actual make file.

Building PEG for a custom target is basically a matter of setting the best-fit options in the file `peg.hpp`, compiling all of the needed PEG source files, and linking them either into a library or directly into your system software.

Most users find that `dospeg.cpp` is an easy starting point for building PEG to run standalone or with a new RTOS. This file contains an implementation of ***PegTask*** that can be run standalone or quickly converted to a single RTOS task.

We suggest that you first build PEG to run in one of the pre-configured environments. This will allow you to experiment with the library and become familiar with the program startup sequence. You can also begin coding as much or as little of your application level software as you desire, as your application level software will NOT have to be modified to run on your final target. After you have read the remainder of this manual and experimented with PEG, you will find it very straightforward to port PEG to run on a custom hardware and software platform. If you have any questions during this process, we encourage you to contact Swell Software for assistance. We are generally able to have customers up and running on their target platforms within one or two working days!

Chapter 2: Common Terms and Concepts

This chapter introduces some of the basic concepts that are common to C++ programming and graphical windowing interfaces. If you are an experienced C++ programmer and you have worked with desktop windowing interface software, you can safely skip this information.

The first section of this chapter introduces a few C++ programming concepts that may be new to some users who have limited experience with C++ programming. PEG is designed such that only a basic understanding of C++ is required to make effective use of the library, and often developers who have only limited 'C' programming experience are able to quickly begin using PEG.

For users who are familiar with the EC++ (Embedded C++) standard, it should be noted that PEG is very nearly fully compliant with EC++. The only exception to this is the occasional use of multiple inheritance, which is not supported by the strictest EC++ standard. Many compilers, however, allow different levels of conformance to EC++, and PEG has been found to compile without error on all compilers tested to date.

The second section of this chapter describes the relationship between objects that compose a graphical interface. Understanding this relationship is vital to making your user interface work the way you intend, while also working as efficiently as possible.

Notes to users new to C++

If your programming experience to date has focused primarily on ‘C’ and assembly languages, you have chosen a terrific means for adding to your programming skills. While C++ does not fit well in all programming tasks, GUI development is one area, perhaps the best area, for application of the C++ object oriented programming methodology.

While it is not our goal to do a full tutorial on C++ programming, this section of the manual will introduce you to a few critical concepts which must be understood before you can effectively develop programs using PEG. After working through the included examples, you will be well on your way to productive use of the C++ language. We also recommend that you continue to advance your knowledge of the C++ language through the study of any one of several excellent books on the subject. In particular, Teach Yourself C++ (Herbert Schildt, McGraw-Hill) is a terrific and easy to read tutorial. The C++ Programming Language (Stroustrup), while an excellent reference for advanced users, is of more use to someone contemplating writing a C++ compiler than to someone searching to gain insight into practical use of the language. Many additional references fill the void between beginning and advanced C++ language instruction.

Objects

Put simply, objects are data structures and associated methods for manipulating those structures. You define an object for the C++ compiler by using the ‘class’ keyword. The data and functions defined within a class declaration are called the class members. If you are a ‘C’ programmer and you are familiar with defining your own data types via the ‘typedef struct’ syntax, you already understand the basics of defining and using objects. In fact, the C++ keyword ‘class’ is interpreted almost identically to the keyword ‘struct’, and the C++ standards committee had heated discussion regarding removing the keyword ‘struct’ entirely since the two were basically identical. The only difference, as any experienced C++ user knows,

Common Terms and Concepts

is that functions and data members of a struct are by default public and members of a class are by default private.

For example, the following two declarations are identical:

```
typedef struct                // the 'C' (or C++) version
{
    int Count;
    char foo;
    long bar;
} Simple;

class Simple                  // the equivalent C++ declaration
{
public:                       // makes everything 'public'
    int Count;
    char foo;
    long bar;
};
```

The neat feature that C++ adds is the ability to include functions in your objects, rather than just data. Before you start worrying, the compiler does NOT generate a complete set of functions every time an object is created (either via **new** or as an automatic), but every time an object of a certain type is created it shares the member functions with other objects of the same type.

Encapsulation

What do *public* and *private* mean? Well, this gets to one of the much discussed tenants of C++, encapsulation. Encapsulation is a big word that simply means when you define an object you can protect its internals from rogue outsiders by defining all or a subset of the class members as 'private'. When you tell the C++ compiler that data and methods are private, the compiler will not allow other code (other objects, C functions, or whatever) access to those members. There are ways to get around this, but that discussion is unnecessary for our purposes. In addition, the

Common Terms and Concepts

term encapsulation is intended to convey the idea that an object is self contained, i.e. you as the user of an object have no need or desire to know exactly how the object goes about its business, you only care that the object does what it is supposed to do.

A third group of members can be defined as protected, which is somewhere between public and private. When a class member is defined as protected, this tells the C++ compiler that unrelated classes or code cannot access the protected member, however classes *derived from* the class containing the protected member can access the protected member function(s) and/or data.

If at some point you begin to examine the PEG library header files, you will notice that the class declarations all follow a similar style. The public class members, who are the class members the application level software can call, are always listed first in the declaration. Protected members (if any) are listed next, followed by private members. Only the public and protected members of the PEG library classes are documented in the programming manual, since you are prevented from direct access of the private member functions and variables.

Constructors

The first and most common member function you will encounter is called the constructor. You recognize a constructor function by the fact that its name is the same as the name of the class of which it is a member. Constructors allow the designer of a class to specify exactly how the class members should be initialized when an instance of an object is created.

Using the structure example above, when working in 'C' you could create a new copy of struct Simple either by declaring an automatic (space for Simple is made on the stack) or by using malloc() to allocate enough memory to hold a Simple structure. The problem with this is that in either case you do not know what values the data members of Simple will contain when Simple is created. If you want all of

the members to be initialized to zero, you have to do this yourself (possibly by using a call to `memset()`) after you create storage space for Simple.

C++ constructors overcome this problem. Every time a class of type Simple is created, the constructor for class Simple (if one is defined) is called to initialize the object. In fact, the designer of class Simple can define many alternate constructors, and the appropriate version is called depending on what parameters you pass when you create the object.

Inheritance

When working with a C++ class library, it is very common to read the documentation of a particular class and say to yourself “Gee, class xyz is almost exactly what I need”. C++ provides a critical method for handling this situation, through the concept of inheritance. Inheritance allows you to create new classes based on other classes. You are telling the compiler “I want to define a new class that is nearly the same as class xyz, but I want to change a few things”. One way to change things is by overriding member functions of the original class. To override a function means that in your new class you define a function that has the same name as the function in the original class. At run time, when the function is called, your replacement version will automatically be called instead of the original. Classes created in this way, that is classes that are based on other classes, are called derived classes. Derived classes are said to inherit from their predecessors, hence the term inheritance. The predecessor class is called the base class. There is nothing unique about a base class, and in fact a derived class can serve as the base class for yet another derived class. In addition to overriding functions, you can also add completely new functions and/or additional data members in your derived classes. You will have the opportunity to create your own derived classes later in this manual as you work through the programming examples.

In the case of a class library such as PEG, the relationship among the classes (called the class hierarchy) which compose a class library is very important, and

Common Terms and Concepts

since there are a large number of classes it can also be difficult to remember. A diagram or written description of the inheritance structure is therefore very important for new users. The PEG class hierarchy diagram is included later in this manual.

PEG uses inheritance heavily in the design of the class library. This allows PEG to do some very powerful things very easily, and minimizes the size of the PEG code because functions for performing certain tasks are used and re-used by all classes in the library.

If all of this seems just a little overwhelming, relax. By the time you work through the example programs which follow you will have learned all you need to know to use PEG effectively. You can then use the example programs as guidelines to start constructing your own graphical interface.

Inlining

The concept of function inlining is familiar to most users of 'C'. Inlining allows the programmer to instruct the compiler to place the actual instruction sequence generated by a function in place of a call to the function. C++ extends this concept to allow a class definition to specify functions that should be inlined. While the C++ standard does not enforce this capability on C++ compiler vendors, all clever C++ compilers do a good job of supporting function inlining. PEG uses inlining heavily to improve performance.

Windowing Interface Terminology

This section introduces some terms that may be new to you if this is your first experience with graphical programming.

Window and Control

These terms are very loose in definition, and you should not read too much into their use when you see these terms in this manual. These terms are only used for convenience when describing program operation as an alternative to itemizing long lists of actual class names. It is sometimes convenient to group the PEG classes into two broad areas, the Window classes and the Control classes. This does not always imply that a Window class is derived from PegWindow or that a Control class is not derived from PegWindow, although that is often the case. The term Window implies only a background object that contains other objects. The term Control is used to refer to an object that is normally a child of a Window, or an object that the end user may interact with directly.

In PEG, there is actually very little difference between a Window and a Control. A Window can be a child of a Control, and Control can contain a Window. Certain features, such as scrolling, are normally associated with Windows, while other features such as notification messages are normally associated with Controls. This does **not** mean that a control cannot scroll, or that a Window cannot send a notification message. These terms are only used to describe the general case. An application can modify and extend this general case at will.

When you see the term Window, infer only that the documentation is referring to any PEG object that is normally used as a background container for other objects. Likewise, when you see the term Control, this is simply a shorthand method of denoting the group of PEG classes that most often do not contain other objects, and are generally used directly by the end user.

Common Terms and Concepts

Parent, Child, Sibling

These terms refer to the relationship between the windows, controls, and other items that are all part of your interface. A control that is attached to a window is termed a *Child* of that window. Likewise, the window that contains the control is termed the *Parent* window. If there are several controls attached to the same window, those controls refer to each other as *siblings*.

While we have just described the most common case, there is nothing internal to PEG that prevents a window class such as *PegWindow* from being the child of a control, such as a *PegButton*. In fact, it is often very useful to construct custom objects using exactly this type of parent-child relationship.

Some GUI platforms place restrictions on the number of parent-child generations that can be nested within the same window, or even within a single application. PEG imposes no such restrictions, nor will anything prevent an object that is a parent object in one case from becoming a child of another object in a different case. This is a powerful feature of PEG, because it allows you to re-use custom objects that you create in a variety of different ways.

Base, Derived, Inherited

The parent-child relationship described above is often confused with the class hierarchy, which describes a completely different relationship among the classes composing your graphical interface. Some of this confusion results from sloppy terminology, in that people often use the terms parent and child when what they are really referring to is base and derived.

The term Base or base class is a relative term, indicating the named class is the foundation for a class that is derived from it. A class that is called a base class in one case could easily also be a derived class, inheriting data and methods from an even more fundamental object.

It is especially important to remember the distinction between these terms when you are reading the description of PEG message flow and message handling. We have made every effort to insure that the correct terminology is used in all cases.

Modal Execution

A window is said to be executed modally when that window must be closed or completed by the end user before other windows are allowed to receive any user input. This is most often used for executing a ‘Modal Dialog’ , which is a dialog window that must be closed before any other open windows can receive user input such as a mouse click.

In PEG, any window can be executed modally. In fact, there can be several modal windows operating at one time in certain multitasking environments. Modal windows capture all input devices, preventing other windows and controls from being active while the modal window is executing.

Common Terms and Concepts

Chapter 3: PEG Execution Model

This chapter introduces the PEG execution module and describes how the fundamental PEG classes work together to create a working interface. This chapter focuses on establishing a macro-view of PEG of the internal components of a graphical presentation created with PEG, while the following chapters detail actual class descriptions, public functions, and class usage.

You should be warned that this chapter and those immediately following contain a large amount of important information, and much of this information is non-trivial. We are now going to dive under the hood and examine how PEG works. We therefore encourage you to take a break, stretch your legs, and then settle in for some concentrated reading. When you reach the programming examples, you will be well on your way to becoming a PEG power-user!

Since many of the topics covered in this chapter are inter-dependant, it is unavoidable that we must occasionally introduce terms or refer to PEG classes that have not yet been fully defined. For this reason, we recommend that you read chapters three, four, and five straight through from top to bottom the first time, and then return and review each section to solidify your understanding. These chapters are followed by several programming examples, which will help you to put it all together.

PEG supports three general execution models, the single-threaded or standalone model, the ***Multithread*** model, and the ***PRESS*** model. In order to avoid confusion caused by overwhelming the reader, most of the information presented in this chapter assumes you are running in the standalone model. All of the concepts presented are valid regardless of your execution model. It is only the ***flow*** of information and processing that changes based on the execution model. A complete discussion of multi-tasking as it relates to PEG and the supported execution models is included in the “PEG Multitasking” chapter.

Overview

The components of PEG that control the execution of your interface are ***PegTask***, ***PegMessageQueue***, ***PegPresentationManager***, and ***PegScreen***. These components work together to insure that your interface operates in a well defined, predictable, and fault-tolerant manner. These components are also central to insuring that your PEG application is portable to a variety of embedded systems. In this chapter we will fully investigate ***PegTask*** and ***PegPresentationManager***, while ***PegMessageQueue*** and ***PegScreen*** are described in the following chapters.

PegTask provides the interface between PEG and the real-time operating system. When running standalone or in a single threaded environment, ***PegTask*** is simply the entry point to the main program loop.

PegMessageQueue provides a FIFO style message queue for sending information between PEG objects.

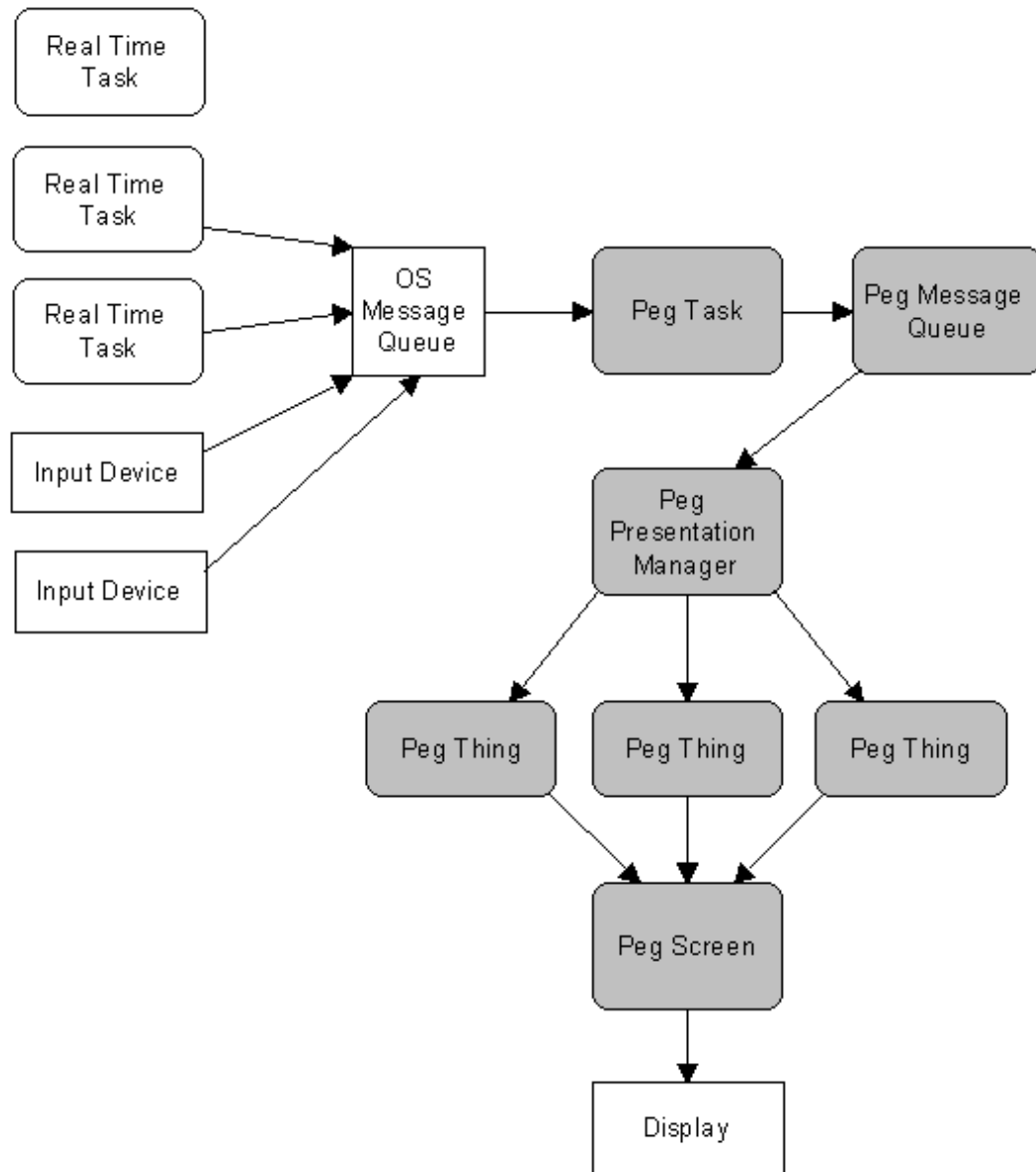
PegPresentationManager keeps order on the visible screen. This involves keeping track of which window(s) are on top of other windows, maintaining the status of each object, and remembering which object should receive user input.

Finally, ***PegScreen*** provides a layer of insulation between PEG and the physical display device. ***PegScreen*** does the dirty work of drawing on the display, and provides all of the low-level drawing functions PEG objects need to present themselves to the user.

Software Block Diagram

A software block diagram of an executing PEG application is shown on the following page. This drawing depicts ***PegTask***, ***PegMessageQueue***, ***PegPresentationManager***, input devices, miscellaneous graphical objects, and the ***PegScreen***. These are the components of every user interface built using PEG.

PEG Execution Model



Program Startup

PEG startup is divided into two layers. The lowest layer, accomplished by **PegTask**, creates the central PEG components and performs any other required initialization. The second startup layer is a function called **PegAppInitialize()**. This is the entry point from the application level software perspective.

PegTask is tailored to the execution environment, while **PegAppInitialize()** is entirely application defined. This segmentation follows the PEG philosophy of insulating all application level software from the target environment, allowing your application software to run un-modified when moving from one of the pre-defined development environments to the final target system.

PegAppInitialize() is where you, the user interface designer, create and display the windows, dialogs, or other graphical elements that will be displayed immediately after system startup. The contents of **PegAppInitialize()** are entirely up to you. You can initially display only one, several, or even no graphical elements at all. It is most common to initially display at least one application window, and then allow subsequent windows and dialogs to appear as defined by the user interface menu and user input actions. We will work through several examples of the **PegAppInitialize()** function in later chapters. As a preview, here is an example of what **PegAppInitialize()** might look like in your application software:

```
void PegAppInitialize(PegPresentationManager *pPresent)
{
    pPresent->Add(new MyWindow());           // create and add your first
window
}
```

We have not covered all of the information you need to fully understand this example, so don't worry about the details. In the above example, we have constructed an instance of a window class called "MyWindow", and added that window to **PegPresentationManager** by calling the **PegPresentationManager**

member function “Add”. This is a typical implementation of the **PegAppInitialize()** function.

After calling **PegAppInitialize()**, PegTask calls the *PegPresentationManager* member function **Execute()**. This is where the central message-processing loop of PEG begins. We will investigate message processing as it relates to PEG in the following chapter, “*PegMessageQueue*”.

PegTask

The graphical interface should be viewed conceptually as a continuous low-priority task in the overall multitasking system. During execution, messages are dispatched from *PegMessageQueue* to *PegPresentationManager*, who in turn routes messages to the various graphical objects for processing. While the graphical interface is not truly running continuously in a multitasking system, the multitasking aspects are transparent to the entire graphical interface with the exception of *PegTask*.

PegTask constructs *PegPresentationManager*, *PegMessageQueue*, and the derived version of *PegScreen* that is required for the target system. These components are required for any PEG application to run. After these components are created, *PegTask* calls **PegAppInitialize()**, which is where you define the initial objects that will be displayed.

PegTask is a conceptual element, and is generally composed of two or more functions. The first function provides the task entry point required by the real time operating system. When running in a standalone DOS environment, *PegTask* is simply the function **main()**. When running in any standalone or single-threaded environment, a second function named **PegIdleFunction()** is also a component of *PegTask*. **PegIdleFunction()** is called by the portable *PegMessageQueue* implementation when there is nothing left for PEG to do. This allows other lower

PEG Execution Model

priority tasks to execute, or in single-threaded systems, **PegIdleFunction()** is a convenient place to perform other processing required by the target system.

In the following example, the ***PegTask*** entry point is a function called, fittingly, **PegTask()**. This function should create the PEG foundation objects, call **PegAppInitialize()**, and begin PEG execution by calling **PegPresentationManager::Execute()**. The following pseudo-‘C’ code illustrates a typical implementation of ***PegTask***:


```

void PegTask(void)
{
    // create the screen interface object:

    PegRect Rect;
    Rect.Set(0, 0, 639, 479);
    PegScreen *pScreen = new TargetScreen(Rect);

    // create the PEG message Queue:

    PegMessageQueue *pMsgQueue = new PegMessageQueue();

    // create the PresentationManager:

    PegPresentationManager *pPresent = new PegPresentationManager(Rect);

    // initialize static pointers used by all PEG objects:

    PegThing::mpScreen = pScreen;
    PegThing::mpPresentation = pPresent;
    PegThing::mpMessageQueue = pMsgQueue;

    // call application initialization function:

    PegAppInitialize(pPresentation);
    pPresentation->Execute();                // run PEG

    // returns only if the application is terminated
    // delete PEG foundation objects
}

```

The example above is generally all that is required to run PEG as a single task in any real time operating system environment.

PegIdleFunction

In the standalone version of PEG, **PegIdleFunction()** is the second component of **PegTask**. **PegIdleFunction()** is called by **PegMessageQueue** when there are no

PEG Execution Model

longer any messages in the queue which require processing. In this case the graphical interface is up to date and does not need the CPU. **PegIdleFunction()** is defined by you, and is typically where you insert an OS-specific suspension mechanism to block (i.e. suspend) PEG until some external stimulus is received which un-blocks PEG and allows GUI processing to continue.

Note that PegIdleFunction() is not called by versions of PEG which have been fully integrated with a commercial RTOS. Versions of PEG which have been integrated by Swell Software with a commercial RTOS do not use this suspension mechanism, but instead use custom implementations of **PegMessageQueue** that suspend the GUI task(s) directly when no messages are available for a particular task.

If you are using PEG for the first time with an OS created in-house or an OS not yet integrated with PEG, you should start the porting process by modifying the DOS version of PEG to run as a single task in your system. This makes it very easy to quickly get PEG running on your target.

The code for **PegIdleFunction()** is obviously somewhat OS dependant. The operation performed by **PegIdleFunction()** is also dependant on whether your input devices are interrupt-driven or polled. If your input devices are interrupt driven, **PegIdleFunction()** should block or suspend indefinitely until a message arrives. If your input devices are polled, **PegIdleFunction** should poll the input devices until a new input message is generated.

The following is an example **PegIdleFunction()** for use with interrupt driven input devices:

PEG Execution Model

```
void PegIdleFunction(void)
{
    // This function is called by the PEG library when there is nothing left for PEG to
    // do.
    // This version suspends the PEG task until a message arrives from an external
    // source.

    MSG *pm;
    PegMessage NewMessage;
    PegThing *pt = NULL;

    // suspend PEG until an external message arrives:

    pm = WaitForMessage(OsPegQueue, INFINITE);

    // convert the OS-format message to a PegMessage:

    NewMessage.wType = ConvertToPeg(pm);

    // place the new message in PegMessageQueue:

    pt->MessageQueue()->Push(NewMessage);

    // return and allow PEG to run
}
```

In the above example, **PegIdleFunction()** waits for an infinite period for a message to arrive at OsPegQueue from some other source in the system. OsPegQueue is an RTOS defined message queue created for interfacing PEG with input devices and other system tasks. When a message is received, the function ConvertToPeg() is called to convert the message from the OS-specific format into the format defined by PEG. The new PegMessage is then passed to **PegMessageQueue**, after which **PegIdleFunction** returns allowing GUI processing to continue.

Note that the above is an example implementation only. In the above example, you should see that input device drivers must be present that are capable of directly posting messages into OsPegQueue. Otherwise, this version of PegIdleFunction

PEG Execution Model

will remain suspended indefinitely and no further GUI processing will take place once PEG calls `PegIdleFunction()`.

In addition, it should be noted that if possible the `ConvertToPeg()` function should be eliminated. This requires that messages are sent through the OS message queue in the `PegMessage` format. If you are able to do this, `PegIdleFunction` should simply transfer messages from the `OSQueue` to `PegMessageQueue`.

An even simpler implementation may be used if your input devices must be polled. The file ***dospeg.cpp***, which is the **PegTask** implementation for running PEG under standard DOS, contains a simple polling loop in **PegIdleFunction()** to retrieve mouse or keyboard events. The following is the DOS version of **PegIdleFunction()** contained in the file *dospeg.cpp*:

```
void PegIdleFunction(void)
{
    PollTime();
    PollMouse();
    PollKeyboard();
}
```

As you can see, this is very simple indeed! If you are running PEG with an RTOS or other system not supported by PEG, this is a very nice starting point for getting your system up and running. After your system is running, you can gradually ‘tune’ **PegIdleFunction()** to eliminate the polling as you create and integrate interrupt driven input drivers.

Many users of PEG with custom operating systems also create a function for sending messages from external tasks into the PEG message queue, defined above as `OsPegQueue`. This function does whatever processing is required to send messages from other tasks to the queue being monitored by **PegIdleFunction()**. Note that the following code sequence is **not** allowed from an external task (i.e. a task other than `PegTask`) when using the portable version of `PegMessageQueue`:

```

void Task2(void)
{
    PegThing *pThing = NULL;
    PegMessage NewMessage;
    NewMessage.wType = PM_EXIT;
    pThing->MessageQueue()->Push(NewMessage);    // DON'T DO THIS!!!
}

```

Why should the above be avoided? Remember that Task2 is not PegTask, but some other task, possibly at higher priority than PegTask. Calling ***PegMessageQueue::Push()*** from an external task could result in the corruption of the FIFO data structures maintained by PegMessageQueue, since there are no list protections in place. The portable version of PegMessageQueue is designed to run only from within PegTask.

There is another less obvious reason why posting messages directly into PegMessageQueue from external tasks is not a good idea. Remember that your user-defined version of **PegIdleFunction()** is probably waiting at an OS-defined message queue for a message to arrive. If you bypass the OS-defined message queue, **PegIdleFunction()** will not ‘wake up’, and PEG message processing will not continue.

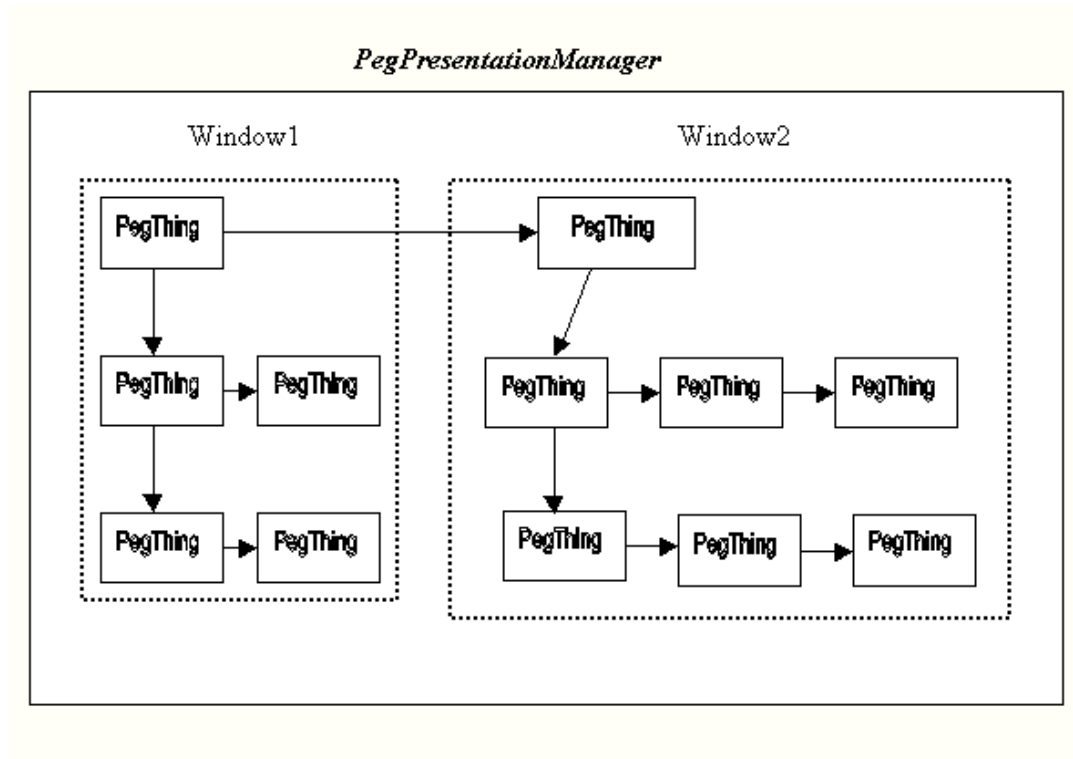
As stated above, this is why most users porting PEG to a new RTOS define a function for sending messages to the OS-defined message queue which is monitored by **PegIdleFunction()**.

The PEG DOS environment is a simple and convenient starting point when porting PEG to a custom target. The entire DOS implementation of PegTask is contained in the file \peg\source\dospeg.cpp. Very little modification is usually required to turn this version of PegTask into a single low priority task running on the target RTOS. For single threaded (i.e. systems not using an RTOS) systems, the DOS environment is also a convenient starting point, since of course DOS is itself a single-threaded environment.

PegPresentationManager

PegPresentationManager keeps track of all of the windows and sub-objects present on the display device. In addition, **PegPresentationManager** keeps track of which object has the input focus (i.e. which object should receive user input such as keyboard input), and which objects are ‘on top’ of other objects. Since there is no limit to the number of window and controls and other objects that may be present on the screen at one time, you can probably imagine that this quickly becomes a complex task.

How does **PegPresentationManager** keep track of all of those windows and their children and grandchildren? By using tree structured lists. Intrinsic to the design of PEG, all objects that can be displayed are derived at some point in their hierarchy from a common base class called **PegThing**. We will give you the details of the **PegThing** class in a later chapter, but for now two important members of **PegThing** are a pointer to each **PegThing**’s first child object, and a second pointer to each **PegThing**’s next sibling. Using these two pointers, **PegPresentationManager** maintains all objects in lists, as shown below.



PegPresentationManager is also derived from *PegWindow*, which is derived from *PegThing*. This means that *PegPresentationManager* is more or less just another window, although in this case the window has no border, can often appear invisible, and always fills the entire screen or display. In essence, *PegPresentationManager* is the great-great-grandfather of all windows, dialogs, and controls you will display during the execution of your system software. We often use the term ‘top level window’ to indicate a window that has been added directly to *PegPresentationManager*. To the end user, a top-level window appears to be standalone, and appears to have no parent. You now know, however, that there is

PEG Execution Model

actually an invisible window that is the parent of all top level windows, and that window is called ***PegPresentationManager***.

In just a little while you will be reading more about a very basic and important PEG class, class ***PegThing***. When you read about ***PegThing***, always remember that ***PegPresentationManager*** derives at some point from ***PegThing***, and so all of the ***PegThing*** member functions are available when you are working with ***PegPresentationManager***. Of special interest are the functions **Add()**, **Remove()**, **Parent()**, **First()**, and **Next()**. These are the functions you will use in your software to modify and examine the tree-structured list of visible objects. Stated another way, these are the functions you will use to add windows to and remove windows from ***PegPresentationManager***.

Event Driven Programming

In viewing the preceding software block diagram you may have wondered “Where is main()?”. PEG follows the event-driven programming paradigm. There is no one central location or super-loop in the application level GUI software.

PEG is message-driven, which may also be called event-driven. This means that real processing is only done in response to messages received from the outside world. The objects that you create and use will be able to send and receive messages. You will be able to invent your own messages and interpret them however you want in order to make your objects do the work you want them to do. In general, you want to keep your messages simple, and the corresponding message processing short, to prevent any one object from dominating your available CPU time. You will work through an example of how to set up your message handling functions later in this manual.

There are several advantages to a message-driven implementation. The fact that PEG objects communicate with each other via messages eliminates the problems associated with callback functions or similar implementations. One PEG object can

communicate easily with another without worrying about how to physically address that object. In the large view, you could accurately state the message-driven systems are distributed systems, and objects can actually be miles apart from each other (physically!) and talk as if they are both running from the same ROM. Of course, the out-of-the-box ***PegMessageQueue*** implementation would require some enhancement to support this example, but these types of things are possible in a message-driven environment.

PegButtons, PegStrings, PegPrompts and other control types also use messaging to notify you when the control has been modified. The messages generated by these types of objects are determined by the object's Id, which is a member variable of every PEG class. This makes the flow of information throughout a PEG-based application very predictable and robust. This type of message passing is so common that PEG defines a unique syntax for handling control notification messages, called ***signals***. ***Signals*** will be described in detail in an upcoming section.

PegPresentationManager supplies the overall control of your PEG application. ***PegPresentationManager::Execute()*** is the main execution loop for your GUI interface. In many embedded systems, ***Execute()*** never returns to the caller, since the graphical interface is intended to run forever. Of course, in a multitasking system you don't really want PEG to execute continuously; rather only when there is real work to do, and even then only when no higher-priority tasks are ready to run. ***PegPresentationManager*** accomplishes this by calling ***PegIdleFunction()*** when there is nothing left for PEG to do.

Input Focus Tree

An additional task of PegPresentationManager is message routing. In a later chapter on PegMessageQueue, you will learn how to route messages to specific windows and controls. However, many system messages, such as mouse and keyboard input messages, are not directed to any particular object. For this reason,

PEG Execution Model

PegPresentationManager internally maintains a pointer to the object that was last selected by the user using the mouse or other input means. This object is called the “current” or “input” object, meaning that by default this object will receive input messages.

PEG views each displayed window and child objects of each window as branches in a tree. When input focus moves from object to object, PegPresentationManager insures that the entire branch of the tree up the actual input object has input focus. You can detect if an object is a member of the input focus branch of the presentation tree at any time by testing the PSF_CURRENT system status flag:

```
if (StatusIs(PSF_CURRENT))
{
    // this object is in the branch of the display tree that has input focus.
}
```

Just because an object is a member of the input focus tree does not mean the object is the end or leaf of the input focus branch. You can obtain a pointer to the final input object by calling the PegPresentationManager::GetCurrentThing() function. This function will return a pointer to the actual default input object, or NULL if no object has been selected to receive input events.

You can also override the user’s input selection and manually command PegPresentationManager to move the input focus at any time by calling the PegPresentationManager::MoveFocusTree() function. This function will set input focus to the indicated object by sending PM_NONCURRENT messages to objects that are no longer members of the input focus branch, and PM_CURRENT messages to objects that are members of the new input focus branch. The effect is that non-directed input messages will be sent to the newly designated input object. In most circumstances, you will not be required to manually adjust the input focus, however this capability is available when you need to use it.

When a new window is added to PegPresentationManager, that window automatically received input focus. Likewise, if that window has any child objects, the first child object of the window receives focus. This continues until a leaf node (an object with no children) is found. The **Add()** function, described in a following chapter, is most commonly used to add child objects to a window. Since the **last** object added to a window becomes the **first** child of the window (unless **AddToEnd()** is used to add the object), the last object added to a window will have input focus when the window is first displayed.

This makes it important to add your child window controls such that the object that should initially have input focus is added to the window last, after all other child objects have been added.

Keyboard Input Handling

Closely related to the input focus tree are PEG keyboard input handling methods. One of the main reasons for keeping track of which object has input focus is to know which control to send keypress messages to when the user operates an interface that has some form of keyboard or keypad input device.

Keyboard input is received by PEG objects when the library is built with the PEG_KEYBOARD_SUPPORT option turned on. PEG_KEYBOARD_SUPPORT does not imply that you need to have a full 100+ key keyboard. Many PEG users have a very limited keypad with only a few key values available. This type of input will work just fine with a PEG application, since PEG requires only a very limited number of key values to navigate through screens and select controls

We will fully describe the format of PEG messages in later section, however it is useful to describe the format of keyboard or keypad input messages here. Keyboard input arrives in the form of PM_KEY messages, meaning that the Message.wType field == PM_KEY.. The actual key value is passed in the Mesg.iData field, and the key flags such as shift key state, control key state, etc.. are passed in the Mesg.lData parameter. Keyboard messages are undirected, meaning that the

message contains no information about which object should receive the message. This makes it the responsibility of PegPresentationManager to know which object should receive keyboard input messages as they arrive from your input device driver.

As described above, when a window is added to PegPresentationManager the first client-area child object of the window gains input focus. A limited number of PM_KEY messages can then be sent to move focus from object to object in the PEG presentation. When PEG_KEYBOARD_SUPPORT is turned on, the end user can fully navigate through a PEG application by sending a very small number of PM_KEY messages to PEG.

The paragraphs below describe the key values that PEG objects are watching for to allow the user to navigate through the graphical interface. The key values are designed to closely follow desktop standards for keyboard input. This does NOT imply that your target system must actually have keys such as TAB or CTRL, this only means that PEG is watching for these key values. If your target system has, for example, 4 arrow keys and an "enter" key, you simply need to map these keys to the best-match key values that PEG is watching for. In some cases, you may need to send different key values for a common input key depending on what type of object the user is interacting with.

PK_TAB	When this key value is received, PEG attempts to move focus to the next child of the current window. If the current child is the last child, focus wraps back to the first child of the current window. If the Shift key is pressed (i.e. the KF_SHIFT flag is set in the PM_KEY message lData member), the tab direction is reversed.
<ctrl> + PK_TAB	This key combination is used to cycle through top level widows.
PK_CR	The carriage return key is used to select the item which has focus. If this is a button object, the object will active or toggle.
PK_F1	This key moves focus to the first menu item of a menu bar added to the current window.
PK_LNUP, PK_LNDN, PK_LEFT, PK_RIGHT	These keys (the arrow keys) move focus from sibling to sibling, and are also used to navigate through PegMenu items.
PK_ESC	This key is used to close an open menu or to escape from a PegString edit operation.
<ctrl> + PK_F4	The key combination is used to close the current window

PEG Execution Model

Mouse or Touch Screen Input Handling

Mouse input is also handled, at least initially, by PegPresentationManager. Mouse and touch screen input message are also undirected, meaning that they are not targetted to any specific object. Mouse and touch screen input message do, however, contain position information for each touch, release, or drag message. This allow PegPresentationManager to quickly determine which object should receive each mouse or touch screen input message.

Chapter 4: PegMessageQueue

PegMessageQueue is a simple encapsulated FIFO message queue with member functions for queue management. ***PegMessageQueue*** also performs timer maintenance and miscellaneous housekeeping duties as will be described later.

How do messages get into the ***PegMessageQueue***? They are placed in the message queue from one of three sources:

- Input devices, such as a mouse, touch screen, or keyboard.
- Any other task in the multitasking system (via ***PegTask***).
- From PEG objects themselves.

The messages placed in ***PegMessageQueue*** are the driving force behind the graphical interface. These messages contain notifications and commands which cause the graphical elements to redraw themselves, remove themselves from the screen, resize themselves, or perform any number of various other tasks. Messages can also be user-defined, allowing you to send and receive a nearly unlimited number of messages whose meaning is defined by you. For example, it would be very common to have a graphical element send a message to another task in the system requesting data for display. The target task receives the request and responds, sending the response message to ***PegTask***.

While PEG is acting upon messages, they must be in the format defined by PEG. While messages are in transit through the OS environment, they must be in the format required by the particular OS. Conversion between these two message formats is the responsibility of ***PegTask***. Many people choose to use the message format defined by PEG throughout their application.

When porting PEG to run on a particular real-time OS, it is best to make use of the OS supplied messaging system in order to achieve the greatest efficiency. If your

PegMessageQueue

PEG distribution has already been tailored to a particular real-time OS, your *PegMessageQueue* implementation has already been customized to use the underlying operating system message passing mechanism. If you are using a new RTOS, you should try to define your RTOS message queue such that little or no translation is required between PegMessage messages and RTOS messages. Most commercial RTOS implementations are designed such that the format of messages and message queues is largely user defined, which allows you to directly send PegMessage formatted messages throughout the entire system.

PEG Message Definition

Messages are defined by PEG as simple structures containing fields indicating the source, target, and content of the message. The definition of this data structure, called PegMessage, is shown below:


```

struct PegMessage
{
    PegMessage() {Next = NULL; pTarget = NULL;}
    PegMessage(WORD wVal) {Next = NULL; pTarget = NULL; wType = wVal;}
    WORD wType;
    SIGNED iData;
    PegThing *pTarget;
    PegThing *pSource;
    PegMessage *Next;

    union
    {
        void *pData;
        LONG iData;
        PegRect Rect;
        PegPoint Point;
        LONG iUserData[2];
        DWORD dUserData[2];
        SIGNED iUserData[4];
        WORD wUserData[4];
        UCHAR uUserData[8];
    };
};

```

On most systems, each PegMessage structure requires 24 bytes of memory.

Messages are identified by the member field wType. This is a 16-bit unsigned integer value, which allows 65,535 unique message types to be defined. Currently PEG reserves the first 5000 message wType values for internal messages, which leaves message values 5000 through 65,535 available for user definition. The number of messages reserved for use by PEG may change slightly in future releases, and the library therefore provides a #define indicating the first message value which is available for user definition. This #define is called **FIRST_USER_MESSAGE**.

PegMessageQueue

Message Flow and Routing

PEG follows a bottom-up message flow philosophy. This means that whenever possible messages pulled from *PegMessageQueue* are sent directly to the lowest level object that should receive the message. If the object does not act on the message, it is passed ‘up the chain’ to its parent, which may be any other type of object, such as a PegGroup or PegWindow. This flow continues until either an object processes the message, or the message arrives at *PegPresentationManager*. If a user-defined message arrives at *PegPresentationManager*, it will be ignored. This occurrence is usually an indication that you forgot to catch a message in one of your window classes.

Many messages, especially user-defined messages, may be directed towards a particular object by the pTarget message field or by the message iData field. If pTarget is anything other than NULL, the message is always sent directly to the object pointed to by pTarget. This type of message is called a *directed* message.

Other messages do not have a particular object as their target. Examples of these messages include mouse, touch screen, and keyboard messages. In these cases the pTarget member of the message is set to NULL, and it is the responsibility of *PegPresentationManager* to determine which object should receive the message. Messages of this type are referred to as *undirected* messages. We will talk more about directed and undirected messages when we discuss the multitasking capabilities of PEG in a later chapter.

When a **user-defined** message is pulled from the message queue and it has a pTarget value of NULL, *the message routing functions assume that the message iData field contains the ID of the object that should receive the message*. This means there are two ways of directing user-defined messages to particular objects. You can load the message pTarget field with an actual pointer to the destination object, which always takes precedence, or you can load the pTarget field of the message with NULL and PegPresentationManager will route the message to the first object found with an ID value matching the message iData member. If you

want to route user-defined messages using object ID values, those objects should have globally defined object IDs to insure that there are never multiple objects visible with duplicate ID values.

Whenever a PEG object sends a system-defined message to its parent window, the message contains a pointer to the object that sent the message. This pointer is contained in the message field called **pSource**. This makes it very easy to identify the sender of the message and perform operations such as modifying the appearance of the object, interrogating the object for additional information etc...

PEG System Messages

PEG messages can be divided into two types. PEG system messages, which are generated internally by PEG to control and manipulate PEG objects, and USER messages, which are defined and used by your application program. Whether a message is a system message or a user message is determined by the value of the message wType field. This is a 16 bit unsigned value. PEG reserves message wType values 1-FIRST_USER_MESSAGE - 1, which is currently equal to 5000. This leaves message types 5000 through 65535 available for user definition.

PEG uses messages internally to command objects to perform certain operations. These internally generated messages are called the system messages. PEG system messages are no different from user defined messages, with the exception that the wType values of these messages are between 1 and **FIRST_USER_MESSAGE**. The definition of these messages is determined by PEG, and PEG objects understand what to do when they receive various system messages.

In addition to defining your own messages, it is very common to want to receive and process the system messages that are generated internally by PEG. This is sometimes called 'intercepting' a message, because you can catch a message that PEG has sent to an object and change the interpretation of the message, or even cause the object to ignore the message entirely. Working examples of how to do this are provided in the programming chapter of this manual.

PegMessageQueue

While at first you may want to avoid intercepting system messages, as your confidence in working with the library grows you will find that this is often the most convenient way to accomplish many tasks. A complete list of the PEG system messages is shown below. If you are reading this chapter of the manual for the first time, we suggest that you continue on to the next section. After you have gained an overall understanding of this material, you should examine the system message definitions and read how each message is used.

System Message List

The following, while not a complete list of the PEG system messages, are the system messages that would potentially be of interest in the application level software. Additional control-specific messages are documented in the section of the reference manual that describes each particular control.

Message	Description
PM_ADD	This message can be issued to add an object to another object. The message pTarget field should contain a pointer to the parent object, and the message pSource field should contain a pointer to the child object.
PM_CLOSE	Recognized by PegWindow derived objects, and causes the recipient to remove itself from its parent and delete itself from memory
PM_CURRENT	This message is sent to an object when it becomes a member of the branch of the presentation tree which has input focus
PM_DESTROY	This message is sent to PegPresentationManager to destroy an object. The pSource member of the message should point to the object to be destroyed
PM_DIALOG_NOTIFY	This message is sent to the owner of a PegDialog when the dialog window is closed if the dialog window is

	executed non-modally. The message iData member will contain the ID of the button used to close the dialog window
PM_DRAW	This message can be sent to an object to force that object to redraw itself
PM_EXIT	This message is sent to <i>PegPresentationManager</i> to cause termination of the application program
PM_HIDE	This message is sent to an object whenever it is removed from a visible parent
PM_KEY	This message is sent to the current input object when keyboard input is received. The message iData member contains the corresponding ASCII character code, if any, and the IDData member of the message contains the keyboard scan code, if available
PM_LBUTTONDOWN	This message is sent to an object when the user generates mouse click input. <i>PegPresentationManager</i> routes mouse input directly to the lowest child object containing the click position. If the child object does not process mouse input, the message is passed up to the parent object. This process continues until an object in the active tree processes the message, or the message ends up back at <i>PegPresentationManager</i> . The position of the mouse click is included in the message Point field.
PM_LBUTTONUP	This message is sent to an object when the user releases the left mouse button. The flow of this message is identical to PM_LBUTTONDOWN
PM_MAXIMIZE	This message can be sent to any PegWindow derived object. If the target window is sizeable (as determined by the PSF_SIZEABLE status flag), it will resize itself to fill client rectangle of its parent
PM_MINIMIZE	Similar to PM_MAXIMIZE, this message can be sent to any PegWindow derived object. If the window is sizeable,

PegMessageQueue

	it will create a proxy PegIcon, add the icon to the parent window, and remove itself from its parent
PM_MW_COMPLETE	This message is sent to the owner of a PegMessageWindow when the message window is closed if the message window is executed non-modally. The message iData member will contain the ID of the button used to close the message window
PM_NONCURRENT	This message is sent to an object when it loses membership in the branch of the presentation tree which has input focus
PM_PARENTSIZED	This message is sent to all children of a PegWindow derived object if the window is re-sized. This makes it very easy for child windows that want to maintain a certain proportional spacing or position within their parent to catch this message and resize themselves whenever the parent window is sized
PM_POINTER_ENTER	This message is sent to an object when the mouse pointer (if any) passes over an object.
PM_POINTER_EXIT	This message is sent to an object when the mouse pointer (if any) leaves the object
PM_POINTER_MOVE	This message is sent to an object whenever the mouse pointer moves over the object
PM_SHOW	This message is sent to an object when it is added to a visible parent, before the object is first drawn. This allows an object to perform any necessary initialization prior to drawing itself on the screen
PM_SIZE	This message is sent to an object to cause it to re-size. This is equivalent to calling the Resize() function. Note that PEG does not differentiate between moving an object and resizing an object. Both are accomplished via the Resize operation. The new size for the object is included in the message Rect field.

PM_RESTORE	This message can be sent to any sizeable PegWindow derived object to cause that window to restore its size and position after it has been maximized or minimized
PM_RBUTTONDOWN	This message is sent in systems that support right mouse button input. PEG objects do not process right mouse button messages
PM_RBUTTONUP	This message is sent in systems that support right mouse button input. PEG objects do not process right mouse button messages
PM_TIMER	This message is sent to an object that has started a timer via the <i>PegMessageQueue</i> TimerSet function when that timer expires. The ID of the timer is included in the iData member of the message

User Defined Messages

Why would you want to define your own messages? This is the way you make your user interface do something useful when the user of your product inputs information. Your interface will be composed of any combination of PEG windows, buttons, strings, etc. along with your custom objects. At some point you will want to perform an action based on the user selecting a button, or typing into a string field. You are notified of this user input via messages sent from the PEG control to the parent window. When you create a control object, you tell the object what message to send back to the parent window when the object is modified by the user by defining the object ID value. Once you have constructed and displayed the control, you simply wait for the arrival of the message that indicates the control has been modified.

There are many other reasons you will want to define your own messages, and it will become more clear as you begin using the library. As an example to get you

PegMessageQueue

thinking in messaging terms, suppose your interface has at some point two separate but related windows visible on the screen. Let us call these windows WindowA and WindowB. WindowA displays several data values, in alphanumeric format, that can be modified by the user. WindowB displays these same data values as a line chart. When the user modifies a data value in WindowA, we want WindowB to update the line chart to reflect the new value. One way of accomplishing this is to define a new message that contains the altered data value. When WindowA is notified by one of its child controls that a value has been changed, it builds an instance of the newly defined message, places the data value into the message, and sends the message to WindowB. When WindowB receives the message, WindowB realizes that the line chart should be redrawn using the new data value.

When you define your own messages, we suggest that you do so by using an enumeration, and that you assign the value of the first enumeration equal to **FIRST_USER_MESSAGE**. For example, the following class declaration includes the typical method of defining messages for use by your application:

```
Class MyWindow::public PegWindow
{
    public:

        enum ThisWindowMessages                // my user-defined
        messages
        {
            TURN_BLUE = FIRST_USER_MESSAGE,    // always start
            with this value
            TURN_GREEN,
            TURN_INVISIBLE
        };
}
```

The above example illustrates a common way to define your own messages. We have implied in this example that you can re-use message values over and over again since the message number enumeration is a member of the class, rather than a global enumeration. This is in fact the case. As long as the object receiving the

message can clearly identify what the message means you don't have to worry about re-using the same message numbers at various points in your application.

How do you send a message from one window to another? There are three ways. First, you can either call the destination window's message handling function directly, passing your message as a parameter. Second, you can load the message **pTarget** field with the address of the window (or any object) that should receive the message and push the message into *PegMessageQueue*. Finally, you can load the message **pTarget** field with NULL, the message **iData** member with the ID of the target window, and push the message into *PegMessageQueue*. The second or third methods are generally preferred, because it adheres to the encapsulation philosophy.

If you load message **pTarget** values with pointers to application objects, you must insure that the object is not deleted before the message arrives. When a user-defined message contains a non-NULL **pTarget** value, there is no verification that the **pTarget** field of the message is a valid object pointer. For this reason, in some situations it is better to use NULL **pTarget** values, and route messages using object IDs. If *PegPresentationManager* is unable to locate an object with the indicated ID, the message is simply discarded.

There are also differences between these methods in terms of the order in which things are done. If you push a message into *PegMessageQueue*, the sending object immediately continues processing, and the target window will receive and process the new message after the sending window returns from message processing. If you call the receiving window's message handling function directly, it will immediately receive and process the message, in effect pre-empting the current execution thread. While these differences are generally inconsequential for user-defined messages, they can be very important for PEG system messages.

Signals

Messages are used to issue commands or send other information between objects that are part of your user interface. In the previous section we learned that a common use for user defined messages is to provide notification to a parent window when a child control has been modified. This usage is so common, in fact, that PEG has defined a simplified method for defining these messages and a corresponding syntax for receiving them. This method is called **Signaling**, and the messages sent and received via **Signaling** are called **Signals**. **Signals** are designed to simplify your programming effort by reducing the complexity associated with windows and dialogs containing a large number of child controls. Signals are also defined to solve some common problems associated with other less friendly messaging systems:

- Very often a single window, such as a modal dialog window, will have a large number of child objects. It can be very difficult to remember all of the unique messages associated with each of these objects.
- Complex control types, such as PegString or PegComboBox, can be modified in several different ways. The result of this is that either multiple message types must be sent by the control to the parent window, or the receiver of a single notification message would have to further interrogate the control to determine exactly why it sent a message.
- Although a control may define several different types of modification, you may not be interested in every type of control modification that can occur. In that case, you do not want the control to waste processing time by generating messages you are not interested in.
- Finally, to facilitate the implementation of a RAD window prototyping tool such as PegWindowBuilder, a consistent, simple, and robust message definition method must be in place.

PEG signaling solves each of these problems. Basically, signaling is nothing more than allowing an object to automatically generate and use multiple user-defined message types based on a single 'object ID' value. As you create a control object that uses signaling, you can further define which signals you are interested in and which you are not. This prevents the object from generating unnecessary messages and wasting CPU time.

A further advantage of the PEG signaling algorithm is that all message values related to signaling are calculated at compile time, and signaling therefore adds no overhead to the run-time performance of PEG.

When you define an object that uses signaling, you only have to specify the objects 'ID' value (which can and should be an enumerated ID name) and which signals you are interested in. In order to process signals generated by that object, you only have to remember the objects ID. In the chapter titled Programming with PEG, the example PEG Signals illustrates the use of PEG signaling.

PEG defines many different signals, or notification messages, that can be monitored for each control. Whenever the control is modified by the user, the control checks to see if you have configured it to notify you of the modification. If you have, the control automatically generates a unique message number based on the control ID and the type of notification. The message source pointer is loaded to point to the control, and the message is then sent to your parent window or dialog.

To receive a signal, PEG defines the **SIGNAL** macro, which is used in your parent window message processing function. The parameters to the **SIGNAL** macro are the object ID and the notification message in which you are interested. The **SIGNAL** macro is a shorthand method for determining the exact message number sent by a control with a given ID and corresponding to one of the 32 possible notification types.

PegMessageQueue

The example on the following page illustrates the use of signals in the definition and run-time processing of a typical dialog window. Since we have not yet discussed all of the information you need to know to fully understand this example, we don't want you to be concerned with the details (in fact, a lot of the details have been left out!). Instead, simply examine the syntax for defining the control object IDs for each of the dialog controls and the message processing statements corresponding to each control.

A few object IDs are reserved by PEG to facilitate the proper operation of PEG modal dialog and modal message windows. These object IDs are defined in the header file `pegtypes.hpp`. These reserved object ID values occupy the upper range of the possible ID values, and for this reason whenever you assign object IDs of your own they should always start with an ID value of 1.

Not all notification signals are needed or supported by all controls. For this reason, the reference documentation for each control type that uses signaling includes a list of the notification messages supported by that control.

Control ID definition and signal processing example

The example on the following page is taken from the Dialog programming example that is found in the programming chapter. We present the message handling function of the dialog window here as a preview allowing you to observe the syntax of PEG signaling. The syntax is slightly unusual, even for experienced 'C' or 'C++' developers.

The first bit of code is from the header file for the dialog window. Each child control is assigned an enumerated ID, as in "USER_NAME" and "HAS_EMAIL". In the second code segment we find the message processing function, where notifications from these controls are caught using the `SIGNAL` macro. The parameters to the `SIGNAL` macro are the ID of the control, and the notification type that we are interested in catching. This syntax has the further advantage of making the code somewhat self documenting, since as you become familiar with

PegMessageQueue

this syntax you will quickly be able to recognize the control type and notification that each case statement is processing.

PegMessageQueue

// Excerpt from MyDialog dialog window header file:

```
Class MyDialog : public PegDialogWindow
{
private:

    enum MyChildControls
    {
        USER_NAME = 1,           // string control ID
        HAS_EMAIL,               // check box ID
        EMAIL_ADDRESS,           // email address string ID
    };

};
```

// excerpt from MyDialog message processing function:

```
switch (Mesg.wType)
{
case SIGNAL(USER_NAME, PSF_TEXT_EDITDONE):
    // add code for user name modification here:
    break;

case SIGNAL(USER_NAME, PSF_FOCUS_RECEIVED):
    // add code here to bring up help for user name:
    break;

case SIGNAL(EMAIL_ADDRESS, PSF_TEXT_EDITDONE):
    // add code for email address change here:
    break;

case SIGNAL(HAS_EMAIL, PSF_CHECK_ON):
    // add code for checkbox turned on:
    break;

case SIGNAL(HAS_EMAIL, PSF_CHECK_OFF):
    // add code for checkbox turned off:
    break;
}
```

Chapter 5: PegScreen

PegScreen is the PEG class that provides the drawing primitives used by the individual PEG objects to draw themselves on the display device. PEG window and controls never directly manipulate video memory, but instead use the **PegScreen** member functions to draw lines, text, bitmaps, etc. Most importantly, **PegScreen** provides a layer of isolation between the video hardware and the rest of the PEG library, which is required to insure that PEG is easily portable to any target environment.

PegScreen is an abstraction, meaning that the **PegScreen** class is an abstract class that defines the functions and function parameters instantiable **PegScreen** derived target-specific interface classes must provide. The term abstract class is a C++ term that means the class contains one or more pure virtual functions. Pure virtual functions are simply placeholders in the class definition. They tell the compiler that the all classed derived from the **PegScreen** base class must provide working versions of the virtual functions. In addition to the virtual functions, **PegScreen** also provides functionality that is common to all implementations.

PegScreen may interface to a large variety of display devices. Standard VGA displays, super-VGA displays, LCD panels, and even printers may be driven by **PegScreen** derived interface classes. Custom implementations may of course extend the required functionality. PEG is delivered with several working examples of **PegScreen** derived interface classes. These are described in more detail in the following sections.

At this point, it should be clearer why the build procedures at the start of this manual instruct you to include different **PegScreen** derived classes depending on your target environment. The files `vgasrn.cpp`, `w32scrn.cpp` etc.. contain derived **PegScreen** classes specific to each environment.

Screen Coordinates

This is a topic of much confusion on many platforms. In some environments, screen coordinates are always relative to the upper left corner of the object that is accessing the screen. In other environments, screen coordinates are always relative to the absolute upper left corner of the screen. In most environments, some combination of the above is used.

Further, the units for screen coordinates are sometimes defined in pixels, inches, sub-inches, or an arbitrary unit derived from a basic font style.

After much discussion, the designers of PEG determined the following complex rules governing screen coordinates:

- PEG screen coordinates are in pixels. When you define the position of a window, the position is in pixels. When you define the position of a dialog, the position is in pixels. When you define the position of a button, the position is in pixels. We think you get the message.
- PEG screen coordinates are relative to the upper left corner of the screen, which is 0,0. While this does not follow trigonometric conventions, this at least is a consistent definition among graphical environments and will be familiar to users who have done previous GUI programming. PEG screen coordinates are not relative to the client area of an object, the client area of an objects parent, or relative to the day of the month. PEG screen coordinates are relative to the upper left corner of the screen, which is 0,0.

If you at some point create custom objects with custom drawing routines, you will need to insure that you always use some corner of the object's client rectangle as the reference point for your drawing routines. While this may at first appear more difficult than alternative approaches, we feel that once you gain experience with this method of coordinate resolution you will find that custom drawing from within PEG is much easier to do than with any other platform.

Video Controllers

PEG is designed to be completely independent of the target system video display channel. In order to accomplish this, the abstract ***PegScreen*** interface class is defined to allow all PEG objects to use a common set of display output functions when drawing to the screen.

PEG can be used with display devices supporting any combination of pixel resolutions and color depths. This includes LCD, HGA, CGA, VGA, and SVGA capable devices.

How Video Controllers Work

Regardless of your target's color depth and screen resolution, all video output controllers operate in at least a somewhat similar fashion. The video controller is responsible for reading pixel data from some system memory area, translating that data into pixel color and intensity values, and driving the display device with analog signals representing these values.

The memory area that contains the pixel information is called the ***frame buffer***. This term is derived from the fact that the video memory storage area usually must contain at least enough memory for one complete refresh, or frame, of the display device. Some systems have multiple frames, meaning that there is enough video memory available to store multiple pages of full-screen pixel data.

How the pixel data stored in the frame buffer and later translated into the analog values used to drive the display device is dependant on the type of display and the mode of the video controller. After determining the color or intensity value for each pixel, the video controller usually has a DAC or series of DACs that convert the digital color and intensity information into the analog value sent to the display.

If a video controller is operating in a palette mode, the pixel data stored in the frame buffer is used to retrieve color values from a set of palette registers internal

PegScreen

to the video controller chip. The values retrieved from the palette registers are applied to the output DACs to generate the analog output signal.

For color systems, the values stored in the palette registers determine the intensity of the red, green, and blue components of each pixel. These individual color components can be specified to varying resolutions, i.e. the palette registers themselves may be from 8 to 32 bits wide.

The number of palette registers available and the width of the output DACs determines how each pixel must be encoded in the video frame buffer. If a controller contains four palette registers, then at most two bits of video data are required to define each output value. If a controller contains 256 palette registers and multiple 8-bit DACs, then a full byte of video memory or more is required to define each output value.

When a video controller is operating in a pass-through or true-color mode, the pixel data contained in the frame buffer is directly applied to the output DACs. This allows an almost unlimited number of colors to be used, and additional video data may be used to define hardware alpha blending or Z-ordering information.

A further responsibility of the video controller is synchronization with the target display. The display device is composed of a matrix of dots, and the layout, density, and size of the dots are dependent on the display device. (The dot analogy is not an accurate physical description for some display technologies, however the concept still applies). In any case, the video controller must feed the display device with a unique analog value for each dot on the display. The display device contains internal circuitry to scan each dot in sequence, usually left to right and top to bottom. As the display device moves from one dot to the next, the video controller must advance to the next analog value at exactly the same moment in time. This requires precise synchronization between the video controller and display device.

A high-resolution display device may contain 1024 columns of dots, with 768 dots in each column or even higher resolution. Typical frame rates (the rate at which the

entire matrix of dots is refreshed) vary from a low of about 30 frames/second to an upper end of 70-80 frames/second. This means that for high-resolution, high-frame rate devices the video controller must output approximately 55 million analog values per second, or a new value each 20 nanoseconds. As you can see, high-end video controllers are among the most sophisticated pieces of silicon you are likely to encounter.

Brief History of Video Standards

PEG is delivered with several instantiable screen drivers derived from *PegScreen*. These screen drivers contain the software necessary to write lines, text, and other information into the video memory frame buffer. There are screen drivers which allow you to get up running quickly on a PC compatible development station, and additional *template* drivers supporting embedded targets with linear monochrome, 2 bit-per-pixel linear grayscale, 2 bit-per-pixel CGA, 4 bit-per-pixel linear, 8 bit-per-pixel palette or packed formats, 16-bit-per-pixel, and 24-bit-per-pixel video output channels. We call these drivers *templates* because they do not contain the actual video controller configuration code, which is of course specific to the target video controller. These templates do contain all of the necessary drawing code for each screen resolution and color depth, which makes it very easy to get up and running on your target system very quickly.

Why are there different derived *PegScreen* classes? Before we can answer this, we need to review some of the history (or Genesis, if you will) of video controllers.

In the beginning, there was IBM. IBM designed the PC, and IBM defined how video controllers worked. The early (up to CGA) video controller chipsets were almost exclusively based on the Motorola 6845 video controller. This standard is still apparent in many modern low-resolution video controllers. IBM later defined another specification, the VGA video controller specification. All chipsets from all manufacturers supported the same standard VGA register set, which meant that all software written for one VGA controller would run on any VGA controller. This

PegScreen

standard became the last true video controller standard, which is still supported somewhat even in today's generation of controller chips.

One major change included with the VGA specification had to do with supporting higher-resolution video modes, including 640x480x16 colors and 320x200x256 colors. ***These are the two highest resolutions included in the VGA standard.*** At the time the VGA standard was introduced, there was as yet no PCI or VESA bus, and a PC running DOS had pre-defined memory windows available for accessing VGA memory. This memory window was defined by IBM as A000:0000 through A000:ffff. For non-Intel processor folks, this is 64Kbytes of memory, total.

When we examine the highest video modes supported by the VGA standard, we find that 320x200x256 colors (i.e. 8 bits/pixel) equals exactly 64K of video memory required. That is how IBM arrived at the 320x200 video mode, which is mode 0x13 on all VGA BIOS implementations. Taking a closer look at the 640x480x16 color mode, we find that this mode requires 153,600 bytes of video memory. Since only a 64K-byte window was allocated to the video memory under DOS, a new and totally complex scheme had to be created for gaining access to the extended video memory.

The scheme implemented involves the use of four memory planes. Using the memory plane architecture involves first telling the video card which memory plane you want to write to, and then actually writing to that plane. If you only want to modify one pixel on the screen, you have to first read from each memory plane, modify the desired bit, and then write the resulting byte back out to each memory plane. This is a **VERY SLOW** way to do things, and consequently video controller manufacturers who wanted to improve video performance quickly circumvented the VGA standard.

The super-VGA (SVGA) resolutions include 640x480x256 colors, all 800x600 modes, all 1024x768 modes, and beyond. The drawback to the SVGA video controllers from a software perspective is that when these controllers were

introduced IBM was no longer driving the standard, and in fact there is no hardware-level standard. In recent years the VESA committee has been formed to bring some conformity back into the video controller arena, but the fact remains that the high-performance SVGA controllers today do not share a common register set, and do not support the same functionality.

The VESA standard has resulted in a set of video BIOS extensions for the PC which allow graphics programmers to read information about the video controller and (hopefully) use the controller in one of the high-resolution modes. This is of no use to an embedded system that has no PC BIOS, and is actually of very little use even to x86 based systems with a PC BIOS because invoking the VESA BIOS extensions is slow, slow, slow. In order to get really good performance out of the new controllers, you have to write a driver specifically for that controller. Not only can you then directly modify the controller registers instead of using the BIOS, but you can also take advantage of any and all advanced features of the controller such as hardware bit-blitting, hardware view-ports, multiple video pages, hardware mouse support and so on. This is why all high-performance video controllers come with drivers for the popular desktop graphics applications such as AutoCad and MS Windows. You simply cannot move around the number of bytes required for high-resolution, high-color depth displays without wide data bus channels and extensive on-chip pixel-moving functions.

The end result of this evolution is that it is virtually impossible for PEG to provide *optimized* driver software for every high-end video configuration available. The keywords here are *high-end* and *optimized*, since generic support for all resolutions and color depths is part of the standard PEG distribution. In fact, this generic type of video driver support is the *only* option available in many competitive graphics packages. As you will see in the following paragraphs, the PEG strategy strives towards providing the best possible video performance by creating fully tuned and optimized PegScreen based driver classes for each target platform. For lower-end LCD panels, no hardware acceleration is available, and the provided linear PegScreen drivers are the best performance solution.

PegScreen

Where does this leave you? If your target system is using a high-resolution hardware accelerated video controller, you will probably want to arrive at a ***PegScreen*** class created specifically for that controller. The functions provided by ***PegScreen*** are straightforward, and by using one the working PEG examples as a template along with the documentation for your video controller most users can accomplish writing their own version of ***PegScreen*** for hardware-accelerated video system in just a few days. If this seems intimidating, or you simply don't have the time to do this yourself, Swell Software can be contracted to perform this work for you. In many cases, we will already have a ***PegScreen*** class that is tuned for your high-end video controller, or at least similar to what you need. Even if that is not the case, contracting Swell Software to create your video driver will result in the best possible video performance, and is usually very cost-effective.

Many embedded applications also reside at the lower end of the spectrum. For example, one popular LCD display is 320x240x4 color gray scale resolution. If this is closer to what your target will be supporting, you will have very little work to do to port PEG to your target system. Your PEG distribution includes template drivers for all common video resolutions and color depths. These driver templates contain all of the necessary drawing routines, and simply require that you add the video controller configuration code specific to your controller.

One nice thing about lower color resolution systems is that a relatively small amount of video memory is required. For the above display, only 19.2 Kbytes of video memory are needed. This means that your video controller may not support some of the more advanced features, because these features simply are not needed when working with a smaller video memory. For a 4-color (2 bits/pixel) display, we can write 4, 8, or 16 pixels (depending on your data bus width) at a time, which results in terrific performance. A quick system can be built around this and similar displays using only a 16-bit CPU running at 16 MHz. A **VERY** quick system for this type of display can be built using a 16-bit CPU @ 16 MHz with a 16-bit wide video memory layout.

The provided interface class templates are usually all that is needed for these lower-resolution video controllers. After adding the video controller configuration logic, the template driver is fully complete and ready to run on your target.

Porting PEG to Your Video Hardware

Porting PEG to run on a custom or semi-custom target platform requires a custom derivation of PegScreen be created that is tuned to the bus architecture, color depth, memory organization, and pixel resolution of the target platform. Creating a custom PegScreen derived class involves 1) Configuring your video controller and 2) Optionally tuning one of the provided template drivers to make the best use of your system configuration. ***This does not mean that you are required to invent algorithms to meet the requirements of these functions.*** The required algorithms are provided in the working PegScreen examples provided with the library and are also included in the provided driver templates. You should always use the nearest provided PegScreen implementation as the basis for your custom screen driver.

Video controller chips are generally not designed to work with only one type of display. All video controllers contain programmable registers that must be initialized to make the controller function in a way that is compatible with your display device. This process of programming the video controller registers is called configuring the video controller.

The video controller and the display device stay ‘in synch’ with each other via horizontal and vertical synch timing signals. The most difficult portion of video controller configuration is insuring that the vertical and horizontal timing signals generated by the controller are within the requirements of the screen being used. This requires that the timing information provided by the screen or LCD display manufacturer be closely correlated with the registers on the video controller that control the sync timing signals. The remainder of a typical controller configuration involves informing the controller of the memory configuration, color values, etc you intend to use.

PegScreen

If your target hardware uses a linear 1 bpp (bit per pixel), 2 bpp, 4 bpp, 8 bpp, or 16 bpp linear frame buffer, the provided template drivers contain everything you need to run PEG on your target. If you are using a high-end hardware accelerated video controller, these template drivers will not take full advantage of hardware acceleration features, but they will allow you to get your system up and running quickly. Tuning your PegScreen driver to take advantage of video hardware acceleration can be accomplished as your project development progresses.

Most two and four color memory systems are linear, with adjacent pixels corresponding to adjacent bits or bit-pairs in video memory. The exception to this is that often for two-color screens separate banks or pages of video memory drive even and odd rows of pixels. Four-color video systems often follow a similar segmentation, with each fourth row of pixels coming from a common bank of video memory.

16-color video systems can be an extension of the linear memory noted above, with each nibble in video memory corresponding to a single screen pixel. A more common form of 16 color video system is the planar organization invented with the IBM VGA controller. This is one of the most difficult memory systems to understand and program, although the provided VGA screen driver utilizes exactly this type of memory layout.

256-color (and above) video systems are all basically linear in nature, requiring one or more bytes of video data for each screen pixel. Calculating pixel addresses is usually very simple for these types of systems. An added complexity for 256 color systems is the use of one or more color palettes.

PegScreen Driver Templates

Several ***PegScreen*** derived screen driver templates are provided in your PEG distribution, in addition to the working VGA, SVGA, and Win32 screen drivers. These templates are general purpose drivers accommodating a wide range of color

depths and screen resolutions. Note that these general purpose drivers do not configure the video controller, and you will need to add the controller configuration before you can use one of these general drivers. For this reason, we call these drivers *templates*. All of the necessary drawing routines are provided and ready to run. You simply have to add the video controller configuration code, and initialize the video frame buffer address. Video controller configuration is of course specific to the controller in use, and for LCD panels is also specific to the panel in use, as the controller frame rate and pixel clock must be matched to the LCD timing specifications.

In all cases, the template drivers must be able to calculate the frame buffer address for each video pixel. This is accomplished using a two-step process. In the class constructor, an array of frame buffer addresses is initialized to contain the address of the left-most pixel of each line on the display. The individual drawing functions then add an x-offset to the line starting address to arrive at the actual pixel address. This implementation improves drawing speed by eliminating the need to perform run-time integer multiplication.

The template screen drivers are organized by color resolution, or bits-per-pixel. Each of the drivers accepts a PegRect parameter to the class constructor, which defines the target system x-y pixel resolution. The template screen drivers can generally accommodate any screen resolution of the indicated color depth.

The individual template drivers are described below.

Monoscrn (.cpp, .hpp)

These modules implement a 1-bit-per-pixel monochrome screen driver. This driver class is named MonoScreen. If your target system is monochrome, you should begin with this driver template.

This driver assumes a linear frame buffer, and byte-wide video memory access. The class constructor accepts a PegRect defining the screen resolution. This allows the

PegScreen

monochrome driver to calculate the starting address of each row of video pixels. This driver also assumes that video scan lines are adjacent in video memory. In some case, the video controller may require that the frame buffer contain a ‘black’ buffer area surrounding the visible screen buffer. If this is the case for your system, you will need to modify the initialization of the array of pointers found in the class constructor.

This driver can be used with nearly any x-y screen resolution.

L2scrn (.cpp, .hpp)

These modules implement a 2 bit-per-pixel, 4 color screen driver. If your target system uses 2 bpp output, you should begin with this driver template. This driver class is named L2Screen.

This template assumes that adjacent display lines are driven by adjacent frame buffer addresses. A common variation of this scheme is that each 4th display line is driven by consecutive frame buffer rows. This is a CGA video arrangement. This template can easily be modified to support this type of system by modifying the scan line address array initialization loop contained in the class constructor.

This driver can be configured for byte, 16-bit, or 32-bit video memory access by adjusting #defines found in the driver header file. This driver can be used with nearly any x-y screen resolution.

L4scrn (.cpp, .hpp)

These modules implement a linear 4-bpp screen driver template. This driver is named L4Screen. This should not be confused with the PEG VGA screen driver, named VgaScreen, which implements a non-linear planar VGA configuration.

If your target system uses a linear frame buffer and 4 bpp video output, you should use this driver template. This driver can be used with nearly any x-y screen

resolution, and can be configured for byte, 16-bit, or 32-bit video memory access by adjusting #defines contained in the driver header file.

L8scrn (.cpp. .hpp)

These modules implement a linear 8-bpp screen driver template. This is very similar to the Win32 screen driver, with the removal of the Windows bitblitting code and the addition of software mouse pointer handling.

If your target system uses a linear frame buffer and 8 bpp video output, you should use this driver template. This driver can be used with nearly any x-y screen resolution, and can be configured for byte, 16-bit, or 32-bit video memory access by adjusting #defines contained in the driver header file.

L16scrn (.cpp. .hpp)

These modules implement a linear 16-bpp screen driver template. If your target system uses a linear frame buffer and 16 bpp video output, you should use this driver template. This driver can be used with nearly any x-y screen resolution, and can be configured for byte, 16-bit, or 32-bit video memory access by adjusting #defines contained in the driver header file.

L24scrn (.cpp. .hpp)

These modules implement a linear 24-bpp screen driver template. If your target system uses a linear frame buffer and 24 bpp video output, you should use this driver template. This driver can be used with nearly any x-y screen resolution, and can be configured for byte, 16-bit, or 32-bit video memory access by adjusting #defines contained in the driver header file.

PegScreen

vgascrn (.cpp, .hpp)

These modules implement a standard VGA screen driver. Note that this driver is not a template, but a fully functional driver that configures the video controller for 640x480x16 color graphics mode operation.

The VGA screen driver is primarily intended for use on x86 platforms. This driver can be configured to use the PC BIOS, or to directly configure the video controller using I/O accesses direct to the controller registers.

We mention the VGA screen driver in this section because if your target system uses a planar 16 color memory organization, you will want to use the VGA screen driver as the basis for your target driver. The planar memory organization adds a good deal of complexity when compared with a linear frame buffer organization, and the VGA screen driver is therefore very useful for embedded targets that use a similar video memory layout.

Profile Mode PegScreen classes

A second complete set of screen driver templates are provided for use with display devices which have been physically rotated 90 degrees clockwise or counter-clockwise. For example, consider a 320(wide) x 240(high) monochrome display screen. There is no mechanical reason why this screen cannot be mounted so as to present a 320(high) x 240 (wide) display to the end user. We refer to this as profile mode, since the display height becomes larger than the display width.

In many cases the target hardware will support this screen rotation either via video controller configuration or jumper settings of the actual display. In other cases, software rotation of the displayed data must be accomplished. For systems other than 8-bpp displays, using a double-buffering technique and rotating the data prior to display requires a large amount of bit swapping and unacceptable overhead. For this reason, PEG provides screen driver templates that are designed to run in profile mode.

PegImageConvert (chapter 11) includes settings to save the converted image data in a format that is more efficient when running with a profile mode screen driver. Therefore, when running with a profile mode screen driver you must insure that any bitmap images converted using PegImageConvert are converted using the matching rotation settings.

Each of the profile mode screen driver header files include settings to define either clockwise or counter-clockwise screen rotation.

ProScrn1 (.cpp, .hpp)

Identical to monoscrn, execept this driver supports physical rotation of the display device.

ProScrn2 (.cpp, .hpp)

Identical to l2scrn, execept this driver supports physical rotation of the display device.

ProScrn4 (.cpp, .hpp)

Identical to l4scrn, execept this driver supports physical rotation of the display device.

ProScrn8 (.cpp, .hpp)

Identical to l8scrn, execept this driver supports physical rotation of the display device.

PegScreen

Accelerated Custom PegScreen classes

A number of hardware accelerated PegScreen derived classes are available for popular high-end video controllers. The currently available drivers are also included in your PEG distribution. If you are using an accelerated controller for which you do not find a custom PegScreen class, Contact Swell Software for an updated list of the available hardware accelerated PegScreen driver classes.

Additional Documentation

The HTML reference manual provides additional information about the PegScreen classes included in your PEG distribution. Be sure to read this documentation to find the best screen class for use with your target system.

Porting to unsupported or non-linear video configurations

If you are running on a video system that is not supported by one of the standard *PegScreen* classes or one of the alternate template implementations, a *PegScreen* class will need to be created for your target system when you are ready to begin executing your application in the target environment. To this end the remainder of this chapter will also detail what is required to create a derived version of *PegScreen* for a custom hardware platform.

It is usually best to begin using PEG in one of the standard development environments using either `vgasrn.cpp` (the VGA screen driver) or `w32scrn.cpp` (the Win32 screen driver), and then move to running PEG on your target system. This allows you to become familiar with the generation operation of PEG and the video output routines. Once you are ready to begin running on your target, you will most often tailor one of the general purpose drivers to your video configuration and begin using this custom screen driver.

If you are running with an unusual video configuration, you may be required to customize the nearest template driver drawing functions. Customizing the drawing

primitive functions involves modifying the pixel address calculations and pixel masking operations found in the nearest example provided to fit the video memory layout of your target hardware. In general, before you can customize the provided drawing primitives you need to fully understand the memory organization required for your display/video controller. This varies tremendously, depending primarily on the color depth of the target display.

Due to all of the variations available in terms of color resolution, hardware acceleration, timing signals, etc.. a document describing exactly how to configure and use every video controller/display combination would fill the Library of Congress. We can provide you with some suggestions towards insuring that if you decide to undergo this process yourself you will be successful:

- 1) Read all of the documentation for your display device, especially the sections describing timing signals and supported resolutions.
- 2) Study the register set for your video controller until you have a firm grasp of how to use each register. Obtain all application notes and supporting documents available from the manufacturer of your video controller. If you try to hurry this step, you will likely meet with many hours of frustration!
- 3) ***Start simply***. Configure the video controller, and fill video memory with a single color. Insure that the display stays 'in synch', and the correct color is displayed.
- 4) Advance slowly. Don't try to add all the PegScreen functions at one time. The first function to implement is the Line() function. You don't have to create a line drawing algorithm, you only have to modify the locations in the provided line drawing function(s) where pixel addresses are calculated. Modify these calculations to fit your video memory layout. Don't worry at first about making your calculations as efficient as possible, you can return and do 'tuning' after you have things working.
- 5) Add each additional function one at a time, and test it thoroughly. If you don't need a particular function, simply define it as an inline 'do nothing' function in your derived class header file.

PegScreen

6) Once you have everything working, re-visit each function and insure that you are getting the most out of your target hardware. If your system supports 16 or 32-bit data transfer to video memory, ***use it!*** If your system provides hardware acceleration for a given function, ***use it!***

7) ***If you experience difficulty at any point in this process, call us! We are more than willing to help you get PEG running on your target platform!***

Configuration of the video controller is usually done in the constructor of the derived PegScreen class. This class is only created once, and no drawing is done before this class is constructed.

The pixel address calculations are usually done in a two step manner to improve speed. During class construction, an array of offsets (or UCHAR pointers) is created. Each entry in this array is the starting address of the corresponding row of pixels in video memory. This eliminates the need to do any multiplication operations in the individual drawing functions. The drawing functions simply add a column offset (and/or bitmask) to the starting row address found in this array. For paged memory systems, the array contains the page and offset within the page to the first pixel of each line on the display device. This two-step address calculation is the method implemented in each of the provided PegScreen driver templates, so this will provide you with a useful example of how to do things.

For very high-end video controllers, it is worth the extra effort to further customize your ***PegScreen*** class to take advantage of any advanced hardware features, such as hardware bit-blitting or hardware mouse pointer capabilities. While given sufficient time this can be accomplished by all users of PEG, you might also want to consider contracting with Swell Software to create an optimized ***PegScreen*** class for your target hardware. Our experience with a variety of video controllers often makes this the most cost effective solution.

In all cases, Swell Software will provide whatever level of support you desire, from simply responding to technical questions to actually performing the

porting development activity. In most cases Swell Software can provide you with a fully optimized PegScreen implementation is less than 40 hours of contract development time.

If you are still defining the chipset you will be using on your final target, we encourage you to contact Swell Software as early as possible in this process. We can often help you avoid costly mistakes in your hardware design, and we enjoy the opportunity to work closely with PEG users during the entire project life cycle.

PEG Palette Considerations

PEG passes color information to the screen interface class through the PegColor structure. The PegColor structure contains data fields of type COLORVAL which is defined to meet the needs of the target system running at a specified color depth. In other words, COLORVAL may be defined as 8, 16, or 32 bits depending on the color depth of the target. In the vast majority of applications, the PegColor structure contains 8-bit COLORVAL foreground and background colors, which are palette indexes or RGB color values. These values are written directly to the video frame buffer, and the video controller converts these values to the final color values visible on the target display using the active palette programmed into the video controller.

When running with a video configuration of 16 colors or less, PEG always defines a fixed system palette that is programmed into the video controller palette registers, or simply intrinsic if the video controller has no palette registers as is the case for a monochrome screen driver. These color values may be found in the header file `\peg\include\pegtypes.hpp`. PegImageConvert, described later, and the PEG image conversion classes support advanced dithering techniques allowing you to make the best possible use of the limited number of colors when displaying high-color images.

PegScreen

For 256 color systems, PEG can operate with a pre-defined fixed palette, a custom palette generated with PegImageConvert, or in packed-pixel mode. The fixed system palette, defined in the header file `\peg\include\pal256.hpp`, is defined such that the first 16 colors in this palette are identical to the fixed 16-color palette of VGA systems. The next 216 entries in the system palette are equal-spaced color values covering the spectrum of RGB values from black (0, 0, 0) to white (256, 256, 256). The remaining 24 palette entries are reserved for future use.

Custom palettes for use in 256 color systems can be generated using PegImageConvert, which is described in a later chapter. Installing and using custom palettes is the responsibility of the application level software, i.e. PEG does not implement a palette manager.

PEG is also in use with systems supporting more than 256 colors. For these color depths greater than 256 colors the PegColor structure **uForeground** and **uBackground COLORVAL** data members are increased to 16 or 32 bit values, which is done automatically via the COLORVAL data type definition. Likewise, the color definitions used by the core library which are contained in the file `\peg\include\pegtypes.hpp` are automatically modified to the equivalent 16 or 24 bit equivalent values as required for the target video system.

PEG does not require large arrays of PegColor values at any time during program execution, and therefore increasing the width of the COLORVAL data type does not have a noticeable affect on memory usage. This also has a small effect on stack use etc.. however these effects are minimal when compared with the extra CPU time required to write/read multi-byte pixel values in video memory.

Double Buffered Video Output

Double buffered output can be a beneficial technique in many graphical systems. By double buffering, we mean that the drawing routines operate within a hidden memory region, and after each drawing operation is completed the updated portion of the local buffer is copied (very quickly!) to the visible frame buffer.

While the provided PegScreen drivers are configured to write directly to the visible frame buffer, any of the provided screen drivers can easily be modified to function in an double-buffered output scheme.

Why would you want to use a double-buffered output technique? For some targets double buffering can dramatically improve the performance, or at least the perceived performance, of your video system. There are several reasons for this.

First, if your CPU is of marginal performance, it is possible that the user will be able to see drawing ‘as it happens’ when drawing is done to the visible frame buffer. While drawing may be slower than desired, this same CPU may be able to do a simple copy from one memory area to another very quickly. In this situation, the perceived performance is improved by using a double buffered output technique. The result is that the user does not see the drawing as it happens, and to the eye it appears that the screen is updated instantaneously.

A second reason for using double buffering is of course the case where the target video system supports direct DMA transfer from a memory area local to the CPU to a frame buffer that is private to the video controller. In this case, the CPU can use all available bus bandwidth while updating the local memory area, and then command the video controller to retrieve the updated region of local memory.

A final example where double-buffered output can be useful is a situation where PEG is used to drive multiple output screens. It is possible to mount several display panels adjacent to one another, and have it appear to the end user that the panels form one much larger display screen. There are several possible ways of accommodating this, however the simplest method is to define a primary frame buffer large enough to span the entire resolution of the combined screens. PEG then works as if there is actually one large screen, the sum total of the actual physical display screens. Each time a drawing operation is completed, the correct portions of the primary frame buffer are copied into the visible frame buffers for each output

PegScreen

screen. There are working PEG installations using exactly this multiple screen output technique.

The drawback to double-buffering is of course the requirement that two memory areas large enough to hold the pixel data for a complete video frame must be maintained. Unless you see a need to improve your drawing performance, you will probably not see a reason for allocating two video frame buffers.

Development Platforms

The following paragraphs outline the different development environments supported by PEG 'out-of-the-box'. These environments allow you to quickly begin using the PEG library, and are often used for an extended period while you wait for the arrival of your target hardware. ***Remember that you will not have to modify your application level code to move from one of these environments to your final target.*** You will simply modify or replace the provided ***PegScreen*** and ***PegTask*** implementations with versions supporting your target hardware and OS.

The ***PegScreen*** derived classes that are part of your PEG distribution are intended to allow you to very quickly begin developing your application program using PEG and one of the standard PC-based compiler/debugger combinations. They also provide you with working examples for use when porting PEG to your target hardware. In the event that your final target system resembles an embedded PC, or has a VGA compatible video controller, one of the provided PegScreen derived classes should run on your target platform with little or no modification. For all other users, a custom ***PegScreen*** class should be created as described above in order to maximize the potential of your CPU and video controller.

PEG is also fully integrated with leading real-time operating systems. If you have not yet chosen your RTOS, we encourage you to evaluate one of the systems supported by PEG. In any event, PEG can be integrated with virtually any professional quality RTOS, or PEG can run in a single threaded environment such as DOS, or PEG can run in a simple standalone configuration. If you have

questions regarding how to configure or integrate PEG with your target environment, we encourage you to contact Swell Software for assistance.

In the provided development environments as well as on your final target, you must include the base ***PegScreen*** class (contained in the file `pscreen.cpp` & `pscreen.hpp`) when you build the PEG library. The base class, while abstract, does provide functionality that is required by all implementations. In addition, you will add one derived ***PegScreen*** class implementation, along with the (pseudo) ***PegTask*** implementation supporting the desired OS, either DOS or MS Windows or your target system.

PEG supports DOS and MS-Windows development environments. The MS Windows environment is probably the easiest to use, because it is most easily configured in terms of target screen size and debugging tools. The DOS environment is also useful for many users. The drawback to the DOS environment is that remote debugging is required, since the DOS-based debuggers do not react well when the application under development switches the video controller to graphics mode, as PEG does. The following paragraphs provide further information about each of the provided development environments.

MS Windows

PEG is delivered with ***PegScreen*** and ***PegTask*** implementations supporting Win32 and Win16 environments. This allows you to prototype and execute your GUI application using the mature MS Windows development tools.

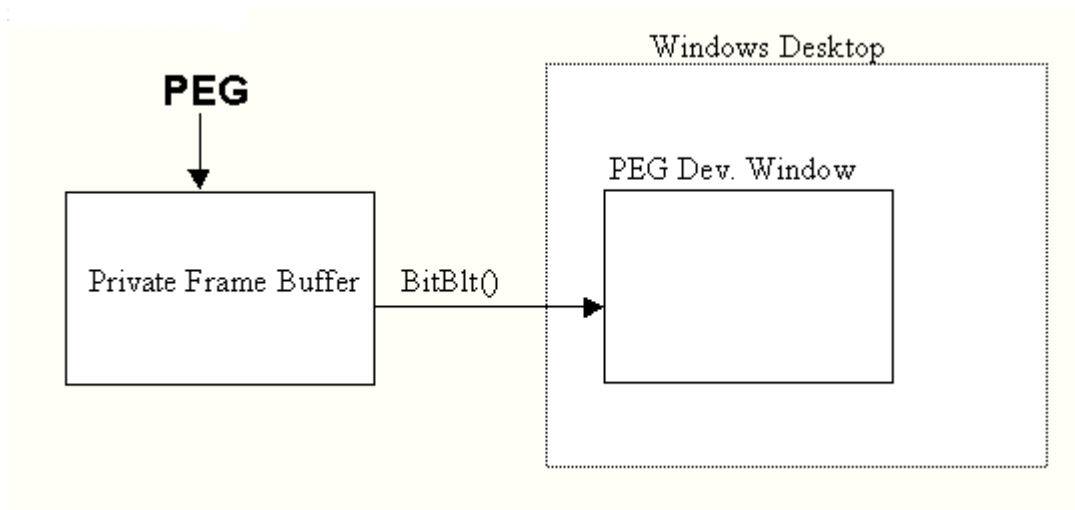
Win32

It should be noted that the PEG screen driver used for Win32 operation is actually a straightforward linear frame buffer driver. Since execution under Win32 does not allow PEG to directly access video memory, PEG actually writes all video output to a private temporary frame buffer. When drawing has been completed, this temporary frame buffer is copied into the visible window using bitmap transfer

PegScreen

functions. ***The Win32 PegScreen driver does not use the Windows GDI to implement the PegScreen drawing primitives.***

The following drawing should help clarify the operation of the PEG screen driver for the Win32 environment:



As shown above, all PEG drawing actually occurs to a privately allocated memory buffer. After each drawing operation, the PEG Win32 screen driver then transfers the updated portion of the private frame buffer to the visible window using a simple bit-blitting operation.

It should be noted that the Win32 screen driver provides a working example of double-buffered video output. Examining the section of the Win32 screen driver that transfers the local frame buffer to the visible screen provides a straightforward example of how to determine the portion of the local buffer that needs to be copied at the conclusion of a drawing operation.

While this approach does not provide the best possible performance, it is a much more representative example of how PEG will work on your target hardware. Of course when you move to your target hardware the final bit-blitting operation will be eliminated. The purpose of the Win32 ***PegScreen*** class is not to provide the best possible performance, but rather to provide you with a very robust and easy to use programming and debugging environment.

A further benefit of the implementation of the PEG screen driver for Win32 is that true WYSIWYG performance is guaranteed. Since PEG is not using the GDI, everything you see on the screen under Win32 appears exactly as it will appear on your final target system.

To build the PEG library for Win32, you need to add the files w32scrn.cpp and winpeg.cpp to your library make file.

W32scrn.cpp is a derived ***PegScreen*** class supporting Win32.

Winpeg.cpp is a small wrapper module that replaces ***PegTask*** when running under MS Windows. This module provides the WinMain() and MS-Windows-to-PEG message translation enabling PEG to run as a MS Windows application.

Win16

To build the PEG library for Win16, you need to add the files w16scrn.cpp and winpeg.cpp to your library make file.

W16scrn.cpp is a derived ***PegScreen*** class supporting Win16.

Winpeg.cpp is a small wrapper module that replaces ***PegTask*** when running under MS Windows. This module provides the WinMain() and MS Windows-to-PEG message translation enabling PEG to run as a MS Windows application.

PegScreen

DOS

PEG currently provides two derived ***PegScreen*** classes for the DOS environment. Both of these drivers directly write to the video frame buffer. The first supports 640x480x16 color VGA. While this implementation is painfully slow due to the limitations of the VGA frame buffer and the ISA architecture, it is virtually guaranteed to run on all VGA-compatible video controllers.

The second ***PegScreen*** class for DOS supports 1024x768x256 color video resolution. This implementation is much faster than the VGA version since the only overhead involves setting the correct 64Kbyte video page, but since this is a SVGA resolution, you may not be able to use this class depending on your PC chipset. Your video controller must be VESA compliant, and must support mode 0x105. If you are not sure about your video controller, you can build this version of the library, and the ***PegScreen*** class for this video mode will report an error in the required mode is not supported.

In either case, the DOS screen classes are intentionally very generic in order to provide working environments for the largest possible number of users. There is no customization done to take advantage of any high-level hardware acceleration. In addition, the DOS screen classes are written to function in DOS real mode. This requires that all video memory access is done using the slow 8-bit ISA bus, rather than one of the speedier busses created in more recent years. Consequently, most users will find that the execution performance on the final target system is much better than the development environment.

DOS VGA

To build the PEG library for a DOS environment with 640x480x16 color resolution, you should include the files `vgasrn.cpp` and `dospeg.cpp`.

`vgasrn.cpp` provides a generic derived ***PegScreen*** implementation supporting 640x480x16 color VGA.

dospeg.cpp provides a replacement **PegTask** that simply creates the required PEG system objects and passes control to **PegPresentationManager**. DosPeg further contains the functions required to poll the mouse and keyboard via the PC BIOS. Dospeg.cpp uses this BIOS-based implementation in order to insure that your development environment will be up and running quickly, regardless of the type of mouse or keyboard your workstation is using. If your target system provides a similar PC-BIOS environment, this implementation may be close to what you will need on your final target. If your system is completely unlike a PC running DOS, dospeg.cpp still provides you with a template illustrating what your final **PegTask** will need to do. The most significant change you will likely make is to replace the polled mouse and keyboard implementation with interrupt-driven equivalents for your input devices.

Notice that the polling functions in dospeg.cpp place messages in **PegMessageQueue** when mouse or keyboard events occur. This is exactly what you will need to do in your target system, although you will typically have another level of indirection in this data passing scheme. On a typical embedded target, your input devices will generate interrupts, and in turn your interrupt service routines will pass OS-type messages to **PegTask** indicating the input event. **PegTask** will then receive the OS-type message, translate that message to the equivalent PegMessage, and place the PegMessage in **PegMessageQueue**. If you need assistance with this activity the engineers at Swell Software are willing and able to work with you when porting to your final target.

DOS SVGA

To build the PEG library for a DOS environment with 1024x768x256 color resolution, you should include the files svgascrn.cpp and dospeg.cpp.

svgascrn.cpp provides a generic derived **PegScreen** implementation supporting 1024x768 SVGA. In this case the driver is still using the ISA bus for video memory access, which is very slow compared to using one of the newer hi-speed

PegScreen

PC data bus architectures. This approach is used to insure that the greatest number of PEG users are able to run PEG out of the box on their development stations. Enhanced versions of the SVGA screen driver which take advantage of newer bus architectures are available from Swell Software.

The module dospeg.cpp provides a replacement *PegTask* that simply creates the required PEG system objects and passes control to *PegPresentationManager*. See the notes on dospeg.cpp in the preceding section.

Tuning your development environment

PegPresentationManager is flexible enough to allow you to customize one of the provided development environments in order to exactly match the pixel-dimensions and very closely emulate the target screen size during your development activity. There is no requirement that *PegPresentationManager* equal the pixel-dimensions of the screen used for development, which allows you to use a subset of your PC-screen to mimic the target screen as closely as possible. A couple of examples should clarify this concept.

If your target system will be using a x-y resolution other than a standard PC resolution, you should use the next-highest resolution supported by PEG, but create a *PegPresentationManager* that is exactly the same size as your target display. For example, if your target system will be running 536x266 resolution (a strange resolution indeed!), you will need to use a *PegScreen* driver supporting 640x480 resolution or higher, and create a *PegPresentationManager* that is exactly 536x266. Remember that *PegPresentationManager* is derived from PegWindow, so you can construct *PegPresentationManager* to have any width and height you require. In this way your application screens will be forced to stay within the confines of your target system, and during development you will simply have an unused black border at the right and bottom of your screen.

It is also possible to configure your development environment so that the *physical size* of your application development screen is very close to the final target screen

size. For example, many users do the bulk of the development work using a 15", 17" or larger video display, while the final target may have a 4", 7", or similar screen size. In this case, you will want to configure your development environment for a much higher screen resolution than actually required, and then define ***PegPresentationManager*** to again be the exact pixel dimension of your target system. A typical implementation would be to configure ***PegScreen*** for 640x480 resolution, but define the ***PegPresentationManager*** limits as 320x240, which is a common LCD display resolution. In this case your application will use only the upper-left ¼ of your development screen, which in many cases is much closer to the physical size of the target LCD display. ***PegScreen*** could be configured for even higher resolution, further reducing the physical size of your screens during development.

Chapter 6: Fundamental Data Types

This chapter introduces the custom data types defined by PEG. These data types include simple 8, 16, and 32 bit data storage types, and more complex types for passing information such as color, position, and bitmap data. After you have been using PEG for a while, these data types will become second nature and you will use them as easily as you now use **char** or **int**. You will find the source code for the following definitions in the file **pegtypes.hpp**. In general, definitions and constants that globally affect all objects are contained in this file. Definitions and constants that are specific to an object type are contained in the header file specific to that object.

The following simple data types are used instead of the intrinsic data types defined by the compiler to avoid conflicts when running on CPUs with differing basic word length and data manipulation capabilities. In all cases, longer bit length types on those machines that do not accommodate 8 or 16 bit data values may replace shorter bit length types. The following definitions, contained in the file **pegtypes.hpp**, may need to be modified to match the word length of your target CPU. The comment next to each data type describes the storage requirements PEG requires for each type:

```
typedef char CHAR           // 8 bit signed
typedef unsigned char UCHAR // 8 bit unsigned
typedef short SIGNED        // 16 bit signed
typedef unsigned short WORD // 16 bit unsigned
typedef int LONG            // 32 bit signed
typedef unsigned int DWORD  // 32 bit unsigned
```

PegPoint

PegPoint is a basic pixel address data type. The x,y position is always relative to the top-left corner of the screen. PegPoint is defined as:

```
struct PegPoint
{
    SIGNED x;
    SIGNED y;
};
```

Note that PegPoint contains SIGNED data values. This means that it is perfectly normal and acceptable during the operation of PEG for at least some portion of an object to have negative screen coordinates. This simply means that the object has been moved partially or entirely off the visible screen. Of course PEG clipping methods prevent the object from trying to access the non-existent area of video memory.

Fundamental Data Types

PegRect

A large part of your programming tasks when working with a graphical interface revolve around defining and calculating rectangular areas on the screen. By providing a very complete set of operators and miscellaneous member functions, the **PegRect** class is designed to facilitate these types of operations. **PegRect** is defined as:

```

struct PegRect
{
    void Set(SIGNED x1, SIGNED y1, SIGNED x2, SIGNED y2)
    {
        wLeft = x1;
        wTop = y1;
        wRight = x2;
        wBottom = y2;
    }
    void Set(PegPoint ul, PegPoint br)
    {
        wLeft = ul.x;
        wTop = ul.y;
        wRight = br.x;
        wBottom = br.y;
    }
    BOOL Contains(PegPoint Test);
    BOOL Contains(SIGNED x, SIGNED y);
    BOOL Contains(PegRect &Rect);
    BOOL Overlap(PegRect &Rect);
    void MoveTo(SIGNED x, SIGNED y);
    void Shift(SIGNED xShift, SIGNED yShift);
    PegRect operator &=(PegRect &Other);
    PegRect operator |= (PegRect &Other);
    PegRect operator &(PegRect &Rect);
    PegRect operator ^= (PegRect &Rect);
    PegRect operator +(PegPoint &Point);
    PegRect operator ++(int x);
    PegRect operator += (SIGNED);
    PegRect operator --(int x);
    PegRect operator -= (SIGNED);
    BOOL operator != (PegRect &Rect);
    BOOL operator == (PegRect &Rect);
    SIGNED Width(void) {return (wRight - wLeft + 1);}
    SIGNED Height(void) { return (wBottom - wTop + 1);}
    SIGNED wLeft;
    SIGNED wTop;
    SIGNED wRight;
    SIGNED wBottom;
}

```

Fundamental Data Types

};

PegColor

PEG is designed to allow you do easily modify the default appearance of all the PEG objects at compile time, and also to change the appearance of individual objects at run time. PEG is internally designed to support display color depths up to 24-bpp 8-8-8 RGB operation. The color definitions used in your PEG source distribution limit PEG to only 16 colors in order to allow PEG to run out-of-the-box on most systems. There are no restrictions on the actual color depth of your display, but PEG objects will not by default make use of this extended color capability. The default color values used by each of the PEG objects are defined in the file **pegtypes.hpp**.

All PEG objects contain four primary color values. These color values determine the default background and default text colors, as well as the background and text colors to use when the object is ‘CURRENT’, or selected. Many PEG objects appear the same way whether selected or not. A few PEG objects define additional color values specific to that object type. In any event, you can modify the default color values used by PEG objects by adjusting *#defines* contained in the file **pegtypes.hpp**, and you can also modify object colors at run time by using the **SetColor()** function, described in chapter 7.

PegColor is also a parameter to many of the screen output functions you will read about shortly. PegColor informs the output functions what foreground and background colors to use, and may also modify the operation of the primitive output functions through the use of miscellaneous color flags.

PegColor is defined as:

Fundamental Data Types

```
struct PegColor
{
    PegColor (UCHAR fore, UCHAR back = PCLR_DIALOG, UCHAR Flags =
CF_NONE)
    {
        uForeground = fore;
        uBackground = back;
        uFlags = Flags;
    }
    PegColor()
    {
        uForeground = uBackground = BLACK;
        uFlags = CF_NONE;
    }
    void Set(UCHAR fore, UCHAR back = BLACK, UCHAR Flags = CF_NONE)
    {
        uForeground = fore;
        uBackground = back;
        uFlags = Flags;
    }
    UCHAR uForeground;
    UCHAR uBackground;
    UCHAR uFlags;
};
```

The member `uForeground` is generally the color used to draw text, lines, and the outline of rectangles and polygons. The member `uBackground` is generally the color used to fill rectangles and polygons and filled text.

The `uFlags` member modifies the operation of many of the screen output functions. The available flags are:

CF_NONE- This flag is used when the default operation is desired.

CF_FILL- This flag is used to fill rectangles, polygons, and text. For text functions, `CF_FILL` will cause the text background area to be filled with the

Fundamental Data Types

background color, while `CF_NONE` will cause only the text foreground to be drawn.

Fundamental Data Types

PegMessage

PegMessage defines the format of messages passed within the PEG environment. PegMessage is defined as:

```
struct PegMessage
{
    PegMessage() {Next = NULL; pTarget = NULL;}
    PegMessage(WORD wVal) {Next = NULL; pTarget = NULL; wType = wVal;}
    WORD wType;
    SIGNED iData;
    PegThing *pTarget;
    PegThing *pSource;
    PegMessage *Next;

    union
    {
        LONG iData;
        PegRect Rect;
        SIGNED iUserData[4];
        WORD wUserData[4];
        PegPoint Point;
        void *pData;
    };
};
```

On most machines, each PegMessage requires 24 bytes of RAM.

For user-defined messages, all but the wType and pTarget message fields can be used in any way desired. The iUserData, wUserData, and pData fields are intended to allow you to easily pass any type of data in your user defined messages.

PegTimer

PEG timers provide a simple means for you to receive periodic timer messages in your windows or controls. Any object derived from a PEG object can start any number of individual timers. When the timer expires, that object will receive a PM_TIMER message from PEG. The message iData member will contain the ID of the timer that expired. If the timer is started with a non-zero reset value, the timer will automatically load itself with the reset value and begin a new timeout.

PEG timers are maintained by *PegMessageQueue*. In order for PEG timers to function, your system software must call the *PegMessageQueue* member function **TimerTick** periodically to indicate to PEG that one tick time has expired. This is normally accomplished in the target specific implementation of *PegTask*. For versions of PEG which have already been customized for a particular real-time operating system, *PegTimer* is fully integrated with the operating system timer services such that an unlimited number of **PegTimers** are driven by a single OS timer.

The **TimerTick** mechanism serves two purposes. First, it insulates PEG from knowing anything about your target hardware time base. Second, it allows you to tailor the frequency in which you strobe the PEG timer. For example, very often it is not necessary for your GUI timers to be nearly as accurate as your low-level timer interrupt. Let's say that you want your PEG timers to be accurate to 50 milliseconds, while your low-level timer interrupt occurs every one millisecond. In that case, you would simply call the **PegMessageQueue::TimerTick()** function once for every ten interrupts received.

You determine the time base for PEG timers. Therefore, the value loaded in a PEG timer is simply a number of ticks, rather than any absolute time value. PEG defines the constant '**ONE_SECOND**', which should be set by you to equal the number of PEG timer ticks that will occur in one second. When you load a PEG timer, you should calculate the tick value based on this **ONE_SECOND** definition. This way if your time-base changes during program development, you will not have to track

Fundamental Data Types

down every location where you are using a PEG timer and modify the tick value used.

You start a `PegTimer` by calling the `PegMessageQueue` member function `SetTimer(SIGNED Id, SIGNED iInitial, SIGNED iRepeat)`. The parameters allow you to specify a timer `Id` value, the first timeout period, and successive timeout periods. The timer `Id` value can be any number greater than zero. If you have one window or control that creates many timers, you will probably want to assign them unique `Id` values so that you can recognize each timer expiration message.

The `iInitial` and `iRepeat` timeout periods determine how many timer ticks will expire before the timer ‘times out’, and these can be the same value. If the `iRepeat` value is zero, the timer will time-out only once and delete itself. This is a one-shot timer.

While you have an active timer running, you will receive a `PM_TIMER` message in your `Message()` handling function each time the timer expires. When you want to stop a timer, you use the `PegMessageQueue` member function `KillTimer(SIGNED Id)`. If you pass an `Id` value of zero to the `KillTimer` function, all timers owned by the calling object are deleted.

`SetTimer()` can be called at any time after an object has been constructed to start a `PegTimer`. All active timers should be deleted with a call to `KillTimer()` prior to an object being destroyed. Further information about the `PegTimer` interface functions is provided in the `PegMessageQueue` class reference, and an example using `PegTimer` is provided in Chapter 9 of this manual.

Your application level code should never instantiate a ***PegTimer*** directly. In fact, the definition of ***PegTimer*** is private to ***PegMessageQueue***, preventing you from directly using ***PegTimer*** in your application level software. ***PegMessageQueue*** provides all of the interface functions you will need to create and use ***PegTimers***. However, for completeness, the definition of ***PegTimer*** is shown below:

Fundamental Data Types

```
struct PegTimer
{
    PegTimer() {pNext = NULL; pTarget = NULL;}
    PegTimer(LONG ICnt, LONG IRes)
    {
        pNext = NULL;
        pTarget = NULL;
        ICount = ICnt;
        IReset = IReset;
    }
    PegTimer(PegTimer *Next, PegThing *Who, WORD wld, LONG ICnt, LONG
IRes)
    {
        pNext = Next;
        pTarget = Who;
        ICount = ICnt;
        IReset = IRes;
        wTimerId = wld;
    }

    PegTimer *pNext;
    PegThing *pTarget;
    LONG ICount;
    LONG IReset;
    WORD wTimerId;
};
```

PegFont

The ***PegFont*** type contains information about each font used in your application. The ***PegScreen*** text output and text information routines require a pointer to a ***PegFont*** structure as a parameter. One ***PegFont*** structure should be defined for each font you intend to use. You do not have to create these structures manually, since the ***PegFontCapture*** utility program will automatically generate this data structure for you, along with the associated offset table and data table.

By default, PEG includes and uses only two fonts, called **SystemFont** and **MenuFont**. **SystemFont** is slightly larger than **MenuFont**, and is used for the title text in windows, and the strings in text boxes and string objects. **MenuFont** is used by the PEG menu bar and menu buttons, and by the ***PegPrompt*** and ***PegTextButton*** objects. You can easily change the font used by any text-related PEG object at run time by calling the **SetFont()** member function for that object after the object has been created.

The two native fonts provided with PEG have been hand-tuned to look nice at small point sizes. Larger fonts are usually very readable without any hand-tuning, and can be used directly as they are output from the ***PegFontCapture*** utility.

Default Fonts

If you create custom fonts for use in your application, you can use them at specific times using the **SetFont()** function as described above. You can also build PEG so that any number of custom fonts are used by default for each of the PEG decorations and controls that use text. If you find that you are constantly changing the font used by a specific object type, it is much easier to simply make that the default font for that object type. The default font used by PEG for each object type is defined in the header file ***pfonts.hpp***. This header file, as delivered, contains the following definitions:

Fundamental Data Types

```
#define DEFAULT_FONT      &SysFont    // prompt, string, and textbox font
#define DEF_CELL_FONT     &MenuFont   // spreadsheet cell font
#define DEF_HEADER_FONT   &SysFont    // spreadsheet header font
#define DEF_TITLE_FONT    &SysFont    // title bar font
#define DEF_BUTTON_FONT   &MenuFont   // menu button and text button
font
#define DEF_STATBAR_FONT  &MenuFont   // status bar font
#define DEF_TAB_FONT      &MenuFont   // notebook tab font
#define DEF_GROUP_FONT    &MenuFont   // groupbox font
#define DEF_MESGWIN_FONT  &MenuFont   // message window client area
font
```

If you want to use a newly created font as the default font for any object, simply insert the address of your new font in the appropriate definition above and rebuild the PEG library. If you replace all of the default fonts with your custom fonts, you can remove the `psysfont.cpp` and `pmenfont.cpp` files from your build to save valuable memory space.

Vector Fonts

Fonts created with `PegFontCapture` are variable width bitmapped fonts. A bitmapped font has the advantages of good performance and excellent appearance. The disadvantage of bitmapped fonts is that a new font is needed for each point size and style used by the application. This can consume a large amount of memory if a large number of font sizes and styles are required.

PEG also includes support for a scalable vector font. This vector font can be used to create any number and size of bitmapped fonts *at run time*. The bitmapped font(s) thus created can then be used just like any other PEG fonts.

Vector font support requires the inclusion of additional functions and data that are not included in the library by default. You can enable support for vector fonts by turning on the definition **PEG_VECTOR_FONTS** in the file

`\peg\include\peg.hpp`. Turning this definition on and then re-building the PEG library is all that is required to install and use vector fonts.

The vector font cannot be directly used to draw text, and cannot be assigned to a PEG object using the `SetFont()` function. This is because by definition the vector font has no inherent size, but must first be ‘rasterized’ at a selected point size. This means that PEG actually draws the vector font into dynamic memory at the requested point size, creating a new bitmapped font. A pointer to the newly created font is returned by the rasterizing function. When you are done using the font, it should be deleted to insure that the memory associated with the new font is returned to the free store.

The functions used to create and delete a new bitmapped font from a vector font are detailed in the **PegScreen()** class description. These functions are named **MakeFont()** and **DeleteFont()**.

The following is an example of using a vector font in an application program. There is also a small application program that demonstrates the use of vector fonts in the directory `\peg\examples\vecfont`.

```

Void MyWindow::Draw(void)
{
    BeginDraw();
    // create 20 point font, not bold, italic:

    PegFont *MyFont = Screen()->MakeFont(20, FALSE, TRUE);

    // use MyFont here to draw text on the screen

    DrawText(....., MyFont);
    .
    .
    .
    // after the program is done with the font, it should be deleted:

    Screen()->DeleteFont(MyFont);
    EndDraw();
}

```

A bitmapped font created at run time can also be assigned to any PEG text related object such as a PegTextButton or PegString. However the user must insure that the font has been created before it is assigned to the object, and it must be deleted after the object has been destroyed.

Remember- PEG fonts may be either bitmapped or vector formats, however only bitmap format fonts can be passed directly to the PegScreen drawing and font information functions.

The ***PegFont*** structure is defined as:

```
struct PegFont
{
    UCHAR uType;           // bitmapped or vector
    UCHAR uAscent;
    UCHAR uDescent;
    UCHAR uHeight;         // max character height, in pixels
    WORD wBytesPerLine;    // bytes in each scan line in data array
    WORD wFirstChar;       // first character present in bitmap
    WORD wLastChar;        // last character present in bitmap
    WORD *pOffsets;        // pointer to offset table
    UCHAR *pData;          // pointer to data array
    PegFont *pNext;
};
```

Multilingual Support

The PegFont data structure can be used to contain almost any number of glyphs, including character sets that exceed 256 characters in number. We will discuss large character sets further in the chapters on PegFontCapture and WindowBuilder. For very large fonts the PegFontCapture will automatically generate multi-page fonts using the pNext parameter of each font to create a single PegFont containing any number of characters. Each page of the font may have different attributes, which allows for the greatest memory savings.

PegBitmap

PegBitmap is a structure used to pass bitmap data to the **PegScreen** bitmap drawing functions. PEG supports bitmaps in 1 bpp (2 color), 2 bpp (4 color), 4 bpp (16 color), 8 bpp (256 color), 16 bpp (65536 colors) or 24-bpp (8-8-8 RGB) formats. Further, PEG bitmaps may be compressed, uncompressed, or may only encode changes from a previous bitmap, which is used for displaying animations. PEG uses RLE compression techniques. While other techniques offer superior compression ratios, RLE compression offers very fast run-time performance.

The ***PegImageConvert*** utility is used to generate bitmaps in the correct format for your application. The data structure shown below, along with all of the actual bitmap data, is generated automatically by this utility. In general, bitmaps larger than 64x64 pixels should be compressed to save storage space. Also, in many cases rendering compressed bitmaps on the screen is faster than rendering the same bitmap in uncompressed format since fewer memory read operations are required.

The ***PegBitmap*** structure is defined as:

```
struct PegBitmap
{
    UCHAR uVersion;           // compressed, uncompressed, delta
    UCHAR uBitsPix;           // 2, 4, or 8
    WORD wWidth;              // in pixels
    WORD wHeight;             // in pixels
    UCHAR *pStart;            // address of bitmap data
};
```

Most PegBitmap structures used in your application will probably be generated prior to compiling using PegImageConvert. However, PegBitmap structures can also be created at run-time using various means. The PEG run-time image conversion classes allow an application to read and decompress GIF, JPG, and BMP files into PegBitmap structures at run time. Further, the PEG screen drivers contain functions allowing you to create, draw into, and finally render on the screen dynamically allocated PegBitmaps of arbitrary size.

PegCapture

The ***PegCapture*** data type is used by PEG to copy a rectangular region of the screen pixel values to or from video memory. This is used by PEG to hide and restore the mouse pointer and for various other operations, but can also be used by the application level software whenever an area of the screen needs to be saved and later restored. The ***PegCapture*** type is defined as:

```
class PegCapture
{
    public:
        PegCapture(void);
        ~PegCapture();
        PegRect &Pos(void) {return mRect;}
        PegPoint Point(void);

        void SetPos(PegRect &Rect);
        BOOL IsValid(void) {return mbValid;}
        void SetValid(BOOL bValid) {mbValid = bValid;}
        void Realloc(LONG lSize);
        void Reset(void);
        void MoveTo(SIGNED iLeft, SIGNED iTop);
        void Shift(SIGNED xShift, SIGNED yShift) {mRect.Shift(xShift, yShift);}
        PegBitmap *Bitmap(void) {return &mBitmap;}

    private:
        PegRect mRect;
        PegBitmap mBitmap;
        LONG    mDataSize;
        BOOL    mbValid;
};
```

Chapter 7: The Mighty Thing

In this chapter you will learn about the most fundamental and important class in all of the PEG library, class ***PegThing***. This chapter describes the overall capabilities of class ***PegThing***, the individual public member functions of the class, and provides several small programming examples illustrating the most common PEG programming operations. This chapter complements the information found in the HTML reference manual, i.e. here we concentrate on useful examples, while the primary goal of the HTML reference is to provide a quick lookup for function names and argument lists. A full class hierarchy diagram is also provided in the PEG class reference.

The first half of this chapter is a rather formal class reference, covering the constructors and member functions of the ***PegThing*** class. The second half of this chapter covers various important topics explaining the purpose of ***PegThing*** member variables and demonstrating the use of ***PegThing*** member functions to accomplish small programming tasks. The following chapter continues in this vein, providing more complete programming examples that will have you well on the way to using PEG to create your graphical interface.

PegThing is the base class from which all viewable PEG objects are derived. While you may never create an instance of an actual ***PegThing*** in your application, it is very possible that you will derive your own custom control types from ***PegThing***. In any event, every window and control you will use is based on ***PegThing***, so you will be using the public functions of ***PegThing*** often when programming with PEG. Learn them well!

A basic precept in the design of PEG is that all graphical objects, from the most complex tabbed notebook or table to the simplest bitmapped button, share a small but significant set of properties. Some of these basic properties include: whether or not the object is visible; if the object has a parent and who that parent is; if the

The Mighty Thing

object has children and who those children are; if the user should be allowed to interact with an object; etc. These and other properties define how each object will participate in your graphical presentation. Class ***PegThing*** maintains this information about each PEG object.

The above paragraphs are not meant to imply that ***PegThing*** is a complex or hard to understand. In fact, ***PegThing*** is actually very straightforward. The following sections will describe in detail the public functions and data members of ***PegThing***. With this information, you will gain a clear understanding of how PEG works, and you will even be able to anticipate how objects will work together when combined to perform complex interfaces. In chapter 10 of this manual you will find further descriptions and practical examples using ***PegThing*** and its derived objects.

PegThing Members

This section will describe each public member function and public variable of class PegThing. Rather than focusing entirely on formal function declarations, parameter descriptions, and return values as is done in the HTML reference, this section of the manual includes many code fragments and useful examples that illustrate how each of the member functions can be used.

This reference does NOT include every member function of the PegThing class. Please refer to the HTML reference manual for an alphabetical list of all PegThing member functions.

Constructor(s)

➤ PegThing(const PegRect &Rect, WORD wld = 0, WORD wStyle = FF_NONE)

This constructor is used when the desired initial position of the object on the screen is known at the time of object creation. Rect contains the starting screen coordinates, in pixels, for the object. The wStyle parameter indicates the object's initial drawing style.

➤ PegThing(WORD wld = 0, WORD wStyle = FF_NONE)

This constructor is used when the object position is not known at the time of object creation. When this is the case, it is necessary to define the objects position some time between when the object is created and when the object is drawn on the screen. This can be done in a derived class constructor, or when the object receives the PM_SHOW message.

The easiest way to set an objects position is to call the member function **Resize()**, which accepts a **PegRect** argument which should contain the desired screen coordinates. Calling **Resize()** is the only acceptable way to set an objects size or position **after** the object is visible.

The Mighty Thing

A more direct method of setting an objects position and size is to directly modify the objects mReal (the absolute bounding rectangle of an object) and mClient (the inside client area of an object) variables. This method must be used with caution since PEG base classes often must insure that mClient remains correctly positioned relative to mReal. Also, you should never directly modify mReal or mClient after an object is visible, since this will usually not have the desired result due to PEG clipping enforcement.

Public Functions

➤ **virtual SIGNED Message(const PegMessage &Mesg)**

This function is called by *PegPresentationManager* to allow an object to process a message. This is the most commonly overridden of all PEG functions, because customizing object behavior is done by adding your own message types and message handling code to the default operation performed by PEG.

Either messages can be those defined internally by PEG, or they can be new messages defined by you. PEG system messages are recognized by the PegMessage.wType field, which is < FIRST_USER_MESSAGE for PEG system messages. For this reason you should always insure that your user message types are greater than FIRST_USER_MESSAGE. A complete list of all PEG system messages is contained in the section of this manual entitled *PegMessageQueue*.

➤ **virtual void Draw(void)**

This function is called by *PegPresentationManager* when an object initially needs to draw itself, or by the application software when an object has been modified. This is one of the most commonly overridden functions in custom classes created by PEG users, because by overriding this function you can define a new object with

a custom appearance. An example of overriding the draw function is given in the section entitled “Introduction to PEG programming”.

Usually when you override the **Draw()** function you will allow the base-class **Draw()** function to execute at some point in your routine. A common question is “When do I call the base-class **Draw()** function?”. This depends on whether you want your custom drawing to appear on-top or below the default operation. If you want your customizations to appear ‘on-top’ (which is usually the case), you should call the base-class draw function before you do your own drawing.

In some cases you may not want to invoke the base-class **Draw()** function at all. This is perfectly OK, as long as you remember a few rules:

- 1) Start your draw function with a call to **BeginDraw()**.
- 2) After you have done your custom drawing, call **DrawChildren()** to insure child objects get their chance to draw.
- 3) After everything is done, call **EndDraw()**.

The calls to **BeginDraw()** and **EndDraw()** should actually be included regardless of whether or not you call the base-class draw function. These calls inform the PegScreen driver when a drawing sequence begins and ends. When you override the **Draw()** function, and call the base-class draw function during your drawing routine, the **BeginDraw()** calls become nested. This is expected by the PegScreen driver, which keeps track of the nesting level and recognizes when the total drawing operation is complete by tracking this **BeginDraw()-EndDraw()** nesting.

A common programming mistake is to modify some attribute of an object, for example the **PSF_ENABLED** status of the object, and tell the object to re-draw, as shown:

The Mighty Thing

```
Something->AddStatus(PSF_ENABLED);           // change the object status
Something->Draw();                             // WON'T WORK!!!
```

This never works unless by luck you are changing the object status when the object already needed to be re-drawn. Why won't this work? Because one of the performance enhancing features of *PegScreen* is that *PegScreen* only allows drawing to occur to areas of the screen which have been *invalidated*, i.e. areas which *PegScreen* has been told are no longer drawn correctly. Screen invalidation will be described in more detail in the *PegScreen* class documentation. In short, if you really want to force an object to re-draw itself on the screen, you should use a programming sequence like this:

```
Something->AddStatus(PSF_ENABLED);           // change the object status
Invalidate(Something->mClient);               // invalidate the objects
client area
Something->Draw();                             // now this works!!
```

You may wonder why an object doesn't automatically invalidate and redraw itself whenever an attribute of that object has been changed. This is a good question and requires a thorough answer.

First, directly modifying object status bits is generally not a good idea, and was done above only for example purposes. You can usually change any attributes you need to by calling object member functions. In any event, since in the above example we directly modified status bits for the object, the object doesn't know what happened and therefore there is no way it would know to re-draw.

Next, PEG objects invalidate themselves automatically when you use a member function to modify some attribute of an object that would cause the object to appear differently. You don't have to manually invalidate an object's screen area when you modify an object through a member function.

The Mighty Thing

Finally, PEG objects DO NOT re-draw themselves automatically when they are modified, even through a member function. On many GUI systems, objects redraw themselves whenever they think there is the slightest possibility that drawing might be a good idea. This is a terrific way to kill the performance of a graphical system, and leads to ugly flicker and herky-jerky operation. This philosophy also assumes that the object knows more about what is going on than the programmer, which is (almost!) never the case.

Consider for example that based on a user input you want to change the text on 3 buttons, remove a textbox, and add a couple of strings. On the desktop GUI platforms this would cause the parent window and all of its children to redraw themselves completely six times!

With PEG, you can make a single modification or a group of modifications without any intervening screens drawing occurring. When you have completed your modifications, you simply tell the parent window to redraw, which in turn causes all of the invalid child objects to re-draw themselves in one simple and fast operation.

It takes a little bit of practice to learn when to re-draw things using PEG. The best way to learn is if you are not sure, don't tell things to re-draw. When you test your application, it will quickly be obvious if something should have been told to redraw after a modification.

Is this extra programming effort worth it? Emphatically, unequivocally, **YES**. Inefficient, unneeded screen drawing can bring even the best CPU to its knees. In an embedded system, you want your CPU cycles going to the really important tasks rather than wasting them redrawing screen objects that should be keeping still!

The Mighty Thing

➤ **virtual void Add(PegThing *Who)**

This function adds **Who** to the current object. **Who** thus becomes a child of **this**. This function is used to make windows and controls members of the presentation tree.

If the object **Who** is already a member of the current object's child list, **Who** is not added again to the child list, but instead **Who** is simply moved to the front of the child list.

If **Who** is not visible at the time this function is called, and the object **this** is visible, a **PM_SHOW** message will be sent to **Who** to inform it that it has become visible. If the calling object is not visible at the time **Who** is added, and the calling object later becomes visible (by addition to a visible object), **PM_SHOW** messages will be sent at that time to the calling object and all of its children.

When constructing complex windows and dialogs, it is best to first add all of the child objects to the main window or dialog, and then add the main window or dialog to ***PegPresentationManager***. This is slightly more efficient than adding each child object to a window or dialog that is already visible.

➤ **virtual void AddToEnd(PegThing *Who)**

Similar to the **Add()** function in all respects, except this function makes **Who** the **last** child object of **this**. This is useful for controlling the tab order of child objects, since the tab order is determined by the order of child objects in the current object list.

➤ **virtual PegThing *Remove(PegThing *)**

This function removes a child object from the current object's child list. This function is the opposite to **Add()**. Attempting to remove an object which is not in the child list has no effect. When an object is removed from a visible parent, it will receive a **PM_HIDE** message to notify it that it has been removed from the screen.

The Mighty Thing

Remove() does not delete the object after it has been removed. In fact the purpose of Remove() is to allow you to remove objects from the screen without deleting them, allowing you to later re-display the object simply by re-adding it to a visible window. If you want to remove and delete an object, the ***PegThing*** member function Destroy() is provided for that purpose.

➤ **const char *Version(void)**

This function returns a pointer to the PEG library version string.

➤ **virtual PegThing *Find(WORD wld, BOOL bRecursive = TRUE)**

This function can be used to find any object based on the object ID value. For example, you may create a PegDialog window that has many child controls. If you need to modify the status of those controls as the dialog is manipulated, you will need to keep or obtain pointers to those child controls. There are two ways you could obtain a pointer to each child control. You could add member pointers to the dialog window that are initialized as each child control is constructed. This is faster than using the Find() function to locate child controls, but requires more memory to store all of the child control pointers. An alternative is to use Find() to obtain a pointer to a child control when the pointer is needed.

The following example illustrates using Find() to locate a child PegString control and testing to see if the PegString has a non-NULL string value. If the string has a null value, the dialog OK button will not close the dialog. For this example, we assume the desired string has the enumerated ID value IDS_MY_STRING:

The Mighty Thing

```
SIGNED MyDialog::Message(const PegMessage &Mesg)
{
    switch (Mesg.wType)
    {
        case SIGNAL(IDB_OK, PSF_CLICKED):

            PegString *pString = (PegString *) Find(IDS_MY_STRING);

            if (pString->DataGet())    // Does string contain text??
            {
                return PegDialog::Message(Mesg);
            }
            break;
    }
    return 0;
}
```

➤ **virtual void SetColor(UCHAR uIndex, UCHAR uColor)**

SetColor is called to override at run time an object's default color values. Every PEG object has at least four color indexes, any of which can be reset using the SetColor function. The color indexes which can be passed in *uIndex* are defined as follows:

PCI_NORMAL:	The normal client area fill color.
PCI_SELECTED:	The fill color when the object is selected.
PCI_NTEXT:	The normal text color for the object.
PCI_STEXT:	The text color to use when the object is selected.

The available *uColor* values are defined in the file `pegtypes.hpp`. These color values will vary depending on the color depth supported on the target system.

A few PEG objects such as ***PegTable*** have additional color values associated with them.

➤ **virtual void DrawChildren(void)**

This function tells each child of the current object to draw itself by calling the individual child object draw functions. In your derived classes, you do not usually need to call this function since this is normally handled automatically by PEG when you call the base class drawing function. However, if you choose not to call the base class drawing function in your custom **Draw()** function, you will usually want to call **DrawChildren()** at some point in your drawing routine to insure that objects which have been added to your parent class draw themselves.

➤ **virtual void Resize(PegRect Rect)**

Any PEG object can resize itself or any other object at any time by calling the **Resize()** function. The new screen coordinates for the objects are passed in the parameter **Rect**. If you maintain or find a pointer to another object, you can also resize that object by calling the same function. The following example illustrates this concept:

```
PegRect Rect(10, 10, 40, 40);
PegButton *MyButton = new PegTextButton(Rect, 0, "Hello");
.
. // at any time, to resize MyButton:
.
Rect.Set(20, 20, 60, 60);
MyButton->Resize(Rect);
```

If an object is visible when it is resized, it will automatically perform the necessary invalidation and drawing. It is perfectly OK to resize an object that is not visible, in fact in many cases this is the best time to do it.

➤ **virtual void Center(PegThing *Who)**

This function will adjust the screen coordinates of **Who** such that **Who** is horizontally and vertically centered over the client area of **this**. **Who** does not necessarily have to be a child of **this**, although this is the most common case. The following example demonstrates centering an object on the screen:

The Mighty Thing

```
PegRect Rect;  
Rect.Set(0, 0, 100, 100);           // create 100x100 pixel window  
PegWindow *MyWin = new PegWindow(Rect);  
Presentation()->Center(MyWin);      // center window on the screen  
Presentation()->Add(MyWin);          // make the window visible
```

➤ **void Destroy(PegThing *Who)**

This function is called to remove an object from view and delete the memory associated with that object. If the object has no parent, it has already been removed from view in which case **Destroy()** simply deletes the object. In the case that **Who == this**, **Destroy()** will post a message to *PegPresentationManager* to delete the calling object.

➤ **PegThing *Parent(void)**

Returns a pointer to the parent object, or NULL if the object has no parent (i.e. the object is not visible).

➤ **PegThing *First(void)**

Returns a pointer to the first child object in the current object's tree.

➤ **PegThing *Next(void)**

Returns a pointer to the current objects next sibling, or NULL if the current object is the end node of the current branch of the object tree.

➤ **UCHAR Type(void)**

Returns the objects enumerated type, held in the private member variable muType. This variable is used to determine the class of an object.

➤ **void Type(UCHAR uType)**

Assigns the value of the object's private muType member. This is normally done by the constructor of the PEG object, although you can define new types for your derived objects.

➤ **Void Id(WORD)**

Assigns the value of the object's mwId member. The default value is zero. Object Ids are used by PEG signaling classes to determine the message number associated with notification messages. For all other class types, muId has no effect on the internal operation of PEG, but can be useful to the application level software for identifying objects at run time.

➤ **WORD Id(void)**

Returns the value of the object's muId member. The muId value is not used by PEG directly, but is useful to the application software for keeping track of individual controls or other objects when a window such as a complex dialog has several instances of a particular object type associated with it. By assigning ID's to each object, the application can determine precisely the source of a control notification by requesting the controls mwId value.

Object IDs are also used to send and receive signals. The message number associated with a particular signal is calculated based on the object ID and the signal being sent.

➤ **BOOL StatusIs(WORD wMask)**

This function is used to test individual bits of an objects private mwStatus variable. This variable contains system status flags common to all PEG classes. An application program generally should never attempt to modify these flags, however it is sometimes useful to read this value to test for certain object states. The system

The Mighty Thing

status flags, defined in the header file `pegtypes.hpp`, are shown on the following page.

- PSF_VISIBLE- The object is visible on the screen. This flag should not be modified by the application level software. Clearing or setting this flag will not have the effect of removing or displaying the object. The *PegThing* member functions **Add** and **Remove** are used for that purpose.
- PSF_CURRENT – This flag indicates that the object is in the current branch of the display tree. If the object is a leaf object (i.e. it has no children) and it is current, then it is the object which will receive keyboard input messages.
- PSF_SELECTABLE – This flag is tested by *PegPresentationManager* to determine if an object is enabled and allowed to receive input messages. The application level software can modify this flag.
- PSF_SIZEABLE – This flag determines whether or not an object can be resized. . The application level software can modify this flag.
- PSF_MOVEABLE – This flag determines whether or not an object can be moved. The application level software can modify this flag.
- PSF_NONCLIENT – This flag , when set, allows a child object to draw outside the client area of its parent. The application level software can modify this flag after the object is constructed but before the object is displayed.
- PSF_ALWAYS_ON_TOP – This flag insures that the object is always on top of its siblings. The application level software can modify this flag.
- PSF_ACCEPTS_FOCUS - This flag indicates that the object will become the receiver of input events when selected. The application level software can modify this flag, but normally this is not advised. If this flag is modified for a particular object, it is important for correct operation that ‘breaks’ in the tree of

objects accepting focus are avoided. In other words, if a parent window cannot accept focus, then neither should any of the window's child objects be allowed to accept focus.

- **PSF_VIEWPORT** - This flag, when set, instructs *PegPresentationManager* that the object should be given a private screen viewport. Objects that have a viewport are drawn differently than objects that do not have a viewport. In general, large objects or objects which have a very complex drawing routine should be given viewports, while small or simple objects should not. By default all *PegWindow* derived objects receive viewports, and all other objects do not. This flag should not be changed except immediately after the object is constructed.

➤ **void AddStatus(WORD wMask)**

This function can be used to modify an object's mwStatus flags. AddStatus will logically OR the wMask parameter with the object's mwStatus variable. This function is used often by the PEG foundation objects to modify the state of visible window or control, but is rarely used by the application level software.

➤ **void RemoveStatus(WORD wMask)**

The opposite of AddStatus(), RemoveStatus() can be used to clear individual bits or a combination of bits in an object's mwStatus variable. This function will logically AND the complement of wMask with the object's mwStatus variable.

➤ **PegScreen *Screen(void)**

This function returns a pointer to the screen interface object. The screen interface object provides all of the drawing functions you will use in custom drawing routines. For information about how to draw on the screen, refer the *PegScreen* class reference in the HTML documentation, and to the examples which follow.

The Mighty Thing

Note:

The **Screen()** function returns the static **PegThing** member variable mpScreen. mpScreen does not have to be set in stone for the life of your application. One possible reason to temporarily replace the mpScreen pointer value is to perform screen printing operations. By defining a **PegScreen** class that drives a printer, you can easily print any PEG window by temporarily setting the mpScreen pointer to point to your print driver, telling the PEG window to re-draw, and then setting the mpScreen member back to its original value.

➤ **PegPresentationManager *Presentation(void)**

This function returns a pointer to the application's instance of **PegPresentationManager**. This value is required in order to interact directly with the top-level presentation. That is, in order to add a new window to the screen you would add the window to **PegPresentationManager** as shown:

```
PegWindow *MyWindow = new PegWindow(Rect);  
Presentation()->Add(MyWindow);
```

➤ **PegMessageQueue *MessageQueue(void)**

This function returns a pointer to the application's instance of **PegMessageQueue**. You will need to use this function in order to post messages to other windows or objects that are part of the application, and to make use of PegTimer facilities.

➤ **void FrameStyle(WORD)**

This function can be used to modify the appearance of the frame for most **PegThing** derived objects. This function is provided for convenience, and is nearly identical to the Style() function shown below with the exception that it guarantees that only the objects frame style is modified, whereas the Style() function can modify all style flags. The available frame styles are:

FF_NONE	// no frame
FF_THIN	// thin black frame
FF_RAISED	// 3D raised frame
FF_RECESSED	// 3D recessed frame
FF_THICK	// 3D thick frame

➤ **WORD FrameStyle(void)**

This functions returns the current frame style of an object.

➤ **void Style(WORD)**

This function is used to set the style flags for an object. The available style flags are shown on the following page. Not all style flags are supported by all classes. In all cases, the desired style flags can be ‘OR’ed together to form one style parameter.

As an aid in remembering the names of the style flags, the flags are grouped into different categories, and the name of each flag starts with an abbreviation of that category. For example, the frame flag names start with ‘FF’ for Frame Flag, and the button flags start with ‘BF’ for Button Flag. The functions for reading and modifying an object’s style flags are described in the following sections. The style flags are:

The Mighty Thing

```
// Frame Flags:
FF_NONE           // no frame
FF_THIN           // thin black frame
FF_RAISED        // 3D raised frame
FF_RECESSED       // 3D recessed frame
FF_THICK          // 3D thick frame

// Text Justification Flags:
TJ_RIGHT          // right justify text
TJ_LEFT           // left justify text
TJ_CENTER         // center justify text

// Title Flags:
TF_SYSBUTTON      // include a system button and menu
TF_MINMAXBUTTON   // include minimize and maximize buttons
TF_CLOSEBUTTON    // include a close button

// Button Flags:
BF_REPEAT         // auto-repeat when held down
BF_SELECTED       // default to selected appearance
BF_DOWNACTION     // send notification on down, rather than up click
BF_FULLBORDER     // slightly different border style
BF_SEPERATOR      // drawn as separator only
BF_CHECKABLE      // menu button includes checkmark
BF_CHECKED        // check-mark button is checked
BF_DOTABLE        // menu button includes radio button style dot
BF_DOTTED         // dotable menu button is selected

// Edit Flags:
EF_CUT            // allow text to be cut
EF_COPY           // object text can be copied
EF_PASTE          // object accepts text paste
EF_EDIT           // object text can be modified
EF_DRAWPARTIAL    // for multi-line text, show incomplete lines
EF_WRAP           // auto-wrap text to fit within client width

// Message Window Flags:
MW_OK             // include OK button
MW_YES            // include YES button
```



```

MW_NO                // include NO button
MW_ABORT             // include ABORT button
MW_RETRY             // include RETRY button
MW_CANCEL            // include CANCEL button

// Table and SpreadSheet Style:
TS_SOLID_FILL        // background area is filled with solid color
TS_DRAW_GRID         // table draws gridlines
TS_PARTIAL_COL        // SpreadSheet displays partial columns
TS_PARTIAL_ROW        // SpreadSheet displays partial rows

// Notebook Style:
NS_TOPTABS           // Notebook tabs appear at top of notebook
NS_BOTTOMTABS        // Notebook tabs appear at bottom of notebook
NS_TEXTTABS          // Notebook tabs contain text (not custom objects).

// Slider Control Style:
SF_SNAP              // snap to even step positions
SF_SCALE             // draw scale marks

// Spin Button Style:
SB_VERTICAL          // vertical orientation

// miscellaneous appearance flags:
AF_TRANSPARENT       // do not fill client area
AF_ENABLED           // draw active and selectable

```

➤ **WORD Style(void)**

This function returns the current style flags for an object.

➤ **Void SetSignals(WORD wSendMask)**

This function is used to identify which notification messages a signaling control should send to its parent. The mask value should be created by using the SIGMASK macro. This enables multiple signals to be enabled with one call to SetSignals, similar to the object style flags. The available signal masks are shown on the following page:

The Mighty Thing

```
PSF_CLICKED // default button select notification
PSF_FOCUS_RECEIVED // sent when the object receives input focus
PSF_FOCUS_LOST // sent when the object loses input focus
PSF_TEXT_SELECT // sent when the user selects all or a portion of a
text object
PSF_TEXT_EDIT // sent each time text object string is modified
PSF_TEXT_EDITDONE // sent when a text object modification is complete
PSF_CHECK_ON // sent by check box and menu button when
checked
PSF_CHECK_OFF // sent by check box and menu button when
unchecked
PSF_DOT_ON // sent by radio button and menu button when
selected
PSF_DOT_OFF // sent by radio button and menu button
when unselected
PSF_SCROLL_CHANGE // sent by non-client PegScroll derived objects
PSF_SLIDER_CHANGE // sent by PegSlider derived objects
PSF_SPIN_MORE // sent by PegSpinButton when up or right arrow is
selected
PSF_SPIN_LESS // sent by PegSpinButton when down or left arrow
selected
PSF_LIST_SELECT // sent by PegList derived objects, including
PegComboBox
PSF_COL_SELECT // sent when PegTable column(s) are selected
PSF_ROW_SELECT // sent when PegTable row(s) are selected
PSF_CELL_SELECT // sent when PegTable cell(s) are selected
PSF_COL_UNSELECT // sent when a PegTable column is unselected
PSF_ROW_UNSELECT // sent when a PegTable row is unselected
PSF_CELL_UNSELECT // sent when a PegTable cell is unselected
PSF_PAGE_SELECT // sent by PegNotebook when a new page is
selected
PSF_NODE_SELECT // sent by PegTreeView when TreeNode is
selected
PSF_NODE_DELETE // sent by selected TreeNode when 'Delete' key is
received
PSF_NODE_OPEN // sent by selected TreeNode if opened by user
PSF_NODE_CLOSE // sent by selected TreeNode if closed by user
PSF_KEY_RECEIVED // sent when an input key that is not supported is
received
PSF_SIZED // sent when the object is moved or sized
```

➤ **Void SetSignals(WORD wId, WORD wSignalMask)**

This function can be used to both assign an object's ID and the associated signal mask.

Public Inline Functions

Class PegThing contains a number of inline functions designed to improve the API syntax and reduce the amount of typing required when calling public functions of the **PegScreen** and **PegMessageQueue** classes. These should be thought of as pseudo-functions. They do no real work, but they are convenient and reduce unnecessary typing effort.

Since it is important to remember that these functions are actually implemented by **PegScreen** and **PegMessageQueue**, only a brief description of these functions is provided here. Complete descriptions of the actual functions are found in the respective class descriptions.

➤ **inline void SetTimer(WORD wId, LONG lCount, LONG lReset)**

Implementation: MessageQueue()->SetTimer(this, wId, lCount, lReset);

➤ **inline void KillTimer(WORD wId)**

Implementation: MessageQueue()->KillTimer(this, wId);

➤ **inline void BeginDraw(void)**

Implementation: Screen()->BeginDraw(this);

➤ **inline void EndDraw(void)**

Implementation: Screen()->EndDraw();

➤ **inline void Line(SIGNED wXStart, SIGNED wYStart, SIGNED wXEnd, SIGNED wYEnd, const PegColor &Color, SIGNED wWidth = 1)**

Implementation: Screen()->Line(this, wXStart, wYStart, wXEnd, wYEnd, Color, wWidth);

➤ **inline void Rectangle(const PegRect &Rect, const PegColor &Color, SIGNED wWidth = 1)**

Implementation: Screen()->Rectangle(this, Rect, Color, wWidth);

➤ **inline void Bitmap(PegPoint Where, PegBitmap *Getmap, BOOL bOnTop = FALSE)**

Implementation: Screen()->Bitmap(this, Where, Getmap, bOnTop);

➤ **inline void BitmapFill(PegRect Rect, PegBitmap *Getmap)**

Implementation: Screen()->BitmapFill(this, Rect, Getmap);

➤ **inline void RectMove(PegRect Get, PegPoint Put)**

Implementation: Screen()->RectMove(this, Get, Put);

➤ **inline void DrawText(PegPoint Where, const char *Text, PegColor &Color, PegFont *Font, SIGNED Count = -1)**

Implementation: Screen()->DrawText(this, Where, Text, Color, Font, Count);

➤ **inline SIGNED TextHeight(const char *Text, PegFont *Font)**

Implementation: return Screen()->TextHeight(Text, Font);

➤ **inline SIGNED TextWidth(const char *Text, PegFont *Font)**

Implementation: return Screen()->TextWidth(Text, Font);

➤ **inline void Invalidate(const PegRect &Rect)**

Implementation: Screen()->Invalidate(Rect);

➤ **inline void Invalidate(void)**

Implementation: Screen()->Invalidate(mClient);

➤ **inline void SetPointerType(UCHAR bType)**

Implementation: Screen()->SetPointerType(bType);

The Mighty Thing

Public Data Members

➤PegRect mReal

This rectangle defines the outer limits of an object, inclusive. Objects are never allowed to draw themselves outside of this rectangle.

➤PegRect mClient

This rectangle defines the client area of a window or control. In some cases mClient may be equal to mReal, but generally mClient is at least a border width of pixels smaller than mReal. Child objects are not allowed to draw outside of their parent's mClient unless they have **PSF_NONCLIENT** system status.

Using PegThing Member Functions

In this section we will examine several common programming tasks and illustrate the use of the **PegThing** class member functions. Remember that nearly all PEG classes are derived at some point from class **PegThing**, and therefore these operations can be performed from within any member function of a derived class.

The following code fragments are not complete programs! These fragments are short usage illustrations. In most cases, these fragments assume that they are executed from within a function that is a member of a class derived from one of the PEG base classes.

Determining the position of an object

One of the most basic properties of all PEG objects is the object's position on the screen. **PegThing** maintains this information, along with clipping and Z-ordering information to insure that objects are only allowed to draw to the areas of the screen that are 'owned' by the object. An object's position is held in the **PegThing** member variable **mReal**, which is a value of type **PegRect**. You can always determine where an object is at any time by examining the object's **mReal** data member. **PegThing** also maintains a separate but related member called **mClient**, which is an additional **PegRect** member that indicates the client rectangle of the object. For many objects, the client area and the real area are one and the same.

For example, by using the **mReal** variables and the **PegRect::Overlap** function, we can easily determine if two objects overlap using the following code segment:

The Mighty Thing

```
PegThing *pThing1 = First();
PegThing *pThing2 = pThing1->Next();

If (pThing1->mReal.Overlap(pThing2->mReal))
{
    // objects overlap
}
else
{
    // objects do not overlap
}
```

Obtaining a Pointer to PegPresentationManager

It is very common to require a pointer to PegPresentationManager during program execution, as you will see in the examples that follow. The PegThing member function **Presentation()** is provided for this purpose. For example, the following code segment could be used to determine if a window is a top-level window (i.e. a child of PegPresentationManager):

```
If (Presentation() == Parent())
{
    // Current object is a top-level object
}
```

Adding PEG objects

The PegThing member function **Add()** is used to attach one object to another. When an object is added to another, it becomes a child of the object it has been added to. Referring to the “tree of visible objects” described in chapter 3, the **Add()** function adds a new node to the linked list of children of the current object.

When an object is added to PegPresentationManager, it becomes visible. If other objects are then added to this ‘top level’ window, they also become visible as they

are added. An alternative and preferred method for constructing complex windows or dialogs is:

- 1) Create the top-level object.
- 2) Add the children to the top-level object.
- 3) Add the top-level object to ***PegPresentationManager***.

Using the sequence above, the top-level window and all of its children become visible at the same time when the top level window is added to ***PegPresentationManager***.

The following code segment creates a button, and adds that button directly to ***PegPresentationManager***:

```
PegTextButton *pButton = new PegTextButton(10, 10, 80, "Hello");
Presentation()->Add(pButton);
```

The following code segment creates a window, adds a button to the window, and then adds the window to ***PegPresentationManager***:

```
PegRect WinRect;
WinRect.Set(10, 10, 200, 180);
PegWindow *pWin = new PegWindow(WinRect, FF_THICK);
PWin->Add(new PegTextButton(20, 20, 80, "Hello"));
Presentation->Add(pWin);
```

Removing Objects

The ***PegThing*** member function **Remove()** is used to detach an object from the object's parent. This is the opposite of **Add()**. In addition, the ***PegThing*** member function **Destroy()** is similar to **Remove()**, although **Destroy()** both removes the object and deletes the object from memory.

The Mighty Thing

The following example removes the object pointed to by pChild from the object's parent:

```
Remove(pChild);
```

The following example removes 'pChild' from the object's parent, and deletes 'pChild':

```
Destroy(pChild);
```

Finding an object's parent

The PegThing member function **Parent()** returns a pointer to the parent of the current object. The returned pointer is also a pointer to a PegThing. If the object has no parent (i.e. the object has not been Add()-ed to another object), the **Parent()** function will return a NULL pointer.

Finding an object's children

The first child of any object can be found using the function **First()**. This returns a pointer to the head of a linked-list of child objects. The linked list can be traversed using the **Next()** function.

For example, an object could count the number of siblings (i.e. object's with the same parent) it has using the following code sequence:

The Mighty Thing

```
PegThing *pTest = Parent()->First();           // first child of my parent
int iSiblings = 0;
while(pTest)
{
    if (pTest != this)
    {
        iSiblings++;
    }
    pTest = pTest->Next();
}
```

PegThing System Status Flags

All PEG objects also have certain system status flags associated with them. The system status flags are important to the correct operation of the library, but are generally not often needed by the application software. In any event, ***PegThing*** maintains an object's system status flags, and provides public functions which allow you to examine and/or modify the system status flags for an object. The system status flags have names that start with PSF_, which stands for PEG System Flag. The system status flags and the meaning of each is listed in the function reference.

The following code segment can be used to discover which child of the current object has input focus:

```
PegThing *pTest = First();

while(pTest)
{
    if (pTest->StatusIs(PSF_CURRENT))
    {
        break;           // this object has input focus
    }
    pTest = pTest->Next();
}
```

The Mighty Thing

```
}
```

PegThing Style Flags

All PEG objects also have a set of ‘style’ flags associated with them. The style flags are very important to you as a user of the library, in that these flags allow you to easily modify many things related to how an object appears and functions. The style flags are interpreted different ways by different object types, and some style flags apply only to certain types of objects. ***PegThing*** provides functions that will allow you to read or modify an object’s style flags at any time. The style flags supported by each object type are listed in each class description.

While the library provides much diversity in allowing you to easily modify the default appearance and/or operation of an object, it is often not enough to simply modify the style flags for an object. In cases where you need to make modifications that are not controlled by the style flags, you can simply derive new classes from the base classes provided by PEG. To modify an object’s operation, you will override the **Message()** function for an object. To modify an objects appearance, you will override the object’s **Draw()** function.

The following code segment can be used to set the AF_ENABLED style flag for a button. ****Note:** this is an example only. The PegButton class provides member functions for accomplishing this task.

```
PegButton *pChild = (PegButton *) First();  
pChild->Status(pChild->Status() | AF_ENABLED));
```

Using Object Types

All PEG objects have a member variable called muType, which is a logical type indicator. You can retrieve an object’s muType value by calling the Type() function.

The Mighty Thing

If you create your own class by deriving from a PEG base class, that class will be assigned the object type of the base class. You can override this value if desired by re-assigning the object type after calling the base class constructor.

Object Type values are divided into two groups. One group is for classes derived from PegWindow, and the other group is for all other object types. When assigning your own object types, you should use the value

FIRST_USER_WINDOW_TYPE or **FIRST_USER_CONTROL_TYPE** as the base for your own custom type values. This insures that your private type values will be unique and will not overlap on the PEG class types. If your derived class has PegWindow as one of its base classes, and you assign a custom type value to this derived class, the custom value should be \geq

FIRST_USER_WINDOW_TYPE. For all other classes, use the value **FIRST_USER_CONTROL_TYPE** as the base offset for custom object types.

This can be useful when you are searching your child object list for objects of a certain type, for example PegString objects. This value is also useful when debugging since at times you may have a pointer to a *PegThing* and wish to know exactly what type of *PegThing* the pointer points to. After checking the **muType** member of a *PegThing*, you can safely upcast a *PegThing* pointer to a pointer to a specific PEG object type. The possible return values of the Type() function are defined in the header file pegtypes.hpp.

The following code fragment illustrates one possible method of locating the status bar attached to a window:

The Mighty Thing

```
PegThing *pTest = First();           // get pointer to first child object

while(pTest)                          // search to the end of list if
necessary
{
    if (pTest->Type() == TYPE_STATUS_BAR)
    {
        PegStatusBar *pStatBar = (PegStatusBar *) pTest;

        // use pStatBar to call member functions or change attributes

        break;                        // found the status bar, exit the loop
    }
    pTest = pTest->Next();             // continue down the list of children
}
```

Of course, it is simpler to call the PegWindow member function StatusBar(), which does exactly what is shown above and returns a pointer to the PegStatusBar object if one is found. However in many cases complex windows have no way of knowing what children are present without searching for them. The above example illustrates that this is a very simple process.

Using Object IDs

A few object ID values are reserved by PEG for proper operation of dialog boxes and message windows. Therefore you should always begin your private control enumeration with a value of 1, so as not to overlap the reserved ID values, which are at the very top of the valid ID range. The reserved object Id's are:

```
enum PegButtonIds {  
    IDB_CLOSE = 1000,  
    IDB_SYSTEM,  
    IDB_OK,  
    IDB_CANCEL,  
    IDB_APPLY,  
    IDB_ABORT,  
    IDB_YES,  
    IDB_NO,  
    IDB_RETRY,  
};
```

Button's with the Id's listed above are given special treatment by dialog and message window classes. For further information see ***PegDialog*** and ***PegMessageWindow***.

Valid user object ID's are in the range between 1 and 999.

Object ID values can be used to identify an object. When an object sends a notification signal to a parent window, the object ID is contained in the iData member of the notification message.

At any time you can locate a child object using the object's ID with the Find() function. Find will search the child list of the current object for an object with an ID value matching the passed in value.

Object IDs are also useful for identifying top-level windows. It is often the case that one window needs to locate another window, and one window does not know if the other window actually displayed. The following code segments illustrate using Window ID values to locate a top-level window:

The Mighty Thing

```
Window1::Window1(...) : PegDecoratedWindow(...)
{
    Id(ID_WINDOW1);
}

PegDecoratedWindow *Window2::FindWindow1(void)
{
    return Presentation()->Find(ID_WINDOW1);
}
```

Signals

All PEG objects support a basic set of signals which are listed below. ***PegThing*** provides storage for the object ID, the signal mask, and member functions for modifying the signal mask. Derived control types add additional signals unique to each control type. Some signals are turned on by default when an object is assigned a non-zero ID value, and the default signals are detailed in each class description. The full list of available signals and the meaning of each is listed in the description of the **SetSignals()** member function. The signals supported by all ***PegThing*** derived objects include:

- **PSF_SIZED** // sent when the object is moved or sized
- **PSF_FOCUS_RECEIVED** // sent when the object receives input focus
- **PSF_FOCUS_LOST** // sent when the object loses input focus
- **PSF_KEY_RECEIVED** // sent when an input key that is not supported is received

Signaling is never enabled if an object has an ID value of 0, since the message number is determined by the object ID and signal type. The **SIGNAL** macro is used to translate an enumerated object Id and signal into a specific message number. The **SIGNAL** macro is defined as:

```
#define SIGNAL (Id, Signal) (FIRST_SIGNAL_MESSAGE + (Id * 16) + Signal)
```


Overriding the Message() function

Overridden message functions should in most cases return a result of 0. A non-zero return value is used to terminate modal window execution. PegWindow derived classes such as PegDialog and PegMessageWindow return non-zero results when a signal from a child control is received that causes the window to close. In all other cases, Message() should return 0 for normal operation.

In cases where you override a PEG class's Message() function, you should make sure that you pass the messages you are not interested in down to the base class to insure that normal default operation occurs, (unless of course you are specifically intercepting a message to prevent some default operation!). In fact, if you decide to act on the receipt of a PEG system message, you should generally pass the system message down to the base class **before** you perform your own processing.

A typical Message() function for a derived class would appear as follows (assuming in this example that the class is derived from PegWindow):

```
SIGNED MyClass::Message(const PegMessage &Mesg)
{
    switch (Mesg.wType)
    {
        case PM_SHOW:
            PegWindow::Message(Mesg);

            // add your own code here:
            break;

        case USER_DEFINED_MSG1:
            // code for your user message
            break;

        case USER_DEFINED_MSG2:
            // code for another user defined message:
            break;

        case SIGNAL(IDB_OK, PSF_CLICKED):
            // code for OK button clicked:
            break;

        default:
            // pass all other messages down to the base class:

            return (PegWindow::Message(Mesg));
    }
    return 0;
}
```

Overriding the Draw() function

You can create a custom interface appearance by deriving your own control types from the PEG control types. For example, you may want to define a button type that has an appearance different than the standard PEG button types provide.

The Mighty Thing

The following code listings illustrate creating a new PegButton derived class, and overriding the Draw() function to generate a custom button appearance. In this case we are going to draw a wide button border, and use custom colors for the button client area and text. This example code can also be found in the example program file \peg\examples\robot\robobutn.cpp. You can view the appearance of this custom button class by running the example program \peg\examples\robot\winrobot.exe.

The following is the class definition for the RoboButton class:

```
class RoboButton : public PegButton
{
    public:
        RoboButton(PegRect &, WORD wld, char *Text);
        void Draw(void);
        void DataSet(char *Text)
        {
            mpText = Text;
            Screen()->Invalidate(mClient);
        }

    private:
        char *mpText;
};
```

The above class definition tells the compiler that we are defining a new class. The new class will be based on PegButton. This definition also informs the compiler that we are going to override the PegButton Draw() function, since we have defined a function named Draw() with the same return type and parameters as are defined in the virtual PegButton Draw() function. ****Note:** If your parameter list does not match the base class parameter list, the compiler will assume that you are **overloading**, not **overriding**, a base class function.

The following listing is the Draw() function implementation for the RobotButton class:

The Mighty Thing

```
void RoboButton::Draw(void)
{
    PegColor Color;
    BeginDraw();                // note 1

    if (Style() & BF_SELECTED)  // note 2
    {
        Color.uForeground = BLACK;
    }
    else
    {
        Color.uForeground = LIGHTGRAY;
    }

    // draw the top:

    Line(mReal.wLeft, mReal.wTop, mReal.wRight, mReal.wTop,
        Color, 3);              // note 3

    // draw the left:

    Line(mReal.wLeft, mReal.wTop, mReal.wLeft, mReal.wBottom,
        Color, 3);

    if (Style() & BF_SELECTED)  // note 4
    {
        Color.uForeground = LIGHTGRAY;
    }
    else
    {
        Color.uForeground = BLACK;
    }

    // draw the right shadow:
    Line(mReal.wRight, mReal.wTop, mReal.wRight,
        mReal.wBottom - 2, Color, 1);
    Line(mReal.wRight - 1, mReal.wTop + 1, mReal.wRight - 1,
        mReal.wBottom - 2, Color, 1);
    Line(mReal.wRight - 2, mReal.wTop + 2, mReal.wRight - 2,
        mReal.wBottom - 2, Color, 1);
}
```

The Mighty Thing

```
// draw the bottom shadow:
Line(mReal.wLeft, mReal.wBottom, mReal.wRight,
     mReal.wBottom, Color, 1);
Line(mReal.wLeft + 1, mReal.wBottom - 1, mReal.wRight,
     mReal.wBottom - 1, Color, 1);
Line(mReal.wLeft + 2, mReal.wBottom - 2, mReal.wRight,
     mReal.wBottom - 1, Color, 1);

// fill in the button client area:

Color.Set(LIGHTRED, DARKGRAY, CF_FILL);

Rectangle(mClient, Color);    // note 5

// draw the text centered:

PegPoint Put;
Put.x = (mClient.wLeft + mClient.wRight) >> 1;
Put.x -= TextWidth(mpText, &SysFont) >> 1;
Put.y = mClient.wTop + 1;

if (Style() & BF_SELECTED)
{
    Put.x++;
    Put.y++;
}

Color.Set(WHITE, BLACK, CF_NONE);
DrawText(Put, mpText, Color, &SysFont);    // note 6
EndDraw();                                // note 7
}
```

In the above listing, there are several points to notice. We have marked the interesting lines with “note xx” for reference.

Note 1: Always start a custom drawing function with `BeginDraw()`. This call informs the screen driver that drawing is about to begin. If this button is being

The Mighty Thing

drawn as part of a larger window drawing operation, the screen driver will recognize that this is a nested call to `BeginDraw()`.

Note2: This if statement is testing the button style flag `BF_SELECTED` to determine if the button is depressed. If the button is depressed, we want to draw the button shadow on the top and left, instead of on the right and bottom. This provides the 3D action desired for this button class.

Note 3: This statement is using the “`Line()`” wrapper function of `PegThing` to call the `PegScreen::Line()` function. The line endpoints, color and width are passed to the `Line()` function. In this case we are drawing a 3-pixel wide line, to create a wide button border.

Note 4: We are again testing the button style flag `BF_SELECTED` to toggle the border colors.

Note 5: On this line we are using the wrapper function `Rectangle()` to draw a rectangle on the screen. The rectangle will have a RED border, and will be filled with the color `DARKGRAY`. This will fill the client area of the button.

Note 6: After calculating where to draw the button text, this call writes the text for the button on the button face using the PEG font `SysFont`. Any other custom font could be used just as well.

Note 7: The `Draw()` function must end with a call to `EndDraw()` to inform the screen driver that drawing is complete. A common mistake is to forget to call the `EndDraw()` function, which can cause unpredictable results in terms of screen appearance.

You can expand on this example to create custom classes of any type. Once you define a custom class, you can use that class just like the stock PEG classes at any point in your application software. Once you become comfortable with creating

The Mighty Thing

your own classes, you will find that defining a new class such as the RoboButton class can be accomplished within an hour or two of coding and testing.

Chapter 8: Programming with PEG

We are ready to start writing some real GUI software using the PEG class library! This chapter presents additional examples to help you on your way to using the PEG library effectively. In this chapter you will learn the fundamentals of creating PEG objects such as `PegWindow` and `PegDialog` and make them appear on the display. You will also learn how to customize the appearance and behavior of PEG either by modifying object flags or by creating your own derived classes. We will also practice responding to signals generated by buttons and menu buttons.

We begin by covering a few miscellaneous topics such as the PEG variable and procedure naming conventions, memory ownership rules, and how to draw on the screen. This is followed by systematic programming examples which will allow you to begin using PEG effectively regardless of your previous level of graphical programming experience.

PEG Naming Conventions

PEG data types and class names all begin with ‘Peg’. This serves two purposes: it prevents PEG class names from conflicting with your own or with those of another included library, and it also makes it very easy to distinguish the GUI sections of your application code from those sections which have nothing to do with the graphical interface. The remaining words in a variable or procedure name always begin with a capital letter, and the rest in lower-case, as in ***PegMessageQueue***. We believe this is a very readable format.

An attempt is made to provide *meaningful* information about a variable by preceding variable names with one or two identifying letters. While this convention does not fully identify every variable type you will use when programming with PEG, we do not believe preceding a variable name with something like ‘lpszf’, as is done in other environments, is very useful either.

Variable names are always preceded by a single lowercase letter to indicate the type of variable referred to, and in `uCount` or `lLongValue`. Pointer variables are always preceded with a lowercase 'p', no matter what type of data they point to.

Class member variables are always preceded by a lowercase 'm', as in `mwStatus` (member Word) or `mpNext` (member pointer). This makes it easy to determine if a variable is a class member, as opposed to an automatic or global variable. Global variables, which are very few in PEG, are preceded with a single lowercase 'g'. Any variables not preceded with an 'm' or a 'g' are by the process of elimination automatic variables.

Constants and `#defines` are always in full uppercase to make them easily recognizable and distinguishable from variables and functions.

PEG procedure and variable names tend to be longer than you might be accustomed to. We feel that the added readability makes the long names worth a little bit of extra typing.

Source and Header files

Most PEG objects are defined in a unique header file, and the implementation of each object is contained in a unique source file. A few closely related classes share common header and source files. This makes it very easy to remove those components that are not necessary to your application when you are building the PEG library. This does not mean that you have to include 40+ PEG header files in each of your application modules that use PEG. The header file **peg.hpp** includes all of the individual PEG header files, in the correct order, and also contains the definitions used to control the build attributes in use when building the PEG library. This is the only file you should need to include in your application modules in order to use PEG.

One additional header file, **pegtypes.hpp**, contains various definitions used globally by all PEG objects. This file also contains the default color definitions

Programming with PEG

used for the various states of each PEG object. You do not have to include `pegtypes.hpp` separately in your source files.

Static/Global classes

If you are an experienced 'C' programmer, you have most likely encountered the start-up routines that are typically required to initialize non-zero global data values before your program enters 'main()'. In C++, the situation gets more complicated because not only does the memory storage for global classes need to be allocated, the class **constructor**, if defined, must also be called for each global or static class to properly initialize the member variables. Depending on the quality of the documentation provided with your C++ compiler, properly executing this startup code can be a large technical hurdle. For this reason, PEG defines no global or static class instances, which means that you can safely forget about this portion of your startup code unless your application defines global or static class instances.

Program Startup Review

In order for your PEG application to run, the ***PegPresentationManager***, ***PegMessageQueue***, and ***PegScreen*** objects must be created. This is usually the first thing done in ***PegTask***. You will find several examples of this in the code which is part of your distribution. For the Win32 environment, these required objects are created from within `WinMain()` in the file `\peg\source\winpeg.cpp`. For the DOS environment, this is done from within `main()` in the file `\peg\source\dospeg.cpp`.

After the required PEG support classes have been created, `PegAppInitialize()` should be called by your PEG startup code. This segmentation was introduced in order to allow you to easily move from one of the PEG development environments to your target platform without modifying any of your application level software. You simply replace the startup module used for the development environment with an equivalent startup module designed for your target environment.

PegAppInitialize() is where you need to create the first object or objects which will be displayed by your application, and add them to the ***PegPresentationManager***. Examples of performing this operation are included in the programming examples which follow.

Rules Of Memory Ownership

This section is a brief tutorial regarding the memory management technique employed within PEG. This is required to insure that your system software does not suffer from memory leaks or other common memory problems

In our experience it seems that most memory problems result from a lack of clear documentation of how and when allocated memory should be deleted and by whom. There are of course also just bad programming practices, such as sharing pointers to globally allocated data blocks between unrelated objects, which usually lead to trouble.

For PEG objects, the rules are simple. When an object is added (i.e. attached) to another object, PEG owns that object. You do not have to worry about deleting that object as long as you have passed it on to PEG. PEG ensures that all children of an object, along with the object itself, are deleted when the parent object is closed.

For example, suppose you create a dialog window using the **new** memory allocation operator. After creating the dialog, you also create a dozen or so controls and add them to the dialog. At this point all of the controls are owned by the dialog. All that you need to do to delete all of the allocated memory is delete the dialog.

Next, assume that you add the dialog to **PegPresentationManager** (i.e. the dialog is now visible). At this point, you have given up all ownership of the dialog and the dialog's child controls. PEG is now responsible for insuring that the dialog and its child controls are deleted from memory when the dialog is closed.

Finally, assume that you manually **Remove()** the dialog from **PegPresentationManager**, without allowing the dialog to close itself in response to user input. In this case, you again own the dialog, because **PegPresentationManager** no longer has any knowledge of the dialog's existence. However, the controls which were added to dialog are still owned by the dialog, so

once again all that needs to be done to delete all memory associated with the dialog and its controls is to delete the dialog.

Creating PegThings

As you will notice when you review the PEG class hierarchy, all viewable PEG classes are derived at some point from ***PegThing***. ***PegThing*** doesn't really do much in terms of what you see on the screen, but it is this common foundation that allows ***PegPresentationManager***, ***PegScreen***, and ***PegMessageQueue*** to perform their tasks so effectively. This is also the underlying reason why PEG objects are so flexible, as you will learn for yourself once you begin using the library.

PegThing contains information about the physical location of the objects on the screen, the client area of an object, the clipping area of an object, the system status flags for the object (selected, sizeable, etc..), and pointers used to maintain the object's position in the presentation tree.

PegThing provides the member function **Add(*PegThing* *what)**, which is how you add one control or window or any PEG object to another. When you call **Add(*PegThing* *what)**, you are inserting **what** into the current object's child list. If the current object is visible, the newly added object becomes visible. The best way to create a complex window is to create the window, create all of the window's child objects and add them to the window, and finally add the window to ***PegPresentationManager***. In this way the window and all of the child objects become visible at the same time.

PegPresentationManager is also a ***PegThing***. This means that internally to PEG there is no difference between adding a complex window to ***PegPresentationManager*** and adding a simple button to a dialog. In both cases you are simply adding one object to another, with the object **added to** becoming the parent of the object **being added**. Another implied feature of this internal structure is that for very complex applications involving multiple physical display devices, it is possible to create multiple ***PegPresentationManagers***.

A further result of the PEG class hierarchy is that it is perfectly reasonable to create a PEG object that you would normally consider to be a self-contained bottom level

object, such as a `PegPrompt`, and add another object, such as a `PegButton`, to the `PegPrompt`. The result is a `PegPrompt` that first displays the text associated with the prompt, and then allows its child objects to draw themselves. In this example, the `PegButton` would appear next to or over the prompt text, depending on the `Prompt` dimensions and text justification flags. While this result may not appear very useful, you should be able to see that by deriving your own version of `PegPrompt` specifically for this purpose, you could easily create a powerful new object type simply by combining these two PEG defined objects.

The following code fragment illustrates the ease of creating and displaying new windows using PEG. The window created will have a title, menu bar, and status bar:

```
// from with another PEG object message handling function, or from with
PegAppInitialize():
.
.
PegRect WinSize;
WinSize.Set(10, 10, 120, 200);
Presentation()->Add(new AppWindow(WinSize, "My First Window"));
.
.

AppWindow::AppWindow (PegRect Rect, char *Title): PegDecoratedWindow(Rect)
{
    Add(new PegTitle(Title));           // add a title to myself
    Add(new PegMenuBar(MainMenu));     // and a menu bar

    PegStatusBar *pStat = new PegStatusBar();
    pStat->AddTextField(80, "Hello");
    pStat->AddTextField(20, "How are you today?");
    Add(pStat);                         // and a status bar
}
```

Deleting/Removing PEG Things

For some reason, deleting objects often causes more confusion and programming errors than creating them in the first place. PEG attempts to make removing and deleting your GUI objects as painless and mistake-free as possible.

The first thing to understand is that removing a ***PegThing*** (i.e. a window, button, dialog, or other ***PegThing*** derived object) from its parent is not the same as deleting it. Removing an object means that you are taking that object out of the active display tree. After being removed, the object no longer has a parent, and it will not be visible. It is possible, even common, to later re-add the object to a visible ***PegThing*** and use it over again.

You remove a ***PegThing*** by calling the ***PegThing*** member function `Remove(PegThing *What)`. It doesn't matter if you tell the parent object to remove the child, or if you tell the child to remove itself, because the `Remove()` function properly handles either case. That is, it is perfectly acceptable to use the following statement:

```
Remove(this);
```

when an object decides based on some message input that it is time to go away.

While `Remove()` can be useful, it is more common to want to both remove the object from its parent, as well as delete the object from memory. There are three acceptable ways to remove and delete a PEG object:

- 1) Send a `PM_DESTROY` message to ***PegPresentationManager***. The `pSource` member of the `PM_DESTROY` message should point to the object which is to be destroyed. This method is most often used when deleting PEG objects from tasks outside of PEG.

- 2) Call the *PegThing* member function `Destroy(PegThing *Who)`. Any *PegThing* can destroy any other *PegThing*, including itself. This does not mean that the `Destroy()` function will end up executing a `delete(this)` statement. The `Destroy` function checks to see if **Who == this**, and if so automatically sends a `PM_DESTROY` message to *PegPresentationManager* to finish the job.
- 3) If you have manually removed a *PegThing* from its parent through use of the `Remove()` function, and the object to be deleted is not **this**, nor is **this** a child of the object being deleted, it is fine to simply delete the object.

In no cases should you ever execute a `delete(this)` statement. **When in doubt, it is always safe to call `Destroy()`**, as PEG insures that the rules of good C++ memory cleanup are followed.

It is not necessary to manually delete the individual children of a *PegThing*, in fact it will cause errors if you attempt to do this. If you are not clear on this subject you should re-read the section of this document entitled “Rules of Memory Ownership.”

Drawing to the Screen

You can draw on the screen at any time by calling the `PegScreen` class drawing functions. This is most often done from within an overridden ***Draw()*** function, as was described in the previous chapter. When you override a `Draw()` function, you simply start with a call to ***BeginDraw()***, do any amount of drawing, and complete your drawing by calling ***EndDraw()***. When PEG recognizes that an object needs to be re-drawn (i.e. the object was just `Add()`-ed, or the object has been moved), it re-draws the object by calling the object’s ***Draw()*** function.

You can also write functions that draw on the screen outside of the ***Draw()*** function. These functions must be members of a *PegThing* derived class, or at least have access to a *PegThing* object, since all of the `PegScreen` drawing functions

Programming with PEG

require as a parameter a pointer to the PegThing object calling the drawing function. PegScreen requires this pointer to insure that an object is not allowed to draw outside of the area it 'owns' on the screen.

PegScreen only allows drawing to occur to areas of the screen which have been ***invalidated***. Areas of the screen are invalidated by calling the Invalidate() function, which is a member of PegScreen but also provided in inline form as a member of PegThing. Under most circumstances the screen invalidation is handled automatically by PEG as the user moves things around on the screen, or as your program adds and removes visible objects. If all of your drawing is done from within an overridden Draw() function, you don't need to worry about screen invalidation, since your Draw() function is called specifically ***because*** an area of the screen has been invalidated.

If you need to draw on the screen outside at random times, or for example based on a periodic timer, you need to remember to invalidate the area you are going to draw to before you start drawing. If you want to be allowed to draw anywhere within the client area of your object, you can simply call the Invalidate() function with no parameters, which invalidates the area of the screen corresponding to an object's client area. You can also calculate and specify a more limiting rectangle to clip your drawing, and pass that rectangle to the Invalidate() function. No matter how large the invalidated rectangle on the screen, you are never allowed to draw outside of an object's borders.

The following function is an example function that could be used to draw a series of lines to the screen at any time. This example will paint the entire client area of the object black, and then fill the client area of the object with RED horizontal lines, 1 pixel wide, spaced 4 pixels apart.

```

void MyObject::DrawLines(void)
{
    PegColor LineColor(RED, BLACK, CF_FILL);
    SIGNED yPos = mClient.wTop;

    Invalidate();                // invalidate my client area
    BeginDraw();                 // prepare for drawing

    Rectangle(mClient, Color, 0); // fill with black

    while(yPos <= mClient.wBottom)
    {
        // draw red lines:
        Line(mClient.wLeft, yPos, mClient.wRight, yPos, Color);
        yPos += 4;
    }
    EndDraw();
}

```

Drawing to Memory

An alternative to drawing directly to the screen is to draw to an off-screen bitmap. Once you have drawn to an off-screen bitmap, you can display that bitmap on the screen at any time, at any location, by calling the PegScreen **Bitmap()** member function. This is the preferred method of displaying flicker-free animation, and can be used for many other purposes as well.

In PEG drawing to memory works almost exactly like on-screen drawing. You still use the PegScreen member functions to draw to an off-screen bitmap. Off-screen drawing is more of a ***drawing mode***, which is to say that you tell the PegScreen class that you are going to draw to a bitmap, you draw to the bitmap, and then you tell the PegScreen class to return to normal operation.

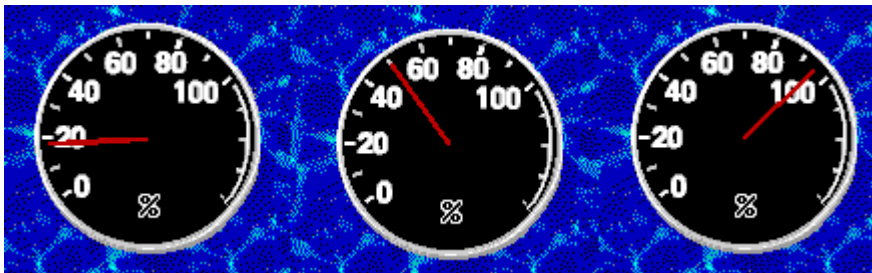
Before you can draw to an off-screen bitmap you must of course get yourself a bitmap to draw to. You do this by calling the PegScreen member function

Programming with PEG

CreateBitmap(). CreateBitmap() returns a pointer to a PegBitmap, as you would expect. The actual format of the PegBitmap is determined by the instantiated screen driver, however you don't really need to know the format of the bitmap to draw into it. The screen driver in use knows how to draw to the bitmap.

Once you get a pointer to a PegBitmap by calling the CreateBitmap() function, you can draw into that bitmap or display it at any time. You draw into the bitmap by calling an alternate form of the BeginDraw() function that accepts a pointer to the bitmap you want to draw into. When you call this alternate BeginDraw function, the screen driver is placed into the off-screen drawing mode. When you are done drawing into the bitmap, you call a similar alternate form of the EndDraw() function to put the PegScreen driver back to normal drawing mode. While the PegScreen driver is in the off-screen drawing mode, you can use any of the normal PegScreen drawing functions to draw into the bitmap.

The following is a screen shot of a program that draws a gauge using off-screen drawing techniques. This example is described in more detail on the following pages.



The following example is a code fragment demonstrating off-screen drawing. This example is provided in source code form in your distribution under the directory \peg\examples\gauge. This program draws a pressure gauge on the screen, as shown in the screen shot above. It does this by drawing a blank gauge bitmap off-screen, drawing the 'needle' or position indicator on top of the background bitmap,

Programming with PEG

and then copying the off-screen bitmap to the visible screen. The resulting effect is that the gauge updates smoothly without any noticeable flicker.

Programming with PEG

```
/*-----*/
// The Draw() function simply copies the off-screen bitmap
// to the visible screen.
/*-----*/
void Gauge::Draw(void)
{
    if (!mpBitmap)    // first time??
    {
        // Create the bitmap, and draw into it:

        mpBitmap = Screen()->CreateBitmap(gbDialBitmap.wWidth,
            gbDialBitmap.wHeight);
        DrawToBitmap();
    }

    // now just copy the bitmap to the screen:

    BeginDraw();
    PegPoint Put;
    Put.x = mReal.wLeft;
    Put.y = mReal.wTop;
    Bitmap(Put, mpBitmap);
    EndDraw();
}

/*-----*/
// This function draws to an off-screen bitmap. This function
/*-----*/
void Gauge::DrawToBitmap(void)
{
    PegPoint Put;
    SIGNED x1, y1, x2, y2;
    SIGNED iRadius = (gbDialBitmap.wWidth / 2) - 8;

    // Open the bitmap for drawing:

    Screen()->BeginDraw(this, mpBitmap);           // Note 1
    Put.x = Put.y = 0;
```

```

// copy the background bitmap into memory bitmap:
Bitmap(Put, &gbDialBitmap);                                // Note 2

// find the center:
x1 = mReal.Width() / 2;
y1 = mReal.Height() / 2;

// find the end:
double angle = ((4.0 * PI) / 5.0) + (((double) miCurrent / 100.0) * PI);
x2 = x1 + cos(angle) * iRadius;
y2 = y1 + sin(angle) * iRadius;

// draw the indicator line:
PegColor LineColor(RED, RED, CF_NONE);                      // Note 3
Line(x1, y1, x2, y2, LineColor, 2);

// Close the bitmap for drawing:
Screen()->EndDraw(mpBitmap);                                // Note 4
}

```

As shown above, the Gauge class **Draw()** function simply copies the off-screen bitmap to the screen. The first time the **Draw()** function is called, the Gauge class uses the PegScreen member function **CreateBitmap** to create the off-screen bitmap large enough to hold the gauge drawing.

The real work of drawing the gauge is done in the class member function **DrawToBitmap**. This function draws the background of the gauge and the gauge indicator line into an off-screen bitmap, which can then be copied to the screen at any time. The following notes should clarify the operation of the **DrawToBitmap** function:

Note 1: This line of code is calling the alternate form of **BeginDraw()**, passing not only a pointer to the Gauge class itself, but also a pointer to the bitmap that the class wants to draw to. This places the PegScreen driver into off-screen drawing mode.

Programming with PEG

Note 2: This line of code is drawing a pre-generated bitmap (the bitmap was generated prior to compile using `PegImageConvert`) into the memory bitmap. This bitmap forms the background of the gauge.

Note 3: This line of code draws the gauge indicator line. The line is draw from the center of the gauge to the outside edge. This line is also drawn to the off-screen bitmap, not to the visible screen.

Note 4: The off-screen drawing mode is terminated by calling the alternative form of `EndDraw()`, which accepts as a parameter a pointer to the bitmap that was drawn to. The off-screen bitmap is now ready to be displayed.

Object Boundries:

All PegThing derived classes have two rectangles associated with them, named mReal and mClient. The rectangle mReal defines the outermost limits of an object. The object and all children of the object are prevented from drawing outside the mReal rectangle.

The mClient rectangle defines the interior boundaries of an object. The mClient rectangle is always a sub-set of the mReal rectangle. All children of an object are clipped to the parent's mClient rectangle, unless the children have PSF_NONCLIENT system status, in which case they are clipped to the parent's mReal rectangle.

For simple objects such as PegButton and PegString, the mClient rectangle is smaller than the mReal rectangle only by the width of the object border. If the object has no border, the mClient and mReal rectangles are identical.

For PegWindow and derived classes, the mClient rectangle is further reduced by the size of the non-client decorations such as a title bar, menu bar, status bar, and horizontal and vertical scroll bars. In other words, non-client children are positioned in the region between the mClient rectangle limits and the mReal rectangle limits.

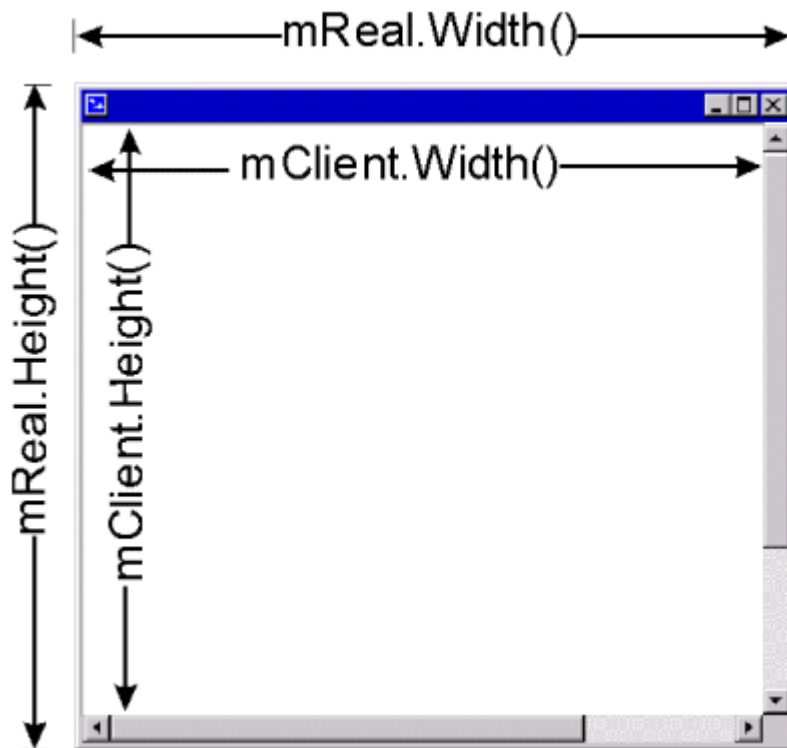
The rectangle you pass to most PEG object constructors defines the outermost limits of the object, hence this rectangle becomes the mReal member rectangle. PEG objects initialize their mClient area by calling the PegThing member function InitClient(), which reduces the mClient area by the object border width. PegWindow performs further operations to reduce the mClient area as decorations are added to the window.

You can create your own non-client area decorations and add them to PEG objects. When you do this, you will also need to add the necessary logic to the parent of

Programming with PEG

these objects to insure that the `mClient` rectangle is reduced correctly to allow space for your new non-client area decorations.

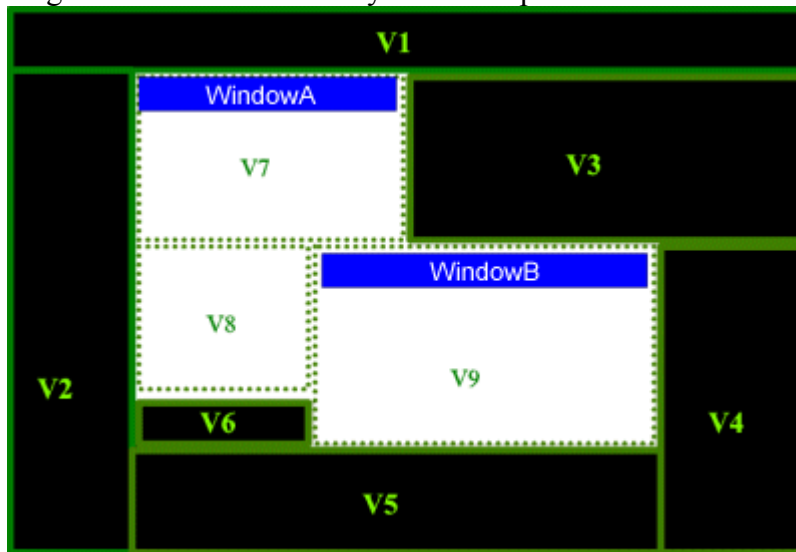
The following diagram illustrates the relationship between the `mReal` and `mClient` member rectangles:



Viewports

PEG uses the concept of viewports to improve drawing efficiency and to allow background drawing operations to occur without overwriting foreground graphics.

Viewports are rectangular areas of the screen owned by certain objects. Each viewport has only one owner, while one object may own several viewports. The diagram below should clarify this concept:



In the diagram above, a typical run-time screen is shown. The black area is the screen background, covered by PegPresentationManager. The two white areas are PEG windows, named WindowA and WindowB. WindowB is on top and partially covering WindowA. In this diagram, the solid outlines depict the viewports owned by PegPresentationManager. In this case, PresentationManager owns viewports V1-V6. WindowA is divided into two viewports, V7 and V8. Finally, WindowB is on top and has one viewport, V9.

PEG maintains the screen viewports, and you do not ordinarily have to concern yourself with how they work. There is one exception, however, that you may need

Programming with PEG

to be aware of. Normally, only PegWindow derived objects have viewport status. That means that other smaller objects like PegButton and PegIcon do not own viewports, and simply inherit the viewport(s) of their parent window.

The viewport management algorithm employed by PEG does not allow there to be breaks in the viewport tree. That is, an object that owns viewports (i.e. a PegWindow derived object) should only be added to another object that owns viewports. This does not mean that you cannot add PegWindow derived objects to objects that are not derived from PegWindow, because you can. However, when you do this you should set the PSF_VIEWPORT status flag of the parent object, to make it a viewport owner.

An example should clarify this concept. Suppose you want to create a simple object container class. This container class will serve as a parent for a group of lists, windows, and other controls. This is a common thing to do, as it allows you to add and remove the entire group of objects at any time simply by adding or removing the container. Since the container class does not need to actually draw anything, you decide to derive it from PegThing, the most basic PEG class. Since at least some of the children of the PegThing container are PegWindow derived objects, you will need to make the PegThing container class a viewport owner. If you don't do this, the PegWindow derived children of the container class won't show up on the screen. You can make the PegThing container class a viewport owner by adding the PSF_VIEWPORT system status in the container class constructor:

```
AddStatus(PSF_VIEWPORT);
```

Now your container class will work correctly, and both PegWindow derived children and simple children will be displayed when the parent container class is displayed.

Programming Examples

The following example programs will allow you to put all you have learned to use. These examples are intentionally not as complex as most real-world applications, allowing you to concentrate on the concepts being presented. However, these examples are fully functional and can be useful as references when developing your own custom application.

In order to gain the most benefit from the following examples, you will need to have a supported compiler and be ready to build and execute these programs. Before you proceed, you should insure that you are able to build and execute the PEG demo application.

This chapter contains several working example programs with complete descriptions of how each program works. The example programs start very simply, and progress to the point of deriving custom windows and dialogs. These examples, while small, are representative of real-world applications in terms of complexity. Large application programs can generally be reduced to using the following techniques repeatedly. In other words, this is ‘as tough as it gets’, and there are no hidden programming tips or secret functions you still need to learn. After you work through these examples, you will know all that is needed to be very productive using the PEG library.

The use of **WindowBuilder** makes it possible to create PEG application programs without ever hand-coding many of the functions and techniques we will present here. You might wonder, therefore, why we are presenting things from the ground up. We feel that you should understand everything about how your PEG application program runs, regardless of whether you are hand-coding your windows and controls or using **PegWindowBuilder** to define them for you. In the end, you should understand how the source code generated by WindowBuilder works.

The instructions in the following examples assume you are using the MS VC++ development environment. You can use any environment you prefer, however you

Programming with PEG

will have to translate the build instructions into instructions that work for your environment.

Example 1- Getting Started

In this example we will create a PegMessageWindow and add the window to PegPresentationManager. This will familiarize you with the PEG startup procedure and give you a chance to verify that you are able to build and run correctly.

For this example, you should create or open a new workspace. The workspace should contain only the PEG library project file. If your workspace contains additional projects, remove them before continuing.

Using your favorite editor or the editor included in the MSVC++ environment, create a file named “startup.cpp” and enter the following lines into the file (****Note:** Each of these example programs are included in the HTML format programming manual. If you are working from the printed manual, you may want to switch to the online version. This will allow you to ‘cut and paste’ the example code, rather than typing it all in by hand.)

```
#include "peg.hpp"

void PegAppInitialize(PegPresentationManager *pPresent)
{
    PegMessageWindow *pWin = new PegMessageWindow("Hello World", "My First
        Window!", FF_RAISED|MW_OK); // note 1
    pPresent->Center(pWin);           // note 2
    pPresent->Add(pWin);               // note 3
}
```

Save the file and insert a new project into your workspace named “pegstart”. Add the file “startup.cpp” to your project, and build. If all goes well, startup.cpp will compile, link with the PEG library and generate the executable file pegstart.exe. You can now run the program to verify how it works.

In the above example we have written a very simple version of PegAppInitialize. PegAppInitialize is called during PEG startup to allow you to define the initial

Programming with PEG

window or windows that will be displayed. In this case, we have created an instance of the stock `PegMessageWindow` and used that as our first window (note 1).

In the source code line labeled “note 2” , we call the `PegThing` function “Center”. Since `PegPresentationManager` is derived from `PegThing`, all of the public `PegThing` member functions are also member functions of `PegPresentationManager`. The `Center()` function centers the message window within `PegPresentationManager`, which effectively centers the window on the screen.

The last line of the `PegAppInitialize` function adds the new window to `PegPresentationManager`. This is how we make the window visible. When you run the program, you see the message window centered within the screen. When you click on the OK button, the window closes and the PEG application program terminates.

You should note that there are a lot of things you did not have to do to create this little program. You did not actually create the OK button, you did not tell the window to close, and you did not tell the window how to draw itself. These functions are all built in to PEG and the `PegMessageWindow` class that you used. All that you had to do is create an instance of `PegMessageWindow`, display it, and let PEG do the rest. Wasn’t that easy??

Example2- Using PegTimer

For this example you will create a derived PegDecoratedWindow class. In this derived window class you will override the Message() function to provide custom functionality. The custom operation is to start a periodic PegTimer and wait for timer expiration messages to arrive. The window will change colors each time the timer expires.

Modify the startup.cpp file you created in example 1 to contain the following:

```
#include "peg.hpp"
#include "startup.hpp"

void PegAppInitialize(PegPresentationManager *pPresent)
{
    PegRect WinRect;
    WinRect.Set(0, 0, 100, 100);
    MyWindow *pWin = new MyWindow(WinRect);
    pPresent->Center(pWin);
    pPresent->Add(pWin);
}
```

Programming with PEG

```
// This is the derived window class constructor:

MyWindow::MyWindow(const PegRect &Rect) : PegDecoratedWindow(Rect,
FF_THIN)
{
    Add(new PegTitle("A colorful Window!"));
    RemoveStatus(PSF_SIZEABLE);
    miColor = 0;
}

// This is the overridden message handling function:

SIGNED MyWindow::Message(const PegMessage &Mesg)
{
    switch(Mesg.wType)
    {
    case PM_SHOW:
        PegDecoratedWindow::Message(Mesg);
        SetTimer(1, ONE_SECOND * 2, ONE_SECOND / 2);
        break;

    case PM_TIMER:
        SetColor(PCI_NORMAL, miColor);
        Invalidate();
        Draw();
        miColor++;
        miColor &= 0x0f;
        break;

    case PM_HIDE:
        KillTimer(1);
        PegDecoratedWindow::Message(Mesg);
        break;

    default:
        return PegDecoratedWindow::Message(Mesg);
    }
    return 0;
}
```

In the above `Message()` function, our derived window class catches three different messages. These are the system messages `PM_SHOW` and `PM_HIDE`, and the `PM_TIMER` messages generated by our timer. The source code entered for each message type is often called a ‘message handler’, i.e. it is the code we want to run to handle each message we are listening for.

The `PM_SHOW` message is received when the window is first displayed. This is a convenient place to start the timer. In this case we set the timer to wait 2 seconds before the first timeout, and to expire every 500 ms (`ONE_SECOND / 2`) thereafter.

This timer ID value is simply set to ‘1’. If you are using many timers, you will probably want to enumerate the timer ID values, but in this case we are only using 1 timer and so we simply hard-coded the timer ID value. Note that the `PM_SHOW` message handler also passes the message on down to the base `PegDecoratedWindow` class. It is important to pass system messages on down to the base class in case the base class is also catching the message.

The `PM_HIDE` message is received when the window is removed. This is a convenient place to stop the timer. You must remember to kill timers that you have started before your windows are deleted, or `PM_TIMER` messages will be sent to invalid destinations and your software will most likely crash. Since a window is always removed before it is deleted, the `PM_HIDE` system message handler is an excellent place to kill any active timers.

The `PM_TIMER` message handler is where we change the window color and re-display the window. A member variable has been defined named `miColor`, and we will use this variable to keep track of which color to display next. This example assumes we are running in 16-color mode, and therefore prevents the color index from passing 15 (0x0f).

Note that after changing the window color by using the `SetColor()` function, we have to tell the window to re-draw. The window does not automatically redraw

Programming with PEG

since you may make several changes to the window and you do not want each change to cause a window re-draw operation.

After making these changes to the file “startup.cpp”, save and close the file. Create a new file in the same directory called “startup.hpp”, and enter the following lines:

```
class MyWindow : public PegDecoratedWindow
{
    public:
        MyWindow(const PegRect &Rect);
        SIGNED Message(const PegMessage &Mesg);
    private:
        SIGNED miColor;
};
```

After you have entered these lines, save the file “startup.hpp”

This header file defines the MyWindow class. As shown above, derived classes do not have to be overly complex. In this case, we simply tell the compiler that MyWindow is derived from PegDecoratedWindow, prototype the class constructor function, and indicate that the Message function is being overridden.

You can now build and execute this new version of startup.hpp. You may want to use your debugger to place breakpoints at the PM_SHOW, PM_HIDE, and PM_TIMER message handlers to verify when each message is received.

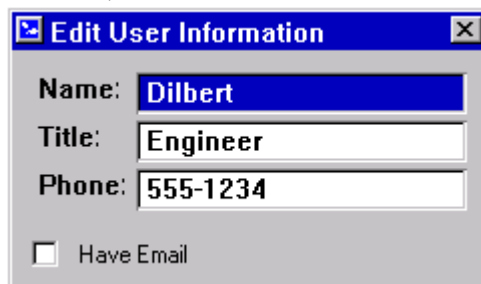
Example 3: More Message Handling and Signals

In this example you will create a new PegDialogWindow and override the window **Message()** function to provide custom operation. You will also learn to use **Signals** to make your dialog window operate interactively with the dialog user.

In your source code distribution, you should find the directory `\peg\examples\dialog`. This directory contains the source files required for this example.

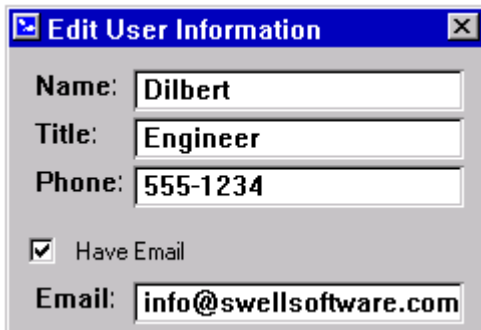
- Create a new Borland or Microsoft Win32 project, called 'dialog'.
- Add the appropriate PEG library you built earlier to your project.
- Add the file **dialog.cpp** to your project.
- Make sure that you include path for your compiler contains the directory `\peg\examples\dialog`, or your compiler may not be able to find the header file `drawwin.hpp`.
- Build the application to generate `dialog.exe`.

Execute `dialog.exe`. You should see a dialog window with several buttons and other controls, as shown below:



Programming with PEG

This dialog, while not very complex, illustrates how to catch messages generated by PEG controls. When you click on the ‘Have Email’ check box, the dialog box grows to include the email string, as shown below:



The dialog has changed size, and a new PegString field has been added to allow the user to type in an email address.

Let’s examine the source code for this application. The first function contained in the file `dialog.cpp` is **PegAppInitialize()**, shown below. As described earlier this is the entry point to your application program. In this case we created a new window of type **DialogWin**, centered the window within `PegPresentationManager`, and displayed the window by adding it to `PegPresentationManager`. **DialogWin** is the newly derived class we have created to make this application operate as shown above.

```

/*-----*/
// PegAppInitialize- called by the PEG library during program startup.
/*-----*/
void PegAppInitialize(PegPresentationManager *pPresentation)
{
    // create the dialog and add it to PegPresentationManager:

    PegRect Rect;
    Rect.Set(0, 0, 240, 140);
    DialogWin *pWin = new DialogWin(Rect);
    pPresentation->Center(pWin);
    pPresentation->Add(pWin);
}

```

The next function contained in the file `dialog.cpp` is the constructor for the newly derived class `DialogWin`. Since we want the new window to have the general appearance of a `PegDialog` object, we have derived `DialogWin` from `PegDialog`. The full source code for the `DialogWin` constructor is shown below.

Programming with PEG

```
/*-----*/
// DialogWin- example dialog window.
/*-----*/
DialogWin::DialogWin(const PegRect &Rect) :
    PegDialog(Rect, "Edit User Information")
{
    RemoveStatus(PSF_SIZEABLE);

    // add the user name string:

    PegPoint Put;
    Put.x = mClient.wLeft + 10;
    Put.y = mClient.wBottom - 24;

    // add a checkbox with the Id value IDB_HAS_EMAIL

    Add(new PegCheckBox(Put.x, Put.y, "Have Email", IDB_HAS_EMAIL));

    // add the phone number string:
    Put.y -= 34;
    Add(new PegPrompt(Put.x, Put.y, "Phone:"));
    Add(new PegString(Put.x + 52, Put.y, mClient.Width() - 72,
        "555-1234"));

    // add the title string:
    Put.y -= 24;
    Add(new PegPrompt(Put.x, Put.y, "Title:"));
    Add(new PegString(Put.x + 52, Put.y, mClient.Width() - 72,
        "Engineer"));

    // add the name string:
    Put.y -= 24;
    Add(new PegPrompt(Put.x, Put.y, "Name:"));
    Add(new PegString(Put.x + 52, Put.y, mClient.Width() - 72,
        "Dilbert"));
}
```

The important thing to note in the above constructor function is the creation of a `PegCheckBox` object with a user defined ID value of `IDB_HAS_EMAIL`. The ID

value `IDB_HAS_EMAIL` is defined in the `dialog.hpp` header file. You should examine this header file with your editor and observe how ID values are usually defined within PEG. The ID value is contained within a private enumeration of the class `DialogWin`. This method should be followed whenever possible in order to insure compatibility with `PegWindowBuilder`. In this case, `IDB_HAS_EMAIL` is simply a value of 1. If the dialog had contained any other controls we were interested in receiving messages from, they would also be contained within this enumeration in the `DialogWindow` class definition.

A final thing to notice in the `DialogWin` class definition (contained in `dialog.hpp`) is the public prototype for the function **`Message()`**. The **`Message()`** function is defined as a virtual function by class **`PegThing`**. **`PegThing`** is the base class for **`PegWindow`**, which is the base class for **`PegDialog`**, which is the base class for **`DialogWin`**. By including a prototype for the **`Message()`** function in the **`DialogWin`** class definition, we are telling the compiler that we want to override this function. This means that whenever a control or even the PEG library calls the window's **`Message()`** function, the new version provided by our new class will be called instead of the default version defined by class **`PegThing`**.

The new **`Message()`** function is shown on the following page. There are three things to observe when examining this function. First, there are two messages we are interested in, and all other messages arrive at the **`default:`** case. When messages arrive at the default case, they are passed down to the base class `PegDialog` for processing. This is a very important thing to remember. If you don't pass the messages you are not interested in (the PEG system messages) down to the base class, your window or dialog will not work at all!

Now let's look at the first case label, which looks a little bit unusual the first time you see the syntax shown. The first case label is making use of the `SIGNAL` macro, which converts an object ID and object signal into a unique message number for this dialog window. This is how we receive messages from the checkbox, which was constructed to have the ID value `IDB_HAS_EMAIL`. The first case label is

Programming with PEG

checking for the checkbox to send a message that it has been turned on. The code which follows re-sizes the dialog window to make it taller, and then adds the email address string to the dialog window. As shown, you do not have to pass messages received from your user-defined objects down to **PegDialog**, since **PegDialog** simply ignores all user-defined messages anyway.

The second case label is similar to the first, except in this case we are watching for the checkbox to send a signal that it has been turned off. When this signal is received, the code which follows makes the dialog window smaller again, and deletes the email prompt and string via the **Destroy()** function, which is a public member of class **PegThing**.

```

/*-----*/
/*-----*/

SIGNED DialogWin::Message(const PegMessage &Mesg)
{
    PegRect NewSize;

    switch(Mesg.wType)
    {
    case SIGNAL(IDB_HAS_EMAIL, PSF_CHECK_ON):

        // make myself taller:

        NewSize = mReal;
        NewSize.wBottom += 24;
        Resize(NewSize);
        Draw();

        // add the new email string:

        mpEmailPrompt = new PegPrompt(mClient.wLeft + 10, mClient.wBottom -
24,
        "Email:");
        Add(mpEmailPrompt);
        mpEmailString = new PegString(mClient.wLeft + 62, mClient.wBottom - 24,
        mClient.Width() - 72, "info@swellsoftware.com");
        Add(mpEmailString);

        // re-draw myself to show the modifications:

        Draw();
        break;

    case SIGNAL(IDB_HAS_EMAIL, PSF_CHECK_OFF):

        // make myself shorter:

        NewSize = mReal;
        NewSize.wBottom -= 24;

```

Programming with PEG

```
        Resize(NewSize);

        // get rid of the email prompt and string:

        Destroy(mpEmailPrompt);
        Destroy(mpEmailString);

        // tell the parent to redraw, to clean up the screen:

        Parent()->Draw();
        break;

    default:
        return(PegDialog::Message(Mesg));
        break;
    }
    return 0;
}
```

Example 4- Deriving a Custom Window

While many of the PEG classes are often used directly as provided, you won't be taking advantage of the full power of PEG until you begin customizing your windows and dialogs by deriving your own classes from the base window and dialog classes provided. The best way to demonstrate this is to work through an example. If you are an experienced C++ programmer, you still may find this example useful as it illustrates a very common PEG programming operation, which is overriding the *PegThing::Draw()* member function.

In your source code distribution, you should find the directory `\peg\examples\drawwin`. This directory contains the source files required for this example. In this example, you are going to create a PEG application program that displays one very simple window. You are then going to derive a new window class that alters the default window drawing to display a custom background.

- Create a new Borland or Microsoft Win32 project, called 'drawwin'.
- Add the appropriate PEG library you built earlier to your project.
- Add the files **drawwin.cpp** and **bricks.cpp** to your project. Drawwin.cpp contains the C++ source code, and bricks.cpp contains the bitmap information that will be used later in this example.
- Make sure that you include path for your compiler contains the directory `\peg\examples\drawwin`, or your compiler may not be able to find the header file `drawwin.hpp`.
- Build the application to generate `drawwin.exe`.
- Execute `drawwin.exe`. As you can see, the program simply displays a rather boring window on the screen. This window doesn't do anything except draw a thick border, fill in its client area, and support resizing via the mouse.

- Open the source file drawwin.cpp with your editor.

You should find a function that looks like this:

```
void PegAppInitialize(PegPresentationManager *pPresentation)
{
    PegRect PutWin;
    PutWin.Set(10, 10, 200, 280);
    Presentation()->Add(new PegWindow(PutWin, FF_THICK));
    // Presentation()->Add(new DerivedWindow(PutWin, FF_THICK));
}
```

The function PegAppInitialize is called automatically by PEG during program startup. This allows you to do whatever user interface initialization you need to do. In this case, we are creating an instance of the PEG class PegWindow. The important thing to gain from this function is that we have created an instance of a PEG object, namely a PegWindow, and added it to **PegPresentationManager**, which is how we made the window appear on the screen.

Now let's put inheritance to work.

- Comment out the line that includes “**new PegWindow**”.
- Un-comment out the line that includes “**new DerivedWindow**”.

What you have done is replaced the original PegWindow with a new custom window type that is derived from PegWindow. This new custom window class is called ‘**DerivedWindow**’. This means that our new window will do everything that PegWindow does, but it can also provide enhanced or modified behavior if we provide functions in our derived class to do so.

The source code for `DerivedWindow` is also included in the file **drawwin.cpp**. The declaration of this class is found in the file **drawwin.hpp**. Once you become familiar with the PEG library you will easily be able to define your own classes such as `DerivedWindow`, but for this example we have provided all of the necessary source code.

This new class customizes the appearance of the window by overriding the **Draw()** function of the base `PegWindow` class. All *PegThing* derived objects, including `PegWindow`, contain a **Draw()** function. This function is called by *PegPresentationManager* when it is determined that an object needs to draw or re-draw itself on the screen.

Take a close look at our new **Draw()** function. You should see a line that contains:

```
PegWindow::Draw();
```

The ‘::’ symbol is called the scope resolution operator. It tells the C++ compiler which **Draw()** function we are talking about, in this case the standard `PegWindow` **Draw()** function. The first thing our custom window does is call the `PegWindow::Draw()` function!! This is very common, in that we want the default operation to occur, and then we want to draw something else ‘on top’ of what the base class does. In this case it would be more efficient to draw the entire window ourselves, since we are effectively painting the entire window client area twice each time we re-draw, but the intent is to keep this example as simple as possible.

The next line is where we provide custom operation. We are calling the `BitmapFill` function, which is a public function of the *PegScreen* class, to fill the client area of the window with the specified bitmap. The variable `mpScreen` is a pointer to the *PegScreen* object instance. This pointer is shared by all *PegThings*, because there can be only one *PegScreen* class in use at a time.

Build and execute the application program with this modified source code. You should now see the window drawn as before, except the client area of the window

Programming with PEG

is now filled in with our bitmap pattern. If you are new to C++, congratulations are in order. You have just used inheritance and overridden a public function!

Further exercises

- 1) Create a new bitmap for filling the window client area using PegImageConvert. Replace the file bricks.cpp with your new bitmap, and run the program to insure that your new bitmap is now used to fill the client area.
- 2) Change the base class for DerivedWindow to PegDecoratedWindow. Add a title bar and status bar to the window. Rerun the program to verify that your new window style is functioning correctly.
- 3) Using the window created above, add a menu bar to the window. The menu bar should contain a button which allows you to toggle between the original bitmap and your newly defined bitmap. This will take you some time to accomplish if this is the first time you have used PEG. Rerun the program and verify that you can toggle between the two bitmaps.

Additional Example Programs

Your PEG library distribution contains several additional example programs. The sub-directories under \peg\examples each contain a complete PEG application program, and a Microsoft project file for building and running the example programs.

The example application programs are organized to provide a quick overview of the stock PEG controls. The example applications will run without any modification on the PEG Win32, Win16, DOS, or integrated real-time environments. However, there are a few differences in performance, colors, etc. There is an obvious performance difference when comparing the Win32 environment with the DOS

VGA environment. This is a result of running the DOS VGA environment using standard VGA mode and using 8-bit ISA-bus memory accesses. The full video memory that is required for 640x480x16 color video is not directly accessible under DOS, which leads to the bit-plane organization describe in the PegScreen chapter.

A typical embedded application will result in much better performance than the DOS environment, since the full video memory is usually directly accessible by the CPU. In addition, many embedded platforms support 16 or 32 bit access to video memory, which greatly enhances performance.

The example applications included as of this printing which are not described above are summarized below.

PegDemo

- Directory: **pegdemo**
- PEG Classes Demonstrated: *PegWindow*, *PegDecoratedWindow*, *PegTextButton*, *PegBitmapButton*, *PegList*, *PegVerticalList*, *PegHorizontalList*, *PegComboBox*, *PegSlider*, *PegStatusBar*, *PegMenu*, *PegDialogWindow*, *PegString*, *PegEditBox*, *PegTextBox*, *PegPrograssBar*, *PegPrompt*, *PegBitmapButton*, *PegGroup*, *PegRadiobutton* and *PegCheckButton*.

This is the published PEG demonstration application using many of the PEG stock objects.

The source code for the standard demo application is in the directory \peg\examples\pegdemo. The only true C++ source code module in this directory is pegdemo.cpp. The remaining files are additional bitmaps and fonts used in the course of the demo application.

Decorated Button

- Directory: **decbtn**
- PEG Classes Demonstrated: *PegDecoratedButton*, *PegToolBar* and *PegToolBarPanel*.

The *PegDecoratedButton* class is a useful class when one needs to combine bitmap decorations and text on a single button. The class also supports “fly-over” behavior where the status of the button visibly changes when the mouse cursor is over the object. This example demonstrates the layout flexibility of the class as well as its ability to be put on *PegToolBarPanel*’s.

Dialog

- Directory: **dialog**
- PEG Classes Demonstrated: *PegDialog*, *PegCheckBox*, *PegString* and *PegPrompt*

This simple example demonstrates how to use the *PegDialog* class to create a simple entry input screen. By checking on the “Have Mail” *PegCheckBox*, the dialog changes its size and displays additional information.

Finite Bitmap Dial

- Directory: **fbdial**
- PEG Classes Demonstrated: *PegFiniteBitmapDial*

This example shows how to use custom bitmaps as backgrounds and needle anchors using the *PegFiniteBitmapDial* class.

Finite Dial

- Directory: **fdial**
- PEG Classes Demonstrated: *PegFiniteDial*

The proper use of the *PegFiniteDial* class is demonstrated in this example.

Graphics

- Directory: **graphics**
- PEG Classes Demonstrated: *PegDecoratedWindow* and drawing primitives in PEG.

This example program uses the extended graphics functions *PatternLine()*, *Polygon()*, and *Circle()*. To build this application, you must turn on the define *PEG_FULL_GRAPHICS* in the header file *\peg\include\peg.hpp*.

Gauge

- Directory: **gauge**
- PEG Classes Demonstrated: *PegBitmap*, *PegThing* and *PegWindow*.

This example program uses the off-screen drawing capability of *PegScreen* to draw an animated gauge style gadget. Off-screen drawing allows you to draw a complex bitmap in a private memory area, and then use the *PegScreen::Bitmap* function to copy the bitmap to the visible frame buffer. This allows smooth animation without any distracting flashing effects.

Image Browser

- Directory: **imgbrows**
- PEG Classes Demonstrated: *PegImageConvert*, *PegJpegConvert*, *PegBitmapConvert*, *PegPngConvert* and *PegGifConvert*.

Programming with PEG

This program demonstrates the use of the run-time image conversion classes for converting .bmp, .gif, and .jpg files to PegBitmap files during program execution.

Keyboard

- Directory: **keyboard**
- PEG Classes Demonstrated: *PegTextButton*, *PegString* and *PegDialog*.

This program demonstrates the ability to provide a user with a full QWERTY style keyboard using only four directional keys. This program also illustrates using the *PegPresentationManager::MoveFocusTree* function to force input focus to move to different objects on the screen.

Notebook

- Directory: **notebook**
- PEG Classes demonstrated: *PegNotebook*, *PegMenu* and *PegDecoratedWindow*.

This program uses the *PegNotebook* window class, and is a useful example of how to populate complex notebook pages.

2D Polygon

- Directory: **p2dpoly**
- PEG Classes Demonstrated: *Peg2DPolygon*.

The *Peg2DPolygon* class is an encapsulation of the polygon primitive. This allows the developer to define a closed polygon using a simple 0,0 upper left reference design. At run time, the polygon object is then manipulated using the class API, not matrix transforms, easing the burden on the developer.

Bitmap Light

- Directory: **pblight**
- PEG Classes Demonstrated: *PegBitmapLight*, *PegTextButton* and *PegDecoratedWindow*.

This example demonstrates how to use a *PegBitmapLight* object. The bitmaps for the example just happen to be a traffic signal, a form of a light. But, the *PegBitmapLight* object itself is really a state object that can represent any state with a unique bitmap.

Circular Dial

- Directory: **pcdial**
- PEG Classes Demonstrated: *PegCircularDial*, *PegString* and *PegDecoratedWindow*.

The *PegCircularDial* object is demonstrated using three dials with unique attributes.

Color Light

- Directory: **pclight**
- PEG Classes Demonstrated: *PegColorLight* and *PegDecoratedWindow*.

This example shows two *PegColorLight* instances, one circular and one rectangular, cycling through color states.

File Dialog

- Directory: **pfdialog**
- PEG Classes Demonstrated: *PegFileDialog*.

Programming with PEG

The PegFileDialog class provides the application designer with a conventional ‘File Open’ and ‘File Save As’ dialog box in an easy to use package.

Currently, this object is only supported on the Linux and Solaris operating systems.

Linear Bitmap Scale

- Directory: **plbscale**
- PEG Classes Demonstrated: *PegLinearBitmapScale*, *PegString* and *PegDecoratedWindow*.

This example demonstrates the flexibility of the PegLinearBitmapScale object. There are examples of using scales both horizontally and vertically, as well as defining the travel of the needle. Notice also that some of the scales have custom bitmapped needles in place of the standard line or polygon needle.

Linear Scale

- Directory: **plscale**
- PEG Classes Demonstrated: *PegLinearScale* and *PegDecoratedWindow*.

Functionally equivalent to the PegLinearBitmapScale, this example shows how to use the standard PegLinearScale.

PopupMenu

- Directory: **popmenu**
- PEG Classes Demonstrated: *PegMenu*, *PegMenuDescription* and *PegDialog*.

This program demonstrates constructing and using a pop-up menu.

Robot

- Directory: **robot**
- PEG Classes Demonstrated: *PegAnimationWindow*, *PegStatusBar*, *PegButton* and *PegPrompt*.

This example program creates several custom derived gadget classes for displaying a custom button style, prompt style, and 2-row status bar.

Spreadsheet

- Directory: **spread**
- PEG Classes Demonstrated: *PegSpreadSheet*, *PegDecoratedWindow*, *PegMenu* and *PegDecoratedWindow*.

This example demonstrates how to set up and manipulate a *PegSpreadSheet*.

Station

- Directory: **station**
- PEG Classes Demonstrated: *PegStripChart*, *PegColorLight*, *PegLinearBitmapScale* and *PegCircularBitmapDial*.

This example brings several of the advanced PEG objects together to demonstrate a simulated nurse's monitoring station.

Strip Chart

- Directory: **stchart**
- PEG Classes Demonstrated: *PegStripChart* and *PegDecoratedWindow*.

Programming with PEG

The `PegStripChart` object is used to display dynamic data in a scrollable, line oriented format.

Table

- Directory: **table**
- PEG Classes Demonstrated: *PegTable* and *PegDecoratedWindow*.

This example program uses the `PegTable` window class, and is a useful example of populating table cells.

Tool Bar

- Directory: **toolbar**
- PEG Classes Demonstrated: *PegToolBar*, *PegToolBarPanel*, *PegBitmapButton*, *PegSlider*, *PegComboBox*, *PegDecoratedWindow* and *PegString*.

This example utilizes the `PegToolBarPanel` and `PegToolBar` classes to show how easy it is to add complete tool bar functionality to a `PegDecoratedWindow` object.

Tree View

- Directory: **treeview**
- PEG Classes Demonstrated: *PegTreeView* and *PegDecoratedWindow*.

This example demonstrates how to populate a `PegTreeView` object with static data. The contents of the tree is the complete PEG class hierarchy.

Unicode

- Directory: **unicode**

- PEG Classes Demonstrated: See the *Peg Demo* listing.

This example is the same as the Peg Demo described above, but it has the ability to display all of the text in either English or Japanese.

Vector Font

- Directory: **vecfont**
- PEG Classes Demonstrated: *PegDialog* and *PegScreen*.

This program uses the PEG vector font functions to allow the user to enter text in a wide range of point sizes.

Final Notes

In this chapter we have illustrated some of the most common PEG programming tasks. As you will see when you begin using WindowBuilder, many of these tasks can be accomplished by simply setting the appropriate configuration information from within WindowBuilder. However, you now know what is happening ‘under the hood’, and you will be able to extend the functionality provided by WindowBuilder whenever required.

Chapter 9: PEG Multitasking

PEG supports three different execution models which can be tailored to your requirements. These execution models are called the ***Standalone*** model, the ***Multithread*** model, and the ***PRESS*** (Peg Remote Screen Server) model. This chapter presents an overview of what these models are, how to view the overall system in each model, and how to use PEG effectively whichever model you choose.

In the first execution model, PEG runs as a single low-priority task in a multitasking system or as a simple super-loop in a standalone system. This is called the ***Standalone*** model, since the entire GUI interface is presented by one task, and all other tasks that desire to affect the graphical presentation must do so by sending messages to the PEG task. Standalone does not mean that you cannot run other tasks in a multi-tasking system, because you can. Our use of the term standalone here implies only that PEG runs as if there are no other tasks and is not aware of the operation of other tasks if such tasks exist. All graphical objects are created from within this single task, and PEG message processing likewise occurs from within the single GUI task.

The ***Standalone*** model is easier for new users to than the ***Multithread*** model. If you are not using an RTOS, or if you are porting PEG to a custom RTOS, you will should at least initially use the ***Standalone*** model.

The second model is the ***Multithread*** model. In the ***Multithread*** model, any number of tasks can **directly** create, display, and manipulate graphical objects. Under this model any task can directly draw to the screen. Message processing for each graphical object usually occurs in the thread of the task that displays the object. Each GUI task can be thought of as an entirely separate application program, running independently of any other GUI tasks.

PEG Multitasking

If you are using an operating system that PEG has been integrated with, you can use either model by simply turning on or off the PEG_MULTITHREAD definition in the file \peg\include\peg.hpp. If you are using a new RTOS or a custom RTOS, you will need to integrate the PEG messaging services with your RTOS messaging services in order to support the *Multithread* model.

The final execution model is the PRESS model. Under this model each process running on the target may execute in a unique, virtual address space. Processes running on the target may even be executed on physically remote processors. The PRESS model details are very specific to the target operating systems under which the PRESS model has been designed. Further details on building and running under the PRESS execution model are available in the document titled **PEG Remote Screen Server Technical Overview**.

Why Different Models?

PEG supports multiple execution models in order to meet the needs of the broadest possible array of applications, while preventing unnecessary overhead in applications that do not require the added complexities of the *Multithread* or *PRESS* execution models.

Many simpler applications work just fine using the *Standalone* model. Often there is very little interaction between the GUI interface and the real-time control oriented tasks in an embedded system. When interaction is required, it is accomplished either through global variables or by sending messages between GUI windows and other system tasks. There are several advantages to this model of execution:

- Absolute minimum overhead
- PEG has no impact on real-time performance.
- Easily integrated with new or custom in-house operating systems.

Under the *Standalone* model, PEG can be ported to any operating system by simply defining one OS message queue for use by PEG, defining a **PegIdleFunction()** that retrieves messages from this queue, and a similar function for use by external tasks for sending messages to this queue.

Conversely, under the *Multithread* model, any number of tasks can create and directly display windows or any type of graphical objects at any time. These tasks can also directly manipulate and update the graphical objects, without interacting with PegTask. This model can simplify the structure and layout of tasks which must interact with the GUI presentation. The only drawback to this model is that a small amount of overhead is added by PEG to insure that everything works correctly, and additional message queues are required to insure that all message processing occurs within the thread of the task that owns a particular window.

The display screen must be treated like any other single-user I/O device, such as a serial port or printer port. It is not acceptable to have one task sending data to a printer, and allow a second task to preempt the first task and begin using the same printer. The final printout would have information from each task printed on the same page, or may even cause a totally unreadable output.

Likewise, the display screen can only be used by one task at a time. The reasons for this are actually more complex than the printer example, but it is sufficient to state that while any number of tasks can directly access the screen, only one task can be actively drawing to the screen at a time. Of course in a multitasking system this exclusion is transparent to the end user, and the appearance is that many operations are happening simultaneously.

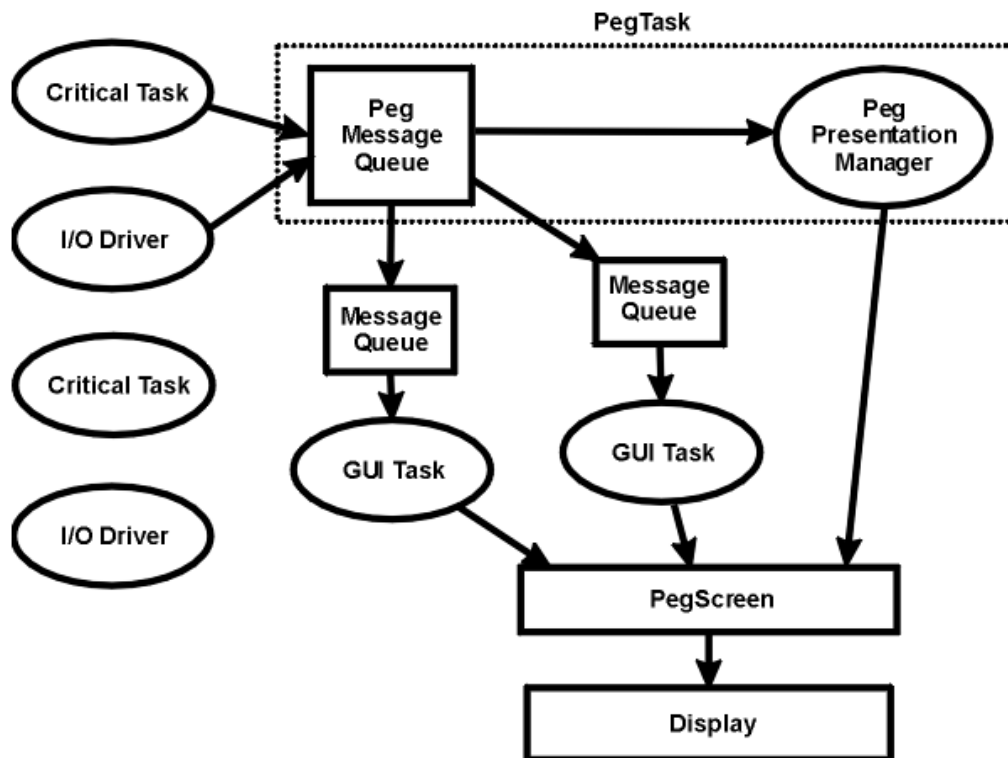
When running under the ***Multithread*** model, PEG internally protects the display screen with a semaphore to insure that only one task can draw to the screen at a time. This is transparent to the application level software, but is mentioned as the user should be aware of this slight added overhead.

In addition, when running under the ***Multithread*** model, PEG creates structures for associating a specific task with each top-level window (i.e. each window added to PegPresentationManager). This also adds a small amount of run-time overhead and requires the creation of additional message queues.

All of the requirements for supporting the ***Multithread*** model are handled internally by PEG for operating systems supported by PEG. The largest area of integration involves re-working PegMessageQueue to work directly based on the underlying RTOS messaging services. This allows your GUI tasks to perform priority-based task scheduling as messages arrive for each task.

PEG Multitasking

The following diagram depicts PEG running under the *Multithread* model:



In the diagram above, we depict a complex system having several critical real-time tasks, I/O drivers, and GUI related tasks. The arrows indicate the general direction of message flow. Note that the significant difference in this model as compared with the graphics server model shown in chapter 3 is that we have several tasks calling **PegScreen** functions, i.e. several tasks are directly manipulating the graphical presentation. This is **not** allowed when running under the *Standalone* model.

Each task that directly uses the PEG API receives a private message queue. This message queue is generally constructed using the messaging services of the underlying RTOS. These private message queues are constructed at run time as GUI task are started, and likewise are deleted if and when the associated GUI task is terminated. This all happens transparently to the application level software.

As shown above, even time critical tasks can participate in the graphical presentation via messaging. However, time critical tasks do not directly manipulate the display or execute windows. Time critical task must limit themselves to sending messages to any of the GUI tasks in order to guarantee that the response time of the critical task is not affected by the operation of the screen.

In the message flow in the diagram above, it may at first appear that the central PegMessageQueue represents a bottleneck, limiting the responsiveness of higher-priority tasks. This is not truly the case, because the PegMessageQueue::Push() function routes messages directly to the private message queue specific to each task whenever possible. Stated more succinctly, messages directed to a particular window are immediately placed in the queue for the task that owns the window. They do not pass through the default PEG message queue.

Also notice that even under the ***Multithread*** model the default PegTask is required. Although there may be no windows executing under this thread, the PegTask thread handles various operations such as mouse movement, background refresh, and message routing for un-directed messages.

In the following sections we will review the operation of the ***Standalone*** model, and use examples to demonstrate the difference between operation under the ***Standalone*** model and the ***Multithread*** model.

Graphics Server Model

The first thing to stress is that the *Standalone* model is capable of doing nearly everything that can be accomplished with the *Multithread* model. The difference is that when you are running with the Standalone model, **all** GUI message processing and drawing operations occur from within PegTask. Tasks other than PegTask are not allowed to directly update the screen or modify the presentation tree.

This does not imply that other tasks are prevented from creating and displaying windows, because they can. However, tasks other than PegTask cannot directly add windows to PegPresentationManager, nor can they directly update any area of the screen. All GUI operations required from tasks other than PegTask are accomplished by sending messages from these outside tasks to PegTask.

When running in the Standalone model, the initial window or windows displayed are created in the function **PegAppInitialize()**. Almost all subsequent window display, update, etc.. is usually done in response to user interaction with these initial windows.

In this model, windows or controls can interact with other system tasks by sending and receiving messages, or through the use of global variables. Tasks other than PegTask cannot directly manipulate the presentation in any way.

Sending messages to PegTask is accomplished by sending messages to the OS-defined message queue depicted back in chapter 4. This is the queue that your **PegIdleFunction()** will retrieve messages from. *Messages should not be directly pushed into PegMessageQueue by tasks other than PegTask, since this could cause queue corruption.*

An example of this should help your understanding of how to organize your application under this model.

Coolant Window Example 1

For this example, suppose your application at some point creates a window for displaying and controlling the temperature of a machine coolant. The coolant temperature window is created in response to the user selecting a menu item on one of the initial windows created in **PegAppInitialize()**. The coolant temperature window displays the current temperature of the coolant, and provides a PegSlider control to allow the user to adjust the temperature setpoint.

The temperature of the coolant is continuously monitored and controlled by a separate real-time task. This second task may read A/D converters, perform averaging, and perform PID calculations to adjust the coolant temperature.

In order for the window to display the coolant temperature, the coolant monitor task stores the current temperature in a global variable. The display window creates a PegTimer in order to periodically compare this global value with the last value displayed, and re-displays the value when a change is detected.

Likewise, the display window stores the current slider value, which is the desired temperature setpoint, in a second global variable. The coolant monitor task continuously reads this global setpoint value in order to properly perform PID calculations.

Coolant Window Example 2

The above example illustrates what is probably the simplest method of interfacing the GUI presentation with other system tasks, via the use of global variables. This method may be the best method for your system, however there are certain drawbacks. First, the use of global variables is often discouraged. Global variables can be changed by any task or piece of code. In a large system, there can be hundreds or even thousands of data values for display, in which case there would be hundreds or thousands of global variables.

PEG Multitasking

Finally, the use of global variables requires that both the display window and the coolant monitor task periodically poll the global values in order to insure that the screen is up to date and the setpoint is correct. There is no ***flow control***, meaning that the rate at which the monitor task updates the global variables is not tied in any way to how often they are updated on the display. The monitor task may spend needless CPU cycles updating the global variable 10 times per second, when the display window is only refreshing the displayed value once every five seconds. In fact, without additional global variables, the monitor task has no way of knowing whether or not the display window is even active. Updating the global temperature variable may be totally unnecessary.

The following offers some improvement on the above example:

To update the temperature displayed, the display window will generate an initial temperature request message when the window is first displayed. This message will be sent to the monitor task asking for the current temperature of the coolant. In creating this message, the window will create a message in the format required by the operating system. One field of the message will indicate the type of message, in this case a TEMP_REQUEST message. Another field of the message will contain a pointer to the coolant display window, (i.e. *this*). This pointer is used by the coolant monitor task to reply to the message.

When the user modifies the temperature setpoint, the display window similarly constructs a message and sends it to the coolant monitor task. This message does not require a response, but will have a different message type indicator to enable the monitor task to understand that this message is a command to change the temperature setpoint for the coolant.

Once the message is created, the coolant window uses the underlying OS message services to send this message directly to the coolant monitor task. The coolant monitor task must periodically check for receipt of messages in whatever way is

supported by the RTOS. When a message is received, the coolant monitor checks the message type, and acts accordingly. If the message is a request for the current temperature, the coolant monitor task constructs a response message and sends the response to PegTask. The response message must contain the window pointer received in the request message, in order for PegTask to direct the translated PegMessage response to the correct window.

The coolant window receives the response just like any other PegMessage. On receipt of the response message, the coolant window updates the displayed temperature, and immediately generates a new request.

This example uses flow control, and does not require polling by the display window. An initial request is generated, and each time a response is received the display window displays the new value and generates a new request. This allows the value to be updated as quickly as possible, while at the same time the monitor task does not calculate and save the current temperature value unless it is requested.

Window Display

Even under the *Standalone* model, it is possible to display, remove, and destroy windows from tasks other than PegTask through the use of messaging. While other system tasks cannot directly add objects to PegPresentationManager, they can create objects and send messages to PegPresentationManager to display, remove, or destroy those objects.

While windows created and displayed in this fashion are visible, they are executing from within the PegTask thread. The task which created the window should not directly manipulate the window or any of its child controls.

FLASH task example

Again an example is useful. A common embedded system may have a FLASH memory update task. This task can erase, read, and write data to FLASH memory. If you have ever worked with FLASH memory, you know that erase and write operations can sometimes require several seconds to complete. Let's say that you would like the FLASH memory task to display a "Flash Update In Progress" message whenever a lengthy operation is occurring. Under the Graphics Server model, the FLASH task cannot simply create the message window and add it to `PegPresentationManager`, nor can it directly remove the window when the operation is complete. Using messaging, however, the FLASH task can accomplish the same effect, as shown below:

```

void FlashTask(void)
{
    // create the message window:
    PegMessageWindow *mwp = new PegMessageWindow("FlashTask", "Updating Flash
    Memory");

    // tell PegPresentationManager to display the window:

    PegMessage NewMessage(mwp->Presentation(), PM_ADD);
    NewMessage.pSource = mwp;           // pass pointer to window to be added
    SendMessageToPeg(NewMessage); // user defined, put message in OsPegQueue

    // do the flash update operation.....

    // If we want to update the status window during the operation,
    // we will have to send further messages (not shown)

    // now the flash operation is complete.....

    // tell PegPresentationManager to destroy the window:

    NewMessage.wType = PM_DESTROY;
    NewMessage.pTarget = mwp->Presentation();
    NewMessage.pSource = mwp;
    SendMessageToPeg(NewMessage);           // user defined, put message in
    OsPegQueue
}

```

In the above example, a message is constructed to tell PEG to display the message window, and after flash processing, a separate message is sent to PEG to remove the window. The function ***SendMessageToPeg()*** is user defined. This function does whatever OS-specific procedure is required to place a message in the message queue that is being monitored by ***PegIdleFunction()***. This transfer of messages between the OS-defined message queue and PegMessageQueue is required under the **Standalone** model to prevent corruption of PegMessageQueue data structures. Messages are only pushed into PegMessageQueue from within the PegTask thread, either in ***PegIdleFunction()*** or as part of PEG message processing.

Modal Execution

A PegWindow or PegWindow derived object (such as PegDialog or PegMessageWindow) may be executed modally by calling the PegWindow member function **Execute()**. In the Standalone model, only PegTask can execute a window modally, and modal execution is global in scope. A window executing modally can display and modally execute another modal window. For example, a modal dialog window can create and display a modal message window, as in “Are you sure you want to close this dialog?”.

No task other than PegTask can perform modal window execution (i.e. only the PegTask thread can call the PegWindow::Execute() function) under the Standalone model.

MULTITHREAD Model

PEG allows multiple tasks to create, display, and interact directly with multiple GUI windows. This mode of operation is enabled by turning on the definition PEG_MULTITHREAD in the file `\peg\include\peg.hpp`.

In all cases, PegTask is still required when using the *Multithread* execution model. PegTask handles normal GUI operations such as moving input focus, changing the window display order, etc. However, you are not required to create and display any graphical objects from within PegTask. When running under the *Multithread* model, it is possible to write a **PegAppInitialize()** function that simply creates and starts the tasks which will display windows, rather than actually displaying any windows from within PegTask.

There are several ways to execute a window under the *Multithread* model. This allows several different types of tasks to participate in the graphical interface, without forcing all GUI tasks to follow the same format. The following sections describe the methods for directly creating and displaying graphical objects from tasks other than PegTask.

Window Execution Under PegTask Thread

When any window is added to PegPresentationManager by calling the PegPresentationManager::Add(Window *MyWin) function, that window is executed under PegTask, regardless of which task actually created and displayed the window. All message processing, including input message processing, is performed from within the PegTask execution thread. This form of window creation is useful for tasks that need to display information, but are not able to become fully involved in GUI interactions and message processing.

PEG Multitasking

A task that presents a window in this way is running outside of the PEG environment. Since the window is executed from within the PegTask thread, the task that presented the window is free to do any non-GUI related processing as required, without worrying about responding to user input events.

Counter Task Example 1

The following is an example of a task which creates and displays a window under the *Multithread* model, but allows PegTask to handle all subsequent message processing. This example is taken from the standard PEG multitasking demo:

```
/*-----*/
void CounterTask(void)
{
    PegRect Rect;
    Rect.Set(400, 360, 600, 478);

    TaskWindow *pt = new TaskWindow(Rect, "Counter Task", 0);

    pt->Presentation()->Add(pt);    // present the window

    while(1)
    {
        Sleep(pt->SliderVal());
        pt->IncCount();
    }
}
```

In this example, the task CounterTask creates a new window called TaskWindow. The window is presented by calling PegPresentationManager::Add, in which case the window executes from within the PegTask thread.

The task then begins normal execution. In this example the task simply sleeps, but the task could perform any other type of processing just as well. Periodically, the task wakes up and **directly** updates a field displayed by TaskWindow by calling a

member function of the TaskWindow class named ‘IncCount()’. The source code for the IncCount() function is shown below:

```
/*-----*/
void TaskWindow::IncCount(void)
{
    char cTemp[34];
    mICount++;           // increment task count variable
    ultoa(mICount, cTemp, 10);
    CountPrompt->DataSet(cTemp);
    CountPrompt->Draw();  // draws to the screen from secondary thread!
}
```

Another result of this mode of execution is that a task can create and display any number of windows, and directly modify any window at any time. Since all message processing for these windows is performed from within PegTask, secondary tasks are free to create additional windows or do any type of non-GUI related processing.

The above example is much different than the Standalone model, which does not allow direct modification of graphical objects from tasks outside of PegTask.

FLASH task example revisited

At this point it is useful to revisit the “FLASH TASK” example used when describing the *Standalone* model, to see how the same operation is simpler to do when running under the MultiThread model.

PEG Multitasking

```
void FlashTask(void)
{
    // create the message window:
    PegMessageWindow *mwp = new PegMessageWindow("FlashTask", "Updating Flash
    Memory");

    // display the window:
    mwp->Presentation()->Add(mwp);           // directly display the window!

    // do the flash update operation.....

    // destroy the window:
    mwp->Destroy(mwp);                       // directly destroy the window!
}
```

Compare the above code fragment with the same code written for the Standalone model. As shown above, it is much easier to display information from within tasks external to PegTask when running in the MULTITHREAD model. A task can create windows, draw to the screen, or do any other operation directly under this model, while in the Standalone model secondary tasks can only interact with the GUI presentation via messages.

Window Execution Under Secondary Thread

A window can be executed from within a new task rather than from within the PegTask thread. In this execution mode, the top level window should be thought of as an entirely new application, possibly running alongside any number of other applications, all within a single PEG presentation.

When a window is executed from a secondary task (i.e. a task other than PegTask), **all** message processing for that window occurs within the thread of the calling task. This is also true for any subsequent windows created by the first or top-level window.

This mode of execution is invoked by calling the PegWindow member function **Execute()** from a secondary task. In this case, the window should not be explicitly added to PegPresentationManager by the secondary task, this will happen automatically when the **Execute()** function is called.

Modal Task Window

The following is an example of creating and executing a window from within a secondary task:

PEG Multitasking

```
void ModalTaskWindow(void)
{
    while(1)
    {
        // do any type of processing here....

        // now create and execute a window:
        PegRect Rect;
        Rect.Set(400, 360, 600, 478);
        TaskWindow *pt = new TaskWindow(Rect, "Counter Task", 0);
        Pt->Execute();          // doesn't return until window closes!

        // do any type of processing here.....
    }
}
```

In the above example, the task directly executes the window. All message processing for the window occurs within the thread of ModalTaskWindow. Like any modal window, the actual PegWindow object is destroyed before control returns to the point of calling the **Execute()** function. The return value indicates the ID of the button or other object used to close the modal window.

The operation of the **Execute()** function must be understood before executing a window in this way. ***Execute() begins modal window execution, and does not return until the window is closed.*** This does not mean that your task is somehow 'locked-up' or disabled, rather the task resumes execution at the **PegWindow::Message()** function as messages are received by the window. This is an important point, and requires that tasks which create and modally display windows are not also required to perform real-time operations.

One finer point should be noted about modal window execution from outside of PegTask. The meaning of modal execution is different in this model than you might expect. While **Execute()** is globally modal when running under the Standalone model, **Execute()** is modal only to the calling thread when running under the

Multithread model. This is a natural extension of the **Multithread** model, and further gives the end user the appearance that each task is a separate application, rather than just separate windows within a single application. A window executing modally is free to create any number of subwindows, as long as those subwindows are children of the first modal window. In other words, ‘modal’ in this case does not mean “click on this window and no where else”, it only means that the modal window cannot create additional top-level windows (i.e. windows added to PegPresentationManager), and control does not return to the point of calling **Execute()** until the top level window is closed..

NOTE: The PegWindow::Execute() function does not return until the window is closed. A task that is involved with real-time control operations should never call the PegWindow::Execute function.

Finally, there may be instances where you want to have a globally modal dialog window. In this case you do not want the user to be able to select any other window on the presentation, including Windows that may have been presented by other tasks. For this reason the PegWindow member function GlobalModalExecute() is provided when running under the Multithread model.

Execution Model Recap

To review, we will summarize in two sentences the difference between the *Standalone* and the *Multithread* models:

Any task can create windows, draw to the screen, or do any other GUI operation directly under the **Multithread** model. Under the **Standalone** model secondary tasks can only interact with the GUI presentation via messages.

Multithread Execution on the Win32 Development Platform

The Multithread capabilities of PEG are available on the Win32 Development Platform for those developers who wish to either simulate their target environment or are planning to deploy their application on a version of the Win32 platform. All of the functionality and threading paradigms discussed in the previous section are applicable to this platform. For instance, every thread of execution has a private PEG message queue that contains PEG messages pertinent to only this thread and vital PEG resources are protected by Win32 style synchronization objects to allow for re-entrancy and thread safe execution.

We'll begin by first looking at a simple example of this execution model, we'll then discuss the programming issues involved with making this model work on the Win32 platform.

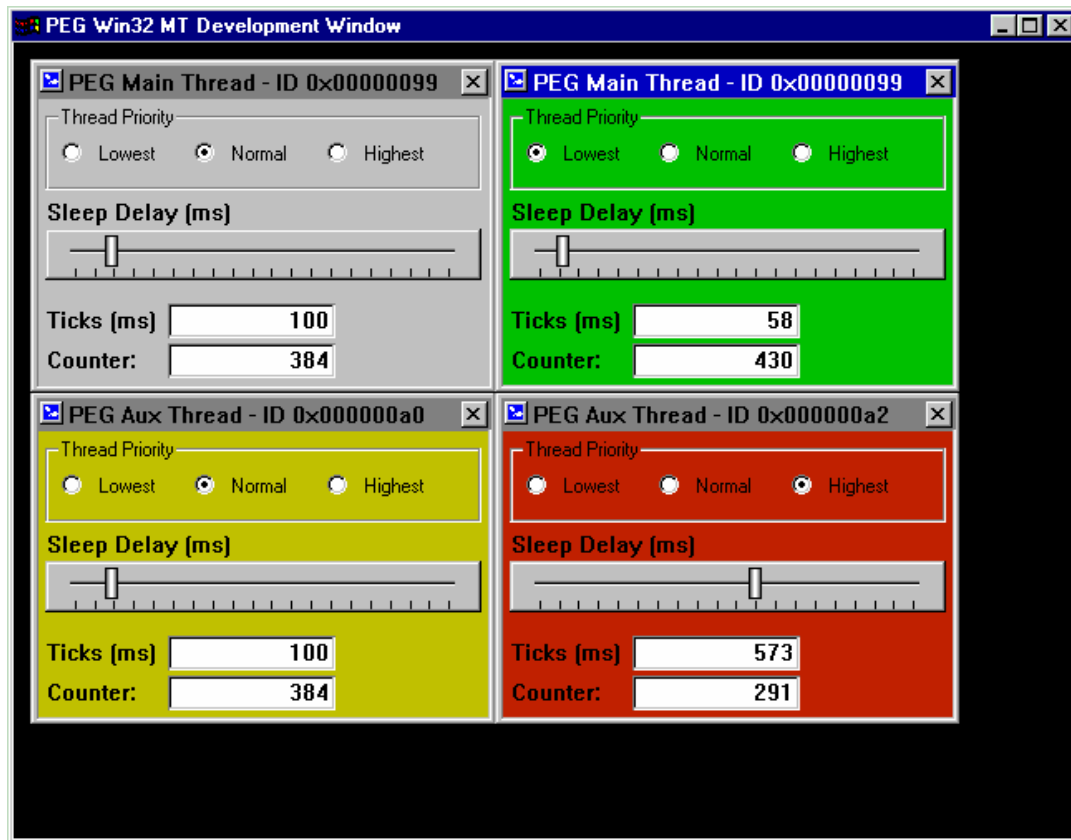
The screen shot below contains four instances of a simple PegDecoratedWindow derived object, MultiThreadWindow. This window has PegRadioButtons for setting the priority of the thread on which it is running, a PegSlider for setting a sleep delay and PegPrompts for displaying the current sleep delay in milliseconds and current counter value. The counter value is actually being updated from an ancillary thread, one thread for each window. This thread is also retrieving its sleep time from the value in the PegPrompt object that it uses as a delay between updates to the counter value.

You can find the source code for this example in your PEG distribution in the \peg\examples\win32mt directory. You may find it helpful to browse some of this code while you are reading through this explanation.

You will notice that the PegTitle object on each window is displaying a thread ID number of the thread on which it is executing. Two of the windows are executing

PEG Multitasking

within the context of the main PegTask, while the other two are operating within the context of ancilliary threads. This demonstrates that any thread may add top level windows to the application at any time and either directly interact with the PEG object, and/or allow other threads to operate on the object.



Two of these MultiThreadWindow instances were added to the PegPresentationManager in the PegAppInitialize function, another was added from an ancilliary thread, and the fourth was added by calling the Execute method of the

PEG Multitasking

window from its own thread. We'll take a look at the code behind this to clarify some of these ideas.

First, two instances of the `MultiThreadWindow` are added to the `PegPresentationManager` in the `PegAppInitialize` function as outlined in the following piece of code:


```

void PegAppInitialize(PegPresentationManager* pPresent)
{
    // Add a window right to the presentation manager that will
    // run in the context of the main PEG thread
    MultiThreadWindow* pWin = new MultiThreadWindow(10, 10, 0, TRUE);
    pWin->SetColor(PCI_NORMAL, LIGHTGRAY);
    pPresent->Add(pWin);

    // Even though we're not using the return value of the thread
    // ID given in the 6th parm to CreateThread, we still have to
    // include a pointer to a DWORD for compatibility with
    // Win95/98/Me since these OS's explicitly need this parm. If
    // this is not included, the call to CreateThread would hang.
    // In a production system, you would want to check this value
    // as well as check the value of the CreateThread return.
    DWORD dwThreadParm;

    // Start up a counter thread for it using the Win32 API function call.
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)CountThreadProc,
        (LPVOID)pWin, 0, &dwThreadParm);

    // Start up a secondary thread that will add it's own
    // window using the Win32 API function call.
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)OtherThreadProc,
        NULL, 0, &dwThreadParm);

    pWin = new MultiThreadWindow(290, 10, 0, TRUE);
    pWin->SetColor(PCI_NORMAL, GREEN);
    pPresent->Add(pWin);
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)CountThreadProc,
        (LPVOID)pWin, 0, NULL);

    // Start up another window that will execute in it's own thread
    // and start up a counter thread for itself.
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)AnotherThreadProc,
        (LPVOID)pWin, 0, &dwThreadParm);
}

```

PEG Multitasking

You will notice that the first two windows are added directly to the `PegPresentationManager`. That implies these windows will execute in the context of `PegTask`. We then start up an ancilliary thread for each window, whose callback function `CountThreadProc`, will then interact with the window.

Next, we create another thread whose callback function is `OtherThreadProc`. And, finally, yet another thread whose callback function is `AnotherThreadProc`. We'll look at these callback functions in a moment.

To clarify this code a little further, the `CreateThread` function is a Win32 API service that allows multiple paths of execution in a single Win32 process. An in-depth discussion of this function is beyond the scope of the manual. If you are new to Win32 threading techniques, consult the documentation that accompanied your compiler, the Microsoft Knowledge Base or any number of third party books from authoritative publishers such as the Waite Group Press or O'Reilly and Associates.

Let's take a look at the first callback function, `CountThreadProc`. Here is the code:

```

DWORD WINAPI CountThreadProc(LPDWORD lpData)
{
    MultiThreadWindow* pWin = (MultiThreadWindow*)lpData;
    // In order to get the REAL handle of the current thread, (not the
    // psuedo handle that GetCurrentThread returns, we have to duplicate
    // the handle. This gives us the thread handle that we can pass to
    // the window that we're controlling so that the window will terminate
    // this thread when the window is deleted.
    HANDLE tCurHandle = GetCurrentThread();
    HANDLE tDupHandle;
    DuplicateHandle(GetCurrentProcess(), tCurHandle, GetCurrentProcess(),
        &tDupHandle, 0, FALSE, DUPLICATE_SAME_ACCESS);
    pWin->SetChildThreadHandle(tDupHandle);
    DWORD dwCount = 0;
    DWORD dwSleep = 0;

    while(1)
    {
        pWin->SetCount(++dwCount);
        dwSleep = pWin->GetSleep();
        Sleep(dwSleep);
    }

    return(0);
}

```

You'll first notice that the `lpData` parameter is cast to a `MultiThreadWindow` pointer. We passed this over in the call to `CreateThread`. For reasons that we'll discuss later, we have to do a bit of Win32 API magic to get the true resource handle of the current thread. It is important that the `MultiThreadWindow` instance have a handle to the thread that is updating the objects on the window, because once the window is deleted, the pointer that the thread has to the window will no longer be valid. Obviously, we have to consider this situation.

The small loop that simply increments a counter and updates the data displayed in the window is the real work that the thread is doing. Consider that this is only an example of what an ancilliary thread is able to do. In a real world application, this

PEG Multitasking

thread may, for example, be responsible for pulling data from a serial buffer that is connected to some piece of hardware and passing that data into a PEG object. It would not be prudent for the PEG object to do this type of processing for reasons already discussed in previous sections.

Before we move on, let's take a look at what the `MultiThreadWindow::SetCount` method does. Here is the code:

```
DWORD MultiThreadWindow::SetCount(DWORD dwNewCount)
{
    EnterCriticalSection(&mtCountCS);

    DWORD dwOldCount = mdwCount;
    mdwCount = dwNewCount;

    PEGCHAR cBuff[20];
    ltoa(mdwCount, cBuff, 10);
    mpCount->DataSet(cBuff);
    mpCount->Draw();

    LeaveCriticalSection(&mtCountCS);

    return(dwOldCount);
}
```

We have abstracted the setting of the `MultiThreadWindow::mpCount` member into this method. This allows for thread safe updating of the value displayed by the object. The entire method is wrapped with a synchronization object (the Win32 API `CRITICAL_SECTION` object, `mtCountCS`). This ensures that any other thread of execution will not corrupt the `mdwCount` member variable. The important point here is that we have not burdened the ancillary thread with this type of issue. It has no idea about how the data is being updated. We have made the owner of the data responsible for keeping the data safe.

Next, the thread calls the `MultiThreadWindow::GetSleep` method. Here is the code for that method:

```
DWORD MultiThreadWindow::GetSleep()
{
    EnterCriticalSection(&mtSleepCS);

    DWORD dwCurSleep = mdwSleep;

    LeaveCriticalSection(&mtSleepCS);

    return(dwCurSleep);
}
```

Again, we have protected the data with a `CRITICAL_SECTION` object. This is important here because the `MultiThreadWindow` can update the value of `mdwSleep` when it receives a `PegMessage` from the `PegSlider` object informing the window that the value of the slider has changed. Here is a look at part of the `MultiThreadWindow::Message` method:

```
SIGNED MultiThreadWindow::Message(const PegMessage& Mesg)
{
    .
    .
    .
    switch(Mesg.wType)
    {
        case SIGNAL(IDC_SLEEP, PSF_SLIDER_CHANGE):
            EnterCriticalSection(&mtSleepCS);
            mdwSleep = Mesg.lData;
            PEGCHAR uBuff[20];
            ltoa(mdwSleep, uBuff, 10);
            mpTick->DataSet(uBuff);
            mpTick->Draw();
            LeaveCriticalSection(&mtSleepCS);
            break;
    }
    .
    .
    .
}
```

Here, too, we operate on the `mdwSleep` member variable, so we wrap the usage in the same `CRITICAL_SECTION` object as the previous method. The concept to note here is that the `PegMessage` pump is executing within the context of the thread in which the instance of the `MultiThreadWindow` is executing; and, the ancilliary thread that is reliant on this data obviously has a separate path of execution so the data must therefore be protected.

Now that we have discussed how ancilliary threads may operate on PEG objects, we'll take a quick look at the remaining two thread call back functions to briefly discuss optional ways to add top level windows to the `PegPresentationManager`.

First, the contents of the callback `OtherThreadProc`:

```

DWORD WINAPI OtherThreadProc(LPDWORD lpData)
{
    MultiThreadWindow* pWin = new MultiThreadWindow(10, 210, 0, TRUE);
    pWin->SetColor(PCI_NORMAL, BROWN);
    // Duplicate the thread handle to get the REAL handle to pass
    // to the window that we're controlling
    HANDLE tCurHandle = GetCurrentThread();
    HANDLE tDupHandle;
    DuplicateHandle(GetCurrentProcess(), tCurHandle, GetCurrentProcess(),
        &tDupHandle, 0, FALSE, DUPLICATE_SAME_ACCESS);
    pWin->SetChildThreadHandle(tDupHandle);
    pWin->Presentation()->Add(pWin);

    DWORD dwCount = 0;
    DWORD dwSleep = 0;
    while(1)
    {
        pWin->SetCount(++dwCount);
        dwSleep = pWin->GetSleep();
        Sleep(dwSleep);
    }

    return(0);
}

```

Much of this is familiar from the `CountThreadProc` discussion, with the exception of how the `MultiThreadWindow` instance is created. You'll remember that in the `CountThreadProc` code, we were passed in an instance of an existing `MultiThreadWindow`, while here we create one for ourselves. This demonstrates that any thread can create and display objects within the PEG Multithread framework.

The last callback, is detailed here:

PEG Multitasking

```
DWORD WINAPI AnotherThreadProc(LPDWORD lpData)
{
    MultiThreadWindow* pWin = new MultiThreadWindow(290, 210, 0, TRUE);
    pWin->SetColor(PCI_NORMAL, RED);

    // Start up a counter thread for it.
    CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)CountThreadProc,
        (LPVOID)pWin, 0, NULL);

    // Execute the window, don't just add it to the presentation
    // manager. When this returns, the window will have been closed,
    // and the counter thread will have been canceled,
    // so we can safely return.
    pWin->Execute();

    return(0);
}
```

This function is an example of yet another way to work with top level PEG windows. Here, we create an instance of a `MultiThreadWindow`, then create another thread that will run the `CountThreadProc` callback, to which we pass a pointer to our new window. We then call the window's `Execute` method, which effectively gives up control in this thread to the PEG object. When `Execute` returns, we know that the window has been removed from the `PegPresentationManager` and has been destroyed, so, we may safely return at that point.

The last programming concept that we will cover here is the relationship between a PEG object, in this case the `MultiThreadWindow`, and ancilliary threads. We touched on this issue earlier while we were looking at the code for the `CountThreadProc`. Since this thread has a pointer to an object that is executing within the context of another thread, the ancilliary thread is taking for granted that the pointer is valid when it works on the object. This is obviously a problem.

There are several techniques available to the Win32 programmer to circumvent this issue. We have chosen here to let the MultiThreadWindow have the power to terminate the thread when it sees fit. This may or may not be the best solution, but for the purposes of this example, it is sufficient to allow for reliable operation.

For the MultiThreadWindow to have the power to terminate the ancilliary thread, it must first know how to address the thread. This is why we go through the trouble of getting the “real” thread handle from the operating system in the first few lines of code in the CountThreadProc. By calling the Win32 API function, DuplicateHandle, Windows returns to us the true handle of the thread as opposed to the psuedo handle we receive in the GetCurrentThread function. (Again, consult your Win32 API documentation regarding this phenomenon.)

We then call MultiThreadWindow::SetChildThreadHandle with the thread handle we are returned. This gives the instance of the window knowledge of the ancillary thread that is updating its PEG child objects.

Here’s the code for the MultiThreadWindow’s destructor:

PEG Multitasking

```
MultiThreadWindow::~MultiThreadWindow()
{
    if(mhChildThread && mbKillChildOnExit)
    {
        EnterCriticalSection(&mtCountCS);
        // Some Win32 documentation states that TerminateThread should
        // only be used for misbehaving threads that don't wish to
        // exit properly, and they warn of memory leakage (since the
        // thread stack is not deallocated, so they say), but there
        // really is no other handy way to terminate a thread without
        // using some overly complicated synchronization techniques.
        // For this example, terminating the thread is probably okay, but
        // you should perhaps explore other possibilities in a
        // production system.
        TerminateThread(mhChildThread, 0);
        LeaveCriticalSection(&mtCountCS);
    }
}
```

Here, we simply call `TerminateThread`. Once again, consult your Win32 API reference documentation regarding the pitfalls of calling this function before you decide to use it in a production system. (The Waite Group Press book, “Win32 API SuperBible” warns of dire consequences that could significantly compromise the robustness of your application.)

To summarize, the PEG Win32 Development Platform affords all of the functionality associated with the PEG Multithread execution model and allows for extensive flexibility when designing your system. It is always best to process real-time or time critical aspects of your system in ancilliary threads, and not within the context of a PEG GUI thread. Finally, be sure to protect your data against corruption.

OS Porting Advanced Topics

Porting PEG to a new operating system involves devising means for servicing input devices, transferring messages between PEG and other system tasks, providing a periodic timer tick, servicing ANSI C++ memory allocation and de-allocation calls, and devising means for supporting the *Multithread* model (if required).

By starting with a Standalone environment such as the DOS environment, most users can do a minimum port of PEG to a new or custom operating system in one or 2 days. This does not include optimizing the message passing or support for the *Multithread* model.

Input Devices

Input devices are generally a touch screen, keyboard, joystick, mouse, or membrane keypad. Other devices can also be incorporated. Incorporating input devices into your PEG system requires two distinct areas of effort. First, the low-level hardware interface must be completed, which usually involves very hardware specific operations and/or an interrupt handler specific to the input device. While implementing this low-level interface is specific to the target platform, Swell Software will work with you to assist with correctly configuring new input devices. Second, the data returned by the low level interface must be enclosed in a PegMessage and the resulting PegMessage must be placed in the *PegMessageQueue*.

PEG input message types are intentionally very generic, which allows you to incorporate any type of input device and decide which type of device action should correspond to each of the device actions. PEG defines input messages for mouse and keyboard input only. Using another input device, such as a touch screen or joystick, requires that you logically map the input device interrupt events to the closest matching mouse events. This is usually very straightforward.

PEG Multitasking

As an example, consider a target system which uses a membrane keypad. A common situation is to place keypad buttons at a regular interval around one or more edges of the screen. Graphics (such as `PegTextButton` or `PegBitmapButton`) are then drawn on the screen at locations corresponding to each membrane keypad button. The graphics at each button location are changed as the user moves from screen to screen or from one operating mode to another. This is commonly called 'softkey' operation.

The above example is easily accomplished using the standard PEG mouse input messages. The keypad input handler must simply create PEG `PM_LBUTTONDOWN` and `PM_LBUTTONUP` messages corresponding to each keypad button. The **PegPoint** member variable of PEG mouse input messages contains the mouse pointer location. For this example, we will simply pass a point on the screen for each keypad button which roughly corresponds to the center position of the graphic on the screen corresponding to each button. The effect is that PEG is tricked into thinking a mouse was clicked over the graphic button, when in fact no mouse exists!

All types of input devices encountered to date are handled using similar means. During the porting process the target input devices are logically mapped to PEG mouse and/or keyboard devices. If you are concerned about using a particular input device with PEG, we encourage you to contact Swell Software to discuss the possible approaches.

It is best if input devices are interrupt driven, and post messages directly to `PegTask`. This allows the removal of `PegIdleFunction`, and task suspension directly by `PegMessageQueue`. Input devices may also be polled via `PegIdleFunction`, as is done in the standard DOS environment.

Interrupt driven input service routines are generally best constructed in two layers. The first layer responds to the hardware interrupt service request, does the

minimum processing required to remove the interrupt, and readies the second level service routine.

The second level service routine determines what type of message should be constructed and passed to PegTask. If the underlying PegMessageQueue is implemented via RTOS message services, the second level interrupt service routine can directly post a message to PegTask. If you are using the PegMessageQueue implementation for DOS and Windows, means must be provided to prevent linked list corruption when posting (and retrieving) messages.

Periodic Timer Service

This service is required for the operation of PegTimers. This is generally very easy to implement with a commercial RTOS. The PEG timer service is built on top of a single RTOS timer. As part of the RTOS timer service routine, a PM_TIMER message is generated with a NULL target and sent to PegTask. PegTask interprets the NULL targeted PM_TIMER message as a timer tick, and calls PegMessageQueue::TimerTick to check for and dispatch actual target specific PM_TIMER messages.

Memory Management Services

PEG uses the standard C++ memory management services, **new**, **new[]**, **delete**, and **delete[]**. Supporting these services on an embedded target is very specific to the RTOS and compiler being used. Some compilers provide very good documentation of how to hook these services, while others do not. Likewise, many RTOS vendors now provide their own hooks for these operations for each supported compiler. Check with your RTOS vendor and compiler documentation for information about supporting these memory management services in a thread-safe manner.

PEG Multitasking

MULTITHREAD support

PEG supports multiple tasks creating windows and interacting with the graphical interface when the PEG_MULTITHREAD define is enabled. In order to support this programming model, several internal data structures must be protected during critical code sections in order to insure that they are not corrupted. These critical code sections use macros defined in `\peg\include\peg.hpp` to invoke semaphore protection of each internal data structure. These macros are named “LOCK_resource_name” and “UNLOCK_resource_name” for each protected resource.

Macros are used to ease the porting effort required when integrating PEG with a new RTOS. You do not have to find each occurrence in the PEG source code where resource locking is required, you simply have to re-define the protection macros to invoke the proper operation with your RTOS. The use of macros also allows you to easily modify the operation of the resource locking and unlocking macros. For single-threaded environments such as DOS, the locking and unlocking macros are simply defined to do nothing.

For performance reasons it is usually best if each internal structure is protected with a unique semaphore. However, care has been taken to avoid deadlock in the event that you decide to protect multiple resources with a single semaphore.

Resource Protection Macros

The protected structures and associated protection macros are shown below:

PegMessageQueue

PegMessageQueue is protected with the macros “LOCK_MESSAGE_QUEUE” and “UNLOCK_MESSAGE_QUEUE”. For completed RTOS integrations, protection of the internal PEG message queue is unnecessary because the entire message queue is implemented via an RTOS message queue. In other cases, this semaphore is used to prevent corruption of a FIFO linked-list of queued messages maintained by PegMessageQueue.

PegTimerList

The high-level timer list is protected via the macros “LOCK_TIMER_LIST” and “UNLOCK_TIMER_LIST”. This semaphore prevents corruption of the linked list of active timers.

PegPresentationTree

This refers to the tree of visible objects, along with the physical screen. These resources are locked with LOCK_PEG, and unlocked with UNLOCK_PEG.

Additional requirements for MULTITHREAD support

In addition to the above resource protection macros, supporting the MULTITHREAD model on a new RTOS requires customization of PegMessageQueue and the addition of several functions used by PegPresentationManager and PegWindow to properly construct and use message queues for each GUI task.

The default PegMessageQueue implementation cannot be used in a MULTITHREAD environment because there are no services for dynamically creating additional queues on an as-needed basis.

These additional macros required for support of the MULTITHREAD model include:

CREATE MESG QUEUE

This macro calls a user or integration defined function to create a new message queue. This macro is invoked by PegPresentationManager when a secondary task calls the PegWindow::Execute function. The called function should return a PEG_QUEUE_TYPE *.

PEG Multitasking

DELETE MESSAGE QUEUE(a)

This macro calls a user or integration defined function to delete a previously created message queue. This macro should return void and accept a PEG_QUEUE_TYPE *.

ENQUEUE TASK MESSAGE(a, b)

This macro calls a user or integration defined function to send a message to a specific queue. The first parameter is a pointer to the message, and the second parameter is a pointer to the desired queue.

CURRENT TASK

This macro should return a PEG_TASK_TYPE indicating the current execution thread. This is typically an RTOS defined task control block pointer.

PEG TASK PTR

This macro should return a PEG_TASK_TYPE indicating the PegTask execution thread. This value is typically saved during PegTask startup for later reference. While supporting the GraphicsServer model is almost trivial, supporting the MULTITHREAD model is can become quite complex and requires a solid understanding of both PEG and the target OS. Assistance with integrating a new RTOS with PEG is available from Swell Software.

Chapter 10: PegFontCapture

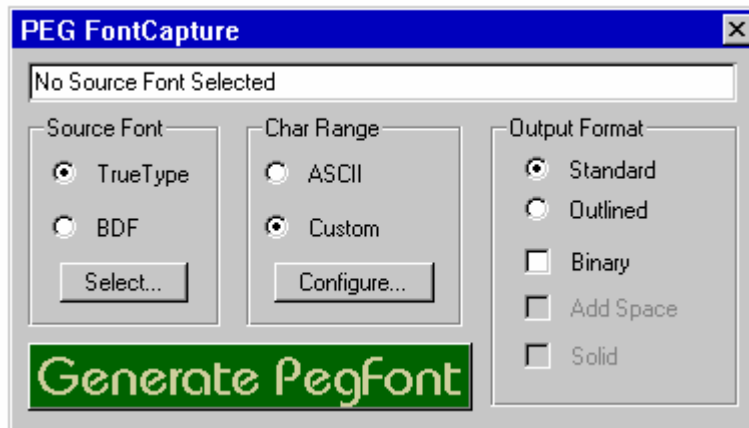
PEG FontCapture is a utility program written using PEG that can be used to generate additional fonts for use by your application program. FontCapture is included with licensed distributions of the PEG library. FontCapture generates .cpp source files or binary data files each containing a PegFont data structure, character widths and character data. Versions of FontCapture are available for MS Windows (all versions) and most Unix/Linux/X11 development stations.

When you choose the standard source-file output format, the .cpp files generated by PegFontCapture can be compiled, linked, and if desired ROM'ed along with the rest of your application code. Since the .cpp files generated by PegFontCapture entirely contain read-only data, additional PegFonts included with your system software should only consume ROM storage.

The binary output format is intended for use with targets which support a run-time file system. In this type of system, an arbitrary number of PegFonts may be stored as binary files and read from the filesystem into memory only when needed.

The PegFont format is a binary 1 or 2 bit/pixel font format. Future version of PEG and PegFontCapture will support additional font formats. The standard 1-bpp format is most commonly used. These fonts can be drawn in any combination of foreground and background colors supported by your target system. The 2 bit/pixel format is used for Outline fonts, which is a font format used primarily in video display systems such as television or set-top box applications. In this format the font includes a single-pixel wide outline. When drawing the font, the foreground color is used for the font outline and the background color is used as the font fill color.

PegFontCapture appears as shown on the following page:



The **Source Font** group allows you to select the source font type. FontCapture supports the conversion of MS Windows TrueType fonts or Adobe Postscript **Glyph Bitmap Distribution (BDF)** fonts. If you select TrueType as your source font type, the Select... button can be used to invoke the ChooseFont Windows common dialog (Note: this option is only available when running FontCapture on a MS Windows host). If you select the BDF source font format, the Select button allows you to select the actual .bdf file from your development station file system. No matter which type of source font you choose, the output of Font Capture is a PegFont data structure in 'C' data array format or binary format.

The **Char Range** group allows you to specify precisely the characters you want to include in your PegFont. For example the desired range for a certain font may include only numeric characters to reduce the resulting font size. For multi-lingual applications, you may need to specify a complex set of character ranges to support all languages included in your system. This sometimes involves using Unicode character encoding, which will be described in detail in later paragraphs. The simplest Char Range to specify is the ASCII character range. This includes characters 0x00 to 0x7f. For any other character range, you must specify in detail

the range or ranges of character you want to include using the range configuration dialog described below.

The ***Output Format*** group allows you to select between normal bitmapped font output and outlined font format.

The ***Outline*** checkbox can be used to generate a font with an added single pixel wide outline of each glyph. This is NOT the typical font type used in PEG applications, but is supported for the minority of applications that require an outlined font capability. When the Outline box is selected, PegFontCapture encodes the output PegBitmap in a 2-bpp format, where bitmap value 0 indicates the pixel should be the foreground color, bitmap value 1 indicates that the pixel should be in the outline color, and bitmap value 2 indicates that the pixel should be either the background color or transparent, depending on the PegColor.uFill value passed to the text drawing function.

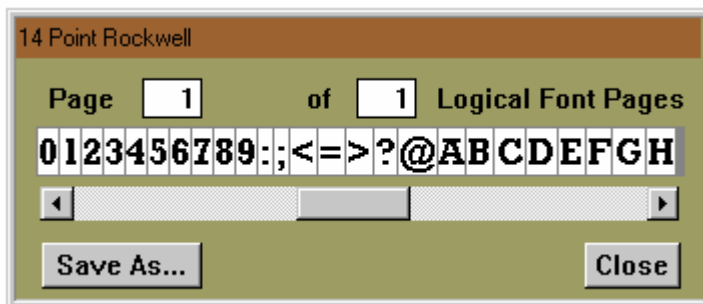
While FontCapture can generate 2-bpp fonts, you should not attempt to use them unless your PegScreen interface class supports this font format. The Win32 screen interface class PegWinScreen includes this functionality as a reference for users who desire to display outlined fonts.

The ***Binary*** output format is used when you want to create a binary PegFont file, rather than the more common C source file.

The ***Solid*** and ***Add Space*** checkboxes are modifiers for the outline font generation mode. The Solid checkbox causes the font outline to appear somewhat heavier than the default outline. The Solid choice is beneficial when working with large fonts. The Add Space option adds a single pixel of spacing between each generated character when generating an outline font. This is beneficial when working with very small outlined fonts. The Solid and Add Space modifiers are ignored if the Outline checkbox is not selected.

PEG Font Capture

The ***Generate PegFont*** button causes FontCapture to convert the source font into a PegFont. This process may occur very quickly for a small font, or it may take several minutes for a very large font containing many thousands of characters. You can capture as many fonts as you like within one session of running FontCapture. After converting the source font, FontCapture will display the window below to preview the resulting PegFont:



You can use this window to examine the PegFont produced, and even compare multiple fonts to find the best appearance. Once you are satisfied with the appearance of your font, you can use the Save As... button on this window to save the font to a source or binary file of your choosing.

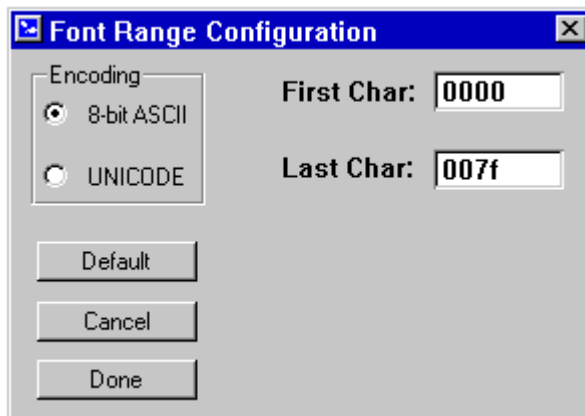
Configuring Character Range

The ***Char Range*** group allows you to specify the range of character glyphs that will be encoded in the output font. When the ASCII option is selected, the range of characters is fixed to ASCII-0 through ASCII-127, which is the normal range for single language applications.

Font Capture also allows you to specify a custom range of characters to be encoded. When you select the Custom option, the Configure... option becomes

active, allowing you to fully define the range of glyphs that will be recorded in the output file.

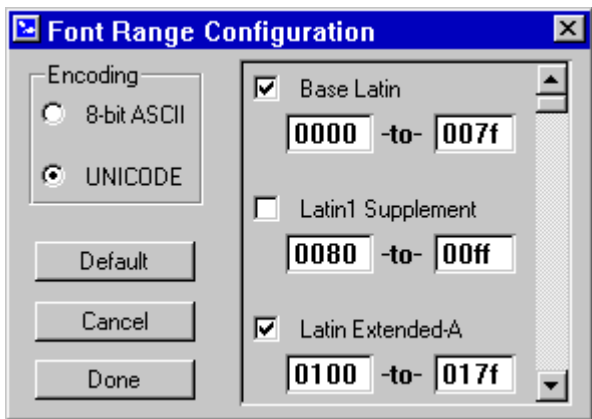
It is often the case that a particular font is only used to display a certain range of characters; for example you may define one font that will be used only for displaying numbers. In this case, you do not need or want to encode the entire ASCII character range in the output file. Instead, you can enter a limited character range by selecting the “Custom” button, and entering the range of characters in the Range Configuration dialog, shown here:



The First Char and Last Char fields allow you to define the start and ending characters to be encoded. Using the numerical example above, you could enter “0030” (i.e. ASCII-‘0’) as the first character, and “0039” (i.e. ASCII-‘9’) as the last character. This will save quite a large amount of memory over capturing the entire ASCII character set.

Multilingual Support and UNICODE

A more advanced use of the Range Configuration dialog deals with UNICODE fonts. When you select the UNICODE option on this dialog, the dialog appearance changes as shown below:



Before we can fully understand how to configure custom UNICODE character ranges, we must first examine what UNICODE is, the options available for supporting multiple languages, and the tradeoffs involved with each approach.

What is UNICODE?

If you are a software developer from North America, you may not be more than vaguely familiar with what UNICODE is, or what it means to your software. UNICODE is a standard definition of 16-bit character encoding that encompass all characters used for all of the most prominent writing structures. For example, the UNICODE standard defines character encodings for characters used to record Latin (~English), Japanese, Korean, and Georgian writings.

To really understand what the UNICODE is, a little clarification in terminology is required. We often confuse or mix the terms 'language', 'alphabet', 'character',

and ‘glyph’. A glyph is a shape representing a character. For example, ‘A’, ‘*A*’, and ‘**A**’ are three individual and unique glyphs, however they are all the same character: ‘Capital Letter A’.

UNICODE defines a unique encoding for each character. UNICODE does not define a font, style, size, or any other attributes for a character. Since there are far more recognized characters in the world (> 28,000) than can be encoded using an 8-bit representation, UNICODE uses 16-bit values to encode each character.

Further Reading

To learn more about the UNICODE standard, we encourage you to purchase The Unicode Standard, Version 3.0 (ISBN 0-201-48345-9)

PEG Character Encoding

The UNICODE font range selection dialog allows you to specify the groups, or code pages, of characters you want to encode. If you select multiple code pages for one font, FontCapture will generate at least one PegFont page for each code page you enable. In all cases the resulting fonts use Unicode character encoding, even if your code page selections leave “holes”, i.e. even if you select a non-contiguous set of character pages. However the multi-page PegFont encoding scheme allows the final font to simply skip any unused range(s) of characters, eliminating memory use for those unsupported code pages.

Font Range Configuration

As stated above, FontCapture allows you to specify precisely the code pages and ranges of characters you need for your application. You enable or disable each code page by selecting the corresponding check box for each page. The numeric range for a code page that is not enabled is ignored.

PEG Font Capture

For each code page that is enabled, you can specify an exact window of character values to capture. These character ranges are entered in hexadecimal format, consistent with Unicode encoding.

The ability to capture limited windows within each code page is very useful for multilingual applications that are attempting to produce a minimal memory footprint. This enables you to select the specific code pages and ranges of characters required in your application, without capturing all of the characters in each page. For example, you may desire to capture code page 1 (Basic Latin) indexes 0020 through 0080, code page 2 (Latin 1) characters 0090 through 0100, and a few additional characters from code page 9 (Cyrillic). You may thus create a custom font containing ≤ 256 characters, but still containing all of the glyphs you need for your multilingual application.

Even if you are using 16-bit character encoding, you will very likely not want to attempt to capture the entire UNICODE character set. Such a character set would require a huge amount of memory, and it is highly unlikely that you will find a font containing anywhere near the entire UNICODE character set. Font Capture allows you to specify exactly which code pages you want to capture from the selected font.

Once you have entered the range configuration, Font Capture saves the configuration (or 'profile') to a binary file for later retrieval. The next time you start the Font Capture program, it will automatically default to the set of ranges defined in the previous usage.

Whenever Font Capture is operated using a Custom font range, the header in the output file produced contains a comment section indicating the mapping of Unicode characters to the character indexes in the captured font. This mapping is required if you want to manually enter 16-bit string values in your application.

Applying Custom Character Filters

FontCapture also allows you to specify a custom range of characters to be encoded by using a character filter file. On the Range dialog you may select the **Use Custom Filter File** checkbox. When this checkbox is selected, you can type the path and filename of a file to be used as a final character filter. In other words, characters selected above will be verified against the filter list and only those characters listed in the filter will be included in the output font.

The custom filter file should have one hexadecimal character encoding per line. An example is shown here:

```
0x3456          // you can put comments
0x3467          // or other notes after the character encoding
0x3786          // as shown here
```

This file format was chosen because it works perfectly with the encoding tables provided with the Unicode Standard version 2.0!! The Unicode standard accepts that many character encoding “standards” are in existence and provides table to map the alternate character encodings to the Unicode encoding. These table can be directly supplied to FontCapture as filter files, allowing you to generate Unicode encoded fonts containing only those characters defined by a previous standard.

Should You Use UNICODE?

If you are working on an application that must support many languages, this is of course the question you are anxious to answer. Supporting multiple languages does not always imply using multiple alphabets or using a single character set containing all of the characters required for each language. All common North-American and most Western European languages can be supported very well by using a single 256 character alphabet. There are, of course exceptions.

PEG Font Capture

There are two schemes in broad use for displaying information in multiple languages. The first scheme uses multiple 256 character font sets, with the characters required for each language or possibly each group of languages encoded in separate font files.

For example, suppose you write down and tabulate every character required to display all strings defined in your application in each language you are required to support. After going through this process, you may find that your application needs to be able to display a total of 500 unique characters. Further, you find that half of the languages can be supported using the first 250 characters, and the other languages can be supported using the second set of 250 characters. In this case, you might decide to use two 256 character font sets, and switch languages simply by switching the fonts used for displaying strings in different languages.

The advantage to the multiple-font approach is that you do not need to use 16-bit character encoding, although your application can draw more than 256 unique glyphs. This can simplify your application development and reduce the amount of memory required to store your character strings. One disadvantage of this approach is that there is not a unique mapping of characters to character encoding, i.e. character 0x0030 in one font may be the glyph ‘0’, while in another font this may be an entirely different glyph.

The alternate approach is to place all required glyphs in a single font file (or multiple font files, if different font sizes and styles are needed). If the number of characters contained in ANY single font file exceeds 256, you will need to run PEG in UNICODE mode, meaning that all PEG strings will be encoded using 16-bits/character. Note that we prefer here to use the term “16-bit encoding”, since as stated in the previous section the font produces with Font Capture may not contain a true Unicode character mapping, depending on the font range configuration. Running PEG in “Unicode Mode” means only that you are using 16-bit character encoding, and does not mean that your string values will be strict Unicode standard encoding.

The advantages to the UNICODE approach are that you do not need to switch fonts when switching between languages, each character encoding is unique and unambiguous, and very large character sets are accommodated with no extra programming effort beyond what is required when first stepping to UNICODE. The disadvantages include increased memory requirements for string storage, and the inability of software debugging applications to display arrays of 16-bit encoded values as “strings”. A further disadvantage is that many run-time string libraries do not support 16-bit character encoding.

You should not take the decision to use 16-bit character encoding lightly. While this string encoding can greatly simplify the long-term programming effort, it will almost certainly take you and your development team a while to become comfortable with using 16-bit characters. Further, it is a time consuming process to manually enter strings that use 16-bit encoding, especially if your compiler has no intrinsic support for 16-bit character encoding. While the WindowBuilder string table editor is provided to aid this process, you will miss the simplicity of entering “String” into your editor!

Defining Unicode Strings

If you decide to run in Unicode mode, you will need to define and initialize your string data such that it is recognized as an array of 16-bit variables. A few compilers have built-in support for this type of data entry, but even in these cases you cannot enter Unicode-formatted strings that use characters above the printable character range using the standard ‘C’ double quoted string syntax.

The easiest way to create your Unicode strings is to allow WindowBuilder (described in chapter 12) to generate them for you. WindowBuilder provides a drag-and-drop string entry editor that allows you to select any character from an extended font file you have generated using FontCapture. You simply pick the characters you want to use, and WindowBuilder generates the associates Unicode formatted initialized array in a portable ‘C’ format.

PEG Font Capture

If you really need to enter your own Unicode format strings, the most portable method is to enter the strings as PEGCHAR arrays, as shown below:

```
PEGCHAR TestString[] = {'H', 'e', 'l', 'l', 'o', ' ', 0x209, 0x210, 0x224, 0};
```

In this example, we have mixed a few characters from the ASCII range with a few characters that are only available in the extended Unicode character set. Note that there currently is no other method available for entering and initializing character codes that fall above the ASCII range unless you are using some type of Unicode enabled editor.

Using Custom Fonts

Using fonts generated with FontCapture is very simple. Every PEG object that supports text output has a member function called `SetFont(PegFont *)`. Therefore, after constructing a PEG object that should use the new font, you simply call that object's `SetFont` function, passing a pointer to your new font. For example, assume that you told `PegFontCapture` to generate a new font called “MyNewFont”, by typing this in the font name field of `PegFontCapture`. In this case, the following code fragment illustrates the process of altering the font used by a PEG object:

```
extern PegFont MyNewFont;  
  
PegTextButton *pButton = new PegTextButton(10, 10, 100, MESG1, "NewFont");  
pButton->SetFont(&MyNewFont);
```

Likewise, if you have overloaded a `Draw()` function for a window or other object, and you are drawing text on the screen, you can simply pass the pointer to your new font to any of the ***PegScreen*** text information or output functions.

Changing Font Defaults

Every PEG object type that supports text has a default font definition. This default font definition is held in a static array which is a member of the `PegTextThing` class. By calling the `PegTextThing::SetDefaultFont` function you can change the default font assigned to all successive instances of any or all `PegTextThing` derived objects. For example, you might use the `SetDefaultFont` function to change the font used for all `PegTitle` or all `PegTextButton` objects.

Chapter 11: PEG Image Convert

Overview

PEG Image Convert is a utility program developed by Swell Software that can be used to convert MS Windows .bmp, CompuServe GIF, PNG, and JPG files into binary or source code formats supported by PEG. All of these file formats are in the public domain, and you may use PegImageConvert to process files which you create in any of these formats. PegImageConvert is a program written using PEG and running under the Win32 development environment.

Input files can be 1, 2, 4, 8, or 24 bit-per-pixel formats. Likewise, PegImageConvert can generate 1, 2, 4, or 8 bit-per-pixel compressed image files. During the image conversion process the palette used to encode the output image can be saved in source format, suitable for use in calling the PegScreen **SetupPalette()** function.

The actual operation of PegImageConvert is heavily dependent on the selected output format. ImageConvert may perform color reduction, dithering, RLE encoding, and transparency encoding for all input file types. In addition, for 8-bpp output format, Image Convert adds optimal palette generation.

While PegImageConvert can output PegBitmap structures in many formats, this is of little use if the PegScreen driver being used on the target is not capable of properly displaying the bitmaps in the format in which they are saved. For example, while PegImageConvert can save PegBitmap structures using 2-bpp encoding, this format should only be used if your derived PegScreen driver class understands and can properly display images saved in 2-bpp format. ***The PegScreen derived interface classes provided with PEG support both 8-bpp bitmap encoding and the native encoding corresponding to the color depth of the target display.***

All of the options that can be selected on the PegImageConvert dialog window control the **output** of PegImageConvert. The format and data content of the input file(s) is determined by PegImageConvert by reading and parsing the input file header information.

If you target supports 256 or more colors, PegImageConvert can also perform advanced palette reduction and optimization, allowing you to create and use any number of color palettes, each of which is optimized for the images displayed in your application. This option is best utilized along with batch image processing (described below), which allows a custom palette to be created for optimal display of multiple images. The input files for list processing can be any combination of the supported file types, and can even have different internal formats in terms of the color resolution associated with each input file. For those readers who are familiar with color quantization methods, PEG Image Convert utilizes an improved form of Heckbert's 'Median Cut' algorithm for color reduction.

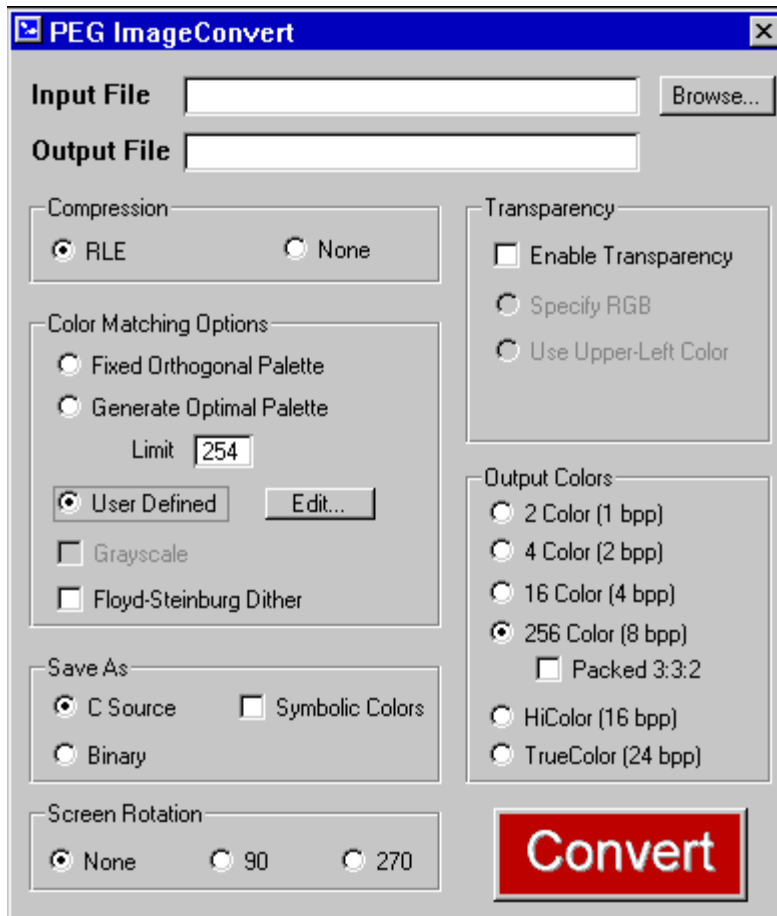
PegImageConvert is not a paint program or ray-tracing package. If you will be creating and using bitmaps and animations in your PEG application program, you will need to obtain a paint and/or graphics program capable of outputting one or all of the supported input file formats. The **Paint** program which is standard with MS Windows provides the minimum functionality you will most likely need, although **Paint** is very limited in terms of palette control and export formats.

If you are working with 256 or more colors, you may also find it useful to purchase an image processing software package capable of translating between several common graphic file formats. ***Paint Shop Pro***, by Jasc, Inc., is one such graphics program that will do everything you will need when creating bitmaps for use with PEG. ***Paint Shop Pro*** is very inexpensive and is available via the Internet for evaluation.

PEG Image Convert

The Conversion Dialog

The PegImageConvert application dialog appears as shown below:



Input File

The input file string allows you to select the source image file. You can either type in the name of the file, or you can use the 'Browse' button to select a file from your computer.

Only one file path\name can be entered in the Input File string field. If you want to process multiple input images at one time, you should enter the image names in an ASCII command file and select the command file as the input file. This is described in more detail in the 'Batch Processing' section.

The basic input file type is determined by ImageConvert based on the input filename extension. For MS or OS2 Bitmap files, the filename extension should be '.bmp'. For GIF files, the filename extension should be '.gif', for PNG file the extension should be ".png", and for JPEG files the filename extension should be ".jpg". To process multiple input files, the filename extension should be '.cmd', which is interpreted as a command file.

PegImageConvert verifies that the file is truly of the type indicated by the file extension by attempting to read the file header information and verifying that the file header makes sense for the indicated file type. An error is reported if the file header information and filename extension do not correspond.

Output File

The output file string field allows you to specify where and to what file name PegImageConvert will save the generated output file. *PegImageConvert also uses this filename as the name of the final PegBitmap.* For this reason, you should enter the filename in exactly for the form you want the resulting bitmap to be named, including upper and lower case characters. PegImageConvert pre-fixes the letters 'gb' to the bitmap name, and post-fixes the letters 'Bitmap'. For example, the following output name:

PEG Image Convert

C:\mybitmaps\House.cpp

will result in the final bitmap being named ***gbHouseBitmap***. This is the name you will use in your source code when referring to the bitmap. To use this bitmap on a PegBitmapButton button, for example, you would then do something similar to the following in your source code:

```
extern PegBitmap gbHouseBitmap;

void MyWindow::MyWindow(.....)
{
    .
    Add(new PegBitmapButton(20, 20, &gbHouseBitmap));
    .
}
```

The naming convention for the resulting PegBitmap structures is slightly different when batch processing, which is described later.

Compression

PegImageConvert can optionally apply a simple RLE compression technique to the output data. The effectiveness of this compression depends on many factors. If your input image is a computer generated image with few colors, RLE compression can be very effective. If your input image was produced with a RAY-tracing package or from an actual photograph, RLE compression is less successful.

When RLE compression is enabled, PegImageConvert is required to save the output data in 8-bpp format, regardless of what format you have selected in the Output Colors field. This means that for 1-bpp, 2-bpp, and 4-bpp input images, turning on RLE compression forces PegImageConvert to first expand the image to 8-bpp format, and then apply RLE compression. Depending on the exact image file,

this can actually cause the final output file to be larger than if compression is not used.

For this reason, selecting RLE compression is actually only a suggestion to PEG Image Convert. If RLE compression is effective at reducing image size, the compression is performed. If compression does not reduce the output image size, the RLE encoding is omitted. This decision is made automatically by Image Convert during the conversion process. Therefore, the only reason to disable RLE compression is if your PegScreen derived screen driver does not support RLE encoded PegBitmap formats. The PegScreen drivers provided with PEG do support RLE encoding.

RLE compression is almost always beneficial if you are using 8-bpp bitmap encoding, especially for very large images. Compression ratios typically vary from 10:1 to 3:2, depending again on the source of the image being processed.

The use of dithering on the output bitmap has an negative impact on RLE compression effectiveness. For this reason, if data size is the most important consideration in your application, you should disable the dithering option. This forces Image Convert to do a “Best Match” color mapping of input to output colors.

Palette Options

Image Convert can apply various color optimization and dithering methods when converting the input images to PegBitmap encoded data structures. You're input images can be any combination of 2, 4, 16, 256, or true-color (i.e. 24-bpp) input images. Image Convert will convert the input images to best possible representations on the target system.

If your source images contain a higher number of colors than are available on your target display, ImageConvert will reduce the number of colors in the source image. This reduction will either perform a best-match remapping or a dithering algorithm, depending on whether or not the dithering option is selected.

For example, suppose you have several full-color GIF images you intend to use on a target system that utilizes a 4 color grayscale LCD display. In this case, you would select 2-bpp output format, and ImageConvert will reduce the full color GIF images to the best possible grayscale representation.

Fixed Orthogonal

The ***Fixed Orthogonal*** palette option instructs image convert to use a pre-defined palette covering the rainbow of colors available for 16 or 256 color targets. This is the only palette option when running in with fewer than 256 colors. When targeting 256-color operation, you must choose between the fixed pre-defined system palette (the ***Fixed Orthogonal*** palette) or an optimal system palette created for your images. The 256-color fixed orthogonal palette used by PEG is contained in the file `peg\include\pal256.hpp`.

Generate Optimal

The ***Generate Optimal*** palette option is only available if you are targeting 256 color (i.e. 8-bpp) output. This is the opposite of using a Fixed Orthogonal palette. When this option is selected, ImageConvert will create a custom palette for use with the input images. The custom palette will be saved at the top of the output file, and will be named *PegCustomPalette*. The custom palette is simply an array of 256*3 unsigned characters, which is passed to the `PegScreen::SetupPalette` function when you want to use the custom palette.

Most users prefer to generate and use a custom palette when running in 256-color or higher modes. This provides you with the best possible image display. Since the resulting palette is generally modified extensively as compared to the palette that was included in the input images, the input images are automatically re-encoded by Image Convert to use the newly created palette. This all happens transparently when the 'Generate Optimal' option is selected in the PegImageConvert dialog.

You do not have to do anything special when you are creating your image bitmap or GIF files in order to use a custom palette.

The custom palette created by PegImageConvert is always named **PegCustomPalette**, and is found at the top of the ASCII output file generated by PegImageConvert. This palette always starts with the 16 standard PEG colors, and is followed by up to 240 colors selected to produce the best possible image display for your input images. The custom palette is simply an array of unsigned characters containing the Red, Green, and Blue components of each color. This array of RGB values should be programmed into your video controller palette registers prior to displaying the associated bitmap(s). This palette can be directly passed to the PegScreen::SetupPalette() function, as shown below:

```
PegScreen()->SetupPalette(PegCustomPalette, 256);
```

It is also possible to use multiple custom palettes. When multiple custom palettes are used, it is the responsibility of the application level software to install the correct custom palette before the corresponding images are displayed. For systems that display 'one window at a time', it is a simple matter to install the correct palette when each window is displayed. For other systems, it can be complex to use multiple palettes, and one optimal palette is generally preferred.

Custom Palette

The Custom Palette option allows you to define with a simple editor any palette you prefer, and ImageConvert will remap each pixel color value in the input file(s) to your preferred palette before generating the output image. The custom palette editor allows you to edit and save the palette RGB values, and you can save the palette definitions to any number of custom palette files.

Floyd-Steinburg Dither

The *Floyd-Steinburg Dither* option instructs image convert to dither your images when re-encoding them to the target palette. Dithering can be used in any of the

PEG Image Convert

output color depths, with or without a custom palette. What is dithering?? You should realize that when an optimal palette is created for multiple images, the actual colors contained in the final palette may not exactly match the original image colors. Likewise when ImageConvert is outputting bitmaps for 16-color targets using input images that contain 256 or more colors, Image Convert must translate those original colors into the best possible representation using only the 16-color palette.

The dithering option tells ImageConvert how to convert your original image colors to the new system palette colors. If dithering is selected, ImageConvert will pick colors such that the average value in each multi-pixel area is equal to the average value of the original input colors for the same multi-pixel area. If dithering is disabled, Image Convert will simply translate each pixel into its nearest color in the target palette.

Should you dither? It depends on your target system, your palette options, and your input images. If you are creating a custom palette, dithering usually has very little effect since the custom palette will contain colors very close to the original image colors. In most other cases, dithering can significantly improve your color display, especially if you are displaying photographic images on a target with fewer than 256 colors. The drawback to dithering is that your images can take on a speckled appearance, especially if large areas of the original image are in a solid color. If your source images are photographic or ray-traced images, you will almost certainly want to dither. If your input images are hand-drawn images with few colors, you may want to disable dithering and use a best-match color mapping.

Output Format

The Output Format option specifies whether PegImageConvert should generate 'C' source code or binary data. If your target system has means for file I/O, you can greatly reduce the RAM or ROM storage requirements for your bitmaps by saving them as binary files, and retrieving them from the file system when they are

needed. Binary data can only be used if your final target system supports means for file I/O.

When ‘C’ source data structures are generated, PegImageConvert writes a normal ASCII file that can be opened and modified with any editor. This ASCII file contains the bitmap data array, along with the corresponding PegBitmap structure definition. If an optimal palette is generated, the ASCII file will also contain the custom palette.

When ‘C’ source code is generated and the output color depth is 8-bpp, you may also select the “Symbolic Colors” option if desired. Normally ImageConvert outputs simple hex values corresponding to each color value. When the Symbolic Colors option is selected, ImageConvert uses the color names (defined in the header file `\peg\include\pegtypes.hpp`) such as BLACK, WHITE, BLUE and so on for each value in the bitmap data array. This is convenient when a common bitmap is going to be used with multiple display devices and/or PegScreen drivers. The symbolic color names are re-defined based on the screen driver in use, allowing the same output file to be used in multiple systems. Unless you will be using a common ImageConvert output file with multiple PegScreen drivers and display devices, you should not select the Symbolic Colors option.

When binary format is selected, PegImageConvert generates a binary data file containing one or more PegBitmap data structures and bitmap data definitions. The binary file starts with an eight byte leader, shown below:

// Leader, one occurrence per binary image file:

<code>char cVersion[4]</code>	// four byte version string, “1.00”
<code>UCHAR uReservedA</code>	// 1 byte reserved
<code>UCHAR uHavePalette</code>	// 1 byte palette flag
<code>UCHAR uReservedB[2]</code>	// 2 unused bytes

PEG Image Convert

The uHavePalette byte of the leader signals the presence or absence of a color palette in the binary file. If the binary file contains a custom palette, uHavePalette will be non-zero and the custom palette immediately follows the file leader, and can be declared as shown below:

```
// Palette, max one occurrence per binary image file

    UCHAR Palette[256*3]      // only present when custom palette is generated.
```

Following the short leader and optional palette data are the bitmap header and bitmap data fields. The bitmap header and data fields are repeated for each image contained in the file. Each bitmap is contained in the binary file as shown below:

```
// repeated for each bitmap in file:

    Char cName[28]           // 28 bytes, bitmap name left justified padded with NULLS
    UCHAR uType              // 1 byte, PegBitmap.uType
    UCHAR uBitsPerPixel      // 1 byte, PegBitmap.uBitsPerPixel
    WORD wWidth              // 2 bytes, PegBitmap.wWidth
    WORD wHeight             // 2 bytes, PegBitmap.wHeight
    WORD wReserved           // 2 bytes, unused
    LONG lSize               // 4 bytes, size of data array in bytes
    UCHAR uData[lSize]       // bitmap data values
```

For each bitmap, lSize bytes of bitmap data immediately follow the bitmap header information. When multiple PegBitmaps are generated using batch conversion, each successive bitmap header immediately follows the previous bitmap data. There are no padding or alignment bytes inserted between bitmaps.

For multi-byte fields such as wWidth and wHeight, byte swapping may be required when reading the bitmap header data depending on the endian type of your CPU and the method used to read the bitmap data value. PegImageConvert always writes multi-byte values MSB (most significant byte) first. Byte swapping is not required for the actual bitmap data, as this section always contains only single-byte values.

Reading a bitmap or series of bitmaps from a binary file can be accomplished with the pseudo-code shown below. Several variations of this example could also be used:

```

    UCHAR *ReadBitmap(FILE *pSrc, PegBitmap &Bitmap)
    {
        UCHAR uTemp[30];
        UCHAR *pPalette;
        LONG  IDataSize;

        fread(uTemp, 1, 8, pSrc);    // read in the leader

        // ** check here for correct version string **

        if (uTemp[5])                // does file contain a palette?
        {
            pPalette = new UCHAR[256 * 3];
            fread(pPalette, 1, 256*3, pSrc);    // read the palette
        }
        else
        {
            pPalette = NULL;
        }

        fread(uTemp, 1, 28, pSrc);    // read image name
    }

```

PEG Image Convert

```
// ** verify image name **

fread(&Bitmap.uType, 1, 1, pSrc);
fread(&Bitmap.uBitsPerPix, 1, 1, pSrc);
fread(&Bitmap.wWidth, 1, 2, pSrc);
fread(&Bitmap.wHeight, 1, 2, pSrc);
fread(uTemp, 1, 2, pSrc);           // skip unused bytes
fread(&IDataSize, 1, 4, pSrc);      // get data size
Bitmap.pStart = new UCHAR[IDataSize]; // get RAM for bitmap data
fread(Bitmap.pStart, 1, IDataSize, pSrc);
return pPalette;
}
```

Note that if the binary file contains multiple bitmaps, the application software could also pass to ReadBitmap() the name of the image file to load. In that case, ReadBitmap would loop through the binary images until the correct bitmap name is found.

Screen Rotation

The screen rotation options change the format of the data produced by ImageConvert. These options should be set to “None” unless you are using one of the profile mode screen drivers. Refer to the chapter on PegScreen for more information on the profile mode screen driver templates.

ImageConvert normally outputs image data in a left to right, top to bottom format. The first data byte corresponds to the upper left image pixel, the next byte corresponds to pixel (1,0), and so on scanning from left to right across the image and moving from the top row to the last row. This is the format expected by the standard PegScreen classes.

The profile mode screen drivers are able to display image data more quickly if the data is stored in a different format corresponding to the screen rotation. The “90” rotation setting should be used when your display device has been mechanically

rotated 90 degrees counter-clockwise. When this is the case, ImageConvert outputs the image data such that the first data byte is the lower-left pixel of the image. Data is then saved in a bottom-to-top, left to right manner.

The “270” rotation setting should be used when your display device has been rotated 90 degrees clockwise (i.e. 270 degrees counter-clockwise). When this is the case, ImageConvert outputs the image data such that the first data byte is the upper-right pixel of the image. Data is then saved in a top-to-bottom, right-to-left manner.

These output format modifications are completely transparent to your application level software. Your application software simply passes PegBitmap addresses to the PegScreen drawing functions. However, it is important that when using the ImageConvert utility you set the Screen Rotation setting to match the display driver you are using.

Note that the pre-defined PEG system bitmaps, contained in the file pbitmaps.cpp, are provided in all forms including non-rotated, clockwise rotation, and counter-clockwise rotation. Only the format corresponding to the screen driver in use is actually compiled and linked into the target system software.

Transparency

The Transparency field can be used to specify a transparent color in the input image. This field only applies to MS Windows or OS2 bitmap files, as GIF and JPG files encode transparency information internal to the input file. In all cases, the transparent color will be saved as index 255 in the output PegBitmap, since index 255 is always interpreted as transparent by the *PegScreen* bitmap functions. Transparency forces the *output PegBitmap structure* to use 8-bpp encoding, since the default transparent color value is 255. This is true even if your source image contains only 2, 4, or 16 colors. If the source image is encoded using less than 8 bits-per-pixel, PegImageConvert will expand the image to 8 bits-per-pixel format when you enable transparency.

PEG Image Convert

MS Windows bitmap files do not inherently support transparency. To use image transparency, the source image(s) must be created such that all areas that should be displayed transparently are ‘painted’ with an otherwise unused color. You must then inform PegImageConvert which color should be interpreted as transparent. There are two methods of specifying the transparent color:

Specify RGB Value

If the source images are 24-bpp (true-color) images, you must specify an actual RGB value to use as the transparent color. If you select the ‘RGB’ button, you should enter the Red, Green, and Blue values in the string fields which are displayed. You can determine the correct values through examination with a quality paint program.

Use Upper-Left Color

When this method is selected, PegImageConvert will assume that the upper left corner pixel is in the transparent color. This method can only be used with 16 and 256 color input images.

Output Colors

The output colors field allows you to specify how the generated PegBitmap structures will be encoded, and tells ImageConvert how many colors are available on the target system. PegBitmap structures can be encoded using 1, 2, 4, 8, 16, or 24 bits-per-pixel. If you are using a system that supports less than 256 colors, saving the output PegBitmap structures in one of these lower bit-per-pixel formats can save a large amount of memory. Note, however, that using 1, 2, or 4 bpp output formats requires that your PegScreen driver class recognizes and properly displays PegBitmaps saved in this format. It is also possible to use a combination of different PegBitmap encodings in one PEG application, as long as your display driver can handle each of the formats in use.

There are two 256-color output formats supported. The default is 256 color palette mode, in which each color value is an index into a set of RGB values maintained in

the video controller color palette. This is the most common 256 color mode. An alternate format is the 8-bit packed-pixel mode. In this mode each data value passed to the video controller represents 3 bits of red, 3 bits of green, and 2 bits of blue color data. This is the only 256 color mode supported by a small set of video controller chips. For this reason you can select the 3:3:2 packed mode when you configure ImageConvert for 8-bpp (256 color) output. Note that when running in packed pixel mode the color palette is not required and is not used.

If transparency or RLE compression are enabled, the resulting PegBitmap structures are saved using 8-bpp encoding, regardless of your selection in the Output Colors field.

The PegScreen driver classes provided in your PEG distribution all support 8-bpp encoding (enabling the use of transparency), RLE encoding, and the native encoding format corresponding to your output color depth.

Batch Conversion

PegImageConvert can be used to perform list or batch conversion of multiple images. This is helpful in almost all cases, and absolutely essential when the goal is to create an optimal palette for multiple images. Batch conversion is selected by specifying an input file with a filename extension of '.cmd', which indicates that the source file is actually a command file, rather than an individual image file.

Command files are simply lists of input images, along with optional comments. They do not instruct PegImageConvert in terms of the type of conversion to perform. This is still done by selecting the appropriate options in the PegImageConvert dialog.

The command file should be an ASCII command file, with one command per line. Each command line should contain a single input image path and filename, and the output image name. The source file and output image name can be separated by any combination of space, comma, and tab characters. The output image name is the

PEG Image Convert

name you will use in your application software when passing the image address to one of the PegScreen bitmap display functions.

When performing batch conversion, all of the resulting output images are saved in one output file, along with the custom palette if a custom palette has been generated.

Very large command files are often easier to maintain by entering comments within the file to indicate where groups of bitmaps files are used. Comment lines are indicated by a single '#' character in the first column of a line.

The following is an example of a typical command file:

```
#
# This is a comment line
# Each command line specifies one input file, and the name of the resulting
# PegBitmap structure.
#
\graphics\bitmaps\stop.bmp  StopSign
\graphics\bitmaps\go.bmp   GoSign
#
# note that .bmp and .tga files can be processed within the same .cmd file
#
\graphics\targa\yield.tga   YieldSign
```

In the above example, the goal is to produce three PegBitmaps and a single optimal palette for use with these three images. The source images are *stop.bmp*, *go.bmp*, and *yield.tga*.

The resulting PegBitmaps will be named *gbStopSignBitmap*, *gbGoSignBitmap*, and *gbYieldSignBitmap*, respectively.

PegImageConvert will process each of the input files using the options selected in the PegImageConvert dialog, and will save all output to the single file specified in the 'Output File' field of the conversion dialog.

Implementation Notes

PEG Image Convert uses the PEG library classes PegImageConvert, PegGifConvert, PegJpgConvert, PegBmpConvert, PegPngConvert and PegQuant to do the work of converting your graphic files into the PegBitmap format and producing optimal color palettes. Since these classes are included with the PEG library, you can also create PEG application programs that read and decompress these common graphic file formats at run time. For most embedded applications, run-time decompression is not desired due to the performance and memory requirements. These systems are better served by using PEG Image Convert to process the application graphics prior to compile time.

The benefit to run-time decompression is of course memory savings. If your application uses a large number of large graphics files, you may decide to use the PEG image conversion classes directly in your application software to perform run-time graphic file decompression.

Chapter 12 WindowBuilder

Overview

WindowBuilder is a rapid prototyping and design tool used to quickly create your PEG windows and dialogs. WindowBuilder is a Win32 application program, and will therefore run on nearly any PC running MS Windows '95, '98, or MS Windows NT. An alternate version of WindowBuilder suitable for use in an X11 development environment is in development and is scheduled for release Q4 2000.

While WindowBuilder may at first glance appear to be a normal MS Windows application, WindowBuilder is actually a PEG application program, running in our Win32 development environment. There are several reasons why WindowBuilder is written entirely using the PEG library. First WindowBuilder is a relatively complex application program and therefore presents a good test case for insuring that the PEG library provides the features and capabilities required by complex programs. Second, since the entire interface is created using PEG library classes, exact WYSIWYG appearance is absolutely guaranteed. The appearance of your PEG objects created from within WindowBuilder will match exactly the appearance of those same objects on your target system, within the normal variations of pixel size, aspect ratio, and color performance. Finally, since WindowBuilder is based primarily on the PEG library, it is possible to port and run WindowBuilder on non-PC platforms.

The main Window Builder window can be re-sized to take full advantage of your screen real estate. In this case, the virtual frame buffer that is always used by PEG under Win32 is re-sized to match the client area of the outer window. This allows you to see your target windows and dialogs at full size.

WindowBuilder Project Files

All work that you do while running WindowBuilder is saved in a binary data structure called your WindowBuilder *Project*. This data structure, when saved to disk, is your WindowBuilder Project file. Window Builder project files have the extension “.wbp”.

The project file accurately maintains information about the source files, target system, images, strings and fonts, etc.. used by your application program. You can save your work at any time, and later re-open the project file and modify your target screens.

Project Path Information

All project file path information, such as the location of image files referenced by your project, is maintained in a relative path format. This means that you can easily copy your WindowBuilder project files from one computer to another as long as you also copy all related font and image files and maintain the same sub-directory structure for your project (if any) in all cases.

Optionally, if WindowBuilder does not find a required image or font file using the relative path information, WindowBuilder always attempts to find the file in the directory containing the WindowBuilder project. This makes it possible to “package” a project and the supporting image and font files in any common directory. WindowBuilder will find the image and font files even if the relative path information is incorrect if the file reside in the same directory as the project itself.

Source Output Files

The goal of WindowBuilder is to produce C++ source files, ready to compile and run on your target system. All of the layout, property settings, images, fonts, etc..

Window Builder

that you use while running WindowBuilder will at some point be exported in the form of C++ source files.

These source files contain standard C++ source code that, when compiled and linked with the PEG library, can be run on your target system. These source files may be one of three types: C++ class definitions, image file data structures, or string table data. Each C++ source file created by WindowBuilder has the extension `‘.cpp’`.

For most `.cpp` source files WindowBuilder also creates a corresponding header file. These header files contain class prototypes, message definitions, control IDs, string IDs, and other definitions required for your application software to compile and run.

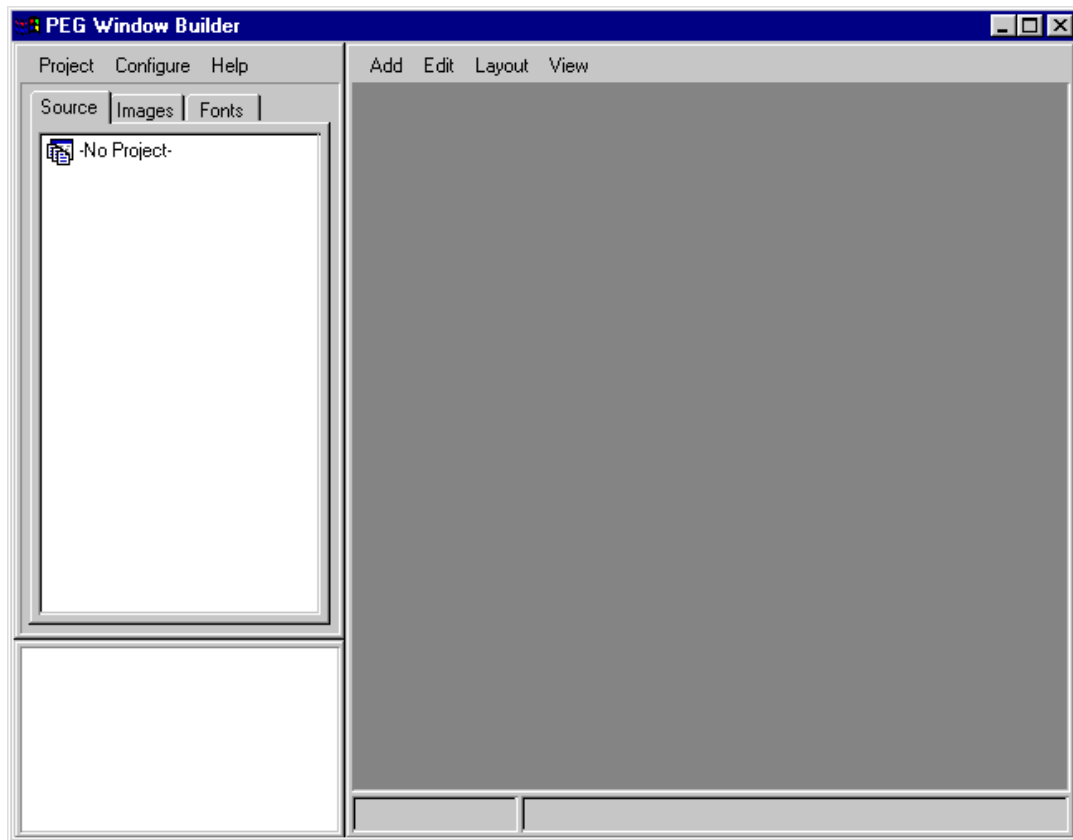
Each top-level module in your WindowBuilder project will generate one C++ source module and one corresponding header file. These files contain the PegWindow derived classes that constitute your application screens.

The source code created by WindowBuilder contains both the object initialization code required to create the window or dialog under construction, and the message handling functions required to process any signals enabled for individual controls. While it is your job to insert the actual signal handling code, WindowBuilder creates a framework for you complete with the individual signal case statements.

If you use BMP, GIF, or JPG images in your project (described in detail in a later section), WindowBuilder will also produce a single image file containing all images added to the project. This is a 'C' format source file containing data structures defining each of the `.bmp`, `.gif`, and `.jpg` images you have added to your project and will be using on your target system.

Screen Layout

When you run WindowBuilder for the first time, you will see screen shown below. This is the default appearance of the WindowBuilder application.



The WindowBuilder environment contains three main windows. These windows are the **Project** window, the **Preview** window, and the **Target** window. The **Project** window is where you can modify global configuration settings, maintain the source files included in the open project, and command WindowBuilder to

Window Builder

perform various operations affecting the entire application program. The **Preview** window displays images and fonts that you have added to your project, and allows you to drag those images and fonts to various PEG target objects. The **Target** window provides true WYSWYG emulation of the target system display screen.

Each of these three main WindowBuilder windows will be described in detail in the following sections.

The Project Window

The window builder project window is where all information global to your project is maintained. A project consists of any number of source files and associated classes, along with fonts, images, and strings used by your system. While it is possible to use multiple project files for a single system, this is discouraged since there is no way in this case for WindowBuilder to prevent the duplication of source code or data.

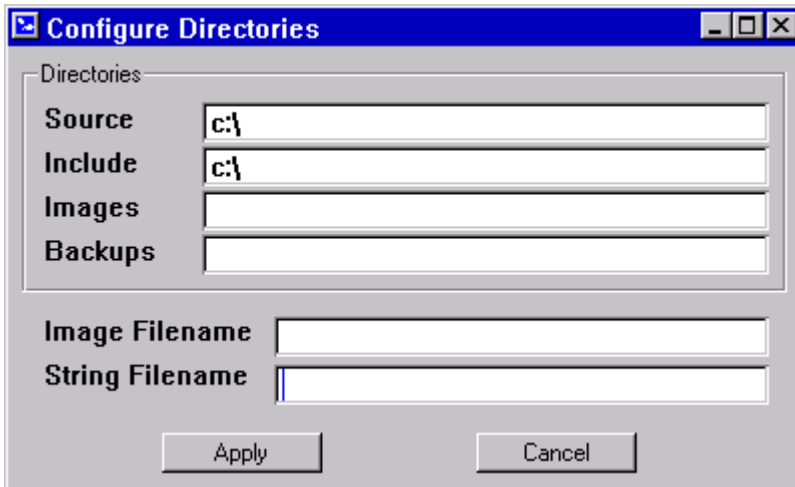
Internally, the Project window correlates directly to and continuously updates the WindowBuilder project file, which has the file extension '**.wbp**', which stands for **WindowBuilder Project**. The project file contains information about the source files included in your project, the class names and types contained within each source file, the fonts used by your project, etc.. Many operations (for example changing the target screen resolution), in addition to affecting the Target window, are also recorded in the '**.wbp**' project file. The project file is not used or included in your system software, but is used only by WindowBuilder. Still, your project file is so important that we recommend you make a backup or tagged version of your WindowBuilder project file every time you make a tagged version of your system software.

The Project window contains a command menu, along with a PegNotebook control. The PegNotebook tabs are titled *Source*, *Images*, and *Fonts*. We will begin by describing each of the project window menu commands, followed by a description of each of the notebook pages.

Project Window Menu Commands

Configure|Directories

This dialog causes the following dialog window to be displayed:



This dialog window allows you to specify the complete path for your source files, include files, and image files. The **Source Files** directory indicates where the source files generated by WindowBuilder will be saved. The **Header Files** directory indicates where the header files generated by WindowBuilder will be saved. This directory can be the same directory as the source files directory if desired.

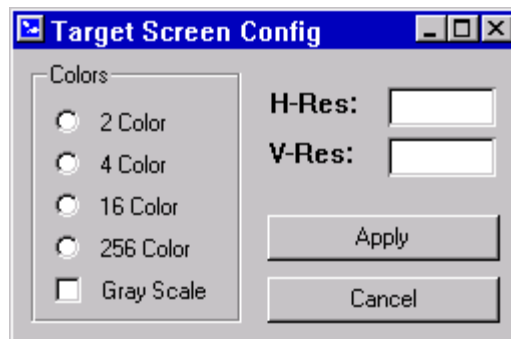
The **Image** directory indicates where the image file produced by WindowBuilder should be placed on your hard drive. The name of this image output file is specified in the **Image File Name** field. The BMP, GIF, and JPG images you incorporate in your project may reside at any location on your system or network, however for ease of transferring WindowBuilder project files it is usually best to place all of your project images into a common directory.

The **String Filename** field allows you to specify the name of the WindowBuilder output file that will contain your project string data. There are actually two output files for string information; one contains the literal string table, and the other is a header file containing the string IDs for each string. Both of these files will have the filename specified in the **String Filename** field, however the literal string table will have the extension .cpp and will be saved in the **Source** directory, while the associated header file will have the extension .hpp and will be saved in the **Include** directory.

The **Backup** directory indicates where WindowBuilder will save backup copies of files before updating them. Backups are created for all source, image, string, font, and project files. To disable file backups, set the Backup directory to a NULL string. It is NOT recommended that you disable file backups.

Configure|Target

This command invokes the following dialog window, which is used to control the appearance of the **Target** window:



The Horizontal Resolution field allows you to specify the horizontal resolution, in pixels, of the target screen.

Window Builder

The Vertical Resolution field allows you to specify the vertical resolution, in pixels, of the target screen.

The color-depth selections are used to configure the appearance of the **Target** window. You can select 2, 4, 16, or 256 colors. In addition, for 16-color screens you can specify either color or grayscale appearance.

Configure|Default Fonts

The command allows you to inform WindowBuilder of the default font settings you would like to use. These settings default to the standard settings provided in your PEG distribution. If you modify the default settings, WindowBuilder must be informed of this to prevent the generation of unnecessary ‘SetFont()’ function calls. If you are using custom fonts for some or all of the default fonts, you must first install these fonts before you can configure them using this command. Installing new fonts is described in the Add|Font command.

Configure|Languages

This command invokes a dialog window that allows you to specify the number of languages supported by the system software and the character encoding used for string storage. This in effect determines the layout of the String Table that will be generated by WindowBuilder, and determines how strings will be entered when new objects are created that display text or string data.

You will be prompted to enter a name for each supported language. These language names will be saved as an enumeration in the string data file. For example, you might configure your system to support three languages, “English”, “German”, and “Dutch”. The first language name is the default language at the start of your PEG application.

Your language names **MUST** be unique within the first seven characters, since these language names are used as a prefix for each string entry in the generated source file. If the language names are not unique within seven characters, you will receive “multiply defined symbol” errors when you compile your string file.

If your application does not require support for multiple languages, or if you for any reason do not wish to maintain your string data in the WindowBuilder string table, you can de-select the “Enable String Table” checkbox on the language configuration page.

Configure|Style

This command invokes a dialog window that allows you to enter several lines of ASCII text that will be included in the source file headers generated by WindowBuilder. You can also modify various style settings which control the format of the source code generated by WindowBuilder. This gives you some control over the appearance of the WindowBuilder source code, hopefully producing source code that is at least similar in style to your own hand-coded software.

Configure|Remote

This command allows you to display the contents of the target screen window in a secondary Microsoft Windows window. This feature is used when the target output device can be driven by a second video card installed in the PC on which WindowBuilder is running. Under certain Windows platforms, a second video card is utilized as an ‘extended desktop’. This capability allows you to drag the remote view window to the second video display, and view your screens on the target display as you create them.

Note that the remote view window is limited to drawing only that portion of the target screen that is visible in the target window. This means that you must resize

Window Builder

your WB window large enough to see the entire target screen in order to see the full window on the second display device.

Project|New

This command creates a new WindowBuilder project file. The project file will initially contain only the default image file, and will use the default Target window configuration.

Project|Open

This command is used to open a previously saved WindowBuilder project file.

Project|Open Recent

This command is used to open a recently edited project file.

Project|Save

This command is used to save the current project file. Note that the associated source files are not updated, rather only the binary project file is saved to disk.

Project|Close

This command is used to close the current project file and exit WindowBuilder. If the project file has been modified you will be prompted to save your changes before the project is closed.

Project|Add Module

This command adds a new source module to the current project. One or more source modules must be added to a new project before you will be able to edit anything using the target screen menu within WindowBuilder.

Project|Add Image

This command adds a new image to the current project. Once an image has been added, it can be applied to any PEG objects which support bitmap images. The source for the image must be a .bmp (Windows or OS2 Bitmap) .gif (CompuServe GIF), or .jpg (JFIF/JPEG) file.

The Images page of the notebook must be selected for this command to be active.

Project|Add Font

This command is used to add any number of custom fonts to your project. Once a font is added, it can be applied to any text-related PEG object simply by dragging the font from the preview window to the object that should be assigned the custom font. The input for adding a font should be a font file created with the PegFontCapture utility program.

The Fonts page of the notebook must be selected for this command to be active.

Project|Update|Source

This command instructs WindowBuilder to update the current selected source and include files to reflect changes in the current project. Before the source files are updated backup copies are saved to the Backup directory, unless this directory has been set to NULL.

Project|Update|Images

This command instructs WindowBuilder to re-process all input image files, and save the resulting PegBitmap information to the **Image File Name** in the source directory. Note that the output image file is completely regenerated each time an image update is performed. It is therefore not advisable to manually edit the output image file, since any changes will be overwritten by an image update.

Window Builder

Project|Update|Strings

This command instructs WindowBuilder to re-create the string data table and associated string ID header file. Note that this file is completely regenerated each time the Update Strings command is issued, and it is therefore not advisable to manually edit the generated string data files.

Project|String Table

This command brings up the string table edit window. This window allows you to define the literal strings and string IDs used for each language in the system. The string table is further described in a following section entitled **The String Table**.

Configure|Directories

This command invokes the directory configuration dialog window.

Working with Modules- The Source Page

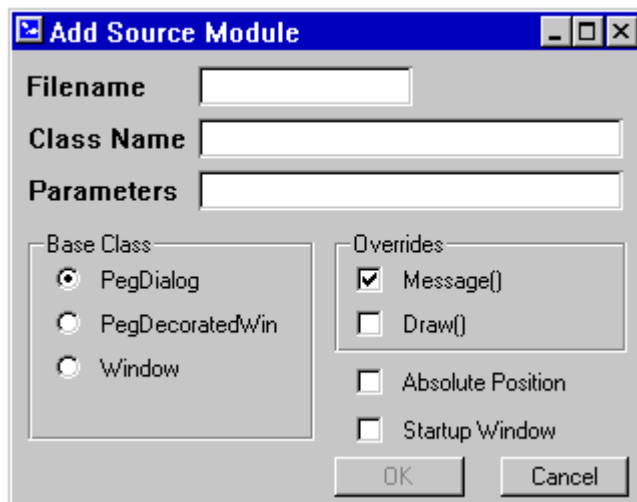
WindowBuilder organizes your application program, containing possibly hundreds of unique application windows, into unique modules. Each module corresponds to one window or dialog, and produces one source file and one include file.

The **Source** page of the Project window notebook control contains a PegTreeView depicting each of the modules included in the current project. Each module is listed in alphabetical order by class name. Each top-level node of the PegTreeView control represents a top level class constructed with WindowBuilder. If you expand a top-level node, you will see the corresponding source and header file associated with that top level class. If you select a source module by left clicking with the mouse, the Target window will display the PEG objects defined within that source module.

The Target Window always operates on the selected module. If no module is selected, none of the Target Window editing commands are operational. ***Therefore the very first thing you must do after creating a new project is add at least one***

module to your project. How to do this is described below, but it is important to remember that the right-hand side of the program window, that target window, is not operational until a module has been added to the project and selected.

You can at any time create a new module by selecting the Project|Add Module command. You will be presented with a dialog window shown below, prompting you to enter the required information, after which the new source module will be added to your project.



The ***Filename*** field allows you to specify the output filename for the source file WindowBuilder will generate for this new module. Any valid filename may be entered into this field. You do not need to specify an extension, as WindowBuilder will automatically write both a .cpp and a .hpp file for this module. .

Window Builder

The ***ClassName*** field allows you to specify a name for the PegWindow derived class you are creating, i.e. the name of the class which will define the window or screen you are developing.

The ***Parameters*** field allows you to specify any user-defined parameters you would like to pass to the class constructor (in addition to the parameters WindowBuilder will always pass to the constructor). If you desire to pass extra parameters, you should type them on this line exactly as they should appear in the constructor prototype, i.e. Type-Name, Type Name, etc.. for each parameter.

The Startup Window checkbox specifies that this window will be the first displayed when your application executes. When this checkbox is selected, WindowBuilder automatically writes the PegAppInitialize function in this module such that this window is created and added to PegPresentationManager during program startup.

The ***Base Class*** field allows you to specify which PEG library class will be the base class for your new Window. This will usually be PegDialog, but could also be PegWindow or PegDialog.

The ***Overrides*** group is used to tell WindowBuilder which function of the base class will be overridden by the class you are defining. Only two options are supported by WindowBuilder (although you can of course add your own function overrides to the completed class). These are the Message function and the Draw function, which you know by now as the most commonly overridden of all PEG member functions. The default setting of this field indicates that you will override the Message() function (to catch signals from child controls) but will not override the Draw() function. This is the most common situation.

The ***Absolute Position*** checkbox allows you to use an alternate form for the class definition. Normally, WindowBuilder produces a class that accepts a left-top corner position as the first two incoming parameters. The window and all child controls are positioned relative to this left-top position. If desired, you can produce a class

that is absolutely positioned, i.e. there is no left-top incoming parameters and the window are child controls use absolute pixel positioning.

When you have completed entering in the required information, a new object of the selected type is created and displayed in the **Target** window, and the new source file is added to the source page tree view control.

To remove a source module and its associated objects from your project, select the source module in the tree control and press the 'Delete' key on your keyboard. Following confirmation, the source module is removed from the current project. Note that the actual source files corresponding to the selected node are NOT deleted from your hard drive. WindowBuilder simply removes all information about the source file from the current project.

To modify the parameters associated with a source module after the module has been created, you can right-click with the mouse on the module in the Source notebook page. This will bring up a dialog window allowing you to change the module name, file name, and other module parameters.

Note that you are not allowed to delete sub-nodes from the PegTreeView displayed within the source page of the notebook. This is accomplished by individually deleting objects from within the Target window, described below.

Working with Images- The Images Page

The second tab of the Project window notebook is named "Images". The Images Notebook page lists the BMP, GIF, and JPG image files included in the current project. Similar to the Source page, this information is presented in a PegTreeView. Each top level node of the tree is one PegBitmap image that can be used in your application program.

Window Builder

Images are imported into your project by using the Project|Add Image command on the menu bar while viewing the Images notebook page. Images can be selected from any location on your local hard drive or network drive. WindowBuilder will maintain relative path information to the image if the image is located on the same drive as the project file, allowing you to easily move entire projects from one computer to another.

When an image is selected with the mouse or keyboard on this notebook page, a preview of the image is shown in the preview window. The image can be applied to a bitmap-based PEG control by dragging the image from the preview window to the target control.

You can display any image on the image page by selecting the image, or by using the up-down arrow keys to move up and down in the tree control. Each image is displayed in the Preview window as it is selected, allowing you to quickly scan through the images included in your project.

Images are deleted by selecting the image in the Image notebook page and pressing the 'Delete' key. Any objects that had been using a deleted image are re-configured to use a default bitmap.

When the Target window is being used to layout a new window or dialog, PegBitmap images from the Images page can be directly dragged-and-dropped onto PEG objects that support image display. For example, if you have created a PegBitmapButton as a child of the current window or dialog, you can simply drag a PegBitmap from the images preview onto the PegBitmapButton. This action causes the PegBitmap to be assigned to the PegBitmapButton. ***It is important to remember that before you can drag an image and drop it in the target window, you must add at least one child object to the target window which is capable of displaying an image.***

When you assign an image to certain types of objects by clicking on the image and dragging it to the object, Window builder will ask you if you would like to re-size the object to fit the image. If you select yes, the target object is re-sized such that the image fits neatly within the object. If you select NO, the image is centered within the client area of the target object and the target object size is not modified. For other object types the resizing is done without question since this is the only mode of operation supported by that object type.

The operation of WindowBuilder when generating the output image file can be quite complex, depending on your target screen color resolution. Take a deep breath before continuing!

If your target system supports 256 or more colors, WindowBuilder will scan each image in the project, create an optimal palette for displaying those images, remap the image colors back to the optimal palette, RLE encode each image, save the custom palette, and save each newly-encoded image file in 'C' style source data structures.

For 16 color targets, the Update Images command is slightly less complex. In this mode, window build dithers each image to a fixed orthogonal 16-color palette, RLE encodes the images for which this is memory efficient, and saves the resulting bitmaps in 'C' style source data structures.

For 2 and 4 color targets, the Update Images command simply saves each image in the selected format with optional dithering.

Working with Fonts- The Fonts Page

The Fonts page is very similar to the Images page. This page lists the fonts that have been added to the project, and allows you to drag-and-drop fonts onto specific objects.

Window Builder

When you first create a new project you will find two fonts listed on the Fonts page. These are the **System** font and **Menu** Font. These fonts are part of the PEG library, and are always available for you to use. Note that you are not allowed to delete these fonts from your project.

You can also add any number of additional fonts to your project and apply them to any text-display gadget that is part of your interface. You must pre-generate the fonts you will add to your project by using the FontCapture utility program. WindowBuilder is able to read the 'C' source files produced by FontCapture and create PegFont data structures in memory using these source input files. The result is that you see exactly the same appearance for your fonts as you will see on your target system.

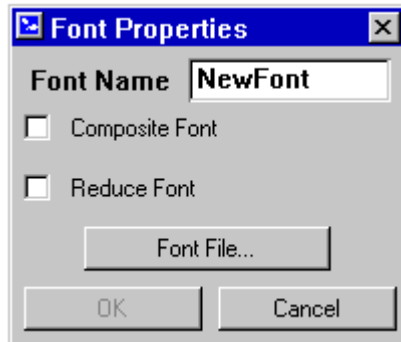
If you delete a font from your project that has been assigned to one or more text objects, you will receive a warning indicating the font is being used by one or more objects. If you delete the font anyway, the text objects that were using the font will revert to the default font based on object type.

The operation of adding a new font to your project varies depending on your language configuration settings (described in a later section). If the current project is configured to use only ASCII characters without the String Table, adding a new font is simply a matter of choosing a font file produced by the FontCapture program.

Composite Fonts and Reduced Fonts

If your project is configured to support multiple languages and Unicode, the operation of adding and defining a font becomes more complex. This is due to several factors, primarily the large number of characters (> 30,000) which may be required for the support of multiple languages. For projects of this language

configuration, the following dialog is displayed when you select the Project|Add Font command:



The **Font Name** field assumes the name of the source font for standard (i.e. non-Composite, non-Reduced) fonts. For Composite or Reduced fonts, WindowBuilder will produce a completely new PegFont from the input font data when the String Table file is generated. In this case, you can assign any name to the new font by typing into the Font Name field.

Composite Fonts are fonts collections, produced and managed by WindowBuilder, containing multiple sub-fonts produced by the FontCapture program. Why are composite fonts needed? To overcome limitations in the character set or alphabet included in most TrueType or BDF font files. In many cases you will find it is impossible to obtain one single TrueType or BDF font that contains all of the characters required by your application program. For example, one TrueType font may contain the Latin and Cyrillic characters, while another contains Kanji and Hangul. It is very rare to find a single font that contains characters for many different alphabets.

In order to avoid re-assigning the font associated with every PEG object when a language change is made, it is desirable to have a single font (or multiple fonts of different sizes, each containing the same character set) that contains all of the

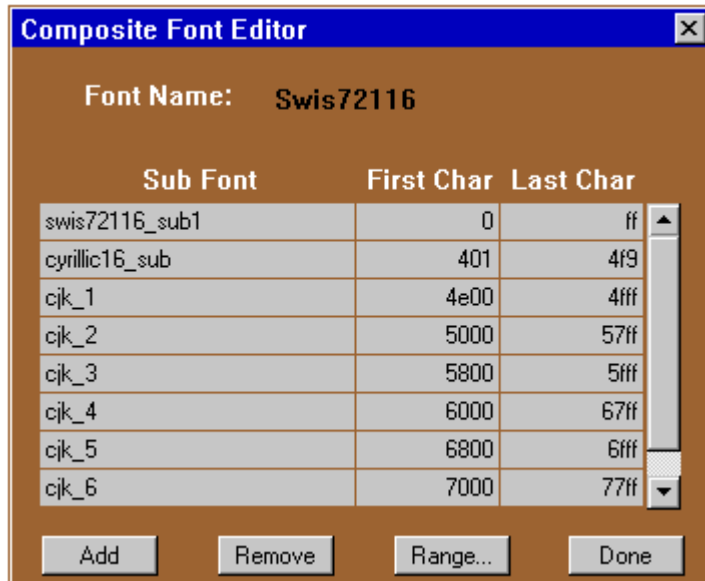
Window Builder

characters used by your application program. This is the reason for Composite fonts, you can combine any number of sub-fonts in to one “SuperFont” potentially containing all the characters from every sub-font.

The true power of Composite Fonts is realized when combined with the Reduce Font option. This option instructs WindowBuilder to produce a new PegFont wherein only those characters actually used by your application strings are included in the new PegFont. This information about which characters to include is obtained by examining all strings found in the project String Table (described below).

By using the Reduce Font option, you can save a tremendous amount of ROM storage for your fonts for languages with very large alphabets, such as Asian languages.

If you select the Composite Font option, you can then select the “Component Fonts” button to edit a table which defines each sub-font that will be include in the composite font. This table is shown here:



In the above example, several sub-fonts have been added to the composition font to yield one “Super Font”. The composition font contains characters from the Latin, Cyrillic, and several pages of CJK (Chinese-Japanese-Korean) alphabetic characters.

For each sub-font that you add to your composite font, the range of characters used from the sub-font will default to the full range of characters contained in the sub-font. Since it is possible that several sub-fonts may contain overlapping characters, you may need to edit the First Char/Last Char ranges displayed in this table so that each sub-font provides a non-overlapping range of characters to the final composite font.

When you have completed defining the sub-fonts that will make up your composite font, you simply close this table by pressing the Done button, and after naming your composite font click the OK button on the Font properties dialog.

Window Builder

You can return and re-edit your Composite font settings at any time by right-clicking on the font name in the Font tree display. Note, however, that while you can change the component font list for a Composite font, ***you cannot change a previously added non-Composite font into a Composite font, nor can you change a Composite font into a normal font.*** Instead, you must delete the font from your project and re-add the font using the desired settings.

For Unicode enabled systems using Reduced fonts, the Project|Update|Strings command does far more than simply write out your strings as C++ string arrays. For Unicode systems, WindowBuilder performs the following operations when the Project|Update|Strings command is issued:

- Scans string tables for all languages, creating global table of required glyphs.
- Re-scans all reduced fonts used by the application, saving only the required glyphs.
- Write new font structures containing only the required characters or glyphs.
- Create C++ wide-string arrays for each language supported.

The Target Window

The WindowBuilder Target window displays as accurately as possible a representation of the target system display screen. This representation is completely accurate in terms of pixel placement of graphical objects and colors used by each object. The Target window does not correct for differences in aspect ratio (i.e. pixel squareness) between your PC screen and your target screen.

When you create objects within the target window, you are actually defining new instances of PEG objects. These objects are dynamically constructed and added to the Target Window, and operate just as any normal PEG objects. This important to remember. As you create your windows and dialogs within WindowBuilder, you are creating a working PEG program. You can at any time interact with the objects you have created, just as the end user of your system software will interact with the final system.

This also causes some confusion at times when you first begin working with WindowBuilder. For example, consider the case where you have created a new modal dialog window, and you set the dialog system status to be non-moveable. Your intention, of course, is that the end-user will not be able to move the dialog window. However, you will not be able to reposition the dialog window from within WindowBuilder either! In order to move the dialog window, you will have to temporarily set the system status to be moveable, move the dialog to the desired position, and then reset the system status to the desired value.

As you work within the Target window, the internal copy of the WindowBuilder project file is updated to reflect all of your changes. The source code files for your project are NOT updated until the Project|Update|Source command is invoked.

The target window becomes active when a source module is selected in the project window. If no source files are included in your project, you must first create a new source module before you will be able to do editing in the target window. When you create a new source module, a default object of the type defined in the new

Window Builder

source module is generated and is the initial object displayed in the target window. After this step has been completed you will be able to use the Target window to modify and/or add children to the initially defined object.

Target Window Status Line

The Target window status bar works to aid you in positioning and sizing the objects you create and place on the Target screen. The first status bar field indicates the type of object that has been selected. The next field indicates the top-left pixel position of the selected object(s). The next field indicates the width and height of the selected object(s). The last field on the status line indicates the current pointer position when the pointer is over the Target window. This pointer position is relative to the top-left corner of the target screen, which is position 0,0.

Selecting Objects in the Target Window

Almost all selection and editing of objects in the Target window is done using the mouse. When you click on an object in the Target window, a dark border is drawn around the object to indicate that the object has been selected. You can re-size any object by dragging the dark border with the left mouse button held down until the desired size of obtained.

You can move an object by either dragging a selected object with the mouse, or by using the keyboard arrow keys.

Multiple objects can be selected by holding the <ctrl> key down while right-clicking on additional objects. When multiple objects are selected, the selection box expands to contain all selected objects.

Target Window Menu Commands

The target window menu commands always operate on the current selected object. You should select an object or group of objects before selecting one of the menu commands. The Target window menu commands are:

Add|Button

This selection brings up a sub-menu of common button objects. These include:

- TextButton
- MLTextButton
- BitmapButton
- DecoratedButton
- CheckBox
- RadioButton
- SpinButton
- Icon

Selecting any of these command adds an object of the selected type to the current selected object. In this case, the current selected object would usually be the top-level window or dialog, or possibly a PegGroup

Add|Text

This selection brings up a sub-menu of common text display objects. These include:

- Prompt
- VertPrompt

Window Builder

- String
- TextBox
- EditBox

Selecting any of these command adds an object of the selected type to the current selected object. In this case, the current selected object would usually be the top-level window or dialog, or possibly a PegGroup

Add|Indicator

This selection brings up a sub-menu of indicator style gadgets. These include:

- Finite Dial
- Bitmap Dial
- Light
- Bitmap Light
- Linear Scale
- Bitmap Scale

Selecting any of these command adds an object of the selected type to the current selected object. In this case, the current selected object would usually be the top-level window or dialog, or possibly a PegGroup

Add|Slide/Scroll

This selection brings up a sub-menu of slider/scroll bar objects. This list includes:

- Slider
- VScroll

- HScroll

Note that adding a Vertical Scroll or Horizontal Scroll using this menu command adds a *client area* scroll bar. This is a user-defined scroll bar rather than a scroll bar which acts to scroll the window client area. Normal non-client-area scroll bars are added by adjusting the window properties.

Add|Container

This selection brings up a sub-menu of container style controls, that is controls that are used to contain or group other child gadgets. These include:

- PegGroup
- ComboBox
- VertList
- HorzList

Add|Chart

This selection brings up a sub-menu of PegChart derived classes that can be added to the current object. These include:

- PegLineChart
- PegStripChart
- PegMultiLineChart

Add|Window

This selection brings up a sub-menu of PegWindow derived classes that can be added to the current object. These include:

- PegWindow

Window Builder

- PegNotebook
- PegTreeView
- PegTable
- PegSpreadSheet

Edit|Properties

This command invokes a properties dialog for the selected object. One and only one object must be selected in order for this menu command to be active. ***You can also invoke the edit properties dialog window by right-clicking with the mouse on the selected object.***

The properties dialog is context sensitive depending on the type of object which has been selected. In general you can adjust the border style, system status flags, and style flags for a given object by selecting each page of the properties dialog notebook control. Many object types have additional settings which can be controlled using the properties dialog.

The properties dialog is also where you specify the text string associated with many object types such as PegPrompt or PegString. For text-based control types, the properties dialog extended properties page includes a field labeled “Initial Text” that allows you to type in a string or, if the String Table is enabled, select the string ID associated with an object. This string ID is a member of the string table maintained by WindowBuilder. You can view and edit the string table by selecting the Project|String Table command in the project window.

If you have disabled the use of the WindowBuilder string table in the Project|Configure|Language dialog, the String page of the properties notebook allows you to directly enter the ASCII string used to initialize a control.

Edit|Copy

Copies the selected object or objects, including all status and style flags. Only one object can be selected when the Edit|Copy command is issued, however that object can have any number of children. When an object such as a PegGroup is copied, and the PegGroup has a number of children, the Group AND all of the group children are copied.

When this command is selected, WindowBuilder automatically changes the selection box to contain the parent of the current object. This allows you to quickly copy and paste an object into the objects parent, which is the most common operation.

Likewise, you can select an object, copy it, and then select an entirely different object to paste the copy into.

Edit|Paste

This command pastes an exact copy of the copied objects into the center of the selected object. WindowBuilder automatically selects the parent of the copied object as the target for the paste command. You can override this operation by selecting any other parent before selecting the paste command.

Window Builder

Layout|Align|Left

Layout|Align|Right

Layout|Align|H-Center

Layout|Align|Top

Layout|Align|Bottom

Layout|Align|V-Center

This group of commands is used to evenly align any number of child controls. Before activating this command any number of child controls should first be selected using the method described above. The above group of commands can then be used to exactly align the group of objects as desired.

Layout|Move To Front

This command adjusts the order in which child objects are added to the parent. The Move To Front command makes the object that last object added to its parent. This is useful for adjusting the tab-order of controls added to a parent window.

Layout|Move To Back

This command adjusts the order in which child objects are added to the parent. The Move To Back command makes the object that first object added to its parent. This is useful for adjusting the tab-order of controls added to a parent window.

View|Test Mode

This command places the target window in test mode. In test mode, all of the WindowBuilder windows are hidden, leaving only your newly created window or dialog on the screen. While in this mode, your new window or dialog will operate exactly as on the final target system, although any message processing code you have added to the window or dialog will not be operational from within WindowBuilder.

While in test mode, you will not be able to select and edit objects. You can exit edit mode by closing the window or dialog under test, or by pressing the “Stop” button placed in the lower right hand corner of the screen.

The String Table Editor

If you have enabled the use of string tables in the WindowBuilder Project|Configure|Language dialog, WindowBuilder will maintain a table containing all strings, for all languages, used by your application program. In this environment, all PegTextThing derived classes are constructed using StringID information, rather than literal strings. This allows your system software to easily convert between different supported languages.

The String Table is composed of an array of literal strings, a two dimensional array of string pointers, and an enumeration of string ID values. String ID values are just indexes selecting the correct row from the two dimensional table. Each table column is associated with one of the supported languages. If your application supports only one language, the String Table is simply a single list of literal strings, an array string pointers, and an associated String ID enumeration.

The correct string table column is selected by the current language, which is maintained in a static member variable of the PegTextThing class. This variable should be loaded with one of the enumerated language names when the active language is selected by calling the PegTextThing::SetLanguage() function. The default language is the first language configured in the Configure|Languages dialog box.

The string table is saved to the filename specified in the Configure|Directories dialog. The enumeration of the language names, and the string table IDs, are saved in the corresponding String Table header file. Each source file that uses the String Table must include the String Table header file in order to resolve the string IDs and language names. This include is added to each source file generated by WindowBuilder.

The String Table is edited by selecting the Project|String Table command on the Project window menu bar. This brings up the string table edit window, shown below:



The String Table

The left hand side of the String Table Editor window displays a PegSpreadSheet object containing each of the strings used in your system. Each row of the table corresponds to a StringID, and each column of the table corresponds to a supported language. The enumerated language names are displayed as the table column headers.

The String Table can be displayed in a two-column or three-column format. You can change the format by right-clicking over the spreadsheet and selecting the desired format in the pop-up menu.

You can also sort the string table entries by using the right-click pop-up menu. This menu provides command to shift the selected entry up or down in the string table.

Window Builder

The first language listed on your language configuration page is your project's "Reference Language". This language will be usually be English, but may be any language desired. The reference language is important because this is the language you are working in when you work in the target window. This is also the language which is always displayed when you view the table in three column mode.

String Edit Fields

The right-hand side of the String Table Editor window displays a series of fields for editing the selected string. The first field, the **ID** field, is where you can modify the string ID name, which is the name associated with each string ID. This name will be included in your string file as an enumerated list, and you will use this name in your application software when you want to refer to a particular string. You can edit this name simply by typing on the keyboard.

You can select the font to use while working in the string table using the drop-down list box labeled **Font**. Once you select the font to use in the String Table Editor, WindowBuilder remembers which font you have selected each time you call up the editor window. For example if you have created a composite font supporting all of your languages, you can specify that this font should be used in the String Table Editor. Each time you call up the String Table Editor, this is the font that will be used unless you select a different font.

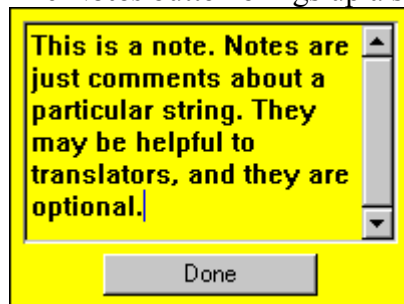
When you select a font to use in the String Table Editor, the font is displayed in the grid in the lower-right portion of the screen. This grid is more than a display, but actually allows you to select characters while editing the current string. This is required since for many languages you may not be able to simply type string values since the language alphabet contains characters that are not included on your keyboard.

The second field on the right side of the String Table Editor window is the string literal edit window. This field displays the string literal value using any of the fonts which are part of your project.

There are three methods for editing strings displayed in the string literal edit window: First, if the current language alphabet is supported by your keyboard, you can simply type the string value. Second, you can simply click on characters displayed in the font viewer window. As you click on the characters, they are inserted into the current string at the current insertion point. Finally, you can type the JIS or Unicode encoding value for the character you wish to insert. For some people who know the encodings for common characters, this is faster than finding the characters in the font display window.

As you edit the selected string, the width of the string (in pixels) is displayed in the **Width** field. This can be useful to insure that the string for every language will fit properly in the display area in which the string is used.

The **Notes** button brings up a small note editor window, shown here:



Notes are useful for including additional information about each string, usually for the benefit of translators who will translate your English or reference language strings into strings for the other languages.

Window Builder

As noted previously the first language you configure in the Configure|Languages dialog is described as your ***Reference Language***. The reference language is the language you will use while working in the Target window, and it is the language you will always view when the String Table Editor is in three-column mode. The reference language will usually be English, but may be any supported language.

The reference language is also important in the event that a translation is not required or available for certain strings in your application. For example, let's suppose that the string "California" is included in your application. Since this is a proper name, the name of a State, translation to another language is usually not required. Therefore, you would leave the string blank for every other language except your reference language, English. When WindowBuilder generates your StringTable file, any blank or NULL strings for secondary languages are automatically filled in with the reference language string pointer. In other words, for every other language in your application, the "California" string entry will be filled with a pointer to the reference English string "California", ***you do not need to duplicate this string for every language.***

Merging String Tables

The ***Merge...*** button on the String Table Editor window invokes a series of dialog that walk you through the merge process. In order to understand the reason for the merge operation, we need to examine the life-cycle of a typical multi-language project development.

Step 1) The system developers define the initial string table. The total number of languages and the language names are defined using the Configure|Languages dialog.

Step 2) The String ID names and the Reference Language (English) strings are initialized for all strings in the application using the String Table Editor.

Step 3) The Window Builder Project file, along with the WindowBuilder executable program, are distributed to translators who will each fill in one column of the string table. These translators may reside at the same location, but often reside all around the globe.

Step 4) The translators return WindowBuilder project files to you, and the returned project files each have one or more additional columns of the string table filled in with translated strings.

The problem should now be obvious: how do you get the translated strings from all of those different translators back into a common project file?? Enter the Merge operation. The Merge operation will merge strings for selected languages from a second project file into the current project file. The process is actually very simple as you are guided by step-by-step through the merge process. When WindowBuilder performs the merge, it looks for matching string ID names in the secondary project. For each matching string ID name, if the selected language in the secondary project has a non-NULL string value, that string value is copied into the current project for that specific string ID and language.

Exporting the String Table

The String Table data can be written out to a C++ source file at any time by selecting the Project|Update|Strings command. This command causes WindowBuilder to write a string file containing all of your String Id Names, your actual literal strings (Hex-Unicode encoded), an array of string pointers for each language, and a function and macro for finding unique strings at run time. The string file is completely re-written each time the Update|Strings command is issued, *therefore you should never manually edit the string file.*

Source Code Generation

The end goal of running WindowBuilder is to produce the C++ source code you will use to display your application screens. ***You will need to edit and add your own program logic to the source files produced by WindowBuilder.*** Most significantly you will need to add program logic to catch signals generated by your child controls. You may also need to make any number of other additions and changes to the source files produced by WindowBuilder.

At the same time, you will want to be able to run WindowBuilder again and again to modify your screens and update the source files without losing any of your hand-coded changes. This is not difficult to do as long as you understand how WindowBuilder updates your source files and follow a few simple rules.

When you instruct WindowBuilder to produce/update the source files using the Project|Update|Source command, WindowBuilder first looks to see if the source file already exists. If it does, WindowBuilder enters “**Merge Mode**”. In Merge Mode, WindowBuilder is very careful not to lose any of your custom modifications. The rules are this: WindowBuilder will find and re-write the section of the source file delimited by the start of the constructor and the comment line which reads:

```
/* WB End Construction */
```

To avoid losing your changes, never make any manual edits between the start of the class constructor and this comment delimiter.

WindowBuilder also searches for the Message() member function, if present, and updates this function to contain any new SIGNAL cases not already present. WindowBuilder will NOT remove case statements from your Message function, even if the control which generated a specific SIGNAL is no longer a child of the window. In short, deleting obsolete sections from your source files is your

responsibility, in the interest of safety WindowBuilder will not delete source lines from your Message function.

Any and all code outside of the class constructor and Message function is maintained without modification during the source code merge process. That is, any other editing that you have done will be preserved entirely during the source file update process.

Pointer Name Control

You can control the type and name of the pointer (if any) used when each child object of the top-level window is created. Controlling how pointers are used is done by adjusting the basic properties, using the properties dialog, for each child control. There are four types of pointers used by WindowBuilder during code generation: Member pointers, Automatic Named pointers, Automatic Temporary pointers, and Implicit pointers. We will describe each type below and describe how you can control the use of pointers in the generated source code.

Implicit Pointers

The most basic pointer type is the implicit pointer. An implicit pointer is used by WindowBuilder when no references to an object are made after the object has been created and you have not chosen to create a member or automatic pointer. In this case WindowBuilder does not need to keep the address of the newly created child in any variable, and therefore uses an in-line, “implicit” pointer to pass the child’s address to the Add() function. The following is an example of source code produced by WindowBuilder that uses an implicit pointer:

```
Add(new PegPrompt(ChildRect, “Text”));
```

Note that the return value from the new operator is not saved, but is passed directly to the Add() function. When no other pointer type is needed, this is the default pointer style used by WindowBuilder.

Window Builder

Temporary Pointers

Next up in the WindowBuilder source code generation process is the temporary pointer. This type of pointer is used by WindowBuilder when reference to an object is required after it has been created, but you have not requested an automatic or member pointer be created. In this case, WindowBuilder will create a temporary automatic pointer to hold the address of the child object instance. The temporary pointer is called “Automatic” because it is created on the execution stack, i.e. space for the pointer is allocated automatically by the compiler on the stack, and the space is destroyed when the function (in this case the class constructor) returns.

A common example of this might be a PegGroup container added to the top level window. During code generation, WindowBuilder needs to maintain the address of the PegGroup instance while creating and adding child controls to the group. WindowBuilder will default to using a temporary pointer for this purpose, which produces source code with the following appearance:

```
PegThing *pChild1;

pChild1 = new PegGroup(.....);    // keep temp pointer to object

pChild1->Add(....);                // add second-generation children to
object
pChild1->Add(....);                // ditto

Add(pChild1);                     // add object to top-level window
```

WindowBuilder will always use the generic names `pChildx` for temporary automatic pointers. WindowBuilder will reuse the temporary pointers for new objects if needed and available during code generation. In some cases, multiple temporary pointers are required simultaneously, in which case WindowBuilder will create and use many temporary object pointers as are needed.

Automatic Named Pointers

Similar to automatic temporary pointers, Automatic Named pointers are created on the execution stack and only exist during the class constructor. Named pointers are created by typing a name into the “Pointer Name” field in the object properties dialog basic properties page and unchecking the “Member Pointer” box. Note that the name must be a valid C++ variable name or your compiler will flag an error when you compile the generated module (no name checking is done by WindowBuilder!).

Automatic Named pointers are very handy to you, the developer, when you want to modify the object created by WindowBuilder in ways that are not supported by the WindowBuilder properties pages. An Automatic Named pointer is used only for the object in question, and more importantly is still valid and available to you at the end of the class constructor (i.e. after the “*/* WindowBuilder End Construction */*” marker). This allows you to further modify an object by calling class member functions via the named pointer prior to returning from the class constructor.

Named pointers also help to improve the syntax and eliminate casting when WindowBuilder must call member functions of a class. For example, if WindowBuilder must call the “SetFont” function for a PegPrompt, it will cast a temporary automatic pointer as follows:

```
((PegPrompt *) pChild1)->SetFont(.....);
```

If an automatic named pointer is used, it will be a pointer to the desired type and no casting is required:

```
PegPrompt *MyPrompt;

MyPrompt = new PegPrompt(....)
MyPrompt->SetFont(....);
```

Window Builder

This can greatly improve the appearance and readability of the constructor source code produced by WindowBuilder.

Member Pointers

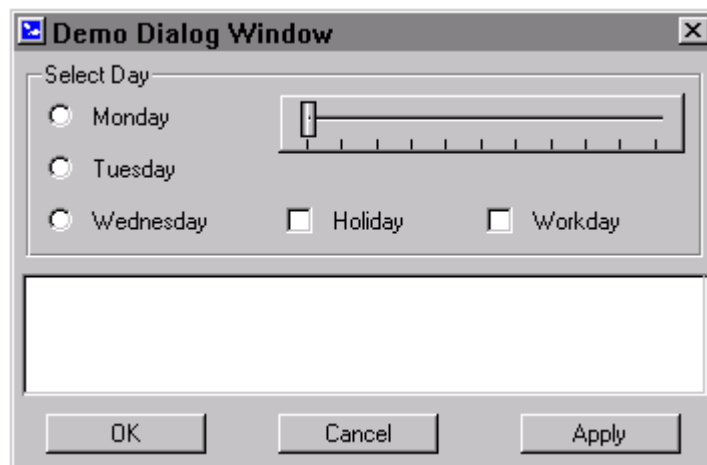
The final pointer option is the member pointer. A member pointer is a pointer to a child object which is maintained as a member variable of the parent window class. This pointer is initialized in the class constructor and used at all times to reference the child object. You can instruct WindowBuilder to create a member pointer for a child object by checking on the “Member Pointer” checkbox in the properties dialog and typing a name in the “Pointer Name” field. Note that the name must be a valid C++ variable name or your compiler will flag an error when you compile the generated module (no name checking is done by WindowBuilder!).

Example 1: Creating a simple PegDialog window

In this example we will walk step-by-step through the procedure required to create a new window builder project, create a new source module, and create a simple PegDialog derived window. This example takes about 15 minutes to complete.

The Example Dialog:

These instructions will take you systematically through the process of creating the simple dialog window shown below. In the following instructions, we will call this the ‘reference dialog’ :

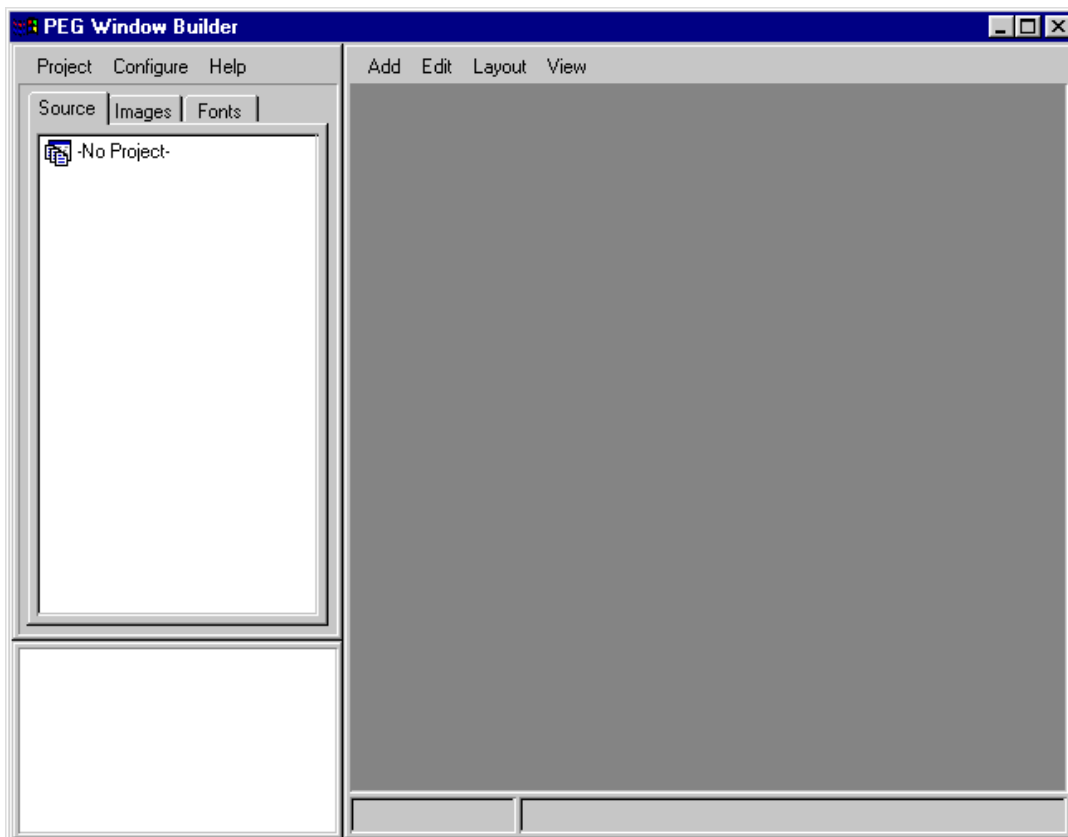


You may find it helpful to refer to the appearance of this dialog as you follow the instructions below.

Window Builder

Creating and Configuring a Project:

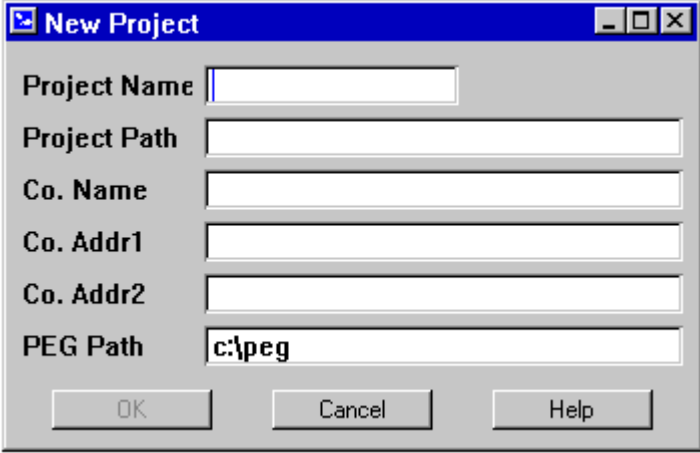
Under MS Windows 95/98/NT/2000, start the WindowBuilder executable program, **pwinbld.exe**. You should see the screen shown here:



This is the WindowBuilder startup screen. This screen allows you to quickly resume work on a previous project, or begin a new project. For this example, we want to begin a new project, so you should select the **New Project** button.

➤ Step 1- Configure Project and Directories

Whenever you begin a new project, WindowBuilder asks you to enter some basic project information such as the project name and where to keep the project file on your computer. WindowBuilder presents the following dialog window to allow you to enter this information:

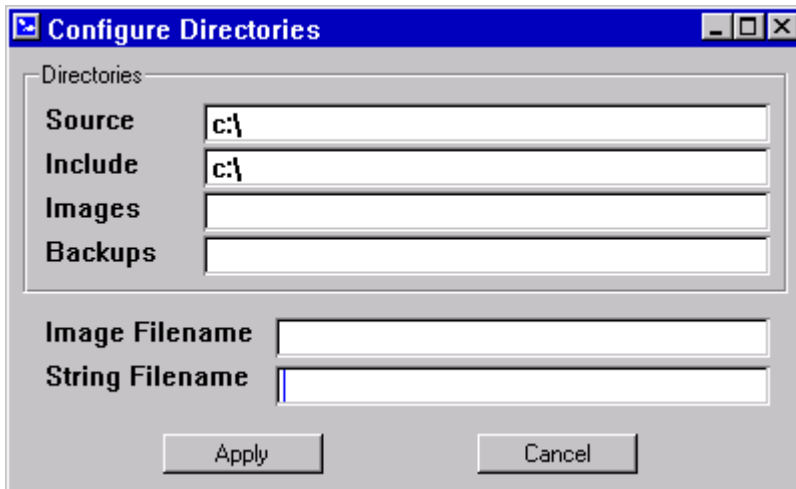
A screenshot of the 'New Project' dialog box in WindowBuilder. The dialog has a blue title bar with the text 'New Project' and standard window control buttons. It contains six text input fields: 'Project Name', 'Project Path', 'Co. Name', 'Co. Addr1', 'Co. Addr2', and 'PEG Path'. The 'PEG Path' field is pre-filled with the text 'c:\peg'. At the bottom of the dialog are three buttons: 'OK', 'Cancel', and 'Help'.

In the project name field of this dialog, type “**DemoProj**”. Your project file will be saved to this name. In the project path field, type any valid drive and directory name. If the directory does not exist, WindowBuilder will create it when you save your project.

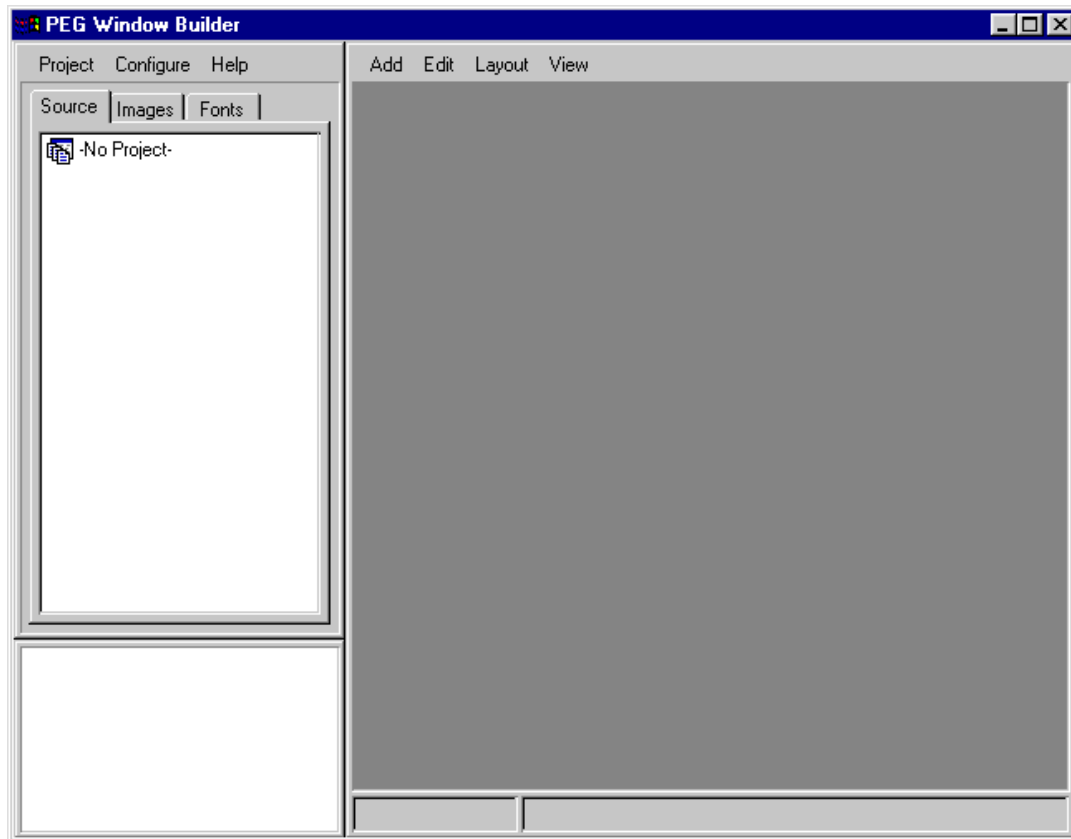
You can also enter your company name and address, although this is not required. If you do enter this information, WindowBuilder will include copyright notifications in the header area of the generated source files.

After you have entered in the required information, select the “OK” button. You are now presented with the following screen:

Window Builder



This dialog allows you to tell WindowBuilder where you would like to save the WindowBuilder output files. For this demo, you only need to enter in valid pathnames for the **Source** and **Include** directories. You can leave these at the default settings, or enter in alternate directory names where you would like to save the source code produced by WindowBuilder. It is acceptable to enter the same directory name for both source and include files. After you have entered the Source and Include directory names, select the “Apply” button. You will now see the screen below:



This is the default appearance of WindowBuilder. WindowBuilder initially contains three main windows. These are the **Project** window (upper left), the **Target** window (right) and the **Preview** window (lower left).

The project window maintains and displays information about all of the source files, images, and fonts that are part of your WindowBuilder project. The Target window provides an exact representation of your target system display screen. The Preview window allows you to view the images (i.e. bitmaps) and fonts you have added to your project.

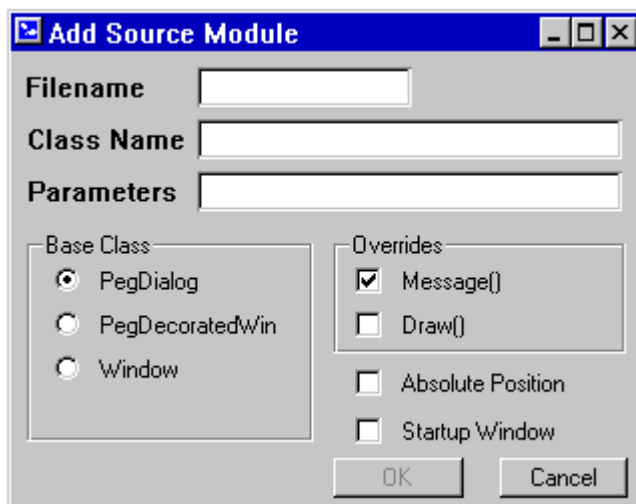
Window Builder

You should now see that the Project window displays the name of the project, i.e. “DemoProj” as the top node in the project source tree.

When a new project is created, the target window begins to display the target screen. The default target screen resolution is 640x480 pixels and uses 16 colors. This can be modified to any supported color depth and resolution using the Configure|Target command. For this example, you should leave the target screen configuration at the default settings.

➤ Step 2- Add a new Module

You are now ready to create a new module. Each WindowBuilder module contains a unique class declaration and class implementation. Select the **Project|Add Module** menu command. You will now see a dialog asking you to enter information about the new class to be created, shown here:



In the filename field, type “**DemoDlg**”. This is the name that will be assigned to the source and header files produced for this class. In the Class Name field, type “**DemoDialog**”. This name will be assigned to the generated class. In the Parameters field, type “**int iCount**”. Any information typed into the parameters field is passed directly to the dialog constructor. While this demo will not actually use the incoming parameter, it is useful to see how this affects the generated source code.

Click on the Startup Window checkbox to turn it on. This will cause WindowBuilder to generate a default PegAppInitialize function for us that will display our example window. Leave the remaining dialog fields at their default values and select “OK”.

You will now see that your project tree contains a new node, with the name DemoDialog. If you expand this node by clicking on the box containing the ‘+’ sign, you will see that the DemoDialog module produces the source file “DemoDlg.cpp” and the header file “DemoDlg.hpp”.

Click on the DemoDialog icon, and the target window now displays the default dialog window that has been created.

Editing the Module:

➤Step 3- Modifying position and size.

You can select the default dialog by left clicking with the mouse anywhere in the dialog window. When the dialog is selected, a dark box is drawn around the dialog window to indicate that it has been selected. You can now use the left-mouse button ‘click-and-drag’ operation to move the dialog window, and you can use the

Window Builder

arrow keys on your keyboard to move the dialog window to any position on the target screen.

If you position the mouse pointer over the dark border around the dialog, the mouse pointer will change shape to indicate that you can also re-size the dialog. You should experiment with moving and resizing the dialog until you are familiar with these operations.

Resize the dialog window under the Width field at the bottom of the screen is approximately equal to “354”, and the height is roughly “232”. You don’t have to be exact, but these are the approximate dimensions of the reference dialog window we are creating in this example.

➤Step 4- Modifying Properties

You can also set the position, size, and other properties of the dialog window by selecting the dialog and then right-clicking with the mouse over the dialog client area. This is the same as selecting the **EditProperties** command. When you do this, you will be presented with the Properties window that allows you to set the basic, extended, and color properties of dialog window.

The first properties page is called the “Basic” properties. These properties are always available, no matter what type of PEG object you are working with. In this case we can leave the Basic properties at the current settings.

Now select the “Extended” tab. This page of properties allows you to adjust parameters that are specific to the dialog window. In the Title field, type “**Demo Dialog Window**”. This assigns the dialog window title. Now select the OK button to apply your changes.

➤Step 5- Add a PegGroup to the Dialog

Make sure that the dialog window is selected, and then select the menu command **Add|Container|Group**. This will add a new PegGroup control to the dialog window. *The **Add** menu command always adds the selected object type to the previously selected parent.* In this case, the parent is the dialog window and the new object type is a PegGroup control.

PEG allows any type of object to be added to any other type of object, however WindowBuilder aids the user by only allowing certain types of objects to be added to other types of objects. For example, WindowBuilder will not allow you to add a PegWindow to a PegButton, although this is actually allowed when you are manually editing source code.

Use the mouse and arrow keys to size the group control so that it is similar in position and size to the reference example. Edit the group properties by right-clicking within the Group object, and on the Extended properties page enter “**Select Day**” in the Text field. This assigns the text value that is displayed as the group title. Select “OK” on the properties dialog and you will see your changes take effect.

➤ Step 6- Add Radio Buttons to the Group

Once you have the PegGroup in position, insure that you select it by left-clicking inside the group with the mouse. Now select the **Add|Control|RadioButton** command. Following this, a new PegRadioButton is added to the center of the PegGroup. This is the general operation of the Add command, in that the selected type of object is created with a default size and positioned at the center of the object’s parent area.

Use the arrow keys to move the radio button to the upper left corner of the group box, and then edit the radio button properties by right-clicking in the radio button client area. On the Basic properties page, enter “**IDRB_MONDAY**” in the ID field. On the extended properties page, enter “**Monday**” in the Text field, and when

Window Builder

you are done select “OK”. The ID value is the value you will use to identify the radio button during program operation. This value is saved in a list of enumerated control IDs in the generated class header file.

Repeat the above procedure to add the two additional radio buttons. Make sure you select the PegGroup parent object before adding each radio button, to insure that the radio buttons are children of the group object. For these buttons, assign the first the ID value “**IDRB_TUESDAY**” and the Text “**Tuesday**”. For the last radio button, assign the ID value “**IDRB_WEDNESDAY**” and the text “**Wednesday**”.

You can use the mouse and arrow keys to position the radio buttons in the approximate order and position you want them to be in. You don’t have to be exact, we will use the Layout commands to insure that the radio buttons are perfectly aligned.

➤Step 7- Using Layout commands.

To insure that the radio buttons are equally aligned, we can use the Layout commands. The layout commands effect collections or groups of objects. In this case, we want to select all three radio buttons before using layout command.

To select the three radio buttons, first select the top radio button with the text value Monday by left-clicking on that radio button. Now hold down the <ctrl> key and left click on the “Tuesday” and “Wednesday” radio buttons in turn. You will see that the selection box grows to enclose all three radio buttons.

Now we want to use the **Layout|Align Left** command to align the left edge of the radio buttons. After selecting this command, you should see that the radio button are all exactly aligned at the left border.

Note that while you have multiple-objects selected, you can use the mouse and arrow keys to move all of the objects as a group. Use the arrow keys now to slide the three radio buttons into a position that “looks right”.

➤ **Step 8- Add remaining children to Group.**

You can add the two checkbox objects to the group by first selecting the Group box, and then selecting the **Add|Control|CheckBox** command. Position the checkboxes using the same methods described above. Assign the first check box the ID “IDCB_HOLIDAY” and the Text “Holiday”. Assign the second check box the ID “IDCB_WORKDAY” and the Text “Workday”.

Again select the group box, and select the **Add|Control|Slider** command. This adds a PegSlider control to the group box. Use the mouse and arrow keys to position and size the slider control as shown in the reference diagram. You do not need to assign any additional properties to the PegSlider control.

➤ **Step 9- Add PegTextBox**

Click on an unused portion of the dialog window to select the window. Be sure that the dark border encloses the entire dialog window. A common mistake is to click inside of the group box, in which case the group box is selected rather than the dialog window. Now select the **Add|Control|TextBox** command. A default size textbox is positioned at the center of the dialog window. You will need to reduce the height of the textbox using the mouse or properties dialog, and move the text box so that it is underneath the group. You can also use the properties dialog to enter an initial text value, such as “**Hello World**”.

➤ **Step 10- Add TextButtons**

Repeating the above procedures, click on an unused portion of the dialog window, and then select the **Add|Control|TextButton** command to add a new button to the

Window Builder

dialog window. The button will again appear at the center of the dialog, and you will need to use the mouse or arrow keys to move the button into position.

Create the three buttons at the bottom of the dialog one at a time, repeating the above process. Use the edit properties command to assign the Text values “OK”, “Cancel”, and “Apply” to each button. Likewise, assign the ID values “IDB_OK”, “IDB_CANCEL”, and “IDB_APPLY” to each button, respectively.

You can use the Layout|Align Top command to insure that the buttons are vertically aligned, and move them as a group until they are centered on the dialog window.

You are now done creating the dialog window!!

Saving Your Work:

➤Step 11- Save the project

At this point you should select the Project|Save command to save your project. This will create the file “DemoDlg.wbp” in the directory you selected in the Configure Directories dialog. Once you have saved your project, you can later open it at any time and modify this dialog or add any number of additional modules to the project.

➤Step 12- Generate Source Code

Make sure that the module “DemoDialog” is selected in the project tree (it should be highlighted). If it is not, left-click with the mouse in the project source tree or use the arrow keys to select the DemoDialog module. Now select the **Project|Update|Source** command to create the C++ source files corresponding to this module. After the source code has been generated, WindowBuilder should inform you that the source or header file has been updated.

➤Step 13- Close WindowBuilder

Select the **Project|Exit** command to exit the WindowBuilder application.

Examining the Source Code:

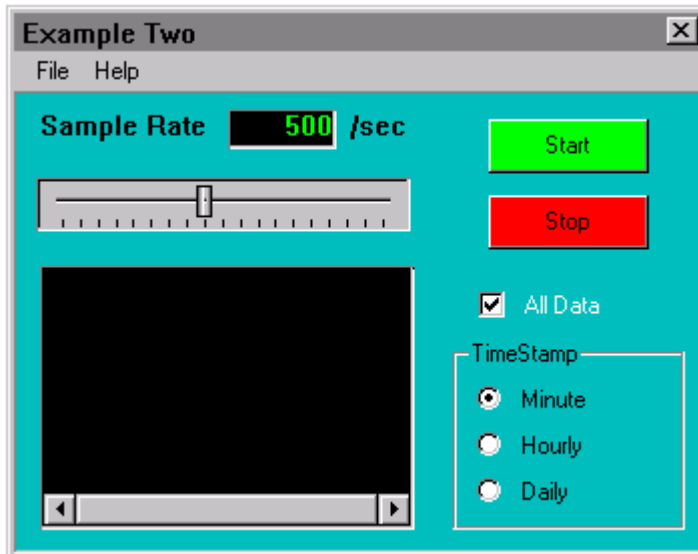
You can now open the source and header files produced by WindowBuilder with your favorite text editor. The generated .cpp file contains the exact C++ source code required to construct the example dialog window at run time, and the generated .hpp header file contains the class declaration and ID enumeration necessary to complete the class description. The source code file also contains a skeleton of the message processing function that you will edit to make your dialog do real work.

Example 2: An advanced PegDialog window

For this example, we will instruct WindowBuilder to create a new PegDecoratedWindow derived window class, and to generate an overridden message handling function. We will also use the ‘Maintain Pointer’ and ‘Send Signals’ options to make the dialog do useful work. When you are done creating the window, you can actually compile the source code and run the application! Be forewarned, this example takes about 60 minutes to complete.

The final appearance of the new window is shown below:

Window Builder



In order to make the example more interesting, assume that this window will display a continuous time chart of some analog data value. For this reason, we have provided the user with a PegSlider control to adjust the 'Sample Rate', Start and Stop buttons to start and stop the sampling process, and a range of timestamp values that indicate how often the recorded data will be saved to a permanent storage media.

We will also add a menu bar to the window. While this is not necessary for this example, it gives us a good chance to demonstrate how you can create and customize menus with WindowBuilder.

Step 1: Create a Project

Start WindowBuilder and create a new project as in example 1. Name this project 'Example2'. Use the Configure|Directories command to set both the source and include output directories to ...\\peg\\wb\\example2. This will enable you to build the resulting application using the provided project file.

Step 2: Add a Module

Create a new module using the Project|Add Module command. Enter **exam2** as the file name, and enter **ExampleTwo** as the class name. It is normally up to you to decide on the file and class names, however we have provided a project file and a version of the PegAppInitialize function that will allow you to build and execute the window after you are done, so in this case it is best to stick with the suggested names.

Before you click on the “OK” button, select the “PegDecoratedWin” button in the Base Class group. This means that your new window will be derived from PegDecoratedWindow, rather than the default PegDialog. Leave the other selections at their default settings.

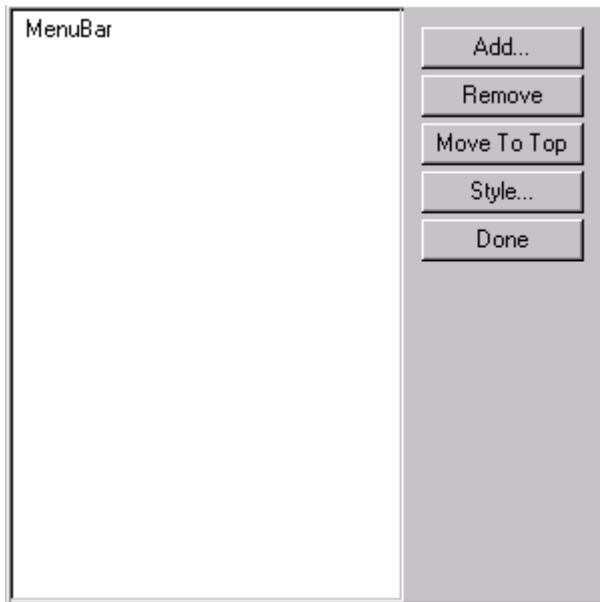
Step 3: Customize the Window

Use the properties dialog to modify the appearance of the window until it resembles the reference example. You will need to select the ‘Color’ tab, and set the PCI_NORMAL color. You will also need to select the Title Style button on the Extended properties page, and turn off the “System Button” and “Min/Max Button”. You should also type “Example Two” in the window title field. On the basic properties page, set the window Height to 280 pixels, and the window Width to 350 pixels. Once you have done this, close the properties dialog by selecting the “OK” button. You should now see that the window has the indicated fill color, and the size and title should match the reference example.

Step 4: Add the Menu Bar

Again enter the window properties dialog. On the Extended properties page, select the “Menu Bar” checkbox. This adds a blank menu bar to the window. Select the “Edit” button next to the Menu Bar check box, and you will see the dialog window below:

Window Builder



This is the WindowBuilder menu editor. The left-hand panel is a PegTreeView control, which will show you all of your menu commands in a convenient tree format. The right-hand panel contains several buttons that you will use to edit the menu bar.

For this example, we are going to create two top-level menu options, called “File” and “Help”. Click on the “MenuBar” node of the tree (something always has to be selected before the menu editing buttons are operational), and then select the “Add” button. You are presented with a small window that allows you to enter the menu “Caption” (the displayed menu text), the menu item ID, and the menu item style.

Type “File” in the Caption field, leave the remaining fields at their default value, and select “OK”. You now see the file menu item displayed in the tree!

Ensure that the “MenuBar” tree node is still selected, and repeat the above procedure to create the “Help” item on the menu bar.

Now we can create any number of submenus. Select the “File” node in the tree, and add the following items:

Caption	ID	Style
Open	IDB_OPENFILE	AF_ENABLED
Close	IDB_CLOSEFILE	AF_ENABLED
		BF_SEPERATOR
Exit	IDB_CLOSE	AF_ENABLED

The style settings are exactly the same as setting the different Style flags when defining a PegMenuDescription. In fact, WindowBuilder will use the information you enter to generate the required PegMenuDescription in source code format for you.

For this example we will not be using the MenuBar commands, so feel free to experiment with the Menu Editor and modify the MenuBar settings. Repeat the above procedures to add “About”, “Search”, and “View” commands to the top level “Help” button.

When you are done, select the “Done” button, and close the properties dialog by selecting “OK”. Your menu has now been added to the decorated window. You won’t be able to actually view the sub-menus from within WindowBuilder, since selecting any non-client window object is intercepted by WindowBuilder to select the parent window. However, you can re-open the properties dialog and edit the menu at any time.

Step 5: Add the Child Controls

We are not going to detail each of the child controls, as by this time you are probably becoming quite adept at adding new controls using the WindowBuilder menu commands. There are, however, a few things that we should point out to help you create a working window that looks like the reference example.

The black window in the lower-left corner is a `PegWindow`, added with `Add|Window|Window`. We have modified the `Frame` style to be `'Recessed'`, changed the `PCI_NORMAL` color to black, and set the `H-Scroll` mode to `"Always"`, which means the window will always display a horizontal scroll bar.

The fields `"Sample Rate"` and `"/sec"` are `PegPrompt` objects.

The field that displays the Sample Rate, and shows the value `"500"`, is also a `PegPrompt` object. This object is important for this example so we will detail its configuration more completely.

In order to obtain the appearance shown, you must first disable the `"Transparent"` style in the prompt Extended properties. When the `Transparent` style is enabled, the prompt always assumes the background color of its parent, so setting the prompt color has no effect. We have also set the `PCI_NORMAL` color to black, and the `PCI_NTEXT` color to green. Finally, the prompt `Frame` is set to `"Recessed"`, and the justification (also on the extended properties page) is set to `"Right"`. Also, select the `"Copy Text"` checkbox on the extended properties page. This instructs the `PegPrompt` object to make a copy of the text it displays, which is required when the text value assigned to the prompt is created dynamically on the stack. We will explain this further later in this example.

Set the prompt ID to `IDP_RATE`, and set the prompt Text to `"500"`. Click on the `"Maintain Pointer"` checkbox in the prompt Basic properties page, and type in the pointer name `"mpRatePrompt"`. This instructs WindowBuilder to define and use a pointer to this `PegPrompt` object that will become a member variable of the parent

window. We will use this pointer to modify the prompt value as the slider control is adjusted. While we could also find a pointer to the prompt at run-time using the Find() function, in this case it is more convenient to simply tell WindowBuilder to create and keep a pointer to the prompt.

The slider control is created with a Recessed frame, an Minimum value of “100”, a maximum value of “1000”, and an initial value of “500”. Set the Tick Interval to 50, which means that a tick mark will be drawn at each increment of 50 in the slider range. Use the slider properties dialog to enter these values. Also, insure that the “Send Signals” checkbox is selected on the slider control Basic properties page. This tells WindowBuilder that we want to receive notification messages from the slider control (in this case PSF_SLIDER_CHANGE signals), and WindowBuilder will create the matching case statements in the window message handling function. Set the slider ID to “IDSL_RATE”.

The Start, Stop, and other remaining child controls are not required for this example. You do not need to assign ID values or select the “Send Signals” checkbox for any of these controls. Simply create them and place them on the window to achieve the appearance shown above.

Step 6: Save Your Work!

Use the Project|Save command to save your project file. You don’t want to lose all of your effort so far do to a power failure or other malady!

Step 7: Generate Source Code

Now select the Project|Update|Source command to generate the source code for your window.

Window Builder

Step 8: Add Message Handling

Open the generated source file in your favorite editor. You will see that WindowBuilder has generated a constructor for the ExampleTwo window, and also the message handling function. Your message handling function should appear very similar to the function below:

```
SIGNED ExampleTwo::Message(const PegMessage &Mesg)
{
    switch (Mesg.wType)
    {
        case SIGNAL(IDB_HELPINDEX, PSF_CHECK_ON):
            // Enter your code here:
            break;

        case SIGNAL(IDB_HELPINDEX, PSF_CHECK_OFF):
            // Enter your code here:
            break;

        case SIGNAL(IDB_ABOUT, PSF_CLICKED):
            // Enter your code here:
            break;

        case SIGNAL(IDB_CLOSE, PSF_CLICKED):
            // Enter your code here:
            break;

        case SIGNAL(IDB_FILESAVE, PSF_CLICKED):
            // Enter your code here:
            break;

        case SIGNAL(IDB_OPENFILE, PSF_CLICKED):
            // Enter your code here:
            break;

        case SIGNAL(IDSL_RATE, PSF_SLIDER_CHANGE):
            // Enter your code here:
            break;

        default:
            return PegDecoratedWindow::Message(Mesg);
    }
    return 0;
}
```

Window Builder

Note that message case statements have been generated for all of the MenuBar commands, and also for the window child controls for which the “Send Signals” option was selected. In this case, verify that you have the case statement “case SIGNAL(IDSL_RATE, PSF_SLIDER_CHANGE):” in your message function. We are going to add the source code here required to make the prompt object display the current slider value. If you do not see this case statement, you should check the properties of the slider control with WindowBuilder, and insure that the “Send Signals” checkbox is checked.

Modify the case statement as shown below:

```
Case SIGNAL(IDSL_SLIDER, PSF_SLIDER_CHANGE):
{
    char cTemp[40];
    itoa(Mesg.lData, cTemp, 10);
    mpRatePrompt->DataSet(cTemp);
    mpRatePrompt->Draw();
}
break;
```

The above code will convert the slider value, which is passed in the message lData field, into an ASCII string for display. It then assigns this string value to the prompt, and redisplay the prompt. Note that the pointer *mpRatePrompt* has been created for us by WindowBuilder, and is defined in the exam2 header file as a private member variable of the Example2 class.

A less obvious note should be made. If you remember we instructed you to select the “Copy Text” checkbox in the range prompt Extended properties page. Normally, PEG objects do not copy text strings in order to reduce memory usage. However, in this case we are creating a new string on the execution stack, via the automatic “cTemp” variable. In this case, we want the prompt object to copy the assigned text, since the cTemp variable will not exist once we return from the Message function.

Step 9: Build and run the program

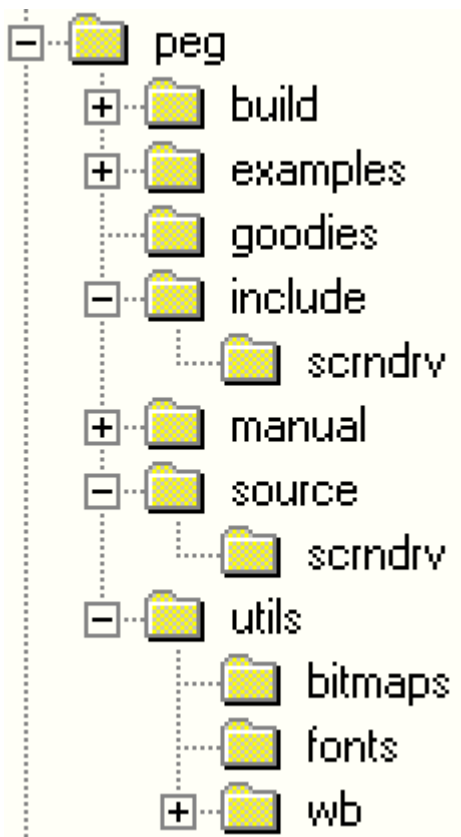
We have provided a version of the startup function `PegAppInitialize` that creates and displays the `ExampleTwo` window in the `\peg\wb\example2` directory. This file is called `startup.cpp`. In order to build and run the program, you will need to do the following (using MS VC++)

- a) Create a new workspace.
- b) Add the project file `\peg\libs\win32\peg.dsp` to your workspace. The project builds the PEG library.
- c) Add a new project to the same workspace. This project will build the example application. You can name the project anything you like. After you have added the project, add the files “`startup.cpp`” and “`exam2.cpp`” to your project.
- d) Make sure that your project dependencies list “`peg.lib`”, as this will build and link the PEG library with your application files.
- e) Build and run the program!

As the program executes, you should be able to adjust the slider control and see the value displayed in the prompt object. You have used WindowBuilder and a little hand coding to create a complex window! We hope you have enjoyed working through this example program.

Appendix A: PEG Directory Structure

This section provides an overview of the directories created during PEG installation and the contents of each directory. After installing PEG, you will find the following directory tree, starting at the root \peg node:



Appendix C: PEG Directory Structure

build- Various make files for building the PEG library using different compilers for different targets. Microsoft, Borland, and several other compiler-specific build files are provided in this directory.

examples- Each folder under this directory contains a complete working example program. Microsoft developer studio project files are also provided for quickly building each of the example programs.

goodies- Miscellaneous gadgets used in the example programs. Not part of the core PEG library.

include- PEG library header files.

include\scrndrv- Target specific PegScreen implementations.

manual- Online documentation, starting with index.htm

source- Source files for the core PEG library.

source\scrndrv- Target specific PegScreen implementations.

utils\bitmaps- This directory contains the PegImageConvert executable program.

utils\fonts- This directory contains the PegFontCapture executable program.

utils\wb- This directory contains the WindowBuilder executable program and examples.