Portable Embedded GUI

PEG Pro/PEG+ Programming Manual

PEG Pro/PEG+ Library Release 2.x

Rev. 3

May 2011

FORWARD

We at Swell Software thank you for choosing PEG!

PEG is by far the most used, best supported, and most adaptable graphical interface software available. Our industry-leading real-time operating system support, hardware integration, and develpment tool compatibility allow complete flexibility as you and your team create next-generation products.

The PEG library and development tools have today been used to create several thousand unique products, and those products have shipped many hundreds of millions of units. The applications utilizing PEG software cover a broad spectrum including various consumer electronics, medical instrumentation, video games, military communications, aeronautics, office equipment, and even desktop applications.

We hope that your own efforts will be equally successful, and we encourage you to use our technical support if you run into any speed-bumps along the way.

In addition to the PEG software package, Swell Software provides consulting and contract programming services to clients in a wide variety 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.

How the manuals are organized

Your documentation set includes four separate manuals:

1) The QuickStart Guide
2) The Programming Manual
3) The WindowBuilder User's Manual
4) This API Reference Manual

The Quickstart Guide is a short tutorial enabling you to easily begin working with WindowBuilder and to create and run your own application in one of our supported desktop environments.

This programming manual provides an 'under the hood' view of the PEG software library internals and introduces basic concepts that are needed to fully understand how PEG works. This is followed by descriptions of the fundamental PEG classes.

The WindowBuilder User's Manual is a guide to the operation of our WindowBuilder WYSIWYG development tool and resource manager.

The API Reference Manual provides extensive information about the fundamental PEG classes. This manual details the Application Programming Interface (API) of the PEG graphics library.

Each of these manuals are provided in both printed and electronic (PDF) formats. The PDF format manuals are always the most recent manual updates, while for practical reason the printed manuals can sometimes be a few months out of date.

Whenever the electronic manuals are updated, they are posted to the Swell Software website. The online manuals can be found at the following address:

http://www.swellsoftware.com/download/documentation.php.

A username and password are required to download the manuals.

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 software.

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 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 primarily at custom embedded systems.*** This distinction is so important that we felt it should be included in the name of our library.

PEG is GUI

The PEG class library provides the building blocks for a powerful and extensible graphical user interface. Extensive thought and research have gone into the design of our product to ensure 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.

In addition to the class library, PEG provides tools for generating graphical fonts, processing, optimizing, and compressing graphical images, designing screens and child controls, creating custom colors, and maintaining multi-lingual string data.

What PEG is NOT

PEG is not an operating system. While PEG can run completely standalone, the library does not provide software for system boot-up, task switching, file system maintenance, or any of the other operating-system level functions your product may require.

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 of using PEG is that our library provides the tools and components that make creating your application level interface a manageable task.

Library Updates

Library updates are posted on the Swell Software website roughly every 90 days. If you are a new PEG customer, you are entitled to a minimum of three months of technical support and library updates. The Download/ Updates 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 http://www.swellsoftware.com/download/updates.php page lists the most recent changes or library enhancements, and also allows you to download the latest release of PEG library source code, supporting utilities, and documentation.

CHAPTER 1

INSTALLING AND BUILDING THE PEG LIBRARY

1.1 Installing PEG

PEG is installed by running the PEG Installer. Your installation key, which is unique to your distribution, is a hexadecimal number encoded to unlock features specific to your distribution. You must enter your installation key each time you run the PEG installer. Installation keys are obtained from the Swell Software sales department, sales@swellsoftware.com.

The PEG installer installs the PEG software library (in source and/or binary formats, depending on your install key), the PEG manuals, example programs, and the WindowBuilder executable program.

PEG can be installed to any destination drive and directory. During the installation process, the PEG installer records the selected installation path for reference later when generating the PEG library configuration header.

1.2 Building the PEG library

The PEG software is comprised of a large number of C++ source files. You can compile and link these source files as part of your monolithic application build process; however, most commonly PEG users compile the PEG source files and archive the resulting object code into an object library, similar to the C run time library or other third-party software libraries. This library is then linked with your application software to produce the final binary image.

Building the library is very simple, although you may need to create a make file or project 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**.

The library is configured via inclusion of the header file ‘peg\include\ pconfig.hpp.’ This header file is automatically generated by WindowBuilder by making your configuration selections; therefore, ***you should not manually edit the pconfig.hpp header file***.

If you have just installed the PEG software, the pconfig.hpp header file is configured to build the PEG library for either the Win32 or X11 desktop environment. You can reconfigure the library at any time to build for your intended target.

This chapter describes each of the library source files and their contents. Several of the source files are optional and only included if required for your target system.

The last section of this chapter describes the preconfigured library make files that are included in your PEG+ or PEG Pro 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 200K of ROM-able code when all options and modules are included. The size of the generated code varies depending on the compiler, optimization levels, and CPU being used. The total code size of the 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 or 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.

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 ensure that all PEG source files and documentation files are named in a consistent manner to avoid problems encountered when moving between Windows, Linux/X11, or other development environments. All source file names use lowercase letters exclusively to avoid case differences when running on Unix or Linux systems.

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

1.3 Build Options

The library is shipped configured for Win32 or Linux/X11 desktop development. This allows you to quickly build the PEG library and begin using the PEG API without concern for hardware porting issues. We recommend that if you are new to PEG, you begin your development in one of these desktop environments. Rest assured that your work will be 100% portable to your final target.

There are many build options and configuration flags, allowing you to precisely tune the PEG library to your requirements. The designers of PEG intentionally structured the library such that many decisions are made at 'compile time' rather than at 'runtime' to improve overall system performance. Configuration flags are also used to tune the library ROM footprint by excluding features that are not required for your application.

To reconfigure the PEG library, you need to run the WindowBuilder program, modify the configuration settings, generate a new pconfig.hpp header file, and recompile each of the PEG source files to generate a new PEG library archive. Refer to the WindowBuilder User's Manual for a complete description of each build option and instructions on generating a new pconfig.hpp header file.

Remember that whenever you change your configuration settings, you need to generate a new pconfig.hpp header file and do a 'clean build' of the PEG library to ensure that your configuration changes have taken effect.

1.4 Library Source Files

This section describes each of the PEG library source files.

The PEG software is distributed with all of the source and header files required to build the library. You only need to include in your make/project file the source files for classes you will be using. Many files can be excluded from your library depending on your system requirements.

While it is possible to simply include the PEG source files in with your application modules as part of your make process, 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, so you will have faster build times if PEG is linked as a library.
- A good linker will not include PEG components that are not used by your application.
- 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.

You will likely find additional source files in your distribution that are not documented here. These source files are specific to the operating system and target hardware that you have requested, and they are documented in your distribution's hardware/RTOS integration notes. This section documents only those source files that are common to every distribution, named the 'core source files.'

Source File Overview

The following source files are included in your PEG distribution:

Source File	Contents
p2dpoly.cpp	Peg2DPolygon class
pal256.cpp	Generic 256 entry color palette
panimate.cpp	PegAnimation class
panimwin.cpp	PegAnimationWindow class
paniprmt.cpp	PegAnimatedPrompt class
pbig5map.cpp	Chinese Big5 to Unicode character mapping tables and API functions
pbitmaps.cpp	Built-in bitmap images for system buttons and mouse pointers
pbmpconv.cpp	Run time image conversion for BMP source
pbmprot.cpp	Run time image rotation functions (PEG Pro Only)
pbmpwrit.cpp	Run time bitmap writer
pbprompt.cpp	PegBitmapPrompt class
pbrush.cpp	PegBrush class implementation.
pbutton.cpp	Various button classes implementation
pcbdial.cpp	PegCircularBitmapDial class
pcdial.cpp	PegCircularDial class
pchart.cpp	PegChart class
pcombo.cpp	PegComboBox class
pdecbtn.cpp	PegDecoratedButton class

pdecwin.cpp	PegDecoratedWindow class
pdial.cpp	PegDial class
pdialog.cpp	PegDialog class
pdirbwr.cpp	PegDirectoryBrowser class
peditbox.cpp	PegEditBox class
peditfld.cpp	PegEditField class
peggrad.cpp	PegGradient class
pegheap.cpp	PegHeapManager class
peginit.cpp	PegCreateFrameWork and PegDestroyFramework functions
pegpci.cpp	PEG PCI-bus access functions
pfbdial.cpp	PegFiniteBitmapDial class
pfdial.cpp	PegFiniteDial class
pfdialog.cpp	PegFileDialog class
pfile.cpp	PegFile filesystem encapsulation
pgifconv.cpp	PegGifConvert class
pgroup.cpp	PegGroup class
phelpbtn.cpp	PegHelpButton class
picon.cpp	PegIcon class
pimgconv.cpp	PegImageConvert class
pjpgconv.cpp	PegJpgConvert class
plist.cpp	PegList, PegHorzList, PegVertList classes
pliteral.cpp	PEG string constants
plnchart.cpp	PegLineChart class
pmenfont.cpp	PegMenuFont
pmenu.cpp	PegMenu, PegMenuBar, PegMenuButton classes
pmesgwin.cpp	PegMessageWindow class
pmessage.cpp	PegMessageQueue class
pmlchart.cpp	PegMultiLineChart class
pmlmsgwn.cpp	PegMultiLineMessageWindow class
pmltbtn.cpp	PegMultiLineTextButton class
pnotebk.cpp	PegNotebook class
ppngconv.cpp	PegPngConvert class
ppresent.cpp	PegPresentationManager class
pprint.cpp	PegPrinter interface class
pprogbar.cpp	PegProgressBar class
pprogwin.cpp	PegProgressWindow class
pprompt.cpp	PegPrompt class
ppwdfld.cpp	PegPasswordField class
pquant.cpp	PegQuant class
prect.cpp	PegRect class
presmgr.cpp	PegResourceManager class
pscreen.cpp	PegScreen class (partial)
pscrline.cpp	PegScreen line, polygon functions
pscrnarc.cpp	PegScreen Arc, Circle, Chord functions
pscroll.cpp	PegHScroll and PVScroll
psincos.cpp	Fixed-point sin/cos implementation
psjmap.cpp	Shift-JIS to Unicode mapping tables
pslider.cpp	PegSlider class

pspin.cpp	PegSpinButton class
pspread.cpp	PegSpreadSheet class
psprompt.cpp	PegScrollPrompt class
pstatbar.cpp	PegStatusBar class
pstchart.cpp	PegStripChart class
psysfont.cpp	System Font
ptable.cpp	PegTable
ptextbox.cpp	PegTextBox
pthing.cpp	PegThing
ptimer.cpp	PegTimerManager class
ptitle.cpp	PegTitle
ptoolbar.cpp	PegToolBar class
ptree.cpp	PegTreeView
ptxtthng.cpp	PegTextThing class
pvecfont.cpp	PegVectorFont
pvprompt.cpp	PegVPrompt
pvvlist.cpp	PegVirtualVertList class
pwindow.cpp	PegWindow
pzip.cpp	PegZip/PegUnzip

Header File Overview

Most source files also have a corresponding header file that prototypes the class. There are additional header files included in your distribution for configuring the PEG library options, defining the PEG data types, etc.

Applications that reference the PEG library need only include the header file ‘peg.hpp.’ This header file first includes pconfig.hpp and conditionally includes the remaining header files required to build the PEG library or to build your application.

1.4.1 Additional files

There are many files in addition to the core library source files listed above. These additional files are related to hardware integration, compiler make/ project files, RTOS integration, screen driver, touch driver, and keyboard driver files. The specific set of additional files included in your installation is dependent on your installation key and cannot be listed here.

For example, if you have ordered PEG for RTOS ‘myRtos’ and compiler ‘myCompiler’ and CPU evaluation board ‘myHardware,’ your installation key will unlock many additional files specific to your RTOS, compiler, and target hardware in addition to the core library files.

Screen Driver

There are many derived versions of the ***PegScreen*** class, not all of which are included in the standard source code distribution. Many versions of ***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 pscreen.cpp file in your PEG library build. You should also include the PEG screen driver template matching your target system color depth. Finally, you must also include one target-specific screen interface class that depends on the platform for which you are building the library.

The screen driver source files provided with your distribution are in the directory **peg\source\screendrv**. The files in this directory will usually include a set of desktop-oriented screen drivers (i.e. screen drivers for running PEG on Windows or Linux/X11), in addition to screen drivers specific to your target hardware.

Additional files that may be required

The touch/mouse driver source files provided with your distribution are in the directory **\peg\source\touchdrv**.

The operating system specific source files provided with your distribution are in the directory **\peg\source\rtos**.

The preconfigured make/project files for building the PEG library are provided under **\peg\build**.

Hardware-specific example applications are provided in the directory **\peg\ targets.** Files in this directory should not be included in your PEG library. These files are typically example applications and target configuration files and should be compiled and linked to make an example executable image.

1.5 Preconfigured Make/Project Files

1.5.1 Building PEG for Windows Desktop

Project files are provided for building the PEG library for execution on a Windows desktop in the \peg\build\win32 directory. Each folder under this directory represents a compiler. Microsoft (many versions), Borland, Green Hills, and possibly other compilers will be listed under this root folder.

Building PEG for DOS

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.

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 PEG for X11 Desktop

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. 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 preconfigured for a specific compiler and target.

Building PEG for other targets

For most distributions, there will also be a folder under \peg\build that is labelled with your RTOS and target compiler name. Under this folder, you will find the correct project/make file to build PEG for your target hardware.

We suggest that you first build PEG to run in one of the preconfigured 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 your are an experienced C++ programmer, you can safely skip ahead to Chapter 3.

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.

Users who are familiar with the EC++ (Embedded C++) standard should note that PEG is fully compliant with EC++. PEG does not use C++ Exceptions, Templates, or RTTI (RunTime Type Identification). PEG does not utilize multiple-inheritance.

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.

2.1 Notes for 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 (Stroustrop), 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 C++ user is aware, 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. Of course, in the C language you can manually maintain pointers to functions within your structures, and this is really all that C++ is doing. The C++ compiler does all of the tricky bits of storing indirect function pointers for you.

Encapsulation

Encapsulation simply means that 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. In addition, the 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.

In addition to public and private members, 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, which 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 API Reference 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)*** that compose a class library is very important, and, 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 reused by all classes in the library.

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.

2.2 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 them 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 most 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.

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 reuse 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 comprising 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 ensure 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 multitasking environments. Modal windows capture all input devices, preventing other windows and controls from being active while the modal window is executing.

CHAPTER 3

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:

```
PEGBYTE       signed 8-bit value
PEGUBYTE      unsigned 8-bit value
PEGINT        signed native int (size unspecified)
PEGUINT       unsigned native int (size unspecified)
PEGSHORT      signed 16-bit value
PEGUSHORT     unsigned 16-bit value
PEGLONG       signed 32-bit value
PEGULONG      unsigned 32-bit value
PEGCHAR       8- or 16-bit character storage type
PEGBOOL       TRUE/FALSE value
PEGCOLOR      color storage type, size dependent on hardware config
```

3.1 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
{
    PEGSHORT x;
    PEGSHORT y;
};
```

Note that PegPoint contains PEGSHORT (signed 16-bit) 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 nonexistent area of video memory.

3.2 PegRect

A large part of your programming tasks in 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(PEGINT x1, PEGINT y1, PEGINT x2, PEGINT y2)
    {
        iLeft = x1;
        iTop = y1;
        iRight = x2;
        iBottom = y2;
    }

    void Set(PegPoint ul, PegPoint br)
    {
        iLeft = ul.x;
        iTop = ul.y;
        iRight = br.x;
        iBottom = br.y;
    }

    PEGBOOL Contains(PegPoint Test);
    PEGBOOL Contains(PEGINT x, PEGINT y);
    PEGBOOL Contains(PegRect &Rect);
    PEGBOOL Overlap(PegRect &Rect);
    void MoveTo(PEGINT x, PEGINT y);
    void Shift(PEGINT xShift, PEGINT 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 += (PEGINT);
    PegRect operator --(int x);
    PegRect operator -= (PEGINT);
    PEGBOOL operator != (PegRect &Rect);
    PEGBOOL operator == (PegRect &Rect);
    PEGINT Width(void) {return (iRight - iLeft + 1);}
    PEGINT Height(void) { return (iBottom - iTop + 1);}

    PEGSHORT Left;
    PEGSHORT Top;
    PEGSHORT Right;
    PEGSHORT Bottom;
};
```

3.3 PegBrush

A ***PegBrush*** is a simple data type passed to drawing functions. A ***PegBrush*** contains the LineColor, FillColor, Pattern, Width, and Style for your drawing. The ***PegBrush*** class is defined as shown:

```
class PegBrush
{
    friend class PegScreen;

    public:
        PegBrush();
        PegBrush(PEGCOLOR LColor, PEGCOLOR FColor,
            PEGINT BStyle = PBS_NO_ALIAS,
            PEGINT LWidth = 1);

        ~PegBrush();

        void Set(PEGCOLOR LColor, PEGCOLOR FColor,
            PEGINT BStyle = PBS_NO_ALIAS,
            PEGINT LWidth = 1)
            {
                LineColor = LColor;
                FillColor = FColor;
                Style = BStyle;
                Width = LWidth;
            }

        PEGCOLOR LineColor;
        PEGCOLOR FillColor;
        PEGULONG Pattern;
        PEGINT   Width;
        PEGINT   Style;
        PegBitmap *pBitmap;

    private:
        PegBitmap *pSysMap;
};
```

The LineColor is the foreground color for drawing text, polygons, etc.

The FillColor is the color used to fill rectangles, polygons, etc.
Width is the outline width for Polygons, Rectangles, etc.
Style is a set of bitwise combined flags to modify the drawing style. The Style flags are:

PBS_SOLID_FILL
The drawing shape (circle, rectangle, etc.) is solid filled.

PBS_BMP_FILL
The drawing shape (circle, polygon, etc.) is filled with a bitmap pattern. The bitmap pattern is specified in the Brush.pBitmap field.

PBS_NO_ALIAS
Text and lines are drawn without anti-aliasing.

PBS_SIMPLE_ALIAS
Text and lines are anti-aliased to a fixed background color.

PBS_TRUE_ALIAS
Text and lines are anti-aliased to the actual background pixel color (PEG Pro only).

PBS_UNDERLINE
Text is underlined.

PBS_ROUNDED0
Lines are drawn with round ends.

PBS_CENTER_LINE
If the Brush.Width field > 1, this flag indicates that the border should be centered on the object perimeter. If this flag is not set, the border is drawn inside the object perimeter.

PBS_PATTERN
This flag is used to draw a patterned rather than solid outline.

3.4 PegMessage

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

```
struct PegMessage
{
    PegThing *pTarget;
    PegThing *pSource;
    PEGUSHORT Type;
    PEGUSHORT Param;
    PegMessage *Next;

    union
    {
        PegRect Rect;
        PegPoint Point;
        PEGLONG ExtParams[2];
        void *pData;
        PEGLONG UserLong[2];
        PEGSHORT UserShort[4];
        PEGUSHORT UserUShort([4];
        PEGUBYTE  UserUByte[8];
     };
};
```

On most machines, each PegMessage requires 24 bytes of RAM if structure packing is enabled.

For user-defined messages, all but the Type and pTarget message fields can be used in any way desired. There is much more information about sending and receiving messages in Chapter 4, ‘PegMessageQueue.’

3.5 PegTimer

PEG timers provide a simple means for you to receive periodic timer messages in your windows or controls. Any PEG graphical object can start any number of individual timers. When the timer expires, that object will receive a PM_TIMER message from PEG. The message Param 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 ***PegTimerManager***. In order for PEG timers to function, your system software must call the ***PegTimerManager*** 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 ***PegTimer***s 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 ten milliseconds. In that case, you would simply call the **PegMessageQueue::TimerTick()** function once for every 5 interrupts received.

You (or your operating system integration) 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 down every location where you are using a PEG timer and modify the tick value used.

You start a PegTimer by calling the PegTimerManager member function SetTimer(PEGUSHORT ID, PEGINT Initial, PEGINT Repeat). 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 Initial and Repeat timeout periods determine how many timer ticks will expire before the timer 'times out,' and these can be the same value. If the Repeat 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 PegTimerManager member function KillTimer(PEGSHORT 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. ***PegTimerManager*** provides all of the interface functions you will need to create and use ***PegTimer***s. However, for completeness, the definition of ***PegTimer*** is shown below:

```
struct PegTimer
{
    PegTimer() {pNext = NULL; pTarget = NULL;}
    PegTimer(PEGLONG Cnt, PEGLONG Res)
    {
        pNext = NULL;
        pTarget = NULL;
        Count = Cnt;
        Reset = Res;
    }

    PegTimer(PegTimer *Next, PegThing *Who, PEGUSHORT Id,
        PEGLONG Cnt, PEGLONG Res)
    {
        pNext = Next;
        pTarget = Who;
        lCount = Cnt;
        lReset = Res;
        wTimerId = wId;
    }

    PegTimer *pNext;
    PegThing *pTarget;
    PEGLONG Count;
    PEGLONG Reset;
    PEGUSHORT TimerId;
};
```

3.6 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) 24-bpp (8-8-8 RGB), and 32-bpp ARGB formats. Further, PEG bitmaps may be compressed or uncompressed and may have an alpha channel. PEG uses RLE compression techniques for compressed bitmaps.

PegBitmap data structures are created by WindowBuilder and saved in your resource file. Your application software utilizes Bitmap IDs, which are converted to PegBitmap data structures by the PegResourceManager.

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

```
struct PegBitmap
{
    PEGUBYTE uFlags; // compressed, transparent, etc.
    PEGUBYTE uBitsPix; // 2, 4, or 8
    PEGUSHORT wWidth; // in pixels
    PEGUSHORT wHeight; // in pixels
    PEGULONG dTransColor; // transparent color for bmps > 8bpp
    PEGUBYTE *pStart; // address of bitmap data
};
```

Most ***PegBitmap*** structures used in your application will probably be generated prior to compiling with ***WindowBuilder***. 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 PNG, GIF, JPG, and BMP files into ***PegBitmap*** structures at run time.

3.7 PegFont

A PegFont is a data structure that contains information to allow the PEG software to draw text. PegFont data structures are generated by WindowBuilder and saved in your application resource file. Your application usually uses font IDs rather than directly referencing PegFont data structures. Font IDs are converted to PegFont data structures by the PegResourceManager.

The PegFont data structure is defined as:

```
struct PegFont
{
    PEGUBYTE   Type;         // bit-flags defined below
    PEGUBYTE   Ascent;       // Ascent above baseline
    PEGUBYTE   Descent;      // Descent below baseline
    PEGUBYTE   CharHeight;   // total height of character
    PEGUBYTE   PreSpace;     // leading space
    PEGUBYTE   PostSpace;    // trailing space
    PEGUBYTE   LineHeight;   // total height with pre and post
    PEGUSHORT  BytesPerLine; // total width of one scanline
    PEGUSHORT  FirstChar;    // first character present in font
    PEGUSHORT  LastChar;     // last character present in font
```

```
    PEGUSHORT  *pOffsets;    // bit-offsets for variable-width font
    PegFont    *pNext;       // page link pointer
    PEGUBYTE   *pData;
};
```

This structure contains all of the information needed by the PEG screen driver to draw text on the graphical screen.

The PegFont.pData member points to a block of constant data that describes the bitmap or rendering of each character. Several formats for this data block are supported, and the specific format is indicated by the PegFont.Type data field.

The PegFont.Type data field bits include:

PFT_VARIABLE // variable width font
PFT_OUTLINE // 2-bpp outline font format
PFT_ALIASED // 4-bpp anti-aliased font format
PFT_ZIPPED // LZW compressed font data
PFT_BMPFONT // 8-bpp bitmap font format
PFT_DROPSHADOW // 4-bpp drop-shadow font format

To summarize, you will not be manually creating or editing PegFont data structures. That is the job of the font generation and editing facilities that are included within the WindowBuilder program. This section is only for reference and to provide some insight into the inner workings of PEG text drawing capabilities.

3.8 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 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);
```

```
        PEGBOOL IsValid(void);
        void SetValid(PEGBOOL bValid);
        void Realloc(PEGLONG Size);
        void Reset(void);
        void MoveTo(PEGINT Left, PEGINT Top);
        void Shift(PEGINT xShift, PEGINT yShift)
        {mRect.Shift(xShift,yShift);}
        PegBitmap *Bitmap(void);

    private:

        PegRect mRect;
        PegBitmap mBitmap;
        PEGLONG    mDataSize;
        PEGBOOL    mValid;
};
```

CHAPTER 4

PEG EXECUTION MODEL

This chapter introduces the PEG execution model and describes how the fundamental PEG classes work together to create a working interface. This chapter focuses on establishing a macro view 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 aware that this chapter and those immediately following contain a large amount of important information, and much of this information is not 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 interdependent, it is unavoidable 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 3, 4, and 5 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, most of the information presented in this chapter assumes that you are running in the Multithread model, which means that you are running PEG within some operating system environment such as a hard-real-time RTOS, or possibly Linux or WinCE.

You can of course also run PEG standalone or in a multiprocessor distributed environment. All of the concepts presented are valid regardless of your execution model. A complete discussion of multitasking as it relates to PEG and the supported execution models is included in the ‘PEG Multitasking’ chapter.

4.1 Overview

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

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. PegTask is not a function name, but rather a conceptual thread of execution.

PegMessageQueue provides a FIFO-style message queue for sending information between PEG objects. PegMessageQueue provides the mechanism for sending user input events to PEG.

PegTimerManager provides a simple interface for low-resolution timing operations.

PegResourceManager organizes and controls access to your application resources. Resources include strings, bitmaps, fonts, and colors.

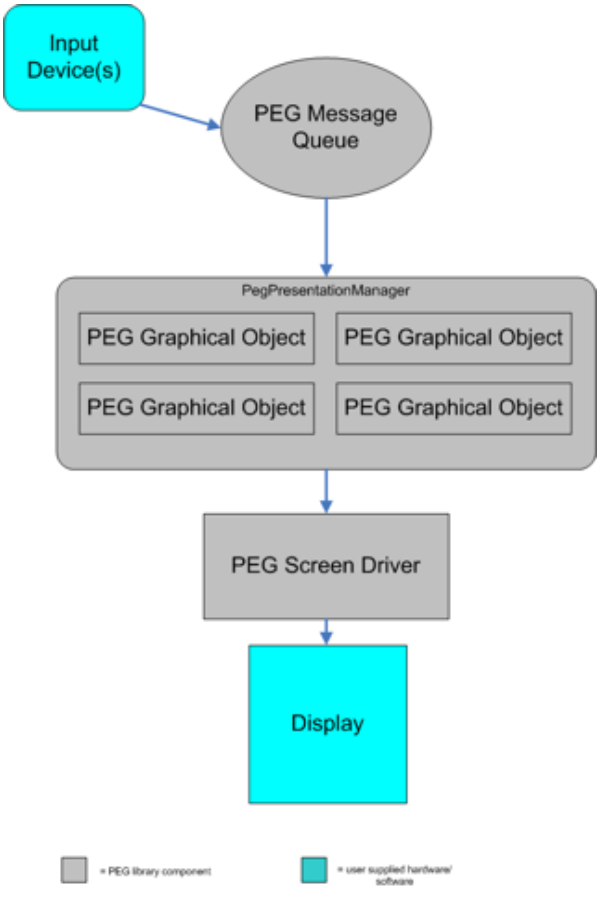
PegPresentationManager keeps order on the visible screen. This involves keeping track of which window(s) are on top of other windows, maintaining the status of each object, and remembering which object should receive user input.

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.

4.1.1 Software Block Diagram

A software block diagram of an executing PEG application is shown on the following page. This drawing depicts input device(s), ***PegMessageQueue,***

PegPresentationManager, miscellaneous graphical objects added to the PresentationManager, and the ***PegScreen*** driver. These are the components of every user interface built using PEG.



4.2 Program Startup

PEG startup is very simple. In PegTask (or in your main function if running standalone), the function named PegCreateFramework() is called to create and initialize each of the above required components. PegCreateFramework() constructs ***PegPresentationManager***, ***PegMessageQueue***, ***PegTimerManager, PegResourceManager***, and the derived version of ***PegScreen*** that is required for the target system. These components are required for any PEG application to run.

After the PEG framework is initialized, PegTask calls the user-supplied function named **PegAppInitialize()**. This is the entry point from the application-level software perspective. The PegAppInitialize() function can be created by WindowBuilder or it can be handwritten.

PegTask is tailored to the execution environment, while **PegAppInitialize()** is application defined and does not change from one environment to another. This segmentation follows the PEG philosophy of insulating all application-level software from the target environment, allowing your application software to run without modification when moving from one of the predefined 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. As a preview, here is an example of what **PegAppInitialize()** might look like in your application software:

```
void PegAppInitialize(PegPresentationManager *pPresent)
{
    // install resource used by your application:

    PegResourceManager::InstallResourcesFromTable(&MyResTable);

    // create and add your first window
    pPresent->Add(new MyWindow());
}
```

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 first

install or register the resources (fonts, bitmaps, etc.) used by the application with PegResourceManger. This resource table is generated by WindowBuilder.

Next, the function constructs an instance of a window class named '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***.'

4.2.1 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***, and PegPresentationManager 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 is a conceptual thread of execution and is implemented in various ways dependent on the underlying operating system. When running in a standalone environment such as DOS, ***PegTask*** is simply the function **main()**.

The following pseudo-'C' code illustrates a typical implementation of ***PegTask.*** The actual implementation will vary slightly from one RTOS integration to another:

```
void PegTask(void)
{
    // Initialize the PEG framework:

    PegRect ScreenSize;
    Rect.Set(0, 0, 639, 479);

    PegCreateFramework(ScreenSize);

    // call application initialization function:
```

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

    // returns only if the application is terminated
    // delete PEG foundation objects
    PegDestroyFramework();
}
```

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

4.2.2 PegIdleFunction

Note that PegIdleFunction() is not required and is not called by versions of PEG which integrated with a commercial RTOS.

In a standalone, single-threaded environment, a second user-supplied function named **PegIdleFunction()** is called by ***PEG*** when there are no longer any messages in the message queue that 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 calls to external system functions that do the real work of your embedded system. Note that PegIdleFunction must return periodically to allow PEG to check for input messages and keep the screen updated. It is not permissible to enter an endless loop within PegIdleFunction().

Versions of PEG that have been integrated by Swell Software with a commercial RTOS do not use this callback mechanism; instead, they automatically suspend or block the PegTask(s) when no messages are available for processing.

Below is an example implementation of PegIdleFunction for a system in which all input devices are polled. The following is the DOS version of **PegIdleFunction()** contained in the file ***dospeg.cpp***:

```
void PegIdleFunction(void)
{
    PollTime();      // see if time has changed
    PollMouse();     // see if mouse has changed
    PollKeyboard();  // see if keyboard has changed
}
```

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 receives input focus. Likewise, if that window has any child objects, the window will (by default) search for the top-left most child object capable of receiving input focus. This continues until a leaf node (an object with no children) is found, and that is the object which will initially have input focus when a new window is displayed.

CHAPTER 5

PEGMESSAGEQUEUE

PegMessageQueue is a simple encapsulated FIFO message queue with member functions for queue management.

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.
- From PEG objects themselves.

The messages placed in ***PegMessageQueue*** are the driving force behind the graphical interface. These messages contain notifications and commands that 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 back to the graphical element.

PEG defines its own message format. The PEG message format never changes from one operating system to another or when running on the desktop vs. running on your target. When running with a real-time operating system, PEG implements the PegMessageQueue by utilizing the underlying operating system services. To your application level software, it always appears simply as PEG messages running through the PegMessageQueue, regardless of the underlying implementation. This of course helps to make your application software completely portable across operating systems.

5.1 PEG Message Definition

PegMessage is a data structure that contains members indicating the source, target, and content of the message. The definition of this data structure is shown below:

```
struct PegMessage
{
    PEGUSHORT Type;
    PEGUSHORT Param;
    PegThing *pTarget;
    PegThing *pSource;
    PegMessage *Next;

    union
    {
        PEGLONG ExtParams[2];
        PegRect Rect;
        PegPoint Point;
        void *pData;
        PEGLONG UserLong[2];
        PEGULONG UserULong[2];
        PEGSHORT UserShort[4];
        PEGUSHORT UserUShort[4];
        PEGUBYTE UserUByte[8];
     };
};
```

On most systems, each PegMessage structure requires 24 bytes of memory. Note that the PegMessage structure contains a union, and these union data fields overlap in memory. You can only assign values to one member of the union.

Messages are identified by the member field Message.Type. This is a 16-bit unsigned integer value that allows 65,535 unique message types to be defined. Currently PEG reserves the first 0x4000 message type values for internal messages, which leaves message values 0x4000 through 0xffff 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 that is available for user definition. This #define is called **FIRST_USER_MESSAGE**.

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 either be passed to a callback function that you define, or if no callback function is defined, the message will be discarded. This occurrence is usually an indication that you forgot to catch a message in one of your window classes.

To define a default message handler that will process all user-defined messages that arrive at PegPresentationManager, you must call the PegPresentationManger::SetUserMessageHandler(...) function, which is fully defined in the API reference manual.

Many messages, especially user-defined messages, may be directed towards a particular object by the pTarget message field or by the message Param 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 Param field contains the ID of the object that should receive the message***. This means that 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 Param member. If you want to route user-defined messages using object ID values, those objects should have globally-defined object IDs to ensure that there are never multiple objects visible with duplicate ID values.

When a PEG object sends a system-defined message to its parent, 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, and so on.

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_DEFINED 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.Type field. This is a 16-bit unsigned value. PEG reserves Message.Type values 1-FIRST_USER_MESSAGE - 1. This leaves message types 0x4000 through 0xffff 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 Type 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 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.

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 filed should contain a pointer to the child object.
PM_ADDICON	This message is sent to a parent window when a child window has been minimized. This tells it to draw an icon at the bottom of the parent window representing the child window. The pData field contains the PegIcon to add.
PM_BEGIN_MOVE	This message is sent by a PegTitle or PegStatusBar object to their parent window. It tells it that the user has started moving the window. The Point field of the message represents the originating point before the move.
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_CUT	This message is sent to cut text from a PegTextThing object.
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 ***PegPresentationManager*** to cause termination of the application program.
PM_GAINED_KEYBOARD	This message is sent to an object when it gains keyboard input focus. This is only used when PEG_KEYBOARD_SUPPORT is enabled.
PM_HIDE	This message is sent to an object whenever it is removed from a visible parent.
PM_HSCROLL	This message is sent to a window to tell it to scroll its client area left or right. The Param field contains the current scroll position and the ExtParams[0] field contains the new scroll position.
PM_KEY	This message is sent to the current input object when keyboard input is received. The message Param member contains the corresponding ASCII character code, if any, and the lData member of the message contains the keyboard scan code, if available.
PM_KEY_RELEASE	This message is sent to the current input object when the user has released a key on the keyboard. This is analogous to a PM_LBUTTONUP message with mouse clicks. The message Param member contains the corresponding ASCII character code, if any, and the Param member of the message contains the keyboard scan code, if available.
PM_LANGUAGE_CHANGE	This message is sent to all PegTextThing objects whenever the language changes by calling SetCurrentLanguage(). This is only used when PEGSTRING_IS_ID is enabled.

PM_LBUTTONDOWN	This message is sent to an object when the user generates left mouse click input. ***PegPresentationManager*** 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 ***PegPresentationManager***. 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_LOST_KEYBOARD	This message is sent to an object when it loses keyboard input focus. This is only used when PEG_KEYBOARD_SUPPORT is enabled.
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 the 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, it will create a proxy PegIcon, add the icon to the parent window, and remove itself from its parent.
PM_MWCOMPLETE	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 Param 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 resized. 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_REMOVETHING	This message is sent to an object to tell it to remove another object. The pSource field points to the object that will be removed.
PM_RESTORE	This message is sent to a window to tell it to restore its dimensions to the previous settings before it became maximized.
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 resize. 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_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 ***PegMessageQueue*** TimerSet function when that timer expires. The ID of the timer is included in the Param member of the message.
PM_VSCROLL	This message is sent to a window to tell it to scroll its client area up or down. The ExtParams[1] field contains the current scroll position and the ExtParams[0] field contains the new scroll position.

User-Defined Messages

User-defined messages are message types you create for your own purposes. You may have other tasks in the system that send user-defined messages to your PEG task, or you may have one PEG window or control send a user-defined message to another window or control.

There are many other reasons you will want to define your own messages, and it will become clearer as you begin using the library. As an example to get you 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
        TURN_GREEN,
        TURN_INVISIBLE
    };
}
```

The above example illustrates a common way of defining your own messages. We have implied in this example that you can reuse 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 reusing the same message numbers at various points in your application.

There are three ways to send a message from one object to another. First, you can call the destination object's message handling function directly, passing your message as a parameter. Second, you can load the message **pTarget** field with the address of the object (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 **Param** member with the ID of the target, and push the message into ***PegMessageQueue***. The last method is most often preferred.

If you load message pTarget values with pointers to application objects, you must ensure 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 most 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 object will receive and process the new message after the sending window returns from message processing. If you call the receiving object's message handling function directly, it will immediately receive and process the message, in effect preempting the current execution thread. While these

differences are generally inconsequential for user-defined messages, they can be very important for PEG system messages.

5.2 Signals

Messages are used to issue commands or send other information between objects that are part of your user interface. The most common use of a message is for a child object to notify the object's parent that is has been modified. For example, a button will notify its parent window that it has been clicked, or a slider control will notify its parent that the slider value has been changed. This usage is so common that PEG defines a special syntax for these notification messages. This syntax 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 handling child notifications:

- 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 PegEditField 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 in which you are not interested.
- Finally, to facilitate the implementation of a RAD window prototyping tool such as PEG WindowBuilder, 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 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 syntax 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 object's '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 object's 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 **PEG_SIGNAL** macro, which is used in your parent window message-processing function. The parameters to the **PEG_SIGNAL** macro are the object ID and the notification message in which you are interested. The **PEG_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 possible notification types.

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 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. 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.

We present the message handling function of the dialog window here as a preview allowing you to observe the syntax of PEG signaling.

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 PEG_SIGNAL macro. The parameters to the PEG_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 self documenting, since as you become familiar with this syntax you will quickly be able to recognize the control type and notification that each case statement is processing.

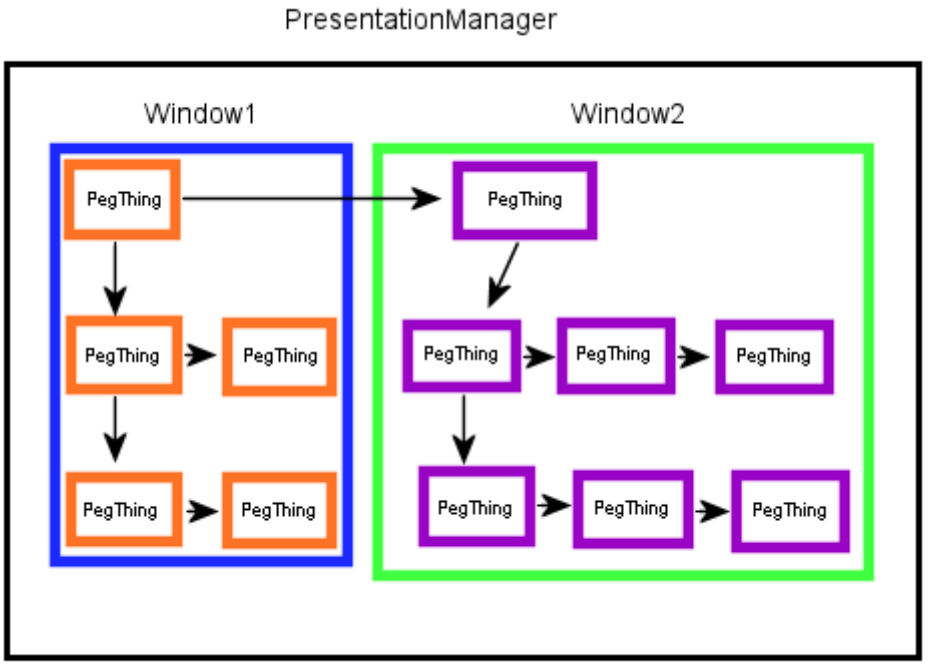
```
// Excerpt from MyDialog dialog window header file:

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.Type)
    {
    case PEG_SIGNAL(USER_NAME, PSF_TEXT_EDITDONE):
        // add code for user name modification here:
        break;

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

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

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

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

CHAPTER 6

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

6.1 Event-Driven Programming

PEG is message-driven, which may also be called event-driven. This means that real processing is normally done in response to messages received from the outside world. PEG follows the event-driven programming paradigm. Controls and Windows respond to input events, and are largely defined by which input events they process and how they process those events.

The objects 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.

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 are described in detail in the chapter titled ***PegMessageQueue***.

PegPresentationManager supplies the overall control of your PEG application. ***PegPresentationManager::Execute()*** enters a continuos loop

popping messages from PegMessageQueue and routing those messages to graphical objects. 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, you want it to execute only when there is real work to do, and even then only when no higher-priority tasks are ready to run. In a multi-tasking environment, PEG automatically suspends itself when there is no work to do.

6.2 Input Focus Tree

An additional task of PegPresentationManager is message routing. Many system messages, such as mouse and keyboard input messages, are not directed to any particular object when they are placed in PegMessageQueue. For this reason, PegPresentationManager internally maintains a pointer to the object that was last selected by the user through 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 ensures 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 that 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 receives input focus. Likewise, if that window has any child objects, the window will (by default) search for the top-left most child object capable of receiving input focus. This continues until a leaf node (an object with no children) is found, and that is the object which will initially have input focus when a new window is displayed.

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 should be sent keypress messages 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 directional and/or select keys available. This type of input will work just fine with a PEG application, since PEG requires only a very limited set of input key types to navigate through screens and select controls

We will fully describe the format of PEG messages in a 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.Type field == PM_KEY. The actual key value is passed in the Mesg.Param data field, and the key flags such as shift key state, control key state, etc. are passed in the Mesg.ExtParams[0] 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.

The paragraphs below describe the key values 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; it only means that PEG is watching for these key values. If your target system has, for example, four arrow keys and an 'enter' key, you simply need to map these keys to the best-match key values PEG is watching for. In some cases, you may need to send different key values for a common input key depending on the type of object with which the user is interacting.

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 ExtParams[0] member), the tab direction is reversed.
<ctrl> + PK_TAB	This key combination is used to cycle through top-level windows.
PK_CR	The carriage return key is used to select the item which has focus. If this is a button object, the object will become 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 PegEditField edit operation.
<ctrl> + PK_F4	The key combination is used to close the current window

Mouse or Touch Screen Input Handling

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

CHAPTER 7

PEGTIMERMANAGER

PEG provides a simple and easy-to-use timer facility for use by the graphical application. PegTimerManager is responsible for creating, deleting, and ticking high-level user-interface (UI) related timers. UI timers are high-level, low-resolution timers useful for graphical operations. They are not directly linked to hardware-level timer interrupts. PegTimerManager can create any number of active timers simultaneously. Each timer is linked to the timer creator, and the timer creator will receive PM_TIMER message(s) when the timer expires. Timers can be configured for either one-shot or continuous (repeating) operation.

In a multitasking system, PegTimerManager usually receives input from one RTOS level timer to serve as the time base for all UI timers. The **PegTimerManager::TimerTick()** function must be called periodically to serve as the time base for all UI timers. PEG distributions for specific operating systems are already configured to perform the RTOS initialization required to driver the high-level PegTimer facility.

When a PEG object is destroyed, PegTimerManager automatically destroys any timers linked to that object. PegTimerManager also supports a single object creating multiple timers, each identified with a unique timer ID.

PegTimers are created and started by calling

PegTimerManager::SetTimer(TIMER_ID, INIT_VALUE, REPEAT_VALUE);

If you are within the scope of a PegThing-derived class, you can omit the class scope resolution operator and simply type this:

SetTimer(TIMER_ID, INIT_VALUE, REPEAT_VALUE);

The TIMER_ID is any integer value you define. This value is passed back to you when the timer expires and you receive a PM_TIMER message. The TIMER_ID value is passed back in the Message.Param field.

The INIT_VALUE is the initial timeout value. The actual value of this time depends on how often the **PegTimerManager::TimerTick()** function is called. The default frequency is ONE_SECOND / 20, or 50 ms per tick.

The repeat value can be zero or any non-negative integer value. A repeat value of zero indicates that the timer is a 'one shot' timer. It will expire once and destroy itself. If the repeat value is not 0, the timer will run continuously, sending repeated PM_TIMER message to the timer owner until the timer is stopped.

A timer can be stopped at any time by calling:

KillTimer(TIMER_ID);

If the TIMER_ID is not zero, only that timer is stopped. To stop all timers owned by an object, pass zero (0) as the TIMER_ID.

An object that starts a timer receives timer expiration message(s). It is important to remember to check the TIMER_ID of a timer message and to ensure that it is a timer you created in your application. If the timer is not your own, you must pass the timer message down to your base class, as several PEG classes create and use their own timers. Consequently, they will not function correctly if you prevent your base class from receiving timer expiration messages. This is an example of a timer message handler:

```
#define MY_TIMER  1

PEGINT MyClass::Message(const PegMessage &Mesg)
{
    switch (Mesg.Type)
    {
    case PM_TIMER:
    if (Mesg.Param == MY_TIMER)
    {
         // do timer event handling here
    }
    else
    {
         // It's not my timer, so
        //  pass the timer even message down to my base class:
         MyBaseClass::Message(Mesg);
    }
    }
}
```

Any object can create any number of simultaneous timers. Each timer should be created with a unique timer ID value.

CHAPTER 8

PEGRESOURCEMANAGER

The fonts, bitmaps, strings, and colors you use in your application are called ***Resources***. In many ways, resources are independent of your application software. You can change and modify your resources to change your user interface without making any changes to your application software. You can even change your resources 'on the fly' as the system is running. An example of this would be a change to the active language or to the color theme of your application.

PegResourceManager manages your system resources. ***PegResourceManager*** allows you to add, remove, and modify resources at compile time and at run time. Resources are registered with the ***ResourceManager***, at which time they are assigned a resource ID. Your application software always refers to a resource using the resource ID rather than using a direct reference to a font, bitmap, or string. This abstraction is what makes it possible to easily modify your resources without requiring any changes to your application software.

8.1 FIDs, BIDs, CIDs, and SIDs

Resources and their corresponding resource IDs come in four flavors, corresponding to each resource type. The resource type is indicated by the prefix of the resource ID. Font resource IDs are always prefixed with FID_ (**F**ont **ID**). Likewise, bitmaps are prefixed with BID_, colors prefixed with CID_, and strings with SID_. This makes it easy to recognize what type of resource is being referred to as you write and examine your application software.

The remainder of the resource ID is any 'C' language compatible variable name that you type as the resource ID name. Examples might include 'BID_BATTERY_ICON,' 'FID_MENU_ITEM,' and 'CID_SLIDER_FILL.' Resource IDs become members of enumeration lists produced by WindowBuilder in your resource header file.

8.2 Registering Resources

The ***ResourceManager*** creates tables to track your resources and resource IDs. Before you can use a resource, it must be registered with the ***ResourceManager***. Resources are registered with the ResourceManger by calling this function

```
PegResourceManager::AddResource(
   PEGUINT RES_ID,
   Resource *,
   PEGUINT Flags,
   PEGBOOL Replace);
```

All public functions of the ***ResourceManager*** are static functions, meaning that they can be accessed from anywhere in your source code without having a pointer to the ResourceManger instance.

The RES_ID parameter is the numeric resource ID. This ID is generally enumerated for you by PEG WindowBuilder, but you can also create your own resource IDs.

The second parameter is a pointer to the resource itself. This will be a pointer to a PegFont, PegBitmap, PegString, or color value depending on the type of resource being registered.

The Flags parameter is used in different ways by different resource types. The Flags indicate special consideration, such as accelerating a bitmap by placing the bitmap in graphics memory, or accelerating a font by placing the font in graphics memory.

The Replace parameter indicates that if a resource already exists in this table position, the ***ResourceManager*** should delete the old resource when replacing it with the new resource. If this parameter is FALSE, the old resource is not deleted from memory and it is the responsibility of the application software to remove the old resource from memory if and when it is no longer needed.

The above AddResource function can be used to register resources individually or one at a time. Resources can also be registered in blocks or multiple resources added *en masse*. This is the method used by WindowBuilder. WindowBuilder simplifies the process of registering your resources by creating a ResourceTable which is a list of multiple resources

of each time. All resources in the **ResourceTable** can be registered at one time by calling this function:

```
PegResourceManager::InstallResourcesFromTable(
    PegResourceTable *pTable,
    PEGBOOL DeleteOld = FALSE
);
```

WindowBuilder will automatically insert this call to install the application's resources in the **PegAppInitialize** startup function if instructed to do so.

WindowBuilder will create resource tables for you, in either source or binary format. Source format is used if you want to compile and link your resources with your application to produce a self-contained binary image. The binary format resource table is used if you want to load your resources from a filesystem. Note that PEG never requires your target to support a filesystem, so both methods are supported.

8.3 String Tables

A string table is an array of string resources. A string table is a two-dimensional array, with each row holding one string entry and each column representing on language. A string table can have between 1 and N rows (string entries), and between 1 and M columns (languages).

PegResourceManager keeps track of the active string table and the active language within that table. The active language is changed by calling ***PegResourceManager***::**SetCurrentLanguage(PEGINT LanguageId).** When the active language is changed, ***PegResourceManager*** sends a message to all visible PEG objects to inform them that the language has changed. Each object that is aware of its string ID automatically retrieves its new string value and invalidates itself; i.e., the object is redrawn. This makes it very easy to change languages at run time.

String Tables are created and maintained by the WindowBuilder String Table Editor. WindowBuilder can also export and import the string table in UNITEXT and XLIFF (an XML specification for string data exchange) formats.

When you generate a resource file with WindowBuilder, you can choose to include or exclude the string table. If the string table is included, you can

additionally choose which languages to include. This makes it possible to create a binary image that supports a base language or set of languages, and to allow the run-time installation of additional language support.

StringTables are managed by ***PegResourceManager*** in 'pages.' StringTable pages can be concatenated to form the global string table. This can be useful when multiple developers are working on different components of an application, and each developer creates a string table page specific to his or her application component. Each developer is given a base StringID to serve as the starting point for his string table page. At compile time (or run time), these string table pages are each registered with ***PegResourceManager***.

Another option is to merge the string tables created by different developers into one 'master project.' The WindowBuilder String Table Editor provides a facility to merge string tables from multiple projects into one master table specifically for this purpose.

8.3.1 Dynamically Created Strings

Nearly all applications use a combination of both statically defined strings, which are included in the string table, and dynamically defined strings, which are created in memory during program execution.

Dynamic strings can be created using any of the standard methods C programmers are familiar with such as using sprintf or **PegLtoA** (a unicode compatible version of the ltoa() function). These strings are created at run time in some user-defined memory area such as heap memory or possibly on the program stack. All PEG objects that support string display (i.e. all object types derived from **PegTextThing**) support string assignment via either a StringID or a pointer to a string. If the string is assigned via a StringID, the object is aware of this StringID and will automatically switch its string display value if the active language is changed.

If a string is assigned to an object using a string pointer, the object in that case does not have a string ID and the object will NOT change its display string when the active language is changed. Many times this is not an issue, because dynamically generated strings are often numerical display fields that do not change with the language. However, if the string does need to change when the language is changed, the application software in

this case is responsible for generating a new string when the active language is changed.

Note also that all PEG object that support string display support a style flag named ‘TT_COPY,’ which stands for TEXT_THING_COPY. This style flag indicates that the object should make and maintain its own copy of the string assigned to the object. If the string is in the string table, the object has no need to make its own copy of the string and the TT_COPY style is not used. If the string created dynamically is volatile memory, the object must make its own copy of the string when the string assignment is made. In this case, the object should be given TT_COPY style when it is created.

8.4 Themes

Themes are collections of font, bitmap, and color resources. Themes can be changed at run time, allowing your to ‘re-theme’ your application on the fly. Multiple themes can be compiled into your application, or themes can be installed at run time using a filesystem or other storage means. Of course, at the designer’s discretion an application might only have one theme and not support run time theme changes.

WindowBuilder supports generating resource tables that contain one theme, all themes, or any combination of themes. Normally, one resource table is generated for each theme. To change themes, the application simply calls the ***PegResourceManager*::InstallResourcesFromTable** function with the address of a new resource table.

Note that most themes will NOT contain a string table. The string table used by your application is usually changed independently of the color theme. In other words, most applications treat changing languages as a different operation than changing the color theme. WindowBuilder therefore does not generate a separate string table for each theme; rather, it creates one master string table that can contain any number of languages.

A theme can contain any subset of the full system resources Any resources not replaced when a new theme is installed simply continue to use the existing resource.

When you are using WindowBuilder to create your themes, WindowBuilder requires that every theme contain exactly the same number of images, fonts, and colors. These resources will have the same ID in each theme. However, each resource can have its own definition.

For example, a bitmap in the first theme may have the ID ‘BID_BATTERY_ICON.’ If you duplicate this theme, the second theme will also have a bitmap resource with ID BID_BATTERY_ICON. However, the actual image link for this resource can refer to a unique source file in the second theme, meaning that the actual bitmap displayed for this ID differs between the two themes.

8.4.1 Resource ID Name Considerations

WindowBuilder allows you to type any ID name you desire for each instance of each resource type. A few hints might be helpful in this regard:

Resource ID names are converted into simple enumerations in your resource header file. Therefore, in order to avoid compile problems, the name you type should follow standard C variable naming conventions and not use special characters like ‘*’ or ‘(‘.

ResourceIDs should indicate how or where the resource is used, not what it looks like. For example, a color ID name like ‘CID_WINDOW_FILL’ would be a good name, and a name like ‘CID_BRIGHT_RED’ would be discouraged. This is because the purpose of using Resources is that they allow you to change the definition of the color without modifying anything else. If you changed the window fill color from bright red to green, the CID_BRIGHT_RED ID name would not fit any longer. However, this is still the WINDOW_FILL color. If your application supports two themes, the name ‘CID_WINDOW_FILL’ will apply equally to both themes, regardless of the actual color appearance.

Similarly, font ID names like ‘BoldItalic20Point’ are discouraged, and a name like ‘FID_MENU_ITEM’ or ‘FID_CHART_TITLE’ are more useful.

8.4.2 Compressing Resource Data

Resource files produced by WindowBuilder can be in source code format or binary format. They can also be compressed or raw format.

Source code format allows the resource data to be compiled and linked with your application image. Binary format resource files are usually loaded from a filesystem and installed at run time.

Compressed format is used to save file system or nonvolatile memory space. Before a resource can be used it must be decompressed, so there must be enough dynamic memory to hold the uncompressed resources for at least one theme for an application to function properly.

Compressed format can be beneficial if an application supports multiple themes. Only one theme can be active at any given time, so only one theme needs to be decompressed in RAM at any given time.

In a similar fashion, different fonts may be required to support different languages. Since only one language is active at any moment in time, only the font(s) required to support that language need to be decompressed in memory.

To produce compressed resource files, WindowBuilder performs LZW compression on Bitmap and Font data. This compression can dramatically reduce the size of both the application image and the binary resource file; however, it comes with a cost. Before the resource (font or bitmap) can be used, it must be uncompressed in RAM.

Resources can be uncompressed either on first use, or when registered with ***PegResourceManager***. The install flag RF_COMPRESSED indicates to the ***ResourceManager*** that the resource is in compressed form. The flag RF_DCOMP_ALWAYS indicates the resource should be decompressed when it is registered with ResourceManger. The flag RF_DCOMP_ON_USE indicates that ***ResourceManager*** will decompress the resource data the first time the resource is used. These flags are included in the PegResourceTable data structure and are passed to the ***PegResourceManager*****::AddResource()** API function**.**

Decompression of compressed resources requires CPU time. This can be a relatively lengthy (~ hundreds of milliseconds) time if you are using large bitmaps and fonts. The exact amount of time required can only be discovered by actual execution on your hardware using your resources. Be aware, however, that using the RF_COMP_ON_USE decompression method will add processing time when a new bitmap or font is displayed on a screen for the first time.

8.5 Retrieving Resources

You will often need to obtain direct information from a resource by retrieving the resource from ***PegResourceManager***. The APIs used to retrieve a resource are:

PegResourceManager::GetBitmap(PEGUINT ResId);
PegResourceManager::GetFont(PEGUINT ResId);
PegResourceManager::GetColor(PEGUINT ResId);

Additional API functions are provided to get specific details about a particular resource. The following two code snippets are two methods for getting the width and height of a bitmap resource with id BID_ICON:

Example 1:

```
PEGUINT ImgWidth;
PEGUINT ImgHeight;

PegResourceManager::GetBitmapWidthHeight(BID_ICON, &ImgWidth,
&ImgHeight);
```

Example 2:

```
PEGUINT ImgWidth;
PEGUINT ImgHeight;
PegBitmap *pMap;

pMap = PegResourceManager::GetBitmap(BID_ICON);
ImgWidth = pMap->Width;
ImgWidth = pMap->Height;
```

All of the PEG screen driver drawing primitives accept resource IDs as input parameters, and translate to actual resource pointers by calling the ***ResourceManager*** when required.

However, when initializing **PegBrush** color fields, it is important to note that the **PegBrush** requires color ***values***, not color IDs. A PegBrush.LineColor and **PegBrush**.FillColor fields are initialized like this:

PegBrush MyBrush;

```
MyBrush.LineColor = PegResourceManager::GetColor(CID_LINECOLOR);
MyBrush.FillColor = PegResourceManager::GetColor(CID_FILLCOLOR);
```

This, of course, assumes that colors have been registered with color IDs CID_LINECOLOR & CID_FILLCOLOR.

The full PegResourceManager API is documented in the PEG API Reference Manual.

CHAPTER 9

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 ensure that PEG is easily portable to any target environment.

PegScreen is an abstraction, meaning 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 all classes 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. Depending on the hardware platform you specified when you ordered the PEG software, there will be several working examples of ***PegScreen***-derived interface classes for your specific hardware. 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. There are many different PegScreen-derived classes for different hardware graphics controllers, different color depths, and different screen orientations. You will normally include only one PegScreen class in your library build, except for special cases where multiple screen drivers are required because of run time changes to screen orientation or graphics hardware.

Screen Coordinates

PEG screen coordinates are in pixels. When you define the position of a window, button, or any other object, the position is in pixels.

PEG screen coordinates are relative to the upper left corner of the screen, which is 0,0. While this does not follow trigonometric conventions, it 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 object's parent, or relative to some abstract convention of a fraction of an inch. PEG screen coordinates are relative to the upper left corner of the screen, which is 0,0.

If, at some point, you create custom objects with custom drawing routines, you will need to ensure that you always use some corner of the object's client rectangle as the reference point for your drawing routines. This ensures that your drawing will be done correctly no matter where your window or other object is positioned on the screen.

9.1 Graphics 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 x,y pixel resolutions, color depths, and color formats. PEG screen derived classes have been created for a wide variety of color depths and pixel data formats. Examples include monochrome, grayscale, 8-bit palette mode, 8-bit packed-pixel (3:3:2), 16-bit 5:6:5 RGB, 16-bit 5:5:5 RGB, 16-bit 4:4:4:4 (ARGB), 24-bit RGB, 24-bit BGR, and 32-bit ARGB and BGRA data formats.

How Graphics Controllers Work

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

The pixel data stored in the frame buffer and later translated into the analog or digital values used to drive the display device is dependent on the type of display and the mode of the video controller. After determining the color or intensity value for each pixel, an analog 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. A digital interface routes the binary pixel data values directly to the display device, such as an 18-bit 6:6:6 format TFT LCD display device.

If a graphics 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 to the video controller chip. The values retrieved from the palette registers are then shifted to the display device.

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.

Graphics controllers must also provide synchronization signals to the display device to specify the start and end of each graphics frame.

Graphics controllers come in many varieties. They may be an internal submodule of your core CPU, they may be an external dedicated module, or they may be ‘fabricated’ using nothing more than a set of GPIO lines from your main CPU and some very fast timers and DMA shift controllers.

The fact that graphics controllers come in so many varieties is the reason the ***PegScreen*** class is really an API definition class, and derived classes are created to accommodate the requirements of each target hardware system.

9.1.1 Porting PEG to Your Graphics Hardware

For most targets, PEG is delivered with a fully-optimized graphics controller driver. However, in some cases customers decide to do the work of creating a custom PEG driver class for their particular hardware. The remainder of this chapter is intended only for those customers creating or writing their own PEG screen driver.

Porting PEG to run on a custom or semi-custom target platform requires that a new driver class is created by deriving from one the PEG screen driver templates. The screen driver templates are software implementations of each required drawing function, each driver template class being specific to one color depth or frame buffer data format.

Once you have defined your own driver class derived from one of the provided templates, you need to add the following: 1) Software to configure your graphics controller and 2) Software to optimize each drawing primitive to make the best use of your hardware capabilities.

This does not mean that you are required to invent algorithms to meet the requirements of the drawing primitive functions. The required algorithms are provided in the working ***PegScreen*** examples and templates provided with the library. You should always use the nearest provided ***PegScreen*** implementation as the basis for your custom screen driver.

Graphics controller chips are generally not designed to work with only one type of display. All graphics 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 graphics controller registers is called configuring the graphics controller.

The graphics controller and the display device stay ‘in sync’ with each other via horizontal and vertical sync timing signals. The most difficult portion of video controller configuration is ensuring 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 graphics 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. that you intend to use.

9.1.2 PegScreen Driver Templates

Several ***PegScreen***-derived screen driver templates are provided in your PEG distribution. These templates are general-purpose drivers accommodating a wide range of color depths and screen resolutions. The provided template drivers contain everything you need to run PEG on your target. If you are using a high-end hardware-accelerated graphics 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. Optimizing the driver to take advantage of hardware acceleration can be accomplished as your project development progresses.

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. All of the necessary drawing routines are provided and ready to run. You simply have to add the graphics controller configuration code and initialize the graphics frame buffer address. Graphics controller configuration is, of course, specific to the controller in use as the controller frame rate and pixel clock must be matched to the display timing specifications.

Controller configuration is performed either by your system software prior to starting PEG, or in the function named **ConfigureController()**, which is a member function (an empty shell) in each template driver.

If you are using an external graphics controller with a relatively slow (compared to CPU-data bus speeds) interface, it is generally most efficient to create a frame buffer in memory local to the CPU that PEG will draw within. This local frame buffer (or the modified portion thereof) is then transferred to the external controller. This transfer happens in the function named **MemoryToScreen()**, which is provided in each driver template.

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 (bpp) monochrome screen driver. This driver class is named MonoScreen. If your target system is monochrome, you should derive your driver class from this template.

L2scrn (.cpp, .hpp)

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

L4scrn (.cpp, hpp)

These modules implement a linear 4-bpp screen driver template for use with systems that support 16 grayscales or 16 colors. This driver is named L4Screen.

If your target system uses a 4-bpp graphics data format, you should use this driver template.

L8scrn (.cpp, .hpp)

These modules implement a linear 8-bpp screen driver template. This template can be configured for either palette mode or 3:3:2 format packed pixel frame buffer data format.

L16scrn (.cpp, .hpp)

These modules implement a 16-bpp screen driver template, supporting either 5:6:5 or 5:5:5 data formats. If your target system uses a 16-bpp graphics data format, you should use this driver template.

L24scrn (.cpp, .hpp)

These modules implement a 24-bpp screen driver template, supporting both R:G:B and B:G:R data formats. If your target system uses a 24-bpp graphics data format, you should use this driver template.

L32scrn (.cpp, .hpp)

These modules implement a 32-bit ARGB or BGRA format screen driver template. If your target system uses 32-bit graphics data format, you should begin your driver by deriving from this template.

PegScreen Templates for Screen Rotation

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) × 240 (high) monochrome display screen. There is no mechanical reason why this screen cannot be mounted so as to present a 240 (wide) × 320 (high) display to the end user. We refer to this as Profile mode, and the native display mounting as Landscape mode, regardless of relative x,y dimensions of your display.

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.

The rotated screen driver templates are provided for those systems in which the graphics data must be rotated in software.

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

Accelerated PEG Screen Driver classes

A number of hardware accelerated PegScreen-derived classes are available for popular graphics controllers. Of course, the list of popular graphics controllers changes weekly, or so it seems. It is a constant effort for the engineers at Swell Software to create and maintain optimized driver classes for hardware accelerated graphics controllers.

An optimized/accelerated driver is a driver customized to take advantage of hardware acceleration capabilities. Drawing primitives are no longer implemented completely in software as in the templates, but are instead customized to use the hardware capabilities to achieve much faster implementations.

For a list of currently supported accelerated screen drivers contact sales@swellsoftware.com

9.1.3 PEG Palette Considerations

PEG passes color information to the screen interface class through the PegBrush. The PegBrush structure contains data fields of type PEGCOLOR. The PEGCOLOR data type is defined to meet the needs of the target system running at a specified color depth. In other words, PEGCOLOR may be defined as 8, 16, or 32 bits depending on the color depth of the target.

When running with a graphics configuration that supports 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.

For 256 color systems, PEG can operate with a predefined fixed palette, a custom palette generated with WindowBuilder, 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 following 16 entries are varying grayscale levels.

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

Double-Buffered Graphics Output

Double buffering is a term meaning that there are two frame buffers, and the graphics controller is dynamically switched to display data from one or the other buffer. All optimized/accelerated PEG screen drivers utilize double buffering.
Double buffering prevents ‘tearing,’ or the appearance of artifacts on a display that is updating rapidly. In a single-buffered system, the graphics controller is shifting data out to the display device from the same memory area into which you are drawing. Depending on timing, this means that the display may, for at least 1/2 of a frame period, receive part of the previous graphics data and part of the updated graphics data. Double-buffering

prevents this from happening by synchronizing the presentation of new graphics data with the graphics controller EOF (End Of Frame) interrupt.

In a double-buffered system, there are two frame buffers referred to as the ‘working’ buffer and the ‘display’ buffer. The display buffer is the buffer actively shifted to the display device. The working buffer is the buffer into which your drawing operations occur.

When drawing operations are completed, the PEG screen driver swaps the two frame buffers. The working buffer becomes the display buffer, and the display buffer becomes the working buffer. This buffer swapping operation is synchronized with the graphics controller EOF (End Of Frame) interrupt.

The two frame buffers must always be kept ‘in sync’ by PEG. This is because only the modified portion of each buffer is updated during drawing operations. So after each buffer switch, the PEG screen driver copies the modified portion of the new display buffer into the working buffer. This copy is usually done using BitBlt or DMA operations for maximum speed.

CHAPTER 10

DESKTOP SIMULATION

Though designed for embedded devices, PEG is delivered with a set of screen drivers, operating system integration files, and project/build files that that allow you to run the PEG library and your application software on your standard PC desktop. Currently, Microsoft Windows and Linux/X11 desktops are supported. Additional desktop environments may be added in the near future.

These environments allow you to quickly begin using the PEG software on your desktop, and are often used for an extended period while you wait for the arrival of your target hardware***.***

For the Windows desktop, Microsoft compilers version 6, version 7 (.NET) and version 8 (2005) are supported. For Linux desktop, the GNU toolchain is supported.

Remember that you will not have to modify your application level code to move from one of these desktop environments to your final target. You will simply replace the ***PegScreen*** and ***PegTask*** implementations with versions supporting your target hardware and OS.

PEG is also sometimes used for cross-platform desktop application development. In other words, the desktop simulation environment and the final product are one and the same. This allows you to write an application once, and run it in several different desktop environments. It should be noted that the PEG WindowBuilder application program is a PEG application program utilizing the desktop simulation environment. This allows WindowBuilder to run on Windows, Linux/X11, or other desktop environments to be supported in the future.

10.1 PegScreen for Desktop Simulation

When building the PEG library for your desktop, you must include the base ***PegScreen*** class contained in the file pscreen.cpp. The base class, while abstract, does provide functionality that is required by all implementations.

In addition, you will include one PEG screen driver template in your PEG library build: the template that corresponds to your desired color depth and pixel data format. Finally, one derived ***PegScreen*** class implementation for either Windows or Linux/X11 is included in your library build.

The templates and desktop drivers are named according to desktop OS and color depth. You should include ONLY ONE template driver and ONLY ONE desktop simulation driver in your library build project or makefile.

The template file l8scrn.cpp and driver file winscr8.cpp are used to run in 8-bit color depth under MS Windows.

The template file l16scrn.cpp and driver file winscr16.cpp are used to run in 16-bit color depth under MS Windows.

The template file l24scrn.cpp and driver file winscr24.cpp are used to run in 24-bit color depth under MS Windows.

The template file l32scrn.cpp and driver file winscr32.cpp are used to run in 32-bit color depth under MS Windows.

The template file l16scrn.cpp and driver file x11scr16.cpp are used to run in 16-bit color depth under Linux/X11.

The template file l24scrn.cpp and driver file x11scr24.cpp are used to run in 24-bit color depth under Linux/X11.

The template file l32scrn.cpp and driver file x11scr32.cpp are used to run in 32-bit color depth under Linux/X11.

10.2 Desktop OS Integration Modules

To build and run with the MS Windows operating system, you should include the file winpeg.cpp in your PEG library build. This integration module supports both single-threaded and multi-threaded operating modes, as determined by the PEG_MULTITHREAD configuration flag.

To build and run with the Linux operating system, you should include the file linuxpeg.cpp in your PEG library build. The Linux integration module supports only multi-threaded execution mode.

10.2.1 Drawing in the desktop environments

When running in desktop simulation mode, all PEG drawing actually occurs to a privately allocated memory buffer. After each drawing operation, the PEG screen driver then transfers the updated portion of the private frame buffer to the visible desktop window using a simple bit-blitting operation.

A benefit of this implementation of the PEG screen drivers for desktop simulation is that true WYSIWYG performance is guaranteed. Since PEG is not using the underlying desktop graphics system, everything you see on your desktop appears exactly as it will appear on your final target system.

10.2.2 Tuning your development environment

It is usually desirable to configure you desktop environment to match your final target system as accurately as possible, and PEG includes built-in facilities to support accurate target system emulation.

Screen Size

When running on your desktop, PEG will create a native OS window such that the ‘client area’ of the native window matches exactly your target system screen size and resolution. The default size of this window client area is determined by the x and y resolution settings contained in the pconfig.hpp file. These settings are set in WindowBuilder by editing the Configure|Screen Driver dialog settings.

The size of the target screen can also be specified on the command line when you run your PEG application. The command line syntax for specifying the screen size is:

/XSIZE=xxxx /YSIZE=yyyy

where xxxx and yyyy are the x and y screen resolution of your target system, in decimal pixels. The command line parameters are case sensitive.

Product Skin

You can display an image or wrapper around your simulated screen when running in the desktop environment. This allows more realism in that you can observe your user interface within the frame of your physical product.

You must first obtain or create an image of your product in BMP or JPG format. This can usually be obtained from marketing literature, technical drawings, or by using a digital camera. The image must be scaled appropriately so that the area reserved for your screen is exactly the correct size.

When using a product skin image, you will usually need to specify an offset from the top-left corner of the skin image to the top-left corner of the display screen. In other words, the display screen is not usually at the very top left corner of your product; instead, there is usually some border area or mounting. The offset between the skin image top left corner and the display screen corner is specified in pixels. The command line syntax for using a product image and screen offset is:

/SKIN=pathname.bmp /XOFFSET=xx /YOFFSET=yy

The pathname is the path and filename of the BMP or JPG product image. The x and y offsets for the screen within this product image are entered in decimal pixels.

An example of using a command line to specify screen size, product image, and screen offset is provided in the \wb\proj\example\run.bat batch file included in your PEG distribution.

Catching user input on Product Skin

When using a product skin to wrap your emulated screen on your desktop, a natural extension is to allow the user to click on the input buttons of the product skin and process those input events in an identical fashion as they will be processed when running on your final target.

For example, suppose your final product will provide the user with a navigation input pad supporting up, down, left, right, and center select buttons. When you display an image of your product with your screen placed within this image, you would like to be able to click on the navigation pad buttons of the product image and observe the operation of your user interface, navigating from screen to screen and selecting items exactly as this will function on your final target.

Since PEG cannot know the desired operation of your product's input pad, the library provides a callback mechanism to allow you to handle input clicks in the product skin area. In other words, if you click within the screen area, PEG will handle these clicks normally (assuming

PEG_MOUSE_SUPPORT is enabled). When you click on the product skin, PEG passes these clicks to your callback function.

Your callback function can then translate the click position into an input message and send it to PEG. In this way, you are emulating the operation of your navigation input pad driver on the final target system.

The prototype of your callback function must match this signature:

```
void (*SkinCallbackFunc) (PEGUSHORT wType, PEGINT X, PEGINT Y);
```

To assign your callback function, you must #define SKIN_CALLBACK_FUNC in your pconfig.hpp file. You can do this by using the Configure|Miscellaneneous dialog, and clicking on the User Defined button. This brings up a dialog which allows you to enter any miscellaneous parameters that will be included in your pconfig.hpp configuration file.

For example, supposing that your callback function is named ‘MySkinCallback,’ you would enter the following in your configuration file:

```
#define SKIN_CALLBACK_FUNC MySkinCallback.
```

You would then implement your callback function like this:

```
void MySkinCallback(PEGUSHORT Type, PEGINT x, PEGINT y)
{


}
```

The parameters to your callback function are the input event type, and the x,y position of the input event. The input event will normally be one of three message types: PM_LBUTTONDOWN, PMLBUTTONUP, or PM_POINTERMOVE.

Your callback function must translate the x,y position of the input event into presses of the keys of your product input keypad. This is normally done by creating a table that defines a bounding rectangle for each button, and the message type you will send to PEG for each button press.

An example of a skin callback function is provided in the example \peg\wb\proj\example\callback.cpp.

CHAPTER 11

PEG MULTITASKING

PEG supports three different execution models that can be tailored to your requirements. These execution models are called the ***Standalone*** model, the ***Multithreaded*** 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 Standalone model, PEG runs as a simple super-loop in a standalone system. PEG implements its own message queue facilities and memory management facilities. In this model, PEG calls the user-supplied function named PegIdleFunction() when the message queue is empty and PEG has nothing to do. This allows you to branch to other parts of your application software that are not related to the user interface.

The second model is the ***Multithread*** model. This model is used when running with an RTOS or, optionally, when running with a desktop operating system. 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 is fully aware of resource protection and critical code section protection when running in the ***Multithread*** execution model. PEG utilizes mutexes or semaphores of the underlying operating system to protect critical code sections when running in the ***Multithread*** model. In addition, PEG often uses the intertask communications facilities of the underlying operating system to implement the ***PegMessageQueue*** interface.

Task-safe execution in the multi-threaded model is the reason an operating system integration module is required when running the PEG software with any RTOS. The integration modules are normally provided as part of your PEG software package when you order PEG for a particular operating system.

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.**

11.1 Multi-threaded Model Overview

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. This model can simplify the structure and layout of tasks that must interact with the GUI presentation. The only drawback to this model is that a small amount of overhead is added by PEG to ensure that everything works correctly, and additional message queues are required to ensure that all message processing occurs within the thread of the task that owns a particular window.

When running in the MULTITHREAD model it is still very common to have only one task that runs PEG. The other system tasks take care of jobs that are not related to the user interface. When the other system tasks need to cause a change in the user interface, they do so by ***sending messages*** into the ***PegMessageQueue***, rather than by directly creating graphical objects or directly calling PEG drawing functions.

To summarize, even though PEG allows any number of tasks to create and display windows in the ***Multithread*** model, the best system design often utilizes only one UI task and messaging passing between the UI task and the other system tasks.

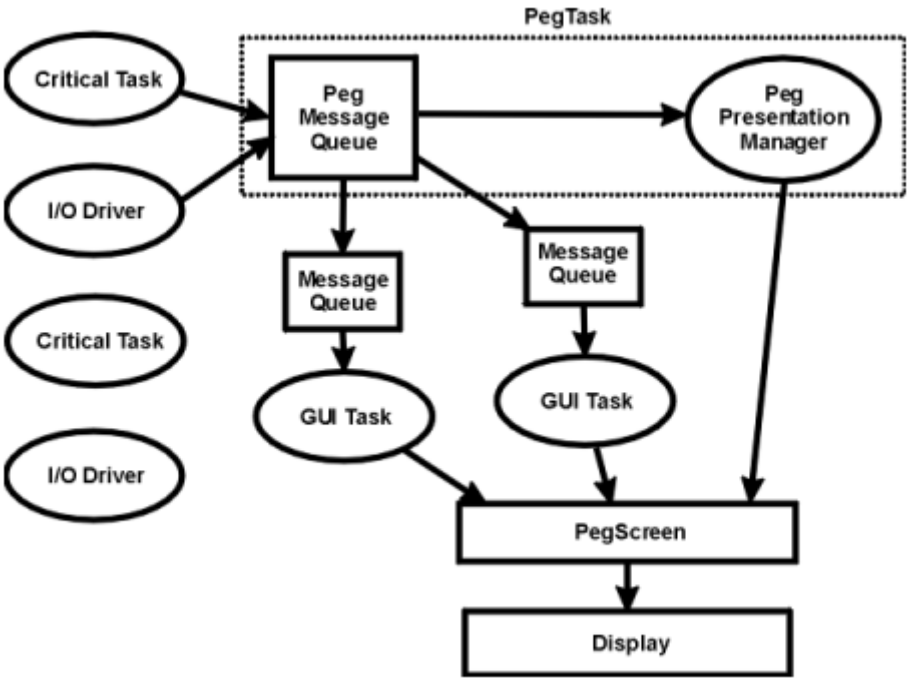
The display screen must be treated like any other single-user I/O device, such as a serial port or printer port. It is not acceptable to have one task sending data to a printer, while a second task preempts the first and begins using the same printer. The final printout would have information from each 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 at a time can actively draw to the screen. Of course, in a multitasking system this exclusion is transparent to the end user, and it appears that many operations are happening simultaneously.

When running under the ***Multithread*** model, PEG internally protects the display screen with a semaphore to ensure that only one task can draw to the screen at a time. Stated another way, PEG prevents any task from pre-empting a drawing task and attempting to do its own drawing. The second task will be suspended or blocked until the first task finishes the drawing operation. The time period during which the second higher-priority task is blocked waiting for the first task to complete is referred to as 'priority inversion.' Priority inversion should normally be avoided. The solution to prevent priority inversion is to organize your tasks such that the higher priority tasks never do drawing, and only tasks of lower priority directly create and interact with the user interface.

The following diagram depicts PEG running under the ***Multithread*** model in a system that uses messages to communicate between high-priority critical tasks the UI task(s), and also has multiple tasks that are directly updating the user interface.



The diagram above depicts a complex system with several critical real-time tasks, I/O drivers, and GUI related tasks. The arrows indicate the general direction of message flow. Note that the critical high-priority tasks communicate with PEG by sending messages.

The above diagram also depicts multiple GUI tasks, meaning that multiple tasks are creating and displaying windows. This level of complexity may not be required for your system, but it is supported by PEG. Your system many only require one GUI task, which is the default PegTask created during system started.

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 tasks 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 competing for the screen resource.

11.1.1 Modal Window Overview

A **PegWindow** or **PegWindow**-derived object (such as **PegDialog** or PegMessageWindow) may be executed modally by calling the PegWindow member function **Execute().** Modal execution means that the user can only interact with this window. The window captures all user input and is forced to stay at the top of the presentation.

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?"

When the **PegWindow::Execute()** function is called, **the call does not return until the model window is closed**. The Execute function actually enters its own local message processing loop. The Execution function does not return to the calling function until message processing returns a non-zero value, which is the value returned to the caller.

In the ***Multithread*** model it is possible to have several tasks each executing a modal window (i.e. multiple tasks have called the **PegWindow::Execute()** function). Modal windows are modal only to the calling thread of execution, not globally modal to all tasks. Think of each task as a completely independent user interface application.

11.1.2 Window Display 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 cannot become fully involved in GUI interactions and message processing.

A task that presents a window in this way 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 that 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());       // periodically update
        pt->IncCount();               // counter value
    }
}
```

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 it 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)
{
    PEGCHAR cTemp[34];
    mlCount++;                      // increment task count variable
    ltoa(mlCount, cTemp, 10);
    CountPrompt->DataSet(cTemp);
    CountPrompt->Draw(); // draws to the screen from secondary
                         // thread!
}
```

11.2 Modal Execution

A window can also 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.

The following is an example of creating and executing a window from within a secondary task:

```
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(); // does not 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. 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.

The **Execute()** function 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.

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.

Sometimes it is desirable to have a window that is 'globally modal'; i.e., the user can only interact with this window no matter how many other tasks may also be displaying modal windows. In that case, the function **PegWindow::GlobalModalExecute()** is used. The GlobalModalExecute() function makes a window globally modal, no matter how many tasks are concurrently displaying modal windows.

If you only have one task that displays user-interface windows, there is no difference between the Execute() and GlobalModalExecute() functions.

11.3 Multithread Execution on the Win32 Development Platform

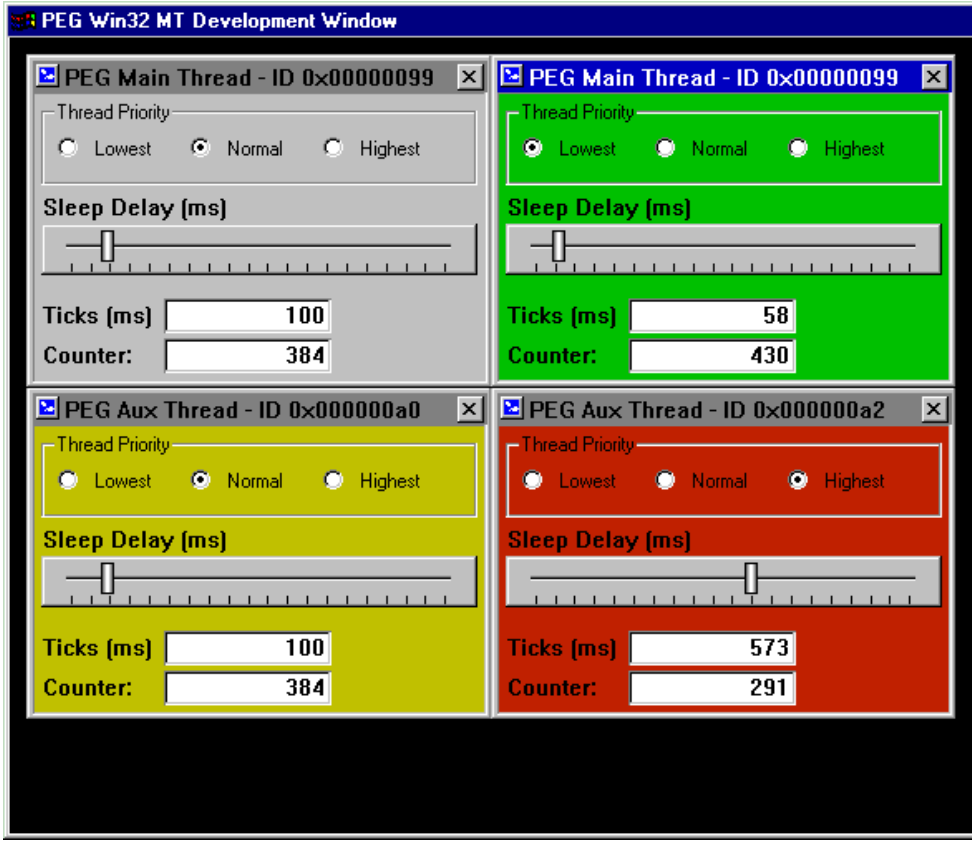
The Multithread capabilities of PEG are available on the Win32 Development Platform for those developers who wish to either simulate their target environment or are planning to deploy their application on a version of the Win32 platform. All of the functionality and threading paradigms discussed in the 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 explaination.

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 within the context of the main PegTask, while the other two are operating within the context of ancillary 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 ancillary thread, and the fourth was added by calling the Execute method of the window from it's 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, CID_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 OSes 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, CID_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);
}
```

You will notice that the first two windows are added directly to the PegPresentationManager. This implies that these windows will execute in the context of PegTask. We then start up an ancillary 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.

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 the lpData parameter is cast to a MultiThreadWindow pointer. We passed this over in the call to CreateThread. For reasons 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 the thread has to the window will no longer be valid. Obviously, we have to consider this situtation.

The small loop that simply increments a counter and updates the data displayed in the window is the real work the thread is doing. Consider that this is only an example of what an ancillary thread is able to do. In a real world application, this 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 the value of the slider has changed. Here is a look at part of the MultiThreadWindow::Message method:

```
PEGINT MultiThreadWindow::Message(const PegMessage& Mesg)
{
.
.
.
    switch(Mesg.wType)
    {
    case PEG_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; furthermore, the ancillary 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 ancillary threads may operate on PEG objects, we'll take a quick look at the remaining two thread callback 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. 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:

```
DWORD WINAPI AnotherThreadProc(LPDWORD lpData)
{
    MultiThreadWindow* pWin = new MultiThreadWindow(290, 210, 0,
        TRUE);
    pWin->SetColor(PCI_NORMAL, CID_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 the window has been removed from the PegPresentationManager and has been destroyed, so, we may safely return at that point.

The last programming concept we will cover here is the relationship between a PEG object, in this case the MultiThreadWindow, and ancillary 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 ancillary 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 ancillary 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 pseudo 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:

```
MultiThreadWindow::~MultiThreadWindow()
{
    if (mhChildThread && mbKillChildOnExit)
    {
        EnterCriticalSection(&mtCountCS);
        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 ancillary threads rather than within the context of a PEG GUI thread. Finally, be sure to protect your data against corruption.

11.4 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).

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 hardware-specific operations and/or an interrupt handler specific to the input device.

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 input events to the closest matching mouse or keyboard events. This is usually very straightforward.

As an example, consider a target system that 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.

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 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 venders 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.

MULTITHREAD support

When the PEG_MULTITHREAD define is enabled, PEG supports the capability of multiple tasks to create windows and interact with the graphical interface. In order to support this programming model, several internal data structures must be protected during critical code sections in order to ensure 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 define the implementation of the protection macros to invoke the proper operation with your RTOS.

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.' 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 macros. They must be defined and implemented to obtain and release a semaphore or mutex on the target system.

The Presentation resource locking can be nested, so the protection mechanism must allow recursive or counted calls to the lock/unlock mechanism from within the same thread. For example, this resource may receive a sequence such as LOCK/LOCK/LOCK - UNLOCK/UNLOCK/ UNLOCK and should handle this sequence appropriately.

Additional requirements for MULTITHREAD support

In addition to the above resource protection macros, supporting the MULTITHREAD model on a new RTOS often 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 *.

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.

As you can see, supporting the MULTITHREAD model can become quite complex and requires a solid understanding of both PEG and the target operating system. Assistance with integrating a new RTOS with PEG is available from Swell Software.

CHAPTER 12

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 API reference manual; i.e., here we concentrate on useful examples, while the primary goal of the API 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 a 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.

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 object has children and who those children are, if the user should be allowed to interact with an object, and so on. These and other properties define how each object will participate in your graphical presentation. It is class ***PegThing*** that maintains this information about each PEG object.

The following sections 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.

12.1 PegThing Members

This section describes each public member function and public variable of class ***PegThing***. Rather than focusing entirely on formal function declarations, parameter descriptions, and return values (as in the PEG Pro API Reference Manual), 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 API reference manual for an alphabetical list of all PegThing member functions.

12.1.1 Constructor(s)

PegThing(const PegRect &Rect, PEGUSHORT Id = 0, PEGULONG Style = 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 Style parameter indicates the object's initial drawing style; FF_NONE indicates that the object will be drawing with no frame (FF_NONE stands for Frame Flag NONE).

PegThing(PEGUSHORT wId = 0, PEGULONG Style = 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 object's 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 object's position is to call the member function **Resize(),** which accepts a **PegRect** argument that 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.

A more direct method of setting an object's position and size is to directly modify the object's 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 ensure 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.

12.1.2 Public Functions

virtual PEGINT 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.

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

The majority of the work of creating a graphical interface using PEG is done by implementing the Message function for your top-level windows. This Message function will receive input events from the child controls of each window, and act on those input events usually by calling additional PEG API functions.

For normal message processing, the Message function returns zero. If a window is executing modally, a non-zero return value indicates to PEG that the window is closing and modal execution should stop. The non-zero return value of the Message function is returned to the caller who invoked the modal window by calling the Window->Execute() function.

virtual void Draw(const PegRect &Invalid)

This function is called by ***PegPresentationManager*** when an object needs to draw or redraw itself. This is the second most commonly overridden function in custom classes created by PEG users, because by overriding this function you can define a custom object appearance.

You have many options when overriding the Draw() function of your base class. Often when you override the **Draw()** function you will first call the base class **Draw()** function to do normal drawing, then you will add your own custom drawing 'on top' of the base class drawing. If you want your customization 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 ensure 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.

The incoming parameter to the Draw() function is the rectangular area that PegPresentationManager has calculated needs to be refreshed. You will pass this rectangle to the BeginDraw() function to inform the screen driver the area to which you will be drawing. All of your subsequent low-level drawing operations will be clipped to this invalid rectangle.

It is normal and acceptable for an object to ignore the incoming Invalid rectangle and always attempt to draw the entire object, regardless of whether the invalid rectangle encompasses the entire object. The screen driver will prevent drawing (and the associated wasted CPU drawing cycles) outside the invalid rectangle. However, for complex graphical objects it is often useful to check the incoming parameter to see which

portions of the object need to be redrawn. The choice of ‘attempt to draw everything’ or ‘just draw what’s needed’ is up to you.

PegPresentationManager calls the Draw() function when an object or part of an object has been invalidated. This call does not happen immediately when the object is invalidated, but happens instead only after all input messages have be processed (refer to the Deferred Drawing section in the PegPresentationManager chapter).

If you want to force an object to update itself, you invalidate the object.

```
SomeThing->AddStatus(PSF_ENABLED); // change the object status
SomeThing->Invalidate();           // invalidate the objec
```

PEG objects invalidate themselves automatically when you use the DataSet or SetColor API functions. For other API functions, such as changes to status or style, you must manually invalidate the object. This is because PEG cannot know which status or style changes cause the object to draw differently, so PEG cannot assume that any status or style change should cause a redraw. PEG always works to prevent time wasted in redrawing objects that do not need to be updated.

virtual void Add(PegThing *Who, PEGBOOL DoShow = TRUE)

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, the object **this** is visible, and DoShow = TRUE, 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, PEGBOOL DoShow = TRUE)

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 order of child objects in a list, such as when adding child objects to a PegVerticalList or PegComboBox.

virtual PegThing *Remove(PegThing *)

This function removes a child object from the current object's child list. This function is the opposite of Add(). Attempting to remove an object that 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.

Remove() does not delete the object after is 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 PEGCHAR *Version(void)

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

virtual PegThing *Find(PEGUSHORT Id, PEGBOOL Recursive = 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 it 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 PegEditField control and testing to see if the PegEditField 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:

```
PEGINT MyDialog::Message(const PegMessage &Mesg)
{
    PegEditField *pString;

    switch (Mesg.Type)
    {
    case PEG_SIGNAL(IDB_OK, PSF_CLICKED):

        pString = (PegEditField *) Find(IDS_MY_STRING);

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

    return 0;
}
```

The Recursive parameter of the Find function indicates whether or not Find() should drill down into children of child objects. For example, to find a top-level window of the PegPresentationManager, without looking at children of the top-level windows, you might do this:

PegWindow *pWin = (PegWindow *) Presentation()->Find(ID_SOME_WIN, FALSE);

The FALSE parameter tells the Find() function to search only the immediate child object of PegPresentationManager, not children of children. Therefore, the above example will return only a top-level window added to PegPresentationManager, and never a nested child object.

virtual void SetColor(PEGUBYTE Index, PEGUINT ColorId)

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 that can be passed in *Index* 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 second parameter, the color ID, is a color ID registered with PegResourceManager. Normally, these color IDs are enumerated in your resource header file, which is generated by PEG WindowBuilder.

Many PEG objects such as ***PegButton*** and ***PegTable*** have additional color indexes associated with them because they require more than 4 unique colors to draw themselves. The API reference manual list the color IDs supported by each object type.

virtual void DrawChildren(const PegRect &Invalid)

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 ensure that objects that have been added to your class draw themselves.

virtual void Resize(const PegRect &Size)

Any PEG object can resize itself or any other object at any time by calling the **Resize()** function. The new screen coordinates for the object are passed in the parameter ***Size***. 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 also perfectly OK to resize an object that is not visible.

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:

```
PegRect Rect;
Rect.Set(0, 0, 99, 99);        // 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.

Using Destroy is the only acceptable method of removing an object from memory. The Destroy function performs several housekeeping tasks, such as making sure any timers owned by the object are stopped and also ensuring that no messages targeted for the object remain in PegMessageQueue.

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.

PEGUSHORT GetType(void)

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

void Type(PEGUSHORT Type)

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

void SetId(PEGUSHORT)

Assigns the value of the object's mId 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.

PEGUSHORT GetId(void)

Returns the value of the object's ID member. The ID value is not used by PEG directly, but it 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 IDs to each object, the application can determine precisely the source of a control notification by requesting the control's ID 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.

PEGBOOL StatusIs(PEGULONG Mask)

This function is used to test individual bits of an object's private mStatus variable. This variable contains system status flags common to all PEG classes. Generally, an application program should never attempt to modify these flags; however, it is sometimes useful to read this value to test for certain object states. The system status flags, defined in the header file pegtypes.hpp, are shown below.

- 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 that 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 ensures 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(PEGULONG Mask)

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

void RemoveStatus(PEGULONG Mask)

The opposite of AddStatus(), RemoveStatus() can be used to clear individual bits or a combination of bits in an object's mStatus variable. This function will logically AND the complement of Mask with the object's mStatus 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 API documentation, and to the examples which follow.

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(PEGULONG Style)

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 object's 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
```

PEGUSHORT FrameStyle(void)

This functions returns the current frame style of an object.

void SetStyle(PEGULONG Style)

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 Style flags supported by each object type are documented in the API reference manual.

void RemoveStyle(PEGULONG Mask)

This function removes the style bit(s) contained in the Mask parameter from the object’s internal mStyle variable.

PEGULONG GetStyle(void)

This function returns the current style flags for an object.

void SetSignals(PEGUSHORT SendMask)

This function is used to modify 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 API Reference Manual lists which signals are supported by each object type. Note that there is a difference between **supported** and **enabled** signals. Note that not all supported signals are enabled (by default) for every object type.

For example, an object type may support the PSF_FOCUS_RECEIVED signal type (all objects in fact support this signal type), but by default this

signal notification is not enabled. If you actually want to be notified when this child object received input focus, you have to enable this notification by calling the SetSignals function for that child object.

The available signal masks are.

```
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
                     // a textobject
PSF_TEXT_EDIT        // sent each time the 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
```

12.2 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, PegTimerManager, PegResourceManager***, 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.

12.2.1 Wrapper function listing

This is a list of primitive functions that are wrapped for convenience by inline functions within class PegThing. There are many additional drawing operations for which no wrapper is provided. Refer to the PegScreen API Reference Manual.

PEGBOOL BeginDraw(const PegRect &Invalid)

Calls PegScreen::BeginDraw(this, Invalid);

void EndDraw(void)

Calls PegScreen::EndDraw();

void Line(PEGINT XStart, PEGINT YStart, PEGINT XEnd, PEGINT YEnd, PegBrush &Brush)

Calls PegScreen::Line(this, wXStart, wYStart, wXEnd, wYEnd, Brush);

void Rectangle(const PegRect &Rect, PegBrush &Brush)

Calls PegScreen::Rectangle(this, Rect, Brush);

void Bitmap(PegPoint Where, PegBitmap *Getmap)

Calls PegScreen()->Bitmap(this, Where, Getmap, bOnTop);

void Bitmap(PegPoint Where, PEGUINT BID)

Calls PegScreen::Bitmap(this, Where,
PegResourceManager::GetBitmap(BID));

void BitmapFill(PegRect Rect, PegBitmap *Getmap)

Calls PegScreen::BitmapFill(this, Rect, Getmap);

void RectMove(PegRect Get, PegRect ClipTo, PEGINT xShift, PEGINT yShift)

Calls PegScreen::RectMove(Get, ClipTo, xShift, yShift);

void DrawText(PegPoint Where, const PEGCHAR *Text, PegBrush &Brush, PegFont *Font, PEGINT Count = -1)

void DrawText(PegPoint Where, const PEGCHAR *Text, PegBrush &Brush, PEGUINT FontId, PEGINT Count = -1)

Calls PegScreen::DrawText(this, Where, Text, Color, Font, Count);

PEGINT TextHeight(PegFont *Font)

PEGINT TextHeight(PEGUINT FontId)

Calls PegScreen::TextHeight(Text, Font);

PEGINT TextWidth(const char *Text, PegFont *Font)

PEGINT TextWidth(const char *Text, PEGUINT FontId)

Calls PegScreen::TextWidth(Text, Font);

void Invalidate(const PegRect &Rect)

Calls PegPresentationManager::Invalidate(this, Rect);

void Invalidate(void)

Calls PegPresentationManager::Invalidate(this, mReal);

void SetPointerType(PEGUBYTE bType)

Calls PegScreen::SetPointerType(bType);

12.3 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 (unless drawing to an unmanaged surface).

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.

12.4 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 with 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 ensure 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:

```
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 to which it has been added. Referring to the ‘tree of visible objects’ described in the PegPresentationManager chapter, 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 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:

```
PegThing *pTest = Parent()->First(); // first child of my parent
PEGINT 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 are 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();
}
```

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 object's 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 specifically for accomplishing this task.**

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

Using Object Types

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

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 ensures 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 ≥ **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 PegEditField 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:

```
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 IDs are:

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

Buttons with the IDs 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 Param 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:

```
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
- PSF_FOCUS_RECEIVED
- PSF_FOCUS_LOST
- PSF_KEY_RECEIVED
- PSF_CLICKED
- PSF_RIGHTCLICK

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.

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 ensure 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):

```
PEGINT 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 PEG_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 following code listings illustrate how to create 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 &, PEGUSHORT wId, char *Text);
        virtual void Draw(const PegRect &Invalid);

    private:
        PEGCHAR *mpText;
};
```

The above class definition tells the compiler that we are defining a new class. The new class will be derived from the PegButton class. 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 that if your parameter list does not match the base class parameter list, the compiler will assume that your are ***overloading***, not ***overriding***, a base class function.

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

```
void RoboButton::Draw(const PegRect &Invalid)
{
    PegBrush Brush;
    BeginDraw(Invalid);       // note 1

    if (GetStyle() & BF_SELECTED)  // note 2
    {
        Brush.LineColor = CLR_BLACK;
    }
    else
    {
        Brush.LineColor = CLR_LIGHTGRAY;
    }
    Brush.Width = 3;

    // draw the top:

    Line(mReal.iLeft, mReal.iTop, mReal.iRight, mReal.iTop,
        Brush); // note 3

    // draw the left:

    Line(mReal.iLeft, mReal.iTop, mReal.iLeft, mReal.iBottom,
        Brush);

    if (Style() & BF_SELECTED) // note 4
    {
```

```
        Brush.LineColor = CLR_LIGHTGRAY;
    }
    else
    {
        Brush.LineColor = CLR_BLACK;
    }
    Brush.Width = 1;

    // draw the right shadow:
    Line(mReal.iRight, mReal.iTop, mReal.iRight,
        mReal.iBottom - 2, Brush);
    Line(mReal.iRight - 1, mReal.iTop + 1, mReal.iRight - 1,
        mReal.iBottom - 2, Brush);
    Line(mReal.iRight - 2, mReal.iTop + 2, mReal.iRight - 2,
        mReal.iBottom - 2, Brush);

    // draw the bottom shadow:
    Line(mReal.iLeft, mReal.iBottom, mReal.iRight,
        mReal.iBottom, Brush);
    Line(mReal.iLeft + 1, mReal.iBottom - 1, mReal.iRight,
        mReal.iBottom - 1, Brush);
    Line(mReal.iLeft + 2, mReal.iBottom - 2, mReal.iRight,
        mReal.iBottom - 1, Brush);

    // fill in the button client area:

    Brush.LineColor = CLR_LIGHTRED;
    Brush.FillColor = CLR_DARKGRAY;
    Brush.Style = PBS_SOLID_FILL;
    Brush.Width = 1;

    Rectangle(mClient, Brush); // note 5

    // draw the text centered:

    PegPoint Put;
    Put.x = (mClient.iLeft + mClient.iRight) >> 1;
    Put.x -= TextWidth(mpText, FID_SYSFONT) >> 1;
    Put.y = mClient.iTop + 1;

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

    Brush.LineColor = CLR_WHITE;
    Brush.FillColor = CLR_BLACK;
    Brush.Width = 1;
    DrawText(Put, mpText, Brush, FID_SYSFONT); // note 6

    // Draw child objects
    DrawChildren(Invalid);       // note 7
    EndDraw();                   // note 8
}
```

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, and to where you will be drawing. If this button is being drawn as part of a larger window drawing operation, the screen driver will recognize that this is a nested call to BeginDraw().

Note 2: 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 and brush style 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. In this example, we are configuring the brush using only predefined color names, rather than color ID values. There are 16 predefined color names, corresponding to the VGA color palette. If we wanted to use other colors, we would need to make a resource file using WindowBuilder, install the color table with PegResourceManager, and use the color IDs when configuring the brush.

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: After the button has completed it’s custom drawing, it calls the PegThing member function DrawChildren() to draw any child objects.

Note 8: 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 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 13

PROGRAMMING WITH PEG

This chapter presents 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 that 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 lowercase, 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.

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 mStatus (**m**ember) 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, #defines, and macros 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; it also contains the definitions used to control the build attributes of the PEG library. ***The header file peg.hpp is the only file you should need to include in your application modules in order to use PEG*.**

A few header files are not specific to class implementations but are globally utilized by the PEG library. For example, the header file **pegtypes.hpp** contains various definitions used globally by all PEG objects. This file also contains the default color definitions used for the various states of each PEG object. You do not have to include pegtypes.hpp separately in your source files, this header is automatically included when you include the “peg.hpp” header file.

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 significant 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***, ***PegScreen, PegResourceManager, and PegTimerManager*** objects must be created. This is usually done in ***PegTask*** by calling the function ***PegCreateFramework()***.

After the required PEG support classes have been created, your function named ***PegAppInitialize()*** will be called by the PEG startup code. This architecture allows you to easily move from one of the PEG development environments to your target platform without modifying any of your application level software. PegAppInitialize is your UI software's 'main.'

PegAppInitialize() is where you install your initial resources, and create the first object or objects that will be displayed by your application. This first object or objects are then added to ***PegPresentationManager*** to create the initial UI display. This might be a temporary splash screen or it may, for example, be the main screen of your application.

13.1 Rules Of Memory Ownership

This section is a brief tutorial regarding the memory management technique employed within PEG. This is required to ensure 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.

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 destroyed. We use the term 'destroyed' to mean that you invoke the ***PegThing::Destroy()*** function to delete the dialog object. This is the function you should always use to remove PEG objects from memory.

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 destroy the dialog object.

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 ensuring 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 once it has been removed (by calling the Remove function). However, the controls that 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 destroy the dialog by calling the ***Destroy()*** function.

13.2 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 easily.

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, 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***.

A further result of the PEG class hierarchy is that it 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. You should see from this that it is very easy to combine different types of PEG objects to create unique appearance. You can 'build up' combinations of display widgets.

You can also derive your own version of PegPrompt or any other PEG display class. Using derivation, you create powerful new object types very easily, by overriding the Message and/or Draw functions (and maybe others) of your base PEG class.

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


AppWindow::AppWindow (PegRect Rect, PEGCHAR *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
}
```

13.3 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 (Destroying) the object. 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 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 two 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 that 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.

In no cases should you ever execute a delete(this) statement. **When in doubt, it is always safe to call Destroy()**, as PEG ensures 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.'

13.3.1 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 require as a parameter a pointer to the PegThing object calling the drawing function. PegScreen requires this pointer to ensure 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 that 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 with 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 can either invalidate the area you wish to

have redrawn, in which case PEG will call your Draw() function automatically, or you can call a function to do drawing directly.

If you want to draw directly, without using PEG's deferred drawing mechanism, you have to calculate the invalid rectangle into which you want to draw, and pass this invalid rectangle as the parameter to your call to BeginDraw().

If you are using deferred drawing, and 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 directly 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)
{
    PegBrush LineBrush(CLR_RED, CLR_BLACK, PBS_SOLID_FILL);
    PEGINT yPos = mClient.Top;

    BeginDraw(mClient); // prepare for drawing

    LineBrush.Width = 0;
    Rectangle(mClient, LineBrush);  // fill with black

    LineBrush.Width = 1;

    while (yPos <= mClient.iBottom)
    {
        // draw red lines:
        Line(mClient.iLeft, yPos, mClient.iRight, yPos, LineBrush);
        YPos += 4;
    }
    EndDraw();
}
```

13.3.2 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 exactly like on-screen drawing. You simply create a DrawingSurface that is a memory bitmap and draw to this surface rather than drawing to a visible surface.

Before you can draw to an off-screen bitmap you must of course create a drawing surface for this purpose. You do this by calling the PegScreen member function **CreateDrawSurface(). CreateDrawSurface()** returns a surface ID, as you would expect. The full prototype for the CreateDrawSurface function is:

PEGINT CreateDrawSurface(PEGINT Type, PEGINT Width, PEGINT Height, PEGINT xOffset, PEGINT yOffset, PEGINT HardLayer = -1, PegThing *pNotify = NULL)

Type is the drawing surface type. To create a memory surface, use PEG_DST_SIMPLE as the Type parameter. If you want drawing to this surface to be clipped to the caller's limits, use PEG_DST_SIMPLE| PEG_DST_CLIPPED as the Type parameter.

Width and **Height** are the drawing surface dimensions in pixels.

xOffset and **yOffset** are the drawing surface display offsets relative to the upper-left corner of the screen. For Memory surfaces, you would generally use 0 for both x and y offsets.

The **HardLayer** parameter is the hardware graphics layer to associate with this surface. For Memory surfaces, this is not normally used and should be set to -1.

The **pNotify** parameter is an optional pointer to an object to be notified when this surface is modified. If this parameter is not NULL, the object pointed to will receive a PM_DRAW_NOTIFY message when the surface is modified. This can be useful if you have one task or object that draws to the surface, and a second task or object that displays the surface bitmap on the visible screen.

Once you have created a memory surface, you can draw into that surface at any time. You draw into the surface by specifying the surface ID when you call the **BeginDraw()** function:

PEGINT BeginDraw(const PegRect &Invalid, PEGINT Surface);

When you call BeginDraw(), the **Surface** parameter defaults to your object's mSurface member variable. However, you can specify any drawing surface when you call the BeginDraw function, and to draw to your memory surface you simply specify that surface ID when you call the BeginDraw function.

After you have drawn to an off-screen memory surface, at some point you normally will want to draw the content of this off-screen surface to the visible screen. You do this by retrieving the PegBitmap associated with the off-screen surface, then drawing the bitmap at any location to a visible drawing surface.

The bitmap associated with a drawing surface is retrieved by calling the **PegScreen::GetSurfaceBitmap(PEGINT Surface)** or **PegScreen::GetSurfaceBitmapAndClose(PEGINT Surface)** functions. The difference between these functions is that the first leaves the off-screen surface active, meaning you can continue to draw to and update the memory surface. The second function retrieves the surface bitmap and closes the memory surface, freeing all memory associated with the memory surface object. You would use the first version if you will be continuously updating the off-screen drawing area, and the second version if you will only drawing to this memory area once.

The following example is a code fragment demonstrating off-screen drawing:

```
void DrawUsingMemSurface(void)
{
   // Create a simple off-screen drawing surface:

   PEGINT MemSurface =
        Screen::CreateDrawSuface(PEG_DST_SIMPLE,
        100, 100, 0, 0, -1, NULL);

   // Open drawing to this surface:

   PegRect DrawRect;
   DrawRect.Set(0, 0, 99, 99);
   BeginDraw(DrawRect, MemSurface);
```

```
    DrawToMem();       // function that does actual drawing

    EndDraw();         // done drawing to memory

    // now put the offscreen bitmap on the screen:

    PegBitmap *pMap =
       Screen()->GetSurfaceBitmapAndClose(MemSurface);

    // Open drawing to visible screen:

    BeginDraw(mReal);   // open drawing to visible screen
    PegPoint Put;
    Put.x = mReal.Left;
    Put.y = mReal.Top;

    Bitmap(Put, pMap);    // draw mem-surface bitmap to visible screen

    EndDraw();            // done drawing
}

// end example code
```

In the above example, the function DrawToMem() is a user-defined function that invokes any combination of drawing API functions to draw to the memory surface. The purpose of the above example code is not to demonstrate actual drawing of which there is an infinite possible variety, but the process and steps needed to draw to an off-screen surface and then transfer this off-screen surface bitmap to the visible screen.

13.4 Object Boundaries

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 subset 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.

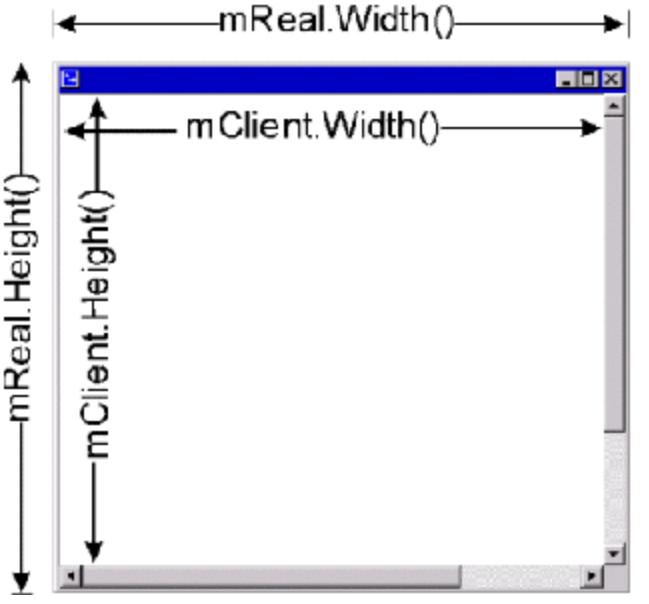
For simple objects such as PegButton and PegEditField, the mClient rectangle is smaller than the mReal rectangle only by the width of the object border. If the object has no border, the mClient and mReal rectangles are identical.

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 these objects to ensure that the mClient rectangle is reduced correctly to allow space for your new non-client area decorations.

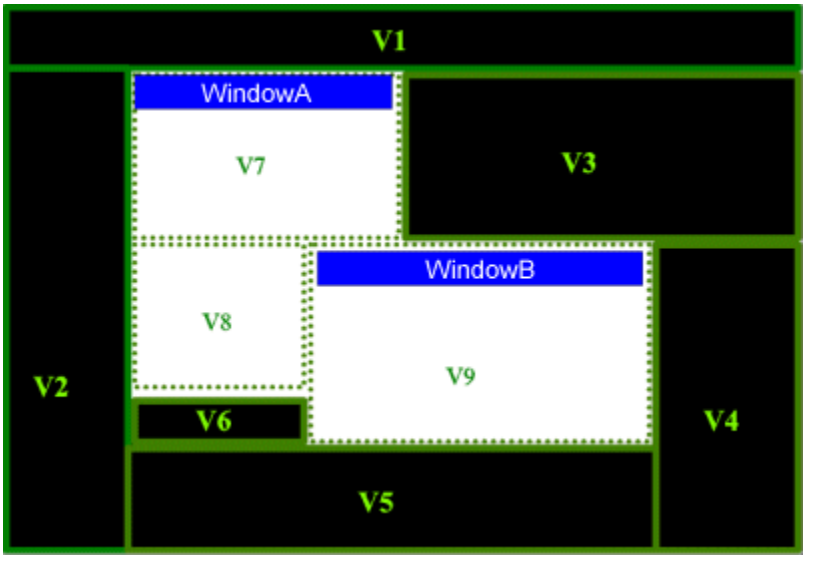
The following diagram illustrates the relationship between the mReal and mClient member rectangles:



13.5 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, of which you may need to be aware. Normally, only PegWindow-derived objects have viewport status. This means 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 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.

Transparent or partially transparent objects should NOT have PSF_VIEWPORT status. PegWindow-derived objects automatically remove PSF_VIEWPORT status when they have AF_TRANSPARENT style.

13.6 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 ensure that you are able to build and execute the PEG demo application found in \peg\examples\pegdemo.

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 ***PEG 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 ***PEG WindowBuilder*** 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 will have to translate the build instructions into instructions that work for your environment.

13.6.1 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:

```
#include “peg.hpp”

enum StringIds {
    SID_HELLO = FIRST_USER_STRING,
    SID_FIRST_WIN
};
```

```
PEGCHAR HelloString[] = {'H','e','l','l','o','
','W','o','r','l','d', 0};

PEGCHAR FirstString[] = {'M','y',' ','F','i','r','s','t','
','W','i','n','d','o','w', 0};


void PegAppInitialize(PegPresentationManager *pPresent)
{

    PegResourceManager::AddResource(SID_HELLO, HelloString);
    PegResourceManager::AddResource(SID_FIRST_WIN, FirstString);

    PegMessageWindow *pWin = new PegMessageWindow(SID_HELLO,
        SID_FIRST_WIN, 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 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).

The first thing to be done in the **PegAppInitialize** function is to register the resources we are going to use with **PegResourceManager**. When you build real applications, you will use WindowBuilder to generate your resource tables. In this example, we simply registered, manually, two string resources with the resource manager. These are the only resources the Hello World application requires.

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 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 into PEG and the PegMessageWindow class you used. All you had to do is create an instance of PegMessageWindow, display it, and let PEG do the rest. Wasn't that easy??

13.6.2 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);
}


// This is the derived window class constructor:

MyWindow::MyWindow(const PegRect &Rect)
    : PegDecoratedWindow(Rect, FF_THIN)
{
    RemoveStatus(PSF_SIZEABLE);
    mColor = 0;
}

// This is the overridden message handling function:

PEGINT 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:
        if (Mesg.Param == 1)  // is this my timer?
        {
            SetColor(PCI_NORMAL, mColor);
            Invalidate();
            mColor++;
            mColor &= 0x0f;
        }
        else
        {
            // not my timer, pass message to base class
            PegDecoratedWindow::Message(Mesg);
        }
        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 one 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. Since a window is always removed before it is deleted, the PM_HIDE system message handler is an excellent place to stop any active timers. If the window is destroyed, PEG will automatically kill any timers associated with the Window.

The PM_TIMER message handler is where we change the window color and redraw the window. A member variable has been defined named mColor, and we will use this variable to keep track of which color to display next. There are always 16 predefined color IDs automatically installed in the resource manager, corresponding to the 16 standard VGA colors; so, for this example, we did not create a color table and register the color table with PegResourceManager. We are simply using the 16 predefined color IDs.

Note that after changing the window color by using the SetColor() function, we have to tell the window to redraw. The window does not automatically redraw since you may make several changes to the window and you do not want each change to cause a window redraw operation.

We tell the window to redraw by calling the **Invalidate()** function. By invalidating the window, we are telling PEG that the window needs to be redrawn.

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);
        virtual PEGINT Message(const PegMessage &Mesg);

    private:
        PEGINT mColor;
};
```

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.

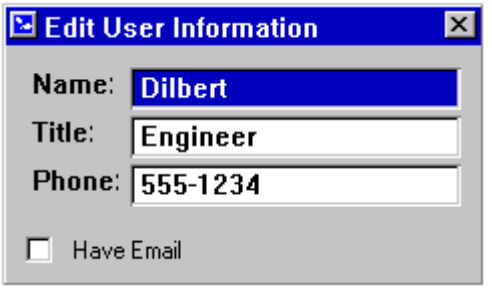
13.6.3 Example 3—More Message Handling and Signals

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:



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 PegEditField 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)
{
PegResourceManager::InstallResourcesFromTable(&Dialog_ResourceTable
);

    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.

```
/*--------------------------------------------------------------------------*/
// DialogWin- example dialog window.
/*--------------------------------------------------------------------------*/

DialogWin::DialogWin(const PegRect &Rect) :
    PegDialog(Rect, SID_EDIT_USERINFO )           /*"Edit User
Information" */
{
    RemoveStatus(PSF_SIZEABLE);

    // add the "Have Email" checkbox:

    PegPoint Put;
    PegRect tempRect;
    PEGSHORT BmpWidth, BmpHeight;

    PegResourceManager::GetBitmapWidthHeight(BID_CHECK_ON,
        BmpWidth, BmpHeight);
    BmpWidth += CBOX_SPACING + TextWidth(LS(SID_HAVE_EMAIL),
        mpFont) + 2;

    Put.Set(mClient.Left + 10, mClient.Bottom - 24);
    tempRect.Set(Put.x, Put.y, Put.x + BmpWidth,
        Put.y + BmpHeight + 2);
   Add(new PegCheckBox(tempRect, SID_HAVE_EMAIL, IDB_HAS_EMAIL));

    // add the phone number prompt and string:

    Put.y -= 34;
    tempRect.Set(Put.x, Put.y,
        Put.x + TextWidth(LS(SID_HEADER_PHONE),mpFont) + 2,
        Put.y + TextHeight(mpFont) + 2);
   Add(new PegPrompt(tempRect, SID_HEADER_PHONE));

    Put.x += 52;
    tempRect.Set(Put.x, Put.y, mClient.Width() - 72,
        Put.y + TextHeight(mpFont));
    Add(new PegEditField(tempRect, SID_THE_NUMBER));

    // add the "Title" prompt and string:
```

```
    Put.y -= 24;
    Put.x -= 52;
    tempRect.Set(Put.x, Put.y,
        Put.x + TextWidth(LS(SID_HEADER_TITLE),mpFont) + 2,
        Put.y + TextHeight(mpFont) + 2);
   Add(new PegPrompt(tempRect, SID_HEADER_TITLE));


    Put.x += 52;
    tempRect.Set(Put.x, Put.y, mClient.Width() - 72,
        Put.y + TextHeight(mpFont)+ 2);
    Add(new PegEditField(tempRect, SID_THE_TITLE));

    // add the name prompt and string:

    Put.y -= 24;
    Put.x -= 52;
    tempRect.Set(Put.x, Put.y,
        Put.x + TextWidth(LS(SID_HEADER_NAME),mpFont) + 2,
        Put.y + TextHeight(mpFont) + 2);
    Add(new PegPrompt(tempRect, SID_HEADER_NAME));

    Put.x += 52;
    tempRect.Set(Put.x, Put.y, mClient.Width() - 72,
        Put.y + TextHeight(mpFont));
    Add(new PegEditField(tempRect, SID_THE_NAME));
}
```

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 ensure 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 in 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 PEG_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 checking for the checkbox to send a message that is has been turned on. The code which follows resizes 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 that 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***.

```
/*--------------------------------------------------------------*/
/*--------------------------------------------------------------*/
PEGINT DialogWin::Message(const PegMessage &Mesg)
{
    PegRect NewSize;

    switch(Mesg.Type)
    {
    case PEG_SIGNAL(IDB_HAS_EMAIL, PSF_CHECK_ON):

        // make myself taller:

        NewSize = mReal;
        NewSize.Bottom += 24;
        Resize(NewSize);

        // add the new email string:
```

```
        PegRect tempRect;
        tempRect.Set(mClient.Left + 10, mClient.Bottom - 24,
            mClient.Left + 10 +
            TextWidth(LS(SID_HEADER_EMAIL),mpFont) + 2,
            mClient.Bottom - 24 + TextHeight(mpFont) + 2);

      mpEmailPrompt = new PegPrompt(tempRect, SID_HEADER_EMAIL);
        Add(mpEmailPrompt);

        PegPoint TopLeft;
        TopLeft.x = mClient.Left + 62;
        TopLeft.y = mClient.Bottom - 24;
        tempRect.Set(TopLeft.x, TopLeft.y,
            TopLeft.x + mClient.Width() - 72,
            TopLeft.y + TextHeight(mpFont) + 2);
       mpEmailString = new PegEditField(tempRect, SID_EMAIL_ADDR);
        Add(mpEmailString);
        break;

    case PEG_SIGNAL(IDB_HAS_EMAIL, PSF_CHECK_OFF):

        // make myself shorter:

        Parent()->Invalidate(mReal);
        NewSize = mReal;
        NewSize.Bottom -= 24;
        Resize(NewSize);

        // get rid of the email prompt and string:

        Destroy(mpEmailPrompt);
        Destroy(mpEmailString);
        break;

    default:
        return(PegDialog::Message(Mesg));
    }

    return 0;
}
```

13.6.4 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 **drawwin_res.cpp** to your project. Drawwin.cpp contains the C++ source code, and drawwin_res.cpp contains the string and bitmap resources that are required by this application.
- Make sure that your 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)
{
   PegResourceManager::InstallResourcesFromTable(
       &drawwin_ResourceTable);

    PegRect Rect;
    Rect.Set(180, 80, 468, 300);
    pPresentation->Add(new DerivedWin(Rect));

}
```

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 registering the application resources and 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.

The source code for DerivedWindow is also included in the file **drawwin.cpp**. The declaration of this class in 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 redraw itself on the screen.

Take a close look at our new **Draw()** function. You should see a line that contains:

```
PegWindow::Draw(Invalid);
```

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.

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 ***PegThing***s, 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 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

Create a new bitmap for filling the window client area using PEG WindowBuilder. Generate a new resource file and run the program to ensure that your new bitmap is now used to fill the client area.

13.6.5 Additional Example Programs

Your PEG library distribution contains several additional example programs. The subdirectories 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 in each of the supported desktop environments and project files and/or makefiles are provided to build each example using

In the base directory of each provided example program is a file named ‘readme.txt.’ This file describes the example program in terms of what the program accomplishes and any special PEG library configuration settings required to build the example.

13.6.6 Final Notes

In this chapter we have illustrated some of the most common PEG programming tasks. As you will see when you begin using PEG WindowBuilder, many of these tasks can be accomplished by simply setting the appropriate configuration information from within PEG WindowBuilder. However, you now know what is happening ‘under the hood,’ and you will be able to extend the functionality provided by PEG WindowBuilder whenever required.