C/PEG™

Portable Embedded GUI

Programming Manual

Third Printing

June 2006

Swell Software, Inc.
2920 Pine Grove Ave
Port Huron, MI 48060
PH: (810) 982-5955
FAX: (810) 982-5949

info@swellsoftware.com

TABLE OF CONTENTS

FORWARD

We at Swell Software thank you for choosing C/PEG!

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

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

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

How are the manuals organized

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

The remainder of the Programming Manual provides an 'under the hood' view of C/PEG library internals and introduces basic concepts that are needed to fully understand how C/PEG works. You will need to read and

understand this material before you begin serious development of your application level software. This is followed by descriptions of the fundamental C/PEG functions. These descriptions contain many working examples that will prove valuable to you as you begin writing your own system software.

The second manual, the PEG Development Toolkit User's Manual, describes our supporting cast of utility programs. These include PEG FontCapture, PEG ImageConvert, and PEG WindowBuilder. The appendices describe the PEG installation directories and the example programs.

The third manual, C/PEG API Reference Manual provides extensive information about the fundamental C/PEG functions. This manual details the Application Programming Interface (API) of the C/PEG graphics library. It is intended as a quick reference guide for developers which may already be familiar with how C/PEG works and may need to review details on individual functions.The Reference Manual is also provided in interactive PDF format. We believe the PDF format class reference is more convenient to use on a day to day basis than the printed manual. This approach also works well in that as you read this manual you are not overloaded (no pun intended!) with member function names and descriptions. Instead, we encourage you to first concentrate on obtaining a high-level understanding of how C/PEG works. Later, as you begin working on your system software, you will probably want to keep the API Reference Manual open at all times.

The Programming Manual and PEG Development Toolkit User's Manual are also provided in PDF format, and the PDF format often contains last minute changes or additions that you will not find here. These additions will be noted in the Release Notes. The online manuals are found at the following address:

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

Username and password are required to download the manuals.

INTRODUCTION

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

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

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

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

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

We believe that you deserve a better solution!

What PEG IS

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

PEG is Portable

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

PEG is Embedded

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

PEG is GUI

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

In addition to the class library itself, PEG provides all of the other tools, documentation, and support you will need to construct a custom graphical interface for your project. This includes utilities for generating graphical fonts ***(PegFontCapture)***; processing, optimizing, and compressing graphical images ***(PegImageConvert)***; and ***PEG WindowBuilder***, a RAD prototyping tool for use with PEG. With the class library and related tools,

PEG without question provides the most powerful, professional, and complete GUI solution available to real-time embedded system developers.

What PEG is NOT

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

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

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

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

Where PEG is going

Over the near term, the core PEG library will continue to grow and improve in terms of the native control types that are available and the flexibility of those controls. Over the longer term, we plan to add entirely new groups of object types that we feel would be useful for embedded systems. Of course, the long-range development list also depends on input we receive from you, the developer using PEG. In any event, the basic library will retain the ability of allowing you to remove unwanted components in order to build a system that exactly fits your size and performance requirements.

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

PEG WindowBuilder, our newest and most powerful support utility, is continually being enhanced to meet the needs of PEG users. Animation, Navigation, Background Images, Hotspots, XLIFF Import and Export, generation of string resource files and zoom capability are just some of the new features as of this manual printing.

Library Updates

Library updates are posted on the Swell Software www site roughly every 90 days. If you are a PEG customer, you are entitled to a minimum of six months of technical support and library updates. The 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. This website has gone through an enhancement phase and now includes a Frequently Asked Questions (FAQ) page with useful contributions from current PEG users.

CHAPTER 1

SYNOPSIS

1.1 What C/PEG Is

C/PEG is a high performance graphics application framework library for resource conscious embedded systems. C/PEG provides the embedded application developer with a full tool set of graphics primitives and high level control objects to easily and quickly construct user friendly, graphical interfaces for any type of embedded device.

The core of C/PEG is written in ANSI conforming C. The architecture of the library is very straight forward and easy to understand for any moderately experienced C programmer.

The C/PEG library is self contained, singly providing all the functionality necessary to create real-world object applications. The core library does not depend on any particular functionality that may or may not be provided by a compiler library or operating system.

C/PEG is designed from the ground up to be portable across operating systems and hardware platforms, thus giving the embedded systems designer the discretion of selecting the technology that best suits the system's application. The core library is platform agnostic and all of the library's extensive functionality is equally available across the entire range of supported hardware and operating environments.

C/PEG is extensible. The library is designed to let a programmer of any skill level easily extend or even replace the behavior or visual representation of library objects. If a stock object does not suit your liking, you are free, even encouraged, to mold the object into a form that better fits the product. This gives embedded systems designers the right tools for building products that stand out from competitors and differentiates the company making the product from the rest of the market.

C/PEG is scalable. From the smallest embedded system with the most modest available resources, all the way up to advanced medical and avionics devices featuring desktop level power, C/PEG is the right graphical application framework. This allows you to select the best hardware platform

for the product without the question of whether or not your GUI has the ability to come along for the ride.

1.2 What C/PEG Is Not

C/PEG is not a desktop graphical user interface that has been 'stripped down' in an attempt to fit into the embedded systems market.

C/PEG does not hide the system from the programmer. It does not attempt to wrap and internalize system functionality that is best left to the application developer. C/PEG deals strictly with the graphical user interface of the system. All other functionality for the application is left to the imagination and skill of the developer. In other words, C/PEG does not force a 'you can do anything you wish, as long as it's what we wanted for you to do' type of development environment.

1.2.1 How C/PEG Works

The execution model of C/PEG is very familiar to any programmer who has ever worked with message driven application models, such as the venerable Win32 API or the mature X Window System. In this model, user interface objects participate in a messaging loop whereby the system notifies the object of certain events. These events may be a by-product of user interaction with the system hardware, such as incoming key or pointing device data; operating system notifications, such as timers; or from interrupts coinciding with external device status. All of these events cause C/PEG to 'do something'. When these events are dormant, so too is C/PEG.

This model also facilitates an easy to understand and highly structured line of communication between C/PEG application objects. It is a very simple process for C/PEG objects to notify any other C/PEG objects with any type of system or user defined message.

1.2.2 Benefits of using C/PEG

Building off of the success of the original C++ version of PEG, C/PEG already has an impressive pedigree and proven track record of enormous benefits to the embedded systems application developer community.

In this day of hurried product development cycles, C/PEG is a tremendous asset to any development team. By providing both low and high level

graphics functionality, application programmers are free to roll up their sleeves and customize the graphics display in any way they like without sacrificing the ability to use familiar GUI objects like text buttons and scroll bars. This translates into a faster time to market with a custom user interface that propels the product past the competitors.

All of this power comes in a small package that won't break the product budget or the chosen hardware platform.

1.3 High Level Overview

1.3.1 Graphics Objects

The C/PEG graphics library is a modular application framework built for embedded systems, and while the code is written in ANSI compliant C, it utilizes some basic principles introduced in C++.

First, the library is logically partitioned into structures. These structures hold both data and pointers to functions. In some ways, these structures resemble a C++ class. But, unlike the typical usage for a C++ class, all of the members of these structures are globally visible. The architecture is designed to allow for maximum application developer flexibility without adding unnecessary overhead. For the sake of discussion, we will refer to these structures as objects.

These objects are then ordered in a hierarchal tree, beginning with the PegThing object, whereby one object takes on the characteristics and functionality of another object, then extends or supplants these characteristics and functionality to create a new object that will fulfill the needs at hand.

For example, the PegThing object implements data members and function pointers that deal primarily with status, ownership and position. Every other object in the C/PEG library is a derivative of PegThing. In other words, every other object in the C/PEG library has the same data members and function pointers laid out exactly the same as PegThing. This allows every object to be a member of a list of objects, own a list of objects, and be aware of it's own state and position on the screen. In terms of C++, this would be termed as inheritance where objects build off the foundation laid down by other objects. This term, in it's classical sense, does not apply too well to the C/PEG architecture. A C/PEG object does not have a compiler determined virtual function table and therefore does not support the early/

late binding paradigm set down by C++. C/PEG objects are collections of data and functions that can be used to easily build new objects independent of the compiler.

Hardware Support

The functions and data that implement the necessary functionality to drive hardware are not part of the object hierarchy of C/PEG objects that begin with PegThing. These objects obviously do not need the same sort of characteristics as an object that is expected to be part of a list of objects that are displaying on the screen.

The functions and data that handle the video hardware are collectively referred to as the PegScreen object. The PegScreen object implements the basic functionality of tracking the invalid regions of the screen, clipping draw operations and providing primitives with which the PegThing derived objects use to display themselves. The PegScreen object also provides a framework for the hardware dependent code that must be filled in for any particular piece of display hardware. This provides a very structured and reliable way to port C/PEG to new video hardware.

A set of display drivers that extend the PegScreen object that are able to actually draw primitives, text and bitmaps are referred to as templates. These templates are arranged by the color depth they support and screen orientation, either landscape or portrait. The templates can do most of the work for any given piece of video hardware, for a given color depth and screen resolution short of actually configuring the hardware and establishing the location of the frame buffer.

The last link in the chain is referred to as the driver. This builds on the template and implements the code necessary to configure the video device as well as discovering the memory location of the frame buffer. These drivers may also implement functionality for drawing primitives, text or bitmaps based on the feature set provided by the particular video hardware. For instance, if the video hardware supports hardware line drawing, then the driver will override the line drawing implemented in the template to provide a more efficient version that takes advantage of the hardware's acceleration scheme.

Comparison with PEG+

C/PEG and PEG+ share many common characteristics. The C/PEG library was developed by the same engineers that first developed PEG+ back in

1997. A great deal of the code contained in C/PEG came directly from PEG+, giving C/PEG a maturity beyond it's age.

But, while the focus of PEG+ is it's ability to scale up to incorporate leading edge features and technologies such as alpha blending and OpenGL, the focus of C/PEG is to scale down to provide even the smallest embedded systems with a complete, efficient and easy to use graphics library.

To that end, C/PEG includes great support for drawing primitives, text and bitmap rendering as well as higher level objects such as buttons and scroll bars. At the same time, C/PEG does not include features that would make it prohibitive to it's target audience.

To summarize, if your embedded application needs a desktop like graphics interface, then you need PEG+. But, if your embedded product needs to display text and bitmaps and would benefit from a good set of higher level objects without sacrificing speed and size, then C/PEG is for you.

1.4 Supported Platforms

Operating Systems

- Windows
- Unix (Linux, Solaris, LynxOS, NetBSD)
- ThreadX (Express Logic)
- smx (MicroDigital)
- OSE (Enea Embedded Technology)
- Stand alone (no operating system required)
- MQX (MQX Embedded)
- eCosPro (eCosCentric, Ltd.)
- PharlapETS (Ardence, Inc.)
- µC/OS-II (Micrium)
- RTXC & RTXC Quadros (Quadros Systems, Inc.)
- CMX-RTX (CMX Systems, Inc)

Processors

- x86 (Intel, AMD)
- ARM (Intel, Cirrus Logic, Sharp, Freescale)

- Coldfire (Freescale)
- PXA (Intel)
- PowerPC (Motorola)

Video

- ARM
- x86 VGA
- Linux frame buffer device
- Epson
- C&T
- ATI
- PowerPC
- Noritake

CHAPTER 2

COMMON TERMS AND CONCEPTS

This section introduces some of the basic concepts and terminology used when discussing the C/PEG library. If you are an experienced C programmer and you have worked with desktop windowing interface software, then much of this material will be familiar territory; but, you are still encouraged to skim through this section to attain a complete grasp of the terminology used throughout this manual is invaluable.

2.1 Structured C Architecture

The core of C/PEG is written in ANSI C. Although this is a moving target, C/PEG is designed to fit the 'lowest common denominator' functionality offered by compilers, compiler libraries and operating systems. After all, the 'P' stand for 'Portable'. If C/PEG doesn't compile and run for every embedded platform available to the C programmer, then it's not doing it's job.

C/PEG isn't fancy about the implementation. It is straightforward, clean and robust. This allows the library to be quickly ported to new platforms, as well as providing a clear road map to extendibility.

Most C programmers groan at the idea their venerable language could be confused in any way with C++. That C++ may be an acceptable language for desktop type applications, but the overhead and, sometimes unpredictable, compiler support for embedded systems demands that C++ and its tenants not be allowed in their world. While, on the other hand, the paradigm introduced by object oriented languages, while not the savior of modern day software, does have it's place and does provide a natural and intuitive way to break down a problem into manageable chunks.

That said, C/PEG uses the C language in a manner most C programmers will understand and find satisfactory.

2.1.1 Structures as Objects

Most graphical interface software written over the past twenty years have an API grounded in the C language. These API's implement dozens or even hundreds of functions to present a powerful environment to achieve incredible graphical applications. Over the last decade, there have been many attempts by many companies to wrap these C language API's into an object oriented C++ blanket. Some are successful, while most are not.

One of the main reasons why most wrapper API's fall by the wayside is they remove the real API from the programmer and, therefore, fall short in providing all of the functionality of the API. This is usually due to the fact it is extremely difficult to wrap a scattered set of functions cleanly into compartmentalized object programming. Any programmer who cut their teeth in graphics programming with the Windows API, then was moved to the macro fortified and obfuscated world of the Microsoft Foundation Class architecture understands this frustration.

C/PEG avoids this problem by being designed from the ground up with the notion that the continuity of the API is the pearl in the oyster. Also, instead of burying the low level functionality of the graphics software, it should be presented to the application developer in a straightforward and intuitive manner that can easily, and safely, be exploited.

So, the big question is: How is this done?

C/PEG logically partitions functionality to present an easy to understand API which encourages exploration by the application developer. The partitioning is focused on melding functionality, that together, is greater than the sum of its parts. Every graphical element that is able to display itself on the screen at run time shares common functionality that is the foundation on which all elements are built. Once this foundation is understood by the application developer using C/PEG, it is a small step to becoming a real graphical interface guru.

In this manual, we will use the term object to a great extent. Don't be put off by this. This is not an inference to a C++ object with it's templates and virtual function tables. Rather, the term object is used to describe collective functionality and data. In C/PEG, these are implemented as nothing more than structures. Each C/PEG graphical object is formed by adding structure elements together to reach the desired level of representation and functionality.

The term derived is also used extensively. Again, this is not meant to imply the C++ meaning of this term. It is intended to convey the idea that one structure shares common data and functionality with a more basic structure. In C/PEG, the basic structure that represents the data and functionality shared by all graphical objects is named PegThing. Every structure that is able to display itself on the screen shares the same data and functional elements found in the same order as they are represented in PegThing. In C/PEG terminology, this is what is meant by derived. It is proper to say PegTextThing is derived from PegThing because PegTextThing holds all of the same members in the same order as PegThing. Thus, a PegTextThing can be safely cast as a PegThing when it is necessary to access members of the PegTextThing structure that are derived from PegThing.

Function Pointers and Callbacks

Bundled in with the data elements of C/PEG objects are function pointers. These function pointers represent the basic functionality of a C/PEG object: how the object is drawn, how the object handles messages, adding and removing objects and so forth. The PegThing object implements a version of this functionality which is used by most every object that is derived from it. Some objects selectively implement their own versions of the functions to achieve a different look and feel than what is already provided.

What this means to the application developer is they are free to create new object types and selectively override, or replace, the default functions with versions of their own which provide the functionality necessary to solve their problem. So, when the term override appears in this manual, this is the type of activity to which is being referred.

For an example, if the application developer wishes to make an object draw differently than a similar object that is already in the C/PEG library, he only needs to prototype a new function that follows the default draw function prototype and assign that function to a C/PEG object's draw function.

In some respects, this may appear to be a callback mechanism, and in loose terms it may fall into that category. The difference here is the state and functionality of an object are self contained. In other words, given the above example, when the draw function is replaced, there is no callback table or some other sort of business going on under the sheets that obfuscates this procedure from the application developer. The mechanism is simply assigning a function to a function pointer within a structure. So,

when that object is being dealt with by any other object, the correct drawing function will be called for the object without the need for any ancillary data.

2.1.2 Graphical Interface Terminology

This section introduces some terms which may be new to programmers not familiar with graphical programming.

Panel and Control

In most graphical API's developed in the past, there are generally two types of graphical elements: windows and controls. Windows are often referred to as top level graphical objects that act as containers for controls, with the controls being child objects of the window. This is often a convenient way to refer to collections of similar objects.

In C/PEG, the concept of a window is replaced by a panel. The PegPanel object implements the functionality necessary to be a top level object and to have child objects with which the user interacts. The name panel was chosen over window because the term window is a very powerful metaphor that implies all sorts of things that may be detrimental in understanding just what is meant by a top level object in C/PEG. For instance, a window is generally thought of as a moveable, sizable, framed and titled object the user can minimize, maximize and close at will. In C/PEG, none of these attributes fit into a panel. A panel is completely controlled by the application developer, giving the application developer complete control over the flow of the application.

Parent, Child and Sibling

These terms refer to the relationship between the panels, controls and other items that are all part of the interface. A control that is attached to a panel is termed a child of that panel. Likewise, the panel that contains the control is termed the parent panel. If there are several controls attached to the same panel, those controls refer to each other as siblings.

While we have just described the most common case, there is nothing internal to C/PEG that prevents a panel object from being a child of a control object. 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 with the same panel, or even within a single application. C/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 C/PEG, because it allows the application developer to reuse custom objects that are created in a variety of ways.

Modal Execution

A panel is said to be executed modally when that panel must be closed or completed by the end user before other panels are allowed to receive any user input. In C/PEG, any panel may be executed modally. In fact there can be several modal panels operating at one time in certain multi-tasking environments. Modal panels capture all input devices, preventing other panels and controls from being active while the model panel is executing.

CHAPTER 3

C/PEG PROGRAMMING REFERENCE

3.1 Building the C/PEG Library

Building the library is a very simple process. For most platforms and compilers, a build file is supplied with the C/PEG distribution. These build files are located in a directory below `cpeg/build` that matches the platform and compiler you are using to build. All of the C/PEG source files can be found in the `cpeg/source` directory. Likewise, the header files are in the `cpeg/include` directory.

The section begins by describing the library configuration flags, which are flags used to include or exclude various components and features in C/PEG. You may need to modify at least a few of these flags before you build the library.

Library Code Size

The code size of the library can vary depending on which resources your application is using and how you have configured the library as well as the compiler, optimization levels and CPU being used.

Since a typical application may not use all of the C/PEG library objects, it is more useful to refer to the typical footprint of the library on an embedded target. The typical footprint is a measure of the ROM or FLASH space actually used by the C/PEG library when linked with your application modules. The measure of the typical footprint is derived from several real-world embedded systems using the C/PEG library. The typical footprint of the library is between 60K and 100K bytes, and can be as small as 48K bytes, depending on which features of the library your application is using.

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

File Naming Conventions

Every effort has been made to insure all C/PEG source files are named in a consistent manner to avoid problems encountered when moving between Windows and Unix environments. All C/PEG source files use the standard DOS 8.3 file name format, and long file names are not used. The source files use only lower case names to avoid problems with case sensitive file systems.

Finally, for Windows users, the C/PEG distribution includes source files in CR+LF format, while the Unix distributions include source files in CR only format.

Build Options

Before building the library, it is necessary to properly set the compiler directives contained in the C/PEG configuration file. This file, `pconfig.h`, is located in the `cpeg/include` directory. These directives determine the target environment for the library, along with other options such as input device drivers, screen drivers, screen resolution and tasking model.

Throughout this manual, the library configuration flags and how they affect library operation are frequently referenced. These configuration flags are simply defined symbols that are listed in the `pconfig.h` file. The terms “turn on” and “turn off” when used to describe these symbols are a reference to either defining or not defining the symbol in the header file.

As delivered, C/PEG is configured to build a library that is suitable for your host development system. This could be either Windows or a Unix variant. This configuration allows you to quickly begin using and experimenting with C/PEG without being concerned about target specific issues.

It is not necessary to fully tune your C/PEG configuration before building the library for the first time. If you wish to begin working as quickly as possible, you may build the library as it is installed from your distribution. You can always return and modify the configuration flags and rebuild the library.

Target Platform

C/PEG is delivered to support your host development environment and target platform. It is suggested you initially build C/PEG for your host development system as these environments allow you to quickly begin using C/PEG and creating your user interface software.

Using C/PEG on a target platform other than those currently supported may require minor modifications to the RTOS interface functions and hardware drivers.

Table 1 shows the list of symbols that configure C/PEG to run with a specific operating system. It is important to note only one of these may be defined at one time. If you are using an operating system that is not yet directly supported by C/PEG, it is usually easiest to simply build C/PEG for stand alone operation and run C/PEG as a single task. This can be enhanced at any time to support true multi-tasking operation as described in later sections.

The primary difference between running stand alone and running with an operating system integration is the method C/PEG uses to provide messaging services. When in stand alone mode, the messaging services are implemented completely by C/PEG. For operating system integrations, the messaging services are implemented using thread safe services provided by the operating system.

A second difference between these configurations is the ability of C/PEG to allow multiple tasks to access the GUI API. Since synchronization is normally a trait of the operating system, when C/PEG is configured for stand alone use, there are no such services available. But, when C/PEG is integrated with an operating system that provides this functionality, C/PEG resources are protected and allow for multi-tasking operation.

Symbols	***OS***
PEG_OS_NONE	No RTOS or unsupported RTOS
PEG_OS_WIN	Windows
PEG_OS_SMX	smx
PEG_OS_THX	ThreadX
PEG_OS_KWIK	AMX
PEG_OS_ETS	ETS
PEG_OS_RTXC	RTXC
PEG_OS_QUADROS	RTXC Quadros
PEG_OS_OSE	OSE
PEG_OS_ONTIME	OnTime RTOS-32
PEG_OS_ECOS	eCos
PEG_OS_RTX	CMX-RTX
PEG_OS_DOS	PEGStandalone for DOS

Symbols	*OS*
PEG_OS_NUCLEUS	Nucleus
PEG_OS_MQX	MQX
PEG_OS_UCOS	uC/OS
PEG_OS_INTEGRITY	INTEGRITY
PEG_OS_UVELOSITY	uvelOSity
PEG_OS_LINUX	Linux
PEG_OS_X11 and PEG_OS_LINUX	Linux with X11
PEG_OS_LYNX	LynxOS
PEG_OS_X11 and PEG_OS_LYNX	LynxOS with X11
PEG_OS_SOLARIS	Solaris
PEG_OS_X11 and PEG_OS_SOLARIS	Solaris with X11
PEG_OS_QNX	QNX
PEG_OS_X11 and PEG_OS_QNX	QNX with X11
PEG_OS_NETBSD	NetBSD
PEG_OS_X11 and PEG_OS_NETBSD	NetBSD with X11

Table 1 (OS Symbols)

Screen Driver

There are a number of screen drivers available for the C/PEG library. The actual driver to use for the specific platform is determined by the selected operating system and color depth, often referred to as bits-per-pixel or bpp, at which to run.

Selecting the operating system is described in the previous section. Selecting the color depth at which to run is done by defining PEG_NUM_COLORS to the appropriate number of colors supported by the platform. For the Windows and X11 screen drivers, PEG_NUM_COLORS is by default, set to 256, but it may be changed to any of the available options. Table 2 outlines the possible values for PEG_NUM_COLORS along with the corresponding color depth.

When first installed on the host development system, C/PEG is configured to run with a screen driver that coincides with the host. So, if you are running on Windows, C/PEG is configured to use the screen driver that runs on Windows. The same for Unix/X11hosts. While these drivers are not usually considered to be of high performance, they do allow you to get C/PEG up and running very quickly.

PEG_NUM_COLORS	Bits per Pixel (bpp)
2	1
4	2
16	4
256	8
65535	16

Table 2 (Color Depths)

To select a screen driver for your target platform, first select the operating system as described in the previous section, then select the appropriate color depth the target hardware will support. You would then add the source modules for the color depth template and the hardware driver to your C/PEG build files. Table 3 lists the names of the template drivers, the source modules where they are implemented and the color depth they support.

Template Name	Source Module Name	Number of Colors Supported
L1Screen	l1screen.c	2
L2Screen	l2screen.c	4
L4Screen	l4screen.c	16
L8Screen	l8screen.c	256
L16Screen	l16screen.c	65535

Table 3 (Screen Driver Templates)

The screen drivers are arranged in three layers. These layers combine from the bottom up to support a particular platform.

The top layer for every screen driver is the fundamental PegScreen object. This object exposes the graphics API to the C/PEG application and is responsible for tracking the invalid region of the screen, appropriately clipping draw operations, and implementing some drawing primitives which are not dependent on color depth.

The middle layer is one of the templates listed in Table 3. The template implements basic drawing primitives as well as text and bitmap drawing for a given color depth.

The bottom layer is referred to as the driver. This portion is responsible for configuring the video hardware and allocating the frame buffer. If the video hardware also has acceleration functions, then the driver implements those as well. For instance, if the video hardware is able to draw lines, then the driver will implement the line drawing functions so that they take advantage of this, and replaces the functions otherwise found in the template.

Every C/PEG library build includes the PegScreen object, one of the color templates and a driver module.

Keyboard or Keypad Input

For the host development environments, keyboard input is configured in the operating system integration. For other targets, the driver may be in a separate source module. If you are unsure of keyboard or keypad support for your target, please contact Swell Software.

The symbol `PEG_KEYBOARD_SUPPORT` determines whether or not the library will internally include support for standard keyboard input. The library code size can be slightly reduced by turning off this definition for targets that do not require support for a standard keyboard. If your target will be using a keyboard or keypad type of input device, then you should turn this symbol on. If your final target will be exclusively using a touch screen or softkeys, you should not define this symbol.

The keyboard handling can be fine tuned in C/PEG by turning on the `PEG_ARROW_KEY_SUPPORT` and/or `PEG_TAB_KEY_SUPPORT` symbols. These symbols determine how C/PEG moves focus between child objects. The operation differs slightly depending on which of these symbols are defined and on which keys are provided for navigating the interface.

Focus Indicators

When operating a GUI with a keyboard or keypad input device, it is usually a requirement various C/PEG objects draw themselves differently when they have keyboard input focus. This provides a visual queue to the end user as to which object is currently active.

When a system operates with only a pointing device, such as a mouse or touch screen, drawing focus indicators is generally not a requirement. If this is the case, the application developer can save library size and run time overhead by eliminating this functionality.

The symbol `PEG_DRAW_FOCUS` determines if C/PEG will include this type of support. This symbol is turned on by default when `PEG_KEYBOARD_SUPPORT` is defined, otherwise it defaults to being turned off.

The application developer may decide to either turn this symbol on or off independent of keyboard support if they so choose.

Mouse Input

C/PEG is delivered with mouse input drivers for the supported development platforms. As with keyboard support, mouse support for any given platform is dependent on the platform and should be configured appropriately. Contact Swell Software for guidance on mouse support for your target hardware.

On all platforms, the `PEG_MOUSE_SUPPORT` symbol can be used to enable or disable internal support for mouse input. If the target system is not using a mouse for user input, then turn this symbol off. This will slightly reduce the code size of the library and also exclude the system bitmaps that are used to draw the mouse cursor, further reducing the size of the run time image.

Touch Screen Input

The `PEG_TOUCH_SUPPORT` symbol configures the library to support touch screen input. This is very similar to mouse input, although a few objects work differently. Specifically, the library does not require messages regarding pointer movement to be received by an object to operate properly when `PEG_TOUCH_SUPPORT` is defined. In most cases, the target system would be configured to run with either a mouse or a touch screen, but not both.

Unicode

The C/PEG library can be built to support 16 bit character encoding, also referred to as Unicode. To enable support for Unicode, define the `PEG_UNICODE` symbol before building the library.

Note some compilers provide intrinsic support for 16 bit string manipulation in their respective run time libraries, while some do not. For this reason, the

symbol PEG_STRLIB is often also required when Unicode support is enabled.

The default setting for this symbol is off.

String Tables

Multi-lingual applications are most often built using string tables rather than using actual string references. This makes it an easy process to change languages when the application is executing. The WindowBuilder tool includes a facility to create and maintain the string table for the application.

To enable this feature, define the PEG_STRING_TABLE symbol. When turned on, this causes functions to be included in the C/PEG library for converting string identifiers to string pointers as well as functions for defining the active string table and functions for assigning the active language.

String Library

The C/PEG library requires a small set of string manipulation functions. These are almost always provided by the compiler's run time libraries. However, many compilers and associated run time libraries do not support 16 bit character encoding. In addition, a few run time libraries force unnecessary functions to be linked into the application to achieve 16 bit character support.

For these reasons, C/PEG provides a sub-set of the standard C string library that provides 16 bit character handling. The provided functions include all of the functions required by C/PEG, thus eliminating the need to link in the compiler run time library.

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

The C/PEG string functions are included by turning on the PEG_STRLIB symbol. The default setting for this symbol is on if PEG_UNICODE is turn on; otherwise it is turned off.

Tasking Model

C/PEG supports two tasking models. The term tasking model refers to the mechanisms C/PEG has in place to allow multiple GUI tasks to operate in parallel on single or multiple processors.

The default tasking model is single threaded. This is the model used for standalone operation and for operating systems that do not support multiple tasks, such as DOS. This model can also be used to interface to operating systems that do support multiple tasks, but have only one task which will access the GUI. The single threaded model provides the lowest overhead and is the simplest to interface. This model can also be used as a starting point when running with an unsupported operating system.

The other tasking model is multi-threaded support. Under this model, any number of tasks may directly create, display, modify and destroy user interface elements. This model is selected by defining the `PEG_MULTITHREAD` symbol. This execution model is the default when C/PEG is configured for a supported operating system other than DOS. For multi-tasking operating systems to which C/PEG has been integrated, `PEG_MULTITHREAD` is defined by default.

The exception to this rule is the NetBSD operating system. As of version 1.6, threading support is not native to the operating system. The C/PEG integration with NetBSD and X11 is not multi-threaded. This is due to the fact the X11 server is not built with threading enabled on NetBSD, and cannot, therefore, reliably support applications that may use an external threading library. The version of C/PEG that is integrated with NetBSD that does not run on X11 does provide multi-threaded support using the GNU pth library.

PegPlotPointView

When the PegScreen object needs to change the value of a single pixel in video memory, it normally uses a macro named `PegPlotPointView` to accomplish this. This macro is defined in each screen template module to carry out this function based on the color depth of the system. If the application will need to switch, at run time, between different C/PEG screen drivers, then this macro must be defined as a function instead of a macro to prevent compiler errors. The `PEG_PLOTPOINTVIEW_AS_FUNCTION` symbol should only be defined to allow for run time driver switching.

Exit Operation

When running C/PEG applications on a development host system, it is usually desirable to allow C/PEG to terminate execution when no C/PEG objects are displayed. Conversely, in most embedded systems, C/PEG should never terminate regardless of whether or not any graphics are displayed. To control this, the symbol `PEG_EXIT_WHEN_LAST_CLOSED` is

provided. When this definition is turned on, C/PEG will exit when all objects are closed. Otherwise, C/PEG will never terminate execution.

This symbol is defined by default when running on a host platform such as Windows or Unix.

Graphics Primitives

C/PEG does not internally use all of the available graphics primitives to draw it's stock objects. If the application level software does not require extended graphics primitives, the code size of the library can be reduced by eliminating these primitives.

It is important to note floating point math is not used in any part of C/PEG. The following primitives do not use floating point math to calculate their respective drawing algorithms. Instead, there is a small lookup table used to determine the sin and cosine of an angle that is used to determine where to plot a point. This table is automatically included when either `PEG_FULL_GRAPHICS` or `PEG_ARC_GRAPHICS` is turned on.

The symbol `PEG_FULL_GRAPHICS` instructs the library to include support for polygons, circles and ellipses.

The symbol `PEG_ARC_GRAPHICS` denotes the inclusion of functions to support drawing arcs.

Memory Management

To portably support different memory allocation schemes supported by disparate operating systems, C/PEG provides a set of macros for allocating and releasing memory on the heap, `PEG_ALLOC` and `PEG_FREE`, respectively. C/PEG uses these macros in every instance that requires memory to be allocated and released.

If the symbol `PEG_MEM` is defined, then C/PEG assumes it will define the `PEG_ALLOC` and `PEG_FREE` macros. On the host development system and standalone builds, these macros map to `malloc` and `free`.

If the compiler or operating system of the target system use their own memory management routines, then the integration substitutes the links and does not define `PEG_MEM`.

Assertions

Throughout the library, C/PEG uses the macro `PEG_ASSERT` to assert pointer arguments as well as return values from functions that allocate memory. This is a nice feature when developing the code base for the application, but is typically not desirable in production code. The symbol `PEG_USE_ASSERT` may be turned on in debug builds then turned off in production builds. In line with many assert implementations, the `PEG_ASSERT` macro evaluates to `((void)0)` when `PEG_USE_ASSERT` is not defined.

LTOA

Two C/PEG objects use the non-ANSI function `ltoa` to convert integer data to string format. Many compiler run time libraries provide this function, while others do not (gcc being one of the latter). For this reason, C/PEG optionally provides it's own version of the ltoa function. If the compiler does not provide the function, define the `PEG_USE_LTOA` symbol.

Internally, C/PEG uses the function PegLtoA to call this function. If `PEG_USE_LTOA` is defined, then this call maps to the C/PEG internal version of the function call. If `PEG_USE_LTOA` is not turned on, then this call maps to using the compiler version of the function named `ltoa`.

The default setting turns on this symbol if either `PEG_UNICODE` is turned on, or the gcc compiler is used, otherwise, the C/PEG version is disabled.

3.1.1 Pre-configured Build Files

Building C/PEG for Windows using Microsoft Visual Studio 6.0 and .NET

Microsoft project files for MSVC++ 6.0 and .NET are provided for building the Windows version of the C/PEG library and each example program. The project files for building the C/PEG library are located in the `cpeg/build/win32/ms60` and `cpeg/build/win32/msnet` directories, respectively. While the project files for building the example applications are in the `cpeg/examples/`*`examplename`*`/win32/ms60` and `cpeg/examples/`*`examplename`*`/win32/msnet`.

As delivered, the C/PEG library is ready to build on this platform for Windows developers.

Before attempting to build the C/PEG library or any example programs, you must first tell MSVC++ where to find the C/PEG include files. For example, using MSVC++ 6.0, go to “Tools|Options|Directories” and add the C/PEG include directory to the include directories and the C/PEG source directory to the source file directories.

Building C/PEG on Linux With X11 Support

There is a GNU make compatible build file in the `cpeg/build/x11/linux` directory. To build the library for Linux/X11, simply switch to this directory and run `make`(1L). This will create a library file that can then be statically linked into any of the example programs.

To build the example programs, go to the `cpeg/examples/`*`examplename`*`/x11/linux` directory and run `make`(1L).

Building for Other Integrated Operating Systems

Build procedures for other operating systems are provided in a separate document that accompanies the operating system integration files. If C/PEG was purchased from Swell Software for a supported operating system, the distribution comes with instructions for configuring and building the library for use with the operating system as well as an example program that demonstrates using C/PEG with the chosen platform.

Building C/PEG for Other Targets

If the target platform does not have a configuration available, it is a simple matter to follow the steps taken to build on other platforms. Building C/PEG for a custom target is basically a matter of setting the best-fit options in `cpeg/include/pconfig.hpp`, compiling all of the needed C/PEG source files and linking them either into a library or directly into the system software.

Most application developers find that the standalone integration is an easy starting point for building C/PEG to run standalone or with a new operating system.

As a rule of thumb, it is best to start working with C/PEG using one of the pre-configured environments, usually Windows or X11/Unix. This will allow the application developer to experiment with the library and become familiar with the program startup sequence. The application level software may also be coded in this environment, since this code will not have to be modified to run on the final target.

If you have any questions during this process, we encourage you to contact Swell Software for assistance.

CHAPTER 4

THE C/PEG EXECUTION MODEL

This section introduces the C/PEG execution model and describes how the fundamental C/PEG objects work together to create a working interface. This section focuses on establishing a high level view of C/PEG and the internal components of a graphical presentation created with C/PEG, while the following sections detail actual object descriptions, functions and usage.

C/PEG supports two general execution models: single threaded and multi-threaded. For the purposes of this discussion, we will be using the single threaded model for an example. All of the concepts presented are valid regardless of the execution model. It is only the flow of information and processing that changes based on the execution model. A complete discussion of multi-threading as it relates to C/PEG is included in a later section “C/PEG Multi-Threading”.

4.1 Overview

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

PegTask provides the interface between C/PEG and the operating system. When running in a single threaded environment, PegTask is simply the entry point to the main program loop.

PegMessageQueue provides FIFO style message queue functionality for sending information between C/PEG objects.

PegPresentation keeps order on the visible screen. This involves keeping track of which objects are on top of other objects, maintaining the status of each object and remembering which object should receive user input.

Finally, PegScreen provides a layer of insulation between C/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 C/PEG objects need to present themselves to the user.

4.1.1 Software Block Diagram

A software block diagram of an executing C/PEG application is shown below in Illustration 1. This drawing depicts PegTask, PegMessageQueue, PegPresentation and PegScreen as well as input devices and miscellaneous graphical objects. These are the components of every user interface built using C/PEG.

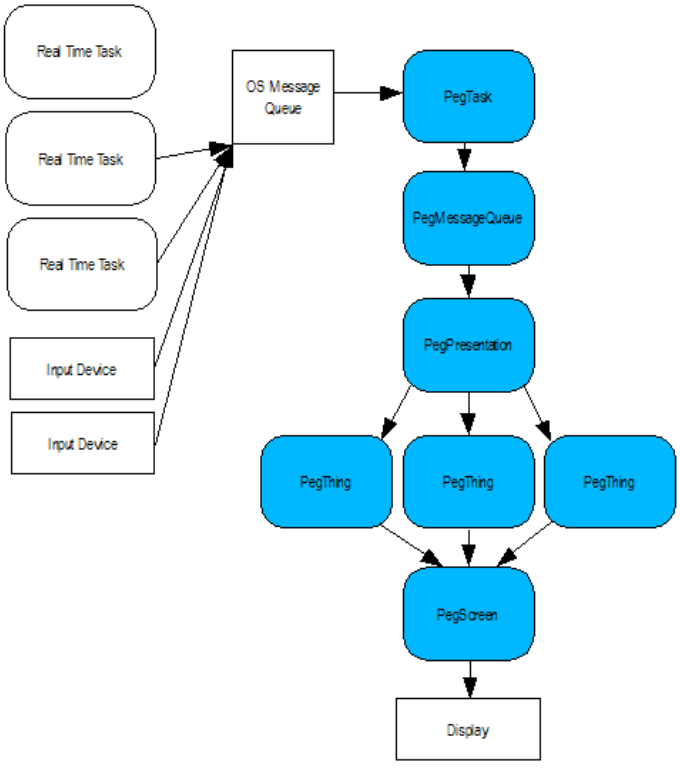

Illustration 1 (Software Execution Model)

4.1.2 Program Startup

C/PEG startup is divided into two steps. The first step is accomplished by PegTask. This step creates the central C/PEG components and performs any other required initialization. The second step takes place in the application level entry point, a function called `PegAppInitialize`.

PegTask is tailored to the execution environment, while `PegAppInitialize` is entirely application defined. This segmentation follows the C/PEG philosophy of insulating all application level software from the target environment, allowing the application software to run unmodified when moving from one of the host development systems to the final target system.

`PegAppInitialize` is where the application developer is allowed to create and display any startup objects for the application. The contents of this function are left entirely to the application developer to do whatever is necessary to launch the system. This may include putting up several graphical elements on the screen. Or maybe only one. Or, maybe none at all. Although, most applications will put at least one object on the screen at this point and allow the user to begin interacting with the system.

We will work through several examples of what this function may look like in later sections. For the moment, here is an example of a PegAppInitialize function that places a PegPanel object on the screen:

```
void PegAppInitialize(PegPresentation *pPresent)
{
   PegAdd(pPresent, MyPanelCreate());
}
```

Text 1 (PegAppInitialize Example)

We have not yet covered all of the information necessary to understand this example, so don't worry about the details. In the above example, we have created an interface object and added that object to the PegPresentation. This is a typical implementation of the `PegAppInitialize` function.

PegTask

The graphical interface should be viewed conceptually as a continuous low priority task in the overall multi-tasking system. During execution,

messages are dispatched from the PegMessageQueue to the PegPresentation, who, in turn, routes the messages to various graphical objects for processing. While the graphical interface is not truly running continuously in a multi-tasking system, the multi-tasking aspects are transparent to the entire graphical interface with the exception of PegTask. PegTask constructs PegPresentation, PegMessageQueue and PegScreen. These components are required for any C/PEG application to run. After these components are created, PegTask calls `PegAppInitialize`, which is the entry point for application code.

PegTask is a conceptual element and is generally composed of two or more functions. The first function provides the task entry point required by the operating system. When running in a standalone environment or on a Unix type of system, PegTask is simply the function `main`. When running in any standalone or single threaded environment, a second function named `PegIdleFunction` is also a component of PegTask. `PegIdleFunction` is called by the portable PegMessageQueue implementation when there is nothing left for C/PEG to do. This allows other lower priority tasks to execute. Or, in a single threaded environment, `PegIdleFunction` is a convenient place to perform other processing required by the target system.

In the following example, the PegTask entry point is a function called, fittingly, `PegTask`. This function should create the C/PEG foundation objects, call `PegAppInitialize` and begin C/PEG execution by calling `PegExecute` and passing the PegPresentation as the argument. Text 2 illustrates a typical implementation of PegTask.

```
void PegTask(void)
{
  PegRect r;
  PegMessage Mesg;
  PegScreen *pScreen;
  PegMessageQueue *pMessageQueue;
  PegPresentation *pPresentation;

  PegRectSet(&r, 0, 0, PEG_VIRTUAL_XSIZE - 1,
     PEG_VIRTUAL_YSIZE - 1);

  /* create the PegScreen interface object */
  pScreen = PegScreenCreate(&r);
  PegScreenPtrSet(pScreen);

  /* create the system message queue */
```

```
  pMessageQueue = PegMessageQueueCreate();
  PegMessageQueuePtrSet(pMessageQueue);

  /* create the PegPresentation object */
  pPresentation = PegPresentationCreate(&r);
  PegPresentationPtrSet(pPresentation);

  /* call into the user application code */
  PegAppInitialize(pPresentation);

  /* execute PegPresentation */
  PegExecute(pPresentation, PEF_NORMAL);

  /* when the above function returns,
   * the application is done */
  PegDestroy(pPresentation);
  PegScreenDestroy(pScreen);
  PegMessageQueueDestroy(pMessageQueue);
}
```

Text 2 (Example PegTask Function)

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

PegIdleFunction

In the standalone version of C/PEG, `PegIdleFunction` is the second component of PegTask. `PegIdleFunction` is called by the PegMessageQueue when there are no longer any messages in the queue which require processing. In this case, the graphical interface is up to date and does not need the CPU. `PegIdleFunction` is defined by the application developer and is typically where an operating system specific suspension mechanism is implemented to block, or suspend, C/PEG until some external stimulus is received which unblocks C/PEG and allows the GUI processing to continue.

It is important to note `PegIdleFunction` is not called by any version of C/PEG which has been integrated with a multi-threading operating system.

Versions of C/PEG that run multi-threaded and are properly integrated with a multi-tasking operating system do not use this suspension mechanism, but instead use custom implementations of PegMessageQueue which suspend the GUI tasks directly when no messages are available for a particular task.

The code for `PegIdleFunction` is obviously somewhat operating system dependent. The operation performed by `PegIdleFunction` is also dependent on whether the input devices are interrupt driven or polled. If the input devices are interrupt driven, `PegIdleFunction` should block or suspend indefinitely until a message arrives. If the input devices are polled, `PegIdleFunction` should poll the input devices until a new input message is generated.

The following is an example `PegIdleFunction` for use with interrupt driven input devices:

```
void PegIdleFunction(void)
{
  OS_MSG *pOsMsg;
  PegMessage NewMessage;

  /* suspend C/PEG until an external message arrives */
  pOsMsg = OSWaitForMessage(OsPegQueue, INFINITE);

  /* convert the OS formatted message to a PegMessage */
  ConvertOSMessageToPegMessage(pOsMsg, &NewMessage);

  /* place the PegMessage into the PegMessageQueue */
  PegMessageQueuePush(PegMessageQueuePtr(),
    &NewMessage);
}
```

Text 3 (PegIdleFunction Example)

In the above example, `PegIdleFunction` waits for an infinite period of time for a message to arrive in the OsPegQueue from some other source in the system. OsPegQueue, in this case, would be an operating system defined message queue created for interfacing C/PEG with input devices and other system tasks. When a message is received, the function `ConvertOSMessageToPegMessage` is called to convert the message from the operating system specific format into a PegMessage format. The new PegMessage is then pushed into the C/PEG system message queue, after which `PegIdleFunction` returns allowing the GUI to process the message.

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

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

In addition, it should be noted that, if possible, the `ConvertOsMessageToPegMessage` should be eliminated. This requires messages are sent through the operating system message queue in the PegMessage format. If the system is able to do this, `PegIdleFunction` should simply transfer messages from the OSPegQueue to PegMessageQueue.

An even simpler implementation may be used if your input devices must be polled. The following listing is an example of what `PegIdleFunction` may look like in this type of scenario.

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

Text 4 (PegIdleFunction Polling Example)

As you can see, this is indeed very simple. If the operating system in use has not been integrated with C/PEG, this is a very nice starting point for getting the system up and running. After the system is running, this function may be tuned to eliminate the polling as the system is integrated and interrupt driven input drivers are added.

Many users of C/PEG with custom operating systems also create a function for sending messages from external tasks into the C/PEG message queue (defined in Listing 3 as OsPegQueue). This function does whatever processing is required to send messages from other tasks to the queue being monitored by `PegIdleFunction`.

Note the following code sequence is not allowed from an external task (i.e. a task other than PegTask) when using the portable (i.e. standalone) version of PegMessageQueue.

Do Not Do This

```
void ExternalTask(void)
{
  PegMessage NewMessage;
  NewMessage.usType = PM_EXIT;
```

```
  PegMessageQueuePush(PegMessageQueuePtr(),
    &NewMessage);
}
```

Text 5 (External Task)

Why should this be avoided? Remember `ExternalTask` is not PegTask, but some other task, possibly of a higher priority than PegTask. Calling `PegMessageQueuePush` from an external task could result in the corruption of protections in place. The portable version of PegMessageQueue is designed to run only from within PegTask.

There is another, less obvious, reason why posting messages directly into PegMessageQueue from external tasks is not a good idea. Remember the user defined `PegIdleFunction` is probably waiting at an operating system defined message queue for a message to arrive. If the application bypasses the operating system defined message queue, `PegIdleFunction` will not wake up, and C/PEG message processing will not continue.

As stated above, this is why most users porting C/PEG to a new operating system define a function for sending messages to the operating system defined message queue which is monitored by `PegIdleFunction`.

The C/PEG standalone environment is a simple and convenient starting point when porting C/PEG to a custom target. The entire standalone implementation of PegTask is contained in the file `cpeg/source/sapeg.c`. Very little modification is usually required to turn this version of PegTask into a single, low priority task running on the target operating system.

PegPresentation

PegPresentation keeps track of all of the objects present on the display device. In addition, PegPresentation keeps track of which object has 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 objects that may be present on the screen at one time, it is easy to see this quickly becomes a complex task.

So how does the PegPresentation do it? By using tree structured lists intrinsic to the design of C/PEG, all objects that can be displayed are derived at some point in their hierarchy from a common base structure called PegThing. This structure is outlined in greater detail in subsequent

sections, but for now two important members of PegThing are a pointer to the PegThing's first child object and a pointer to the PegThing's next sibling. Using these two pointers, PegPresentation maintains all objects in lists as shown in the following illustration.

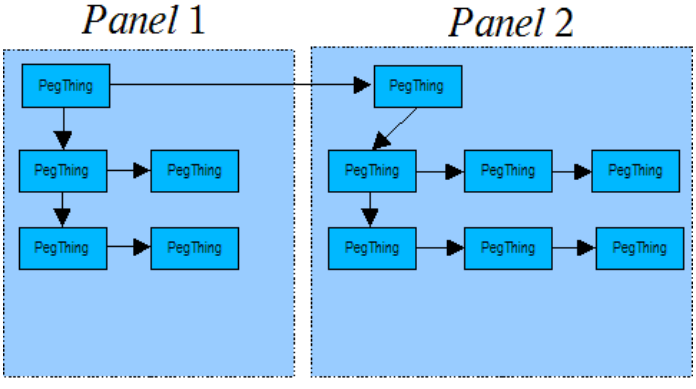

Illustration 2 (PegPresentation)

PegPresentation is also derived from PegPanel, which is derived from PegThing. This means PegPresentation is more or less just another panel, although in this case the panel has no border, can often appear invisible and always fills the entire screen or display. In essence, PegPresentation is the great-great grandfather of all panels and controls that will be displayed during the execution of the application. The term 'top level panel' is often used to refer to a panel that has been added directly to the PegPresentation.

Event Driven Programming

In viewing the example PegTask function above, you may have wondered “Where is main?”. C/PEG follows the event driven programming paradigm. There is no one central location or super-loop in the application level GUI software.

C/PEG is message driven, which may also be referred to as event driven. This means real processing is only done in response to messages received from the outside world. The objects that are created and used in the application software will be able to send and receive messages. The application developer is able to invent unique messages and interpret them in any way in order to make the C/PEG objects do the work the application demands. In general, messages should be simple and the corresponding

message processing should be short. This prevents any object from dominating the available CPU time. There is an example of creating a custom message later in this manual.

There are several advantages to a message driven implementation. The fact C/PEG objects communicate with each other via messages eliminates the problems associated with callback functions or similar implementations. One C/PEG object can communicate easily with another without worrying about how to physically address that object. In the large view, it could be stated the message driven systems are distributed systems, and objects could actually be physically miles apart from each other and talk as if they are both running from the same ROM. Of course, the out of the box PegMessageQueue implementation would require some enhancement to support this example, but these types of things are possible in a message driven environment.

PegButtons, PegEditFields, PegPrompts and other control types also use messaging to notify their parent objects when the control has been modified. The messages generated by these types of objects are determined by the object's internal integer identification, which is a data member of every PegThing derivative. This makes the flow of information through a C/PEG based application very predictable and robust. This type of message passing is so common that C/PEG defines a unique syntax for handling control notification messages, called signals. Signals will be described in detail in an upcoming section.

Input Focus Tree

An additional task of PegPresentation is the message routing. In a later chapter on PegMessageQueue, this mechanism will be looked at closer, and demonstrated how to route messages to specific objects. However, many system messages, such as mouse and keyboard input messages, are not directed to any particular object. For this reason, the PegPresentation internally maintains a pointer to the object that was last selected by the user using the mouse or by some other means. This object is called the *current* or *input* object, meaning that, by default, this object will receive input messages.

C/PEG views each displayed panel and child objects of each panel as branches in a tree. When input focus moves from object to object, PegPresentation insures the entire branch of the tree up to the actual input object has input focus. It is easy to detect if an object is a member of the

input focus branch of the PegPresentation tree at any time by testing the system status flag for PSF_CURRENT.

Just because an object is a member of the input focus tree does not necessarily mean the object is the end, or leaf, of the input focus branch. A pointer to the final input object may be obtained by calling the PegCurrentThingGet 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.

The application level software may also override the user's input selection and manually command PegPresentation to move the input focus at any time by calling the PegFocusTreeMove 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 non-directed input messages will be sent to the newly designated input object. In most circumstances, it is not necessary for the application level software to manually adjust the input focus, however, this capability is available if needed.

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

This makes it important to add child objects such that the object that should initially have input focus is added to the panel last.

Keyboard Input Handling

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

Keyboard input is received by C/PEG objects when the library is built with the PEG_KEYBOARD_SUPPORT symbol turned on.

PEG_KEYBOARD_SUPPORT does not imply the system is outfitted with a full AT style keyboard. Many C/PEG users have a very limited keypad with only a few key values available. This type of input will work just fine with a C/PEG application, since C/PEG requires only a very limited set of key values to navigate through screens and select controls.

The PegMessage object will be discussed in further detail 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 the PegMessage.usType data member is set to PM_KEY. The actual key value is passed in the PegMessage.sData field, and the key flags such as shift key state, control key state are passed in the PegMessage.u.lData field. Keyboard messages are undirected, meaning the message contains no information about which object should receive the message. This makes it the responsibility of the PegPresentation to know which object should receive keyboard input messages as they arrive from the system input device driver.

As described above, when a panel is added to PegPresentation, the first child object of the panel gains input focus. A limited number of PM_KEY messages can then be sent to move focus from object to object. When PEG_KEYBOARD_SUPPORT is turned on, the end user can fully navigate through the presented objects by sending a very small set of PM_KEY messages to C/PEG.

The table below describes the key values C/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 the target system must actually have keys such as TAB or CTRL, this only means C/PEG is watching for these key values. If the target system has, for example, four arrow keys and an Enter key, the application software simply needs to map these keys to the best match key values for which C/PEG is watching. In some cases, the application may need to send different key values for a common input key depending on what type of object with which the user is currently interacting.

C/PEG Key Value	Action Taken
PK_TAB	When this key is received, C/PEG attempts to move focus to the next child of the current panel. If the current child is the last child, focus wraps back to the first child of the current window.
<CTRL> + PK_TAB	This key combination is used to cycle through top level panels.
PK_CR	The carriage return key is used to select the item which has focus. If the object is a PegButton, the object will activate or toggle.
PK_LNUP PK_LNDN PK_LEFT PK_RIGHT	These keys (the arrow keys) move focus from sibling to sibling.
PK_ESC	This key is used to escape from a PegEditField operation.
<CTRL> + PK_F4	This key combination is used to close the current panel.

Table 4 (C/PEG Key Actions)

Mouse or Touch Screen Input Handling

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

CHAPTER 5

PEGMESSAGEQUEUE

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

Messages are placed into the queue from any one of three sources:

1) Input devices such as a mouse, touch screen or keyboard
2) Any other task in a multi-tasking system (via PegTask)
3) From C/PEG objects themselves

The messages placed in PegMessageQueue are the driving force behind the graphical interface. These messages contain notifications and commands which cause the graphical elements to draw themselves, remove themselves from the screen, resize themselves or perform any number of various other tasks. Messages can also be user defined, allowing the application developer to send and receive a nearly unlimited number of messages whose meaning is defined by the application software.

For example, it would be very common to have a graphical element send a message to another task in the system requesting data for display. The target task receives the request and responds, sending the response message to PegTask.

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

When porting C/PEG to run on a particular operating system, it is best to make use of the operating system supplied message system in order to

achieve the greatest efficiency. If the C/PEG distribution has already been tailored to a particular operating system, the distribution has a PegMessageQueue implementation that has already been customized to use the underlying operating system's message passing mechanism. If the target's operating environment is new to C/PEG, then the system developer should try to define the operating system's message queue such that little or no translation is required between PegMessage messages and operating system messages. Most commercial operating system implementations are designed such that the format of messages and message queues is largely user defined, which allows the system developer to directly send PegMessage formatted messages throughout the entire system.

5.1 PegMessage Definition

Messages are defined by C/PEG as simple structures containing fields indicating the source, target and content of the message. The definition of the structure, named PegMessage, is shown below in Text 6.

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

```
typedef union _uPegMessage
{
     struct _PegRect Rect;
     struct _PegPoint Point;
     void *pData;
     PEGLONG lData;
     PEGLONG lUserData[2];
     PEGULONG ulUserData[2];
     PEGSHORT sUserData[4];
     PEGUSHORT usUserData[4];
     PEGUBYTE ubUserData[8];
} uPegMessage;

typedef struct _PegMessage
{
     PEGUSHORT usType;
     PEGSHORT sData;
     void *pTarget;
     void *pSource;
     struct _PegMessage *pNext;
     union _uPegMessage u;
} PegMessage
```

Text 6 (PegMessage Structure)

Messages are identified by the member field `usType`. This is a 16 bit unsigned short integer value, which allows for 65,535 unique message types to be defined. Currently, C/PEG reserves the first 5,000 message type values for internal messages, which leaves message values 5,000 through 65,535 available for user-defined messages. The number of messages reserved for use by C/PEG may change slightly in future releases, therefore the library provides a constant indicating the first message value which is available for user definition. The constant is implemented as a #define with a symbol of `PEG_FIRST_USER_MESSAGE`.

Message Flow and Routing

C/PEG follows a bottom-up message flow philosophy. This means 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 tree to its parent. This flow continues until either an object processes the message, or the message arrives at PegPresentation. If a user-defined message arrives at PegPresentation, it will be ignored. This occurrence is usually an indication the application software failed to catch a message in a top level panel object.

Many messages, especially user-defined messages, may be directed towards a particular object by specifying the `pTarget` field or the `sData` field in the message. If the `pTarget` field 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 direct message.

Other messages do not have a particular object as their target. Examples of these messages include mouse, touch screen and keyboard input messages. In these cases, the `pTarget` member of the message is set to NULL, and it is the responsibility of PegPresentation to determine which object should receive the message. Messages of this type are referred to as undirected messages. This concept will be further explored in the discussion regarding the multi-tasking capabilities of C/PEG in a later section.

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's `sData` field contains the ID of the object that should receive the message. This means there are two ways of directing user-defined messages to a particular object. The application software may load the `pTarget` field of the message with an actual pointer to the destination

object, which always takes precedence, or it can load the `pTarget` field of the message with NULL and PegPresentation will route the message to the first object found with an ID value matching the `sData` field of the message. If the application developer wishes to route user-defined messages using object ID values, those objects should thus have globally defined object ID's to insure there are never multiple objects visible with duplicate ID values.

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

C/PEG System Messages

C/PEG messages can be divided into two categories. C/PEG system messages, which are generated internally by C/PEG to control and manipulate C/PEG objects, and user messages, which are defined and used by the application. Whether a message is a system message or a user message, the type of message is determined by the value of the message `usType` field. This is a 16 bit unsigned value. C/PEG reserves message types values from 1 to `PEG_FIRST_USER_MESSAGE`.

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

C/PEG allows the application developer great flexibility in message handling. Not only is the application developer allowed to create custom messages, it is also common to receive and process the system messages that are generated internally by C/PEG. This is sometimes referred to as intercepting a message, because the application can catch a message that C/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 section of this manual.

System Message List

The following is a list of C/PEG system messages that may generally be of interest to application level software. Additional control specific messages are documented in the section of the reference manual that described each particular control.

C/PEG System Message Type	Description
PM_ADD	This message can be issued to add an object to another object. The message pTarget field should contain a pointer to the parent object, and the message pSource field should contain a pointer to the child object.
PM_CLOSE	Recognized by PegPanel 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 PegPresentation tree which has input focus.
PM_DESTROY	This message is sent to PegPresentation to destroy an object. The pSource member of the message should point to the object to be destroyed.
PM_DRAW	This message can be sent to an object to force that object to redraw itself.
PM_EXIT	This message is sent to PegPresentation to cause termination of the application program.
PM_HIDE	This message is sent to an object whenever it is removed from a visible parent.
PM_KEY	This message is sent to the current input object when keyboard input is received. The message sData member contains the corresponding ASCII character code, if any, and the u.lData member of the message contains key modifiers, if available.

C/PEG System Message Type	Description
PM_LBUTTONDOWN	This message is sent to an object when the user generates mouse click input. PegPresentation routes mouse input directly to the lowest child object containing the click coordinates. 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 PegPresentation. The position of the mouse click is included in the `u.Point` field of the message.
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_NONCURRENT	This message is sent to an object when it loses membership in the input focus branch.
PM_PARENTSIZED	This message is sent to all children of a PegPanel derived object if the panel is resized. This makes it very easy for child objects 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 available on the system) moves into the section of the screen occupied by the object.
PM_POINTER_EXIT	This message is sent to an object when the mouse pointer (if available on the system) moves out of the section of the screen occupied by the object.
PM_POINTER_MOVE	This message is sent to an object when the mouse pointer (if available on the system) moves within the boundary of the section of the screen occupied by the object.

C/PEG System Message Type	Description
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 PegResize function. Note that C/PEG does not differentiate between moving an object and resizing an object. Both are accomplished via the PegResize operation. The new size for the object is included in the message u.Rect field.
PM_RBUTTONDOWN	This message is sent in systems that support right mouse button input. C/PEG objects do not process right mouse button messages.
PM_RBUTTONUP	This message is sent in systems that support right mouse button input. C/PEG objects do not process right mouse button messages.
PM_TIMER	This message is sent to an object that has previously started a timer via the PegTimerStart function. The integral ID of the timer is included in the sData member of the message.

Table 5 (C/PEG System Messages)

User Defined Messages

The underlying motivation for user defined messages is giving the application engineer the ability to use the C/PEG message system in the same manner as system messages to pass messages to and from C/PEG objects and custom application objects. This allows the application to do something useful when the end user interacts with the GUI elements that are specific to a particular product.

There are many reasons for the application engineer to use this mechanism. These will become clear once the developer is familiar with C/PEG and begins building his own applications.

For an example, suppose there are two separate but related panels visible on the screen. We will call these Panel A and Panel B. Panel A displays several data values, in alphanumeric format, that can be modified by the user. Panel B displays these same data values as a line chart. When the user modifies a data value in Panel A, we want Panel B 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 Panel A is notified by one of it's child controls that a value has been changed, it builds an instance of a newly defined message, places the data value into the message and sends the message to Panel B. When Panel B receives the message, Panel B realizes the line chart should be redrawn using the new data value.

It is important to note both Panel A and Panel B should have visibility to the user defined message type integer value. For this reason, it is often necessary to put these types into a header file that can be shared among all objects that will be interested in receiving or sending that particular message type. Remember to always begin with `PEG_FIRST_USER_MESSAGE` when enumerating message types. While it is not an absolute necessity for every user defined message type to be unique, it can save confusion in the long run.

There are three techniques that can be employed for sending user defined messages from one object to another. The first two imply the caller has direct visibility to the receiver by having a pointer to the receiving object.

First, the receiving object's `Notify` function may be called directly by the caller, passing the message as a parameter. The general form of this call is as follows:

```
PEGINT PegNotify(void *pTarget, const PegMessage *pMesg);
```

Second, the message's `pTarget` field may be loaded with the address of the receiver and pushed into the PegMessageQueue. Finally, the message's `sData` member can be loaded with the ID of the target object while the `pTarget` field is set to `NULL` and pushed into the PegMessageQueue. The second or third methods are generally preferred because it adheres to the basic partitioning philosophy of the C/PEG library.

If the `pTarget` field in the message is loaded with a pointer to a C/PEG object, the application developer must insure the object is not deleted

before the message arrives. When a user defined message contains a non-NULL `pTarget` value, there is no verification the `pTarget` field of the message is a valid object pointer. (If `PEG_USE_ASSERT` is defined, then an invalid `pTarget` may throw an assert in the message handling routine of PegPresentation. This is not a desirable action on a production system.) For this reason, in some situations, it is better to use `NULL pTarget` values and, instead, route the message to the object using the object's ID. If PegPresentation 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 events are processed. If a message is pushed into PegMessageQueue, the sending object immediately continues processing and the target object will receive and process the new message after the sending object returns from message processing. If the receiving object's message handling function is called directly by the sender, the receiver will immediately receive and process the message. In effect, pre-empting the calling object and it's execution thread. While these differences are generally inconsequential for user defined messages, they can be very important for C/PEG system messages.

5.2 Signals

As we have seen, messages are used to issue commands or to send other information between objects that are part of the user interface. In the previous section we learned a common use for user defined messages is to provide notification to a parent object when a child object has been modified. This usage is so common, in fact, C/PEG has defined a simplified method for defining these messages and a corresponding syntax for receiving them. This method is called *signaling*, and the messages sent and received via signaling are referred to as *signals*. Signals are designed to simplify the programming effort by reducing the complexity associated with panels that contain a large number of child controls.

Signals are also defined to solve some common problems associated with other, less friendly, messaging systems:

- Very often, a single panel 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 either multiple

message types must be sent by the control to the parent panel, or the receiver of a single notification message would have to further interrogate the control to determine exactly why the child object sent the message.

- Although a control may define several different types of modifications, the application code may not be interested in every type of control modification that can occur. In that case, it would be a waste of processing time for the child object to generate messages in which the parent is not interested.
- Finally, to facilitate the implementation of a rapid application development (RAD) tool such as PEG WindowBuilder, a consistent, simple and robust message definition method must be in place.

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

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

When an object which uses signaling is defined, only the object's ID value needs to be specified along with the signals the object is interested in. In order to process signals generated by that object, the parent only needs to remember the child object's ID. A later section will show this in detail.

C/PEG defines many different signals, or notification messages, which can be monitored for each control. Whenever the control is modified by the user, the control checks to see if it has been configured to notify the parent of the modification. If so, the control automatically generates a unique message number based on the control's ID and the type of notification. The message source pointer is loaded to point to the control, and the message is then set up to the parent panel or object.

To receive a signal, C/PEG defines the `PEG_SIGNAL` macro, which is used in the parent panel's message processing function. The parameters to the `PEG_SIGNAL` macro are the object ID and the notification message in which the object is 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 32 possible notification types.

The example below illustrates the use of signals in the definition and run time processing of a typical parent panel. Since most of the code presented in the example has not yet been fully discussed or explained, it is sufficient to examine the syntax for defining the control object ID's for each of the panel's controls and the message processing statements corresponding to each control.

A few object ID's are reserved by C/PEG to facilitate the proper operation of C/PEG modal panels. These object ID's are defined in the header file `pegtypes.h`. These reserved objects ID values occupy the upper range of possible ID values, and for this reason, ID values assigned to application objects should always start with an ID value of 1.

Not all notification signals are needed or supported by all control types. 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 below presents the message handling function of a panel as a preview allowing the application developer to observe the syntax of C/PEG signaling. The syntax is slightly unusual, even for experienced C programmers.

The first bit of code is from the header file for the panel. Each child control is assigned an enumerated ID, as in `IDC_USER_NAME` and `IDC_HAS_EMAIL`. In the second code segment, we find the message processing function, MyPanelNotify, 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 object, and the notification type we are interested in catching. This syntax has the further advantage of making the code somewhat 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.

```
PEGINT MyPanelNotify(void *pThing, const PegMessage *pMesg);
enum MyPanelChildIds
{
    IDC_USER_NAME = 1,  /* edit field ID */
    IDC_HAS_EMAL,       /* check box ID */
    IDC_EMAIL_ADDRESS   /* email address edit field ID */
};

PEGINT MyPanelNotify(void *pThing, const PegMessage *pMesg)
{
```

```
    switch(pMesg->usType)
    {
    case PEG_SIGNAL(IDC_USER_NAME, PSF_TEXT_EDITDONE):
        /* add code for user name modification here */
        break;

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

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

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

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

    default:
        return(PegPanelNotify(pThing, pMesg));
    }

    return(0);
}
```

Text 7 (Message Handling Function Example)

CHAPTER 6

PEGSCREEN

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

The PegScreen object contains all of the callable user API's for graphics drawing. If it is able to process the draw request in a portable manner, then it does so. If not, it hands the call off to the second layer of the graphics layer, the template. The template is a portable set of drawing functions that work in a particular color depth. For platforms with non-accelerated hardware, the template is able to display graphics directly into the frame buffer. If the hardware does support any type of hardware acceleration, as in line drawing, for instance, then the template hands the draw call off to the lowest level of the graphics layer, the driver.

The following illustrations exemplifies the relationship between these members of the graphics layer.

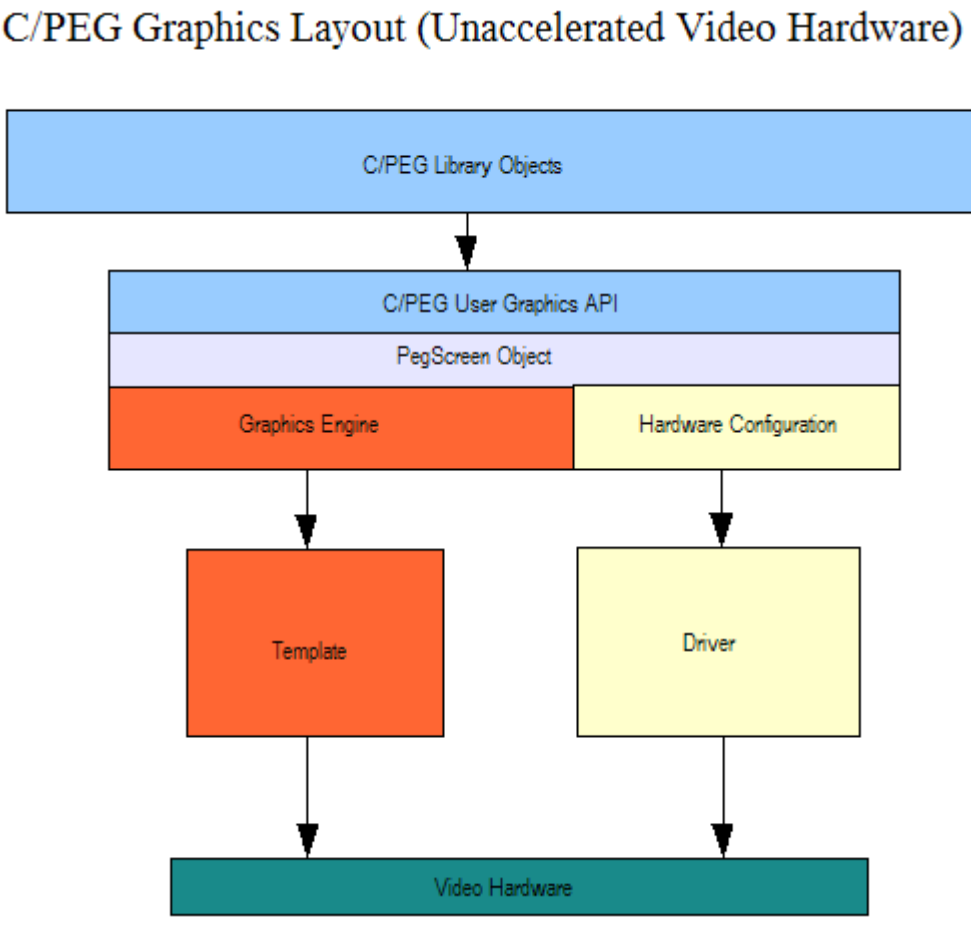

Illustration 3 (Unaccelerated Graphics Layout)

C/PEG Graphics Layout (Accelerated Video Hardware)

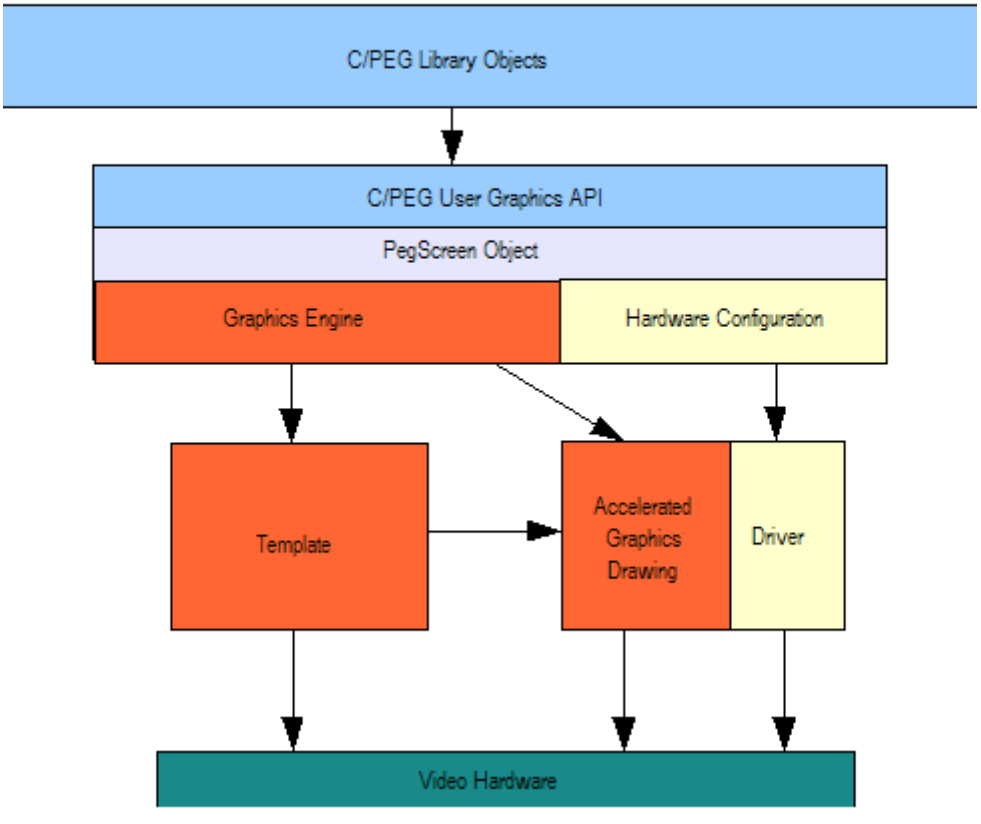

Illustration 4 (Accelerated Graphics Layout)

This layout makes the graphics API portable across platforms. It also provides a clear path for supporting new hardware.

PegScreen may interface to a large variety of display devices. VGA and SVGA displays, LCD panels and even printers may be driven by the PegScreen interface object. Custom implementations may, of course, extend the required functionality. C/PEG is delivered with several working examples of template and driver implementations. These are described in more detail in the following sections.

At this point, it should be clear why the build procedures at the start of the manual contain instructions for including different template and driver modules depending on the target environment.

Screen Coordinates

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

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

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

C/PEG screen coordinates are in pixels. When the application developer defines a position of a panel, the position is in pixels. When the application developer defines the position of a button, the position is in pixels. When the application developer defines the starting position for text drawing, the position is in pixels. We think you get the message.

C/PEG screen coordinates are relative to the upper left corner of the screen, which is 0, 0. While this does not exactly follow trigonometric conventions, this is at least a consistent definition among graphical environments and will be familiar to users who have done previous GUI programming. C/PEG screen coordinates are not relative to the client area of an object, the client area of an object's parent or relative to the day of the month. C/PEG screen coordinates are relative to the upper left corner of the screen, which is 0, 0.

For custom drawing routines, it is important to bear in mind the application developer will always need to use some corner of the object's client rectangle as the reference point for the drawing operations. While this may at first appear more difficult than alternative approaches, we feel once you gain experience with this method of coordinate resolution, you will find custom drawing from within C/PEG is much easier to do than with any other platform.

Video Controllers

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

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

Porting C/PEG to Custom Video Hardware

Porting C/PEG to run on a custom target platform requires a custom driver layer for the graphics engine and configuration routines that is tuned to the bus architecture, color depth, memory organization and pixel resolution of the target platform. Creating a custom PegScreen involves configuring the video controller and optionally tuning one of the provided templates to make the best use of the system configuration. This does not mean it is necessary to invent algorithms to meet the requirements of these functions. The required algorithms are provided in the PegScreen object and templates provided with the library. It is best to start with the nearest template implementation as the basis for your custom driver.

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

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

If the target hardware uses a linear 1 bpp (bits per pixel), 2 bpp, 4 bpp, 8 bpp, or 16 bpp linear frame buffer, the provided templates contain everything you need to run C/PEG on your target. If the video hardware

contains acceleration functions, these templates will not take full advantage of these features, but they will allow you to get a system up and running quickly. Tuning the driver to take advantage of video hardware acceleration can be accomplished as the project development progresses.

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

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

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

PegScreen Templates and Drivers

Several PegScreen templates are provided in your C/PEG distribution. These templates are general purpose drivers accommodating a wide range of color depths and screen resolutions. Note that these general purpose templates do not configure the video controller, that is the job of the driver layer. All of the necessary drawing routines are provided and ready to use for a given color depth at any screen resolution in the templates.

To implement a new driver layer, it is best to start with an existing driver that most closely resembles the target video hardware.

CHAPTER 7

FUNDAMENTAL DATA TYPES

This section introduces the custom data types defined by C/PEG. The data types include simple 8, 16 and 32 bit data storage types as well as more complex types for passing information such as color, position and bitmap data. After using C/PEG for a while, these data types will become second nature. The data types are defined in the include file `pegtypes.h`. The more complex objects are found in separate header files that are listed along with the description of the type.

Simple Data Types

- Defined in file: `pegtypes.h`

The following simple data types are used instead of the intrinsic data types defined by the compiler to avoid conflicts when running on platforms with different 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 may need to be modified to match the word length of the target platform.

C/PEG Data Type	C Data Type	Datatype
PEGBYTE	signed char	8 bit signed
PEGUBYTE	unsigned char	8 bit unsigned
PEGSHORT	signed short int	16 bit signed
PEGUSHORT	unsigned short int	16 bit unsigned
PEGINT	signed int	signed, native size
PEGUINT	unsigned int	unsigned, native size
PEGLONG	signed long	32 bit signed
PEGULONG	unsigned long	32 bit unsigned
PEGCOLORVAL		varies on color depth
PEGCHAR		8 or 16 bits

Table 6 (Simple Data Types)

Whenever possible, the C/PEG library uses the PEGINT and PEGUINT data types to produce the most efficient code for the target processor. If the variable in question does not need to be an exact bit width, the PEGINT and PEGUINT data types are used. The smaller data types are used only when the library requires a specific variable size, or when data space savings require that a short data type is specified.

PegPoint

- Header module: `ppoint.h`
- Source module: `ppoint.c`

Definition:

```
typedef _PegPoint
{
    PEGSHORT x;
    PEGSHORT y;
} PegPoint;
```

PegPoint is a basic pixel address data type. The x and y position is always relative to the top left corner of the screen.

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

PegRect

- Header module `prect.h`
- Source module `prect.c`

Definition:

```
typedef struct _PegRect
{
    PEGSHORT sLeft;
    PEGSHORT sTop;
    PEGSHORT sRight;
    PEGSHORT sBottom;
} PegRect;
```

A large part of graphical interface programming revolves around defining and calculating rectangular regions on the screen. By providing a PegRect structure and support functions, C/PEG easily facilitates these types of operations.

See the “C/PEG API Reference Manual” for support functions that relate to PegRect.

PegBrush

- Header module: `pbrush.h`
- Source module: `pbrush.c`

Definition:

```
typedef struct _PegBrush
{
    struct _PegBitmap *pBitmap;
    struct _PegRect Clip;
    PEGULONG ulPattern;
    PEGINT iWidth;
    PEGCOLORVAL Foreground;
    PEGCOLORVAL Background;
    PEGUBYTE ubFlags;
} PegBrush;
```

C/PEG is designed to allow the application developer to easily modify the default appearance of all of the C/PEG objects at compile time, and also to change the appearance of individual objects at run time. C/PEG is internally designed to support display color depths up to 16 bpp (5-5-5 or 5-6-5 RGB) operation. The color definitions used in the C/PEG source distribution limit C/PEG to actually only using 16 colors. This restriction allows C/PEG to run out of the box on most systems. There are no restrictions on the actual color depth of the target display, but C/PEG objects will not by default make use of this extended color capability. The default color values used by each C/PEG object are defined in `pegtypes.h`.

All C/PEG objects contain four primary color values. These color values determine the default background and default text colors, as well as the background and text colors to use when the object is current, or selected. Many C/PEG objects appear the same way whether selected or not. A few C/PEG objects define additional color values specific to that object type. In any event, these default colors may be modified by adjusting the definitions in the `pegtypes.h` file. Object colors may also be modified at run time by calling the PegColorSet function described in detail later.

PegBrush is also a parameter to many of the screen output functions. PegBrush informs the output functions what foreground and background color to use, how to modify the drawing of the primitive, a pointer to a fill bitmap and a PegRect structure that may be used to perform additional clipping.

The PEGCOLORVAL data type may be 8, 16 or 32 bits, depending on the color depth of the target platform. This is configured automatically by the C/PEG header files based on the selected value of PEG_NUM_COLORS.

The member Foreground is generally used to draw text, lines and the outlines of rectangles, circles, arcs and polygons. While the Background member is generally used as the fill color for text, circles, arcs and polygons.

The iWidth member is used as the width for drawing lines. It is also used as the outline width for drawing rectangles, circles, arcs and polygons.

The ubFlags member modifies the drawing operation of many of the screen output functions.

The following table describes the valid values for this member as well as their meaning on specific draw operations. It is acceptable to use some of these flags together to achieve the desired effect by logically OR'ing the flags together.

It is important to note by using flags which will cause the draw operations to evaluate the pBitmap or Clip members of the brush structure, the respective members must be appropriately filled in, or else there will be a run time error.

.

PEGBrush ubFlags	Description
CF_NONE	No fill, pattern, bitmap or clipping
CF_FILL	Fill rectangles, circles, arcs and polygons with the Background color
CF_XOR	Do not use any color, instead XOR with the background
CF_PATTERN	Applies to line. Uses the value in ulPattern to draw the line. The default for the pattern is defined as 0xccccccccUL by the PEG_DEF_FILL_PATTERN definition.

PEGBrush ubFlags	Description
CF_TILE	Tiles pBitmap over a given rectangle. Uses the Clip member to determine the tile region
CF_CENTER	Centers pBitmap over a given rectangle. Uses the Clip member to determine the region over which to center.
CF_STRETCH	Fills the rectangle defined by Clip with a resized copy of the bitmap in pBitmap. This functionality is not available on all platforms.
CF_CLIP	Clips any draw operation to the Clip member

Table 7 (PegBrush Flags)

The Clip member of a PegBrush allows the application developer to exert fine control over the draw operations. Every draw operation is always clipped to the C/PEG object that is doing the drawing. Therefore, a C/PEG object is never allowed to draw outside of the portion of the screen it occupies. The Clip member enforces even more stringent policing on the part of the PegScreen. Not only are draw operations clipped to the calling object, the draw operations will also be clipped to the Clip rectangle in the PegBrush, if one is supplied and CF_CLIP is part of the ubFlags member.

The ulPattern member is used when drawing patterned lines. The member holds an unsigned long data value that determines the pattern which is used to draw the line. A bit that is turned on in this value, starting from the high order bit, indicates the corresponding pixel on the screen should be drawn in the foreground color of the brush. A bit that is off indicates drawing should be skipped for the corresponding pixel. The default value for this member when the brush is created using the PegBrushCreate functions is PEG_DEF_FILL_PATTERN, which is defined as 0xccccccccUL, which is two bits on followed by two bits off, repeated eight times.

PegMessage

- Include module: `pmessage.h`
- Source module: `pmessage.c`

Definition:

```
typedef union _uPegMessage
{
     struct _PegRect Rect;
     struct _PegPoint Point;
```

```
        void *pData;
        PEGLONG lData;
        PEGLONG lUserData[2];
        PEGULONG ulUserData[2];
        PEGSHORT sUserData[4];
        PEGUSHORT usUserData[4];
        PEGUBYTE ubUserData[8];
    } uPegMessage;

    typedef struct _PegMessage
    {
        PEGUSHORT usType;
        PEGSHORT sData;
        void *pTarget;
        void *pSource;
        struct _PegMessage *pNext;
        union _uPegMessage u;
    } PegMessage;
```

PegMessage defines the format of messages passed within the C/PEG environment. On most machines, each PegMessage requires 24 bytes of memory if structure packing is enabled.

For user defined messages, all but the `usType` and `pTarget` message fields can be used in any way desired. The `uPegMessage` union member field is intended to allow the application developer to easily pass any type of data necessary in the user defined message.

PegTimer

- Include module: `pmessage.h`
- Source module: `pmessage.c`

Definition:

```
    typedef struct _PegTimer
    {
        struct _PegTimer *pNext;
        struct _PegThing *pTarget;
        PEGLONG lCount;
        PEGLONG lReset;
        PEGUSHORT usTimerId;
    } PegTimer;
```

Support functions:

```
void PegTimerStart(PegMessageQueue *pQ, void *pThing,
  PEGUSHORT usId,PEGLONG lCount, PEGLONG lReset);
void PegTimerKill(PegMessageQueue *pQ, void *pThing,
  PEGUSHORT usId);
```

```
void PegTimerTick(PegMessageQueue *pQ);
```

C/PEG provides a simple means for the application developer to receive periodic timer messages in C/PEG objects. Any object derived from a C/PEG object may start any number of individual timers. When the timer expires, that object will receive a `PM_TIMER` message from PegMessageQueue. The message's `sData` member will contain the ID of the timer that has 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 time out.

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

The PegTimerTick mechanism serves two purposes. First, it insulates C/PEG from knowing anything about the target hardware time base. Second, it allows the application designer to tailor the frequency in which the C/PEG timer is strobed. Very often it is not necessary for the GUI timers to be nearly as accurate as low-level timer interrupts. For an example, if it is desired for a C/PEG timer to be accurate to 50 milliseconds, and the low-level timer interrupt occurs every one millisecond, the system software would call the PegTimerTick function once for every 50 timer interrupts received.

The application designed determines the time base for C/PEG timers. Therefore, the value loaded in a C/PEG timer is simply a number of ticks, rather than any absolute time value. C/PEG defines the constant `PEG_ONE_SECOND`, which should be set by the application designer to equal the number of C/PEG timer ticks that will occur in one second. When a C/PEG timer is loaded, it is good practice to calculate the tick value based on this `PEG_ONE_SECOND` definition. This way, if the time base changes during program development, it will not be necessary to track down every location in the application code where timers are being set in order to modify the tick value used.

PegTimers are started by calling the `PegTimerStart` function. The parameters specify the timer ID, the first time out period and successive

time out periods. The PegMessageQueue pointer will usually be the default C/PEG system message queue which can be obtained by calling the PegMessageQueuePtr function. If a single C/PEG object needs to create more than one timer, it is important to assign the timers specific ID values so the C/PEG object is able to recognize each timer expiration message. The theoretical limit for timer ID values on most systems is 32K since the ID is sent to the object in the `sData` member of a PegMessage, which is a PEGSHORT data type.

The lCount and lReset time out periods determine how many timer ticks will expire before the timer 'times out'. These can be the same value. If the lReset value is zero, the timer will time out only once and delete itself. This is generally referred to as a one shot timer.

While there is an active timer for an object, the object will receive PegMessages with a `usType` set to `PM_TIMER`.

To stop a timer, the object should call PegTimerKill. If a value of zero is passed in the `usID` parameter, then all timers for that object will be deleted.

The application level code should never create an instance of a PegTimer object directly. All of the necessary functionality for timer are provided in the timer functions described above.

PegFont

- Header module: `pfont.h`
- Source module: `pfont.c`

Definition:

```
typedef struct _PegFont
{
     PEGUBYTE ubType;
     PEGUBYTE ubAscent;
     PEGUBYTE ubDescent;
     PEGUBYTE ubHeight;
     PEGUSHORT usBytesPerLine;
     PEGUSHORT usFirstChar;
     PEGUSHORT usLastChar;
     PEGUSHORT *pOffsets;
     struct _PegFont *pNext;
     PEGUBYTE *pData;
} PegFont;
```

The PegFont type contains information about each font used by the application. The PegScreen text output and text information routines

require a pointer to a PegFont structure as a parameter. One PegFont structure should be defined for each font that is used by the application. The good news is, these fonts can easily be generated using the PEG FontCapture utility program. This program will automatically generate this data structure along with the associated offset and data tables.

By default, C/PEG includes only two fonts named PegBoldFont and PegNormalFont. As suggested by their names, PegBoldFont is slightly larger and heavier than PegNormalFont. PegBoldFont is used for string objects, prompts and in the title bars in message panels. The PegNormalFont is used for all of the button objects as well as the group object. The font used by any particular object is very easy to change by making a call to PegFontSet.

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

Default Object Fonts

If custom fonts are created for use in an application, they may be used at specific times using the PegFontSet function as described above. This can become tedious and subject to mistakes if the application must set the font for each particular object of a given type. To that end, C/PEG provides functionality to allow the application developer to set the default font for any object type that uses text. The default font used by C/PEG for each object type is defined in the header file pfonts.h.

C/PEG maintains a small table that maps a particular font to a particular C/PEG type of object. This type is independent of the type that is stored in the object. This type is more of an aggregate of like C/PEG objects types into a more general classification. These types can be found in the `pfonts.h` header file defined in the `PEG_DEFAULT_FONT_INDEX` enum. The following table outlines the default font classifications and to which C/PEG objects these pertain.

C/PEG Default Font Index	*C/PEG Objects Affected*
`PEG_DEFAULT_FONT`	`None`
`PEG_TITLE_FONT`	`PegTitle`
`PEG_TEXT_BUTTON_FONT`	`PegTextButton, PegMLTextButton, PegDecoratedButton`

C/PEG Default Font Index	C/PEG Objects Affected
PEG_RADIO_BUTTON_FONT	PegRadioButton
PEG_CHECKBOX_FONT	PegCheckBox
PEG_PROMPT_FONT	PegPrompt, PegMLPrompt
PEG_EDIT_FIELD_FONT	PegEditField
PEG_TEXTBOX_FONT	PegTextBox
PEG_GROUP_FONT	PegGroup
PEG_ICON_FONT	PegIcon
PEG_PROGBAR_FONT	PegProgressBar
PEG_PANEL_FONT	PegPanel
PEG_MESSAGE_PANEL_FONT	PegMessagePanel, PegStatusPanel

Table 8 (Default Font Index)

Any object that derives at some point from PegTextThing contains a pointer to a PegFont structure it uses to pass to the PegScreen object when it wishes to draw text. These types of objects, at some point in the process used to create them, set their internal font pointer by getting a pointer at the default font for their classification. The point here is every PegTextThing derived object calls into this font pointer table to retrieve the font they are suppose to use for drawing. Therefore, if the application modifies the font table, subsequent object creation will obtain the new font from the font table, if their classification's font pointer was modified.

It would be obvious to assume, then, any modifications to the font table should be done early in the application process, preferable before any PegTextThing derived objects are created. This would ensure the objects are created with the correct font.

To do this, the application only needs to call PegDefaultFontSet with the correct object category and a pointer to a PegFont structure. After this call, all new objects of that category will internalize the new PegFont and use that for drawing.

The application developer may also wish to add new categories to this font pointer table to support new category types that are used in the application. This is simple to do as well. The total number of font pointers allowed in the table is controlled by the `PEG_NUM_FONTS` define. This define is the sum of two other defines: `PEG_NUMBER_OF_DEFAULT_FONTS` and `PEG_SPARE_FONT_INDEXES`. As delivered, the C/PEG library defines the latter to be 0. By modifying `PEG_SPARE_FONT_INDEXES` to a number greater than 0,

the application developer may then add that many new category indices and font pointers to the font table.

It is important to note any customer user objects that wish to use the new font must call into the font table during the creation process. This will ensure the application behaves as expected.

It is also possible to modify the font table pre-compile time. Therefore, instead of calling PegDefaultFontSet, the application developer may wish to simply replace the default font in the particular category slot pre-compile time, thus ensuring that every object of that category will always use the desired font without the necessity of calling PegDefaultFontSet at the start of the application. Of course, the down side to this is if the application developer updates the C/PEG library, these changes may be lost during the upgrade process.

Outlined Fonts

PegFonts are normally 1 bpp bitmapped fonts, but, outlined fonts are encoded in 2 bpp format. These fonts can be drawn with any combination of foreground and background colors that are supported on the target hardware. An additional PegFont type is the outlined font type. This font looks identical to the application and library software, however the internal format is modified to draw with a different appearance. The PegScreen recognizes the outline font type and will draw the font accordingly.

Anti-Aliased Fonts

Anti-aliased PegFonts are also bitmapped fonts, but they are encoded using a 4 bpp format to allow each pixel to be defined as one of a possible sixteen levels of off to on. The sixteen shading levels are usually sufficient to provide a nice appearance on most displays.

Anti-aliased fonts are used and assigned to C/PEG library objects just like normal 1 bpp fonts. The only difference is in the internal format. The anti-aliased font format is recognized by the PegScreen object as such and is drawn accordingly.

Multilingual Support

The PegFont data structure can be used to contain almost any number of characters, or glyphs, including characters sets that exceed 256 characters in number. For very large fonts, the PEG FontCapture utility application will automatically generate multi-page fonts using the `pNext` member of each font to create a single PegFont containing any number of characters. Each

page of the font may have different attributes, which allows for the greatest memory savings.

PegBitmap

- Include module: `pbitmap.h`
- Source module: `pbitmap.c`

Definition:

```
typedef struct _PegBitmap
{
     PEGUBYTE ubFlags;
     PEGUBYTE ubBitsPix;
     PEGUBYTE sWidth;
     PEGUBYTE sHeight;
     PEGULONG ulTransColor;
     PEGUBYTE *pStart;
} PegBitmap;
```

The PegBitmap structure is used to pass bitmap data to the PegScreen object during draw operations that wish to draw bitmaps. C/PEG supports bitmaps from 1 to 16 bpp formats. Further, the bitmap data may be compress, uncompressed or may only encode changes from a previous bitmap, which is used for displaying animations. C/PEG uses RLE compression techniques. While other techniques may offer superior compression ratios, RLE compression offers very fast run time performance.

The PEG ImageConvert utility application is used to generate bitmaps in the correct format for the application and target hardware. The data structure, along with the actual bitmap data, is automatically generated by this utility. In general, bitmaps larger that 64 x 64 pixels should be compressed to save storage space. On the other hand, if the target system has plenty of memory available at run time, it may be a performance enhancer to not compress the bitmaps. Every platform and architecture is different, so you may need to experiment with compression to find a suitable level of performance and memory usage for your platform.

While most bitmaps will be converted to a PegBitmap structure pre-compile time, it is also possible to create a bitmap at run time and draw into it using the same draw routines that are used to draw to the screen. The only difference is in the opening call to PegDrawBegin or PegDrawBeginBitmap. The first function implies drawing directly to the screen, while the second takes a pointer to an existing bitmap. The application developer may create a bitmap for this use by calling PegBitmapCreate and passing over the size

and color depth of the requested bitmap. There is an example of using this technique later on in the manual.

CHAPTER 8

THE MIGHTY THING

In this section the most fundamental and important object in all of the C/PEG library will be discussed: **PegThing**. This section describes the overall capabilities of the PegThing object, the associative functions of the objects and provides several small programming examples illustrating the most common C/PEG programming operations. This information complements the API reference manual in that here the concentration is on useful examples, whereas the primary purpose of the API reference manual is to provide a quick lookup for function names and argument lists.

The first half of the section is a rather formal function reference, covering object creation, deletion and functions dealing specifically with PegThing objects. The second half covers various important topics explaining the purpose of PegThing members and demonstrating the use of PegThing support functions to accomplish small programming tasks.

PegThing is the base object from which all viewable C/PEG objects are derived. As mentioned previously, the term derived in this context refers to the fact every viewable C/PEG object contains the same data and function pointer members as PegThing in the same order as they are contained in PegThing. While it may not be typical to create an instance of an actual PegThing object in the application, it is very possible that application level code may wish to derive custom object types directly from PegThing.

A basic precept in the design of C/PEG is all graphical objects, from the most complex to the simplest, share a small but significant set of properties. Some of these basic properties include: whether or nor the object is visible; if the object has a parent and who the 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 forth. These and other properties define how each object will participate in the graphical representation.

This is not to imply PegThing is overly complex or difficult to understand. In fact, PegThing is actually very straight forward. The following sections will describe in detail the functions and data members of PegThing. With this information, the application developer will gain a clear understanding of

how C/PEG works, and will be able to anticipate how objects will work together when combined to perform complex interfaces.

PegThing Support Functions

This section describes each support member and data member of the PegThing object. Rather than focusing entirely on formal function declarations, parameter descriptions and return values as is done in the C/PEG API Reference Manual, this section instead includes many code fragments and useful examples to illustrate how each of the functions can be used.

This manual does not include every function associated with the PegThing object. Please refer to the API manual for an alphabetic list of all PegThing functions.

Creation

Every PegThing derived object in the C/PEG library has a set of functions for creating, initializing and setting up the members of the objects. They all share the same function name scheme: the name of the object type (PegThing, in this case), then an action word or phrase that describes what it is that is being done. The action words and phrases are common for each object, only the object name changes.

For instance, to create a default PegThing object, which is properly initialized, but does not have a size, position, ID or style, use the function PegThingCreateDefault. Likewise, to create a PegTextButton with the same attributes (properly initialized, but not having size, position, ID or style), use PegTextButtonCreateDefault. The function names an internal sequence of what function is called by which other function is exactly the same for every PegThing derived object.

This provides a very definable and reproducible model that, once understood, allows the application developer to easily create custom objects derived from an existing C/PEG base object. Once the object is created, there is a clear road map on how to properly initialize the object for use.

The illustration below shows the five functions which are common to all PegThing derived objects and how these work together to create an object of a given type. The illustration uses PegThing for the example, but the same order holds true for any PegThing derivative, only the function names are different to match the object type.

It is very common for a derived object to call the object from which it was derived for default functionality for a given function. For example, the PegButton object, which is derived from PegTextThing, calls PegTextThingSet from within its own PegButtonSet function. This ensures the derived object properly inherits the correct setup procedure.

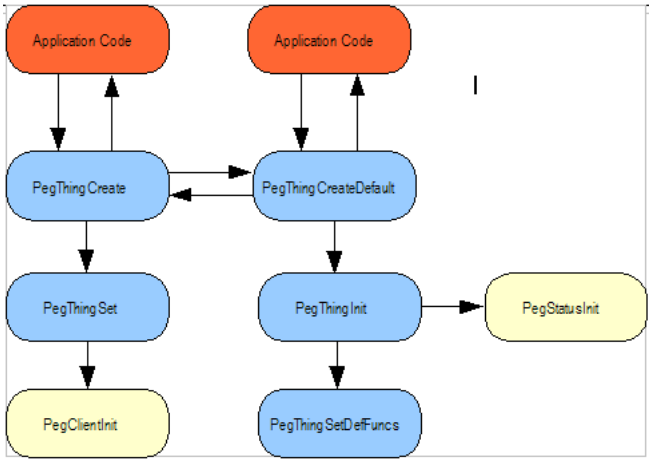

Illustration 5 (PegThing Creation)

```
PegThing *PegThingCreateDefault(void);
```

This function is used when the desired initial position of the object on the screen 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.

The returned PegThing pointer is a fully formed PegThing object. All of it's function pointers have been initialized. It's type has been set to `PEG_TYPE_THING` and it's default status has been set. The remaining data members have all been set to 0.

The easiest, and proper, way to set an objects position is to call the PegResize function, which accepts a pointer to the object that is to be resized along with a pointer to a PegRect structure that is used to resize the object to the desired screen coordinates. Calling PegResize is the acceptable way to set the size of an object or position after the object is visible.

```
PegThing *PegThingCreate(PegRect *pRect, PEGUSHORT usId,
       PEGUSHORT usStyle);
```

This function is used when the desired initial position of the object on the screen known at the time of object creation. This will also assign the ID to the object as well as its initial style.

Internally, this function calls PegThingCreate to obtain a fully formed PegThing object. It then uses the parameters to 'set up' the object. The obvious benefit to this is there really is only one function that does the actual work of allocating the memory necessary for a PegThing object.

```
void PegThingInit(PegThing *pThing);
```

This function is called by PegThingCreateDefault to properly initialize the PegThing object. This function sets the objects type, default status and calls PegThingSetDefFuncs to set up the objects function pointers.
It is suggested any objects which are derived from PegThing objects in application code may wish to call this function after the custom object has been allocated. This ensures the basics of the object are setup properly before doing any custom modifications on the custom object.

```
void PegThingSet(PegThing *pThing, PegRect *pRect,
       PEGUSHORT usId,PEGUSHORT usStyle);
```

This function is called by PegThingCreate after the object has been allocated. The PegRect parameter is used to set the Real member of the object (this member keeps track of the over all size and position of the object) as well as the Client and Clip members, by calling PegClientInit. Also, the ID and style of the object are set.

If the application developer creates a custom object derived directly from PegThing, it is a good idea to call this function from the function that creates the custom object to correctly set up the position, style and ID of the object.

```
void PegThingSetDefFuncs(PegThing *pThing);
```

This function sets up the function pointers in the passed over PegThing object. This function is called by PegThingInit to properly set up the function pointers.

Creation Examples

This may seem a bit confusing at the onset, but the creation procedure for a PegThing or derivative is actually partitioned like this to allow flexibility in creating new object types. Likewise, it is actually very easy to use from the

application software stand point. For example, here is the code to create a default PegThing object:

```
PegThing *pThing = PegThingCreateDefault();
```

And that's it.

Internally, the execution flow followed the right side of the above diagram, but that is not the concern of the application code. So, PegThingCreateDefault just quietly does it's job and returns a fully formed pointer to a new PegThing object.

Here's the code to create a PegThing object that has size, position (both provided by the PegRect object), style and ID:

```
PegRect r = { 5, 20, 225, 140 };

PegThing *pThing = PegThingCreate(&r, ID_OBJECT_ID,
       FF_THIN);
```

This is very simple as well. The rectangle determines the size and position of the object, we give it an ID and tell it that its border style is thin.

Destruction

It is important for the application developer to live by a simple rule of thumb:

"If C/PEG created it, then C/PEG destroys it"

This is a golden rule of C/PEG and can not be overstressed. If an object is created using a create function in C/PEG, then that object must be destroyed using the object's complimentary destroy function. If this simple rule is not followed, the application will quickly suffer from memory leaks and execution issues and has no chance of not failing.

There are many reasons for this. First, some objects contain pointers to exterior data that will not be released back to the system if the object is freed by the application code. Second, the object will not be cleared from system message queue if freed by application code. This could quickly spell doom for the application if there are any messages bound for the freed object. Finally, if the object had a parent, the parent would be unaware the object has disappeared and will, at some point, attempt to access that object through it's child pointer.

For an illustration, if you call this:

```
PegThing *pThing = PegThingCreate(&r, ID, FF_NONE);
```

or this:

```
PegThing *pThing = PegThingCreateDefault();
```

You had better do this to destroy pThing:

```
PegDestroy(pThing);
```

It's as simple as that and extremely important to ensure the integrity of the system.

For a more in depth look at how C/PEG manages objects and memory, see the 'Rules of Memory Ownership' section below.

Implementation of the Function Pointers

There are 11 function pointers contained in the PegThing structure declaration. This allows for a elementary implementation of encapsulation, which is one of the corner stones of object-oriented programming. This encapsulation mechanism allows any PegThing derived object to implement any one of a group of functions to better suit the problem at hand, without the need for the object to register itself with the system.

As an example, the draw function which every PegThing derived object is able to customize has a function pointer assigned to it in the PegThing structure. If a new object wishes to draw itself differently than how a PegThing object draws itself, then the application developer only has to prototype a new function in the same fashion as the default draw function and assign that new function to the function pointer in the structure. This is done on an object by object basis and is usually part of the creation process for the object.

Without this mechanism, getting an object to look and behave in any way distinct from the default implementation would be a messy proposition. On most desktop GUI systems, the application developer must register a type with the system and provide a callback for system messages which are directed to that object. This often leads to application code that spends more time dealing with system restrictions and policy than it deals with the business of the actual application implementation. In C/PEG, the idea is to make developing custom objects natural and structured, without introducing a level of overhead which renders the notion of customization unusable.

The 11 functions are first described in detail to give you a better understanding of the tools that are available, then several examples are given in order to clarify when and how to override these functions to hone the object into a tool better suited to the application.

An important note before we move on; there are support functions for setting and calling these functions. It is strongly suggested the application developer exclusively make use of these support functions and does not touch the structure's function pointer members directly. This insulates the application code from the actual implementation of the mechanism for maintaining the function pointers within the structure. The following tables list the functions, (which are currently implemented as macros to save the overhead of a function call), used to call the individual PegThing functions as well as the function used to set the function pointers in the structure.

Function Prototype	Internal Function Pointer Called
`PegDraw(void *pThing)`	`funcDraw`
`PegNotify(void *pThing, const PegMessage *pMesg)`	`funcNotify`
`PegAdd(void *pParent, void *pAdd, PEGBOOL bRedraw)`	`funcAdd`
`PegAddToEnd(void *pParent, void *pAdd, PEGBOOL bRedraw)`	`funcAddToEnd`
`PegRemove(void *pParent, void *pRemove, PEGBOOL bRedraw)`	`funcRemove`
`PegDestroy(void *pThing)`	`funcDestroy`
`PegResize(void *pThing, PegRect *pRect)`	`funcResize`
`PegDrawBorder(void *pThing, PEGCOLORVAL background)`	`funcDrawBorder`
`PegDrawFocus(void *pThing, PEGBOOL bRedraw)`	`funcDrawFocus`
`PegEraseFocus(void *pThing)`	`funcEraseFocus`
`PegParentShift(void *pThing, PEGINT xshift, PEGINT yshift)`	`funcParentShift`

Table 9 (Function Pointer Wrapper Functions)

Based on these function wrapper macros, to call an object's add function, for instance, the application code should use this form:

```
PegAdd(pParent, pAdd, TRUE);
```

and never this form:

```
pParent->funcAdd(pParent, pAdd, TRUE);
```

Every PegThing and derivative supports all 11 of these function pointers. Some are implemented as custom functions specific to that object, while others defer the call to the structure from which they derived. It is standard practice, for instance, to pass a function call to a base object when the derived object wishes to implement the default behavior for a function call based on the status of the object. This type of function deferring is most evident within the notify construct, where a derived object wishes to catch some special messages and, at the same time, wishes C/PEG system messages to be processed by the base object.

As an example, if a PegThing derived object, MyObject implements a custom notify function, but wishes to pass C/PEG system messages to PegThing to implement, the notify function would look something like the following:

```
PEGINT MyObjectNotify(void *pThing, const PegMessage *pMesg)
{
     switch(pMesg->usType)
     {
     case PEG_SIGNAL(IDB_POPUP_BUTTON, PSF_CLICKED):
          /* do something when IDB_POPUP_BUTTON is pressed */
          break;
     default:
          return(PegThingNotify(pThing, pMesg));
     }
     return(0);
}
```

Notice that the custom notify function wishes to handle the instance where the `IDB_POPUP_BUTTON` is clicked, but nothing more. Every other message that is sent to the object by C/PEG is deferred to the base object from which MyObject was derived.

It is important to note that, in this case, it is incorrect for the object to call PegNotify instead of PegThingNotify. This is because by calling PegNotify, the MyObjectNotify function will be called, which would make this recursive and enter into an infinite loop. In all cases when overriding a base object's default function, it is correct to call the base object's default function directly

if the derived object wishes to optionally implement base object functionality and not call the generic function wrapper as listed in the above table.

There is more discussion of these principles in the programming section later in this manual.

The function used to set an individual function pointer is declared as the following:

```
PEGBOOL PegFuncPtrSet(void *pThing, PEGINT iFuncID, void
      *pFunc);
```

This allows the application developer to set the function pointer for any given PegThing derived object. The valid arguments for iFuncID are listed below.

Value Function Identifier	Set Function Pointer
PFP_NOTIFY	funcNotify
PFP_DRAW	funcDraw
PFP_ADD	funcAdd
PFP_ADDTOEND	funcAddToEnd
VPFP_REMOVE	funcRemove
PFP_DESTROY	funcDestroy
PFP_RESIZE	funcResize
PFP_DRAWBORDER	funcDrawBorder
PFP_DRAWFOCUS	funcDrawFocus
PFP_ERASEFOCUS	funcEraseFocus
PFP_PARENTSHIFT	funcParentShift

Table 10 (Valid Function Identifiers)

The third parameter is a pointer to a user defined function that will replace the default implementation of the particular function pointer in the object.

For example, to set the draw function in an object, the application developer would do the following:

1. Prototype the replacement function

```
void MyObjectDraw(void *pThing);
```

2. Set the function pointer in an object that has already been created:

```
PegFuncPtrSet(pMyObject, PFP_DRAW, MyObjectDraw);
```

3. Implement the MyObjectDraw function to do the custom drawing.

There are more detailed examples of using the function pointers in PegThing in the programming sections later in this manual.

Notify

Prototype:

```
PEGINT PegThingNotify(void *pThing, const PegMessage
       *pMesg);
```

This is the PegThing version of the notify function. This version implements handlers for most of the basic C/PEG system messages as defined in a previous section. Most objects derived from PegThing and implement a custom version of this function do so to handle mouse events or user input in a different manner than what is available here.

Draw

Prototype:

```
void PegThingDraw(void *pThing);
```

The default implementation of the PegThing's draw function merely calls PegDrawChildren, and therefore does not directly implement any draw operation on it's own behalf. This function is implemented differently for the remaining objects in the C/PEG library and, along with the notify function, is often customized by application level objects.

Add

Prototype:

```
void PegThingAdd(void *pParent, void *pAdd, PEGBOOL
       bRedraw);
```

This implementation provides the basic functionality necessary to add one PegThing or derivative to another. This function is rarely customized, and if it is, the custom function will almost always call this version of the function at some point to ensure the objects are parented correctly.

AddToEnd

Prototype:

```
void PegThingAddToEnd(void *pParent, void *pAdd, PEGBOOL
      bRedraw);
```

This is basically the same call as PegThingAdd, the exception being that the object is added at the end of the parent's child list, instead of to the front.

Remove

Prototype:

```
void *PegThingRemove(void *pParent, void *pRemove, PEGBOOL
      bRedraw);
```

The is the opposite of adding an object. Again, this function is usually not customized, and if so, this version is almost always called to make sure the objects were properly decoupled.

Destroy

Prototype:

```
void PegThingDestroy(void *pThing);
```

This function releases the memory associated with the PegThing or derived object. In the event the application developer creates a custom object whose size differs in any way from the base object upon which it is derived, the custom object should implement a custom version of the destroy function.

For instance, if the custom object has a pointer to some data which is allocated at run time, the custom object must define it's own destroy function to properly release the memory associated with the data pointer.

Resize

Prototype:

```
void PegThingResize(void *pThing, PegRect *pRect);
```

The PegThing version of this function properly updates the object's internal size and clipping rectangles. Also, if the object has been moved as well as resized (the top-left coordinates of the new rectangle do not match the top-left coordinates of the object's internal rectangle), then the object's children,

if any, are alerted their parent object has been shifted by calling PegParentShift for each child object.

DrawBorder

Prototype:

```
void PegThingDrawBorder(void *pThing, PEGCOLORVAL
       background);
```

The PegThing version of this function is able to draw the raised, recessed and thin border styles that most PegThing objects use. (The thick frame, as on the PegPanel, is supported by the PegPanel version of this function).

If the custom object wants to display itself in a rectangular type of border, then this function may be called from within the draw function of the derived object.

DrawFocus

Prototype:

```
void PegThingDrawBorder(void *pThing, PEGBOOL bRedrawAll);
```

The PegThing version of this function does not actually do any drawing. Derived objects implement a custom version of this function for their own use. Most notably, PegButton derivatives implement drawing code that puts a thin rectangle just inside of their client boundary when the object is current to visually inform the user which object has focus.

EraseFocus

Prototype:

```
void PegThingEraseBorder(void *pThing);
```

As with the draw border function, the PegThing version of this function does not do any actual drawing.

ParentShift

Prototype:

```
void PegThingParentShift(void *pThing, PEGSHORT xshift,
       PEGSHORT yshift);
```

This function is called by the PegThingResize function when an object that has children is moved or resized and the new placement or size forces the top-left coordinates of the object to change. The PegThing version of this

function updates the child objects positional rectangles, as well as calls the same function for it's child, if any.

This function is not usually customized, but may be, if needed. An example of when this might be necessary would be if the object maintains positional data that is relative to it's current placement on the screen, like chart points on a graph. If the parent object is moved, then this object would need to be informed and update it's own chart points relative to the new position on the screen.

Overrides and Deferrals

The above listed functions are available in every PegThing derived object. The object may not provide a object specific implementation of the function, instead deferring the call to the object type from which it is derived, but it is never an error to refer to the object's version of the function.

For example, the PegButton object, which is derived from PegTextThing, which is itself derived from PegThing, does not directly implement the add function. It, instead, defers the function call to it's base object, PegTextThing, to handle. In this case, PegTextThing does not actually implement the add function, either. Instead, PegTextThing defers the call to PegThing. This is where the call actually lands. But, it is not an error for the application call to do this:

```
PegButtonAdd(pButton, pAdd, TRUE);
```

This is because C/PEG is built to know this really means to call PegTextThing's add function, which C/PEG also knows is a call to PegThing's add.

This is important for a number of reasons. The first is this allows the application developer to follow a very simple set of rules when overriding functions to create custom objects. When overriding functions, it is always safe to call the implementation of the function from which the custom object was derived. Therefore, the following is safe to do:

```
void MyObjectAdd(void *pParent, void *pAdd, PEGBOOL bRedraw)
{
    /* do some kind of custom processing */

    /* call the base object's version to do the C/PEG work */
    PegButtonAdd(pParent, pAdd, bRedraw);
}
```

Of course, this makes the assumption MyObject derived from PegButton.

This provides a clear roadmap to application developers. Instead of searching through header files finding which object implemented which function and who derived from who, it is far easier to always remember to call the object from which the custom object was derived.

Second, this provides an alternative to the function call wrappers which call the current object's version of a function. To illustrate, the previous example will be examined a little closer.

There is a macro that calls the current object's version of the add function, PegAdd. Easy enough. During the course of creating a PegButton object, the function pointers for the PegButton object are set up to point to, first, PegThing functions, then a few from PegTextThing, then a few PegButton implements on it's own. It would be very difficult for the application developer to remember which function is mapped to which of the object's function pointers and how to call the function and so forth. Therefore, there is a set of macros which coincide with each of the function pointers in PegThing objects. They follow the same form as the actual function prototypes as shown above: PegAdd looks just like PegThingAdd, but the object name was dropped. Easy enough, too.

The power of this is the application developer may always do the following to add any object to any other object, regardless of the `pParent` object's actual implementation of the add function:

```
PegAdd(pParent, pAdd, TRUE);
```

Under the hood, this macro expands to call the function that is pointed to by pParent's funcAdd pointer. This insures the correct function is always called given any sort of parent object. Easy enough, still.

The only place where making a call like this is a problem would be inside of the function pointed to by pParent's funcAdd pointer, and giving pParent as the first argument. That said, the treachery is obvious, right? This is not to say there may not be instances where this type of recursion is wrong, but they are probably far off edge cases that would infrequently show themselves.

So, to really drive home the lesson, don't ever do anything resembling this:

```
void MyObjectAdd(void *pParent, void *pAdd, PEGBOOL bRedraw)
{
    PegAdd(pParent, pAdd, bRedraw):
```

```
}
```
This would dig a very deep and dark hole very quickly from which the task would never be liberated.

This holds true for any code within any type of function override. It is always an error to call the callers version of the current function from within the callers version of the current function.

To solve this, the application developer may defer the call to the base object's version of the function, hence, this is correct:

```
void MyObjectAdd(void *pParent, void *pAdd, PEGBOOL bRedraw)
{
     PegButtonAdd(pParent, pAdd, bRedraw):
}
```

Of course, the question now is: “How much is this going to cost me?”

Here's the beauty: It's free. It costs nothing. Nada. Zilch. Zero.

And here's why. Every PegThing derived object, in the object's header file, implements a list of all of the function's that are implemented as function pointers in the object. Some of the list are actual function prototypes for functions the object is overriding. Every derived object overrides at least one function, otherwise, there wouldn't be a point to the object. The remainder of the list are macros that substitute a call to that object's version of a particular function with a call to it's base object's version of that function. These macros chain together until an actual function prototype is found, and that's where the preprocessor lands. Simple, robust and free.

To illustrate, we'll continue on with the PegButton example. Here's an excerpt from the PegButton header file:

```
PEGINT PegButtonNotify(void *pThing, const PegMessage *pMesg);
void PegButtonDraw(void *pThing);
#define PegButtonAdd(_t,_a,_d)        PegTextThingAdd(_t,_a,_c)
#define PegButtonAddToEnd(_t,_a,_d)   PegTextThingAddToEnd(_t,_a,_c)
```

And on it goes down the line until all of the functions have, in some way, been implemented.

Now, here's the same four functions as described in PegButton's base object's, PegTextThing, header file.

```
#define PegTextThingNotify(_t,_m)   PegThingNotify(_t,_m)
```

```
#define PegTextThingDraw(_t)          PegThingDraw(_t)
#define PegTextThingAdd(_t,_a,_d)     PegThingAdd(_t,_a,_d)
#define PegTextThingAddToEnd(_t,_a,_d)PegThingAddToEnd(_t,_a,_d)
```

Obviously, these are all implemented as macros which defer calls to PegThing. Of course, PegThing implements all of these functions in some way as demonstrated from the declarations for the example four functions listed here from PegThing's header file:

```
PEGINT PegThingNotify(void *pThing, const PegMessage *pMesg);
void PegThingDraw(void *pThing);
void PegThingAdd(void *pThing, void *pAdd, PEGBOOL bRedraw);
void PegThingAddToEnd(void *pThing, void *pAdd, PEGBOOL bRedraw);
```

From this we can draw the conclusion the function that the funcAdd pointer in the PegButton structure points to is PegThingAdd. So, by calling:

```
PegAdd(pButtonObject, pAdd, TRUE);
```

the call goes to PegThingAdd. Likewise, this is the equivalent:

```
PegButtonAdd(pButtonObject, pAdd, TRUE);
```

The first is done by directly accessing the object's function pointer, which is accurate at run time, the second is done with macros and the preprocessor takes care of the work.

This is not to suggest custom objects need to implement the preprocessor function table macros. These macros exist to give the application developer a clear direction for program execution when dealing with the foundation objects in the C/PEG library. They are not needed in the custom, derived, objects, since these objects are not used as base objects for even further derived objects.

We will spin the current example a little further to illustrate the point. From application code, to execute the MyObject's version of the add function, the call would look like this:

```
PegAdd(pMyObject, pAdd, TRUE);
```

Since this function is a macro that binds the call at run time. But, from within the custom add function for MyObject, to defer the call to the base object, the code would do this:

```
PegButtonAdd(pMyObject, pAdd, TRUE);
```

This is a logical partition of duties. For large applications, it is not unusual for a team of developers to work on custom objects, while another team works on implementing those objects and assembling the actual application. For this illustration, we will refer to these groups as designers and developers, respectively. This makes it very easy to draw this conclusion:

Designers call PegButtonAdd, developers call PegAdd.

A designer would call PegButtonAdd in the add function of MyObject to implement the standard C/PEG functionality to add one object to another. That's fine. The designer knows the custom object. He knows what it derives from. He knows how it works and when to call the base object's version of the function to achieve the results he's looking for.

The developer, on the other hand, may not know what MyObject derived from. Actually, he really shouldn't care. He should not be concerned with such details. All he should know is when to use MyObject in the application based on the requirements for the application's functionality at some given point. And, assuming the designer set up the object's function pointers properly, a call the PegAdd will get the job done for the developer.

To take this example a step further, the folks that write C/PEG would be considered the designers and the folks that use C/PEG to build applications are the developers. The only difference lies in the notion C/PEG is made to customize, so there has to be mechanisms in place to allow the designer group of the developer population to easily extend the C/PEG library objects. So, the macros are a convenience to the designer group of the developer population. This aids them in getting their job done without worrying about what object derives from what other object and when it's appropriate to call which function from the tree of derived objects that end at their custom object. (That sentence was meant to sound confusing, because the situation can be).

To summarize, C/PEG implements a deferral system to shield the developer community at large from the overwhelming task of committing to memory every C/PEG object and its list of functions. The application designers and developers, following very simple rules, are able to quickly and easily get up to speed with the basics of using C/PEG and are able to extend the objects in any way they wish.

Style and Status

Every PegThing object and derivative share a group of members that relate to the state of the object and how that object reacts to certain events.

The style of an object deals with how the object is presented on the screen. The type of frame an object has is part of it's style. How the text is aligned when drawn is part of the object's style. If the object is enabled or not is part of it's style. Basically, anything that has to do with an object's graphical representation is part of the object's style.

The status of an object is similar to it's style, but this state pertains more to how the rest of the C/PEG library sees this object. If the object may be resized or moved is part of it's status. If the object accepts user input focus is part of it's status. If the object is visible or current is part of it's status. Although some of these status flags may be taken into consideration in the graphical representation of an object, they are predominately used for C/PEG objects to discover capabilities and state within the C/PEG system.

The following tables lists valid style and status flags, as well as which type of objects pay attention to that particular flag. For any style or status flag, if a base object supports that flag, it's derivatives will as well. Therefore, if PegThing is listed as the supportive object, all objects derived from PegThing may also support that flag.

Style Flags

Style Flag	Meaning	Supportive Objects
FF_NONE	No border	PegThing
FF_THIN	Thin Border	PegThing
FF_RAISED	Raised, 3D looking border	PegThing
FF_RECESSED	Recessed, 3D looking Border	PegThing
FF_THICK	Thick, 3D Border	PegPanel
TJ_RIGHT	Text justify right	PegTextThing
TJ_LEFT	Text justify left	PegTextThing
TJ_CENTER	Text justify center	PegTextThing
TF_NONE	No close button on title bar	PegTitle
TF_CLOSE BUTTON	Close button on title bar	PegTitle
TT_COPY	Copy text data into private buffer	PegTextThing

Style Flag	Meaning	Supportive Objects
LS_WRAP_SELECT	Wraps list select to first when moving past last	PegList
BF_REPEAT	Button repeats sending clicked messages	PegButton
BF_SELECTED	Button is current pressed down	PegButton
BF_DOWNACTION	Button sends clicked message when pushed down	PegButton
BF_FULLBORDER	Adds a thin border around a button in addition to a raised border	PegButton
BF_ORIENT_TR	Controls text/bitmap placement	PegDecorated Button
BF_ORIENT_BR	Controls text/bitmap placement	PegDecorated Button
EF_EDIT	Allows editing of a string	PegEditField
PF_COPY	Copy point data into a private buffer	Peg2DPolygon
PF_FILLED	Draw polygon filled	Peg2DPolygon
MP_OK	Add an OK button	PegMessagePanel
MP_YES	Add a Yes button	PegMessagePanel
MP_NO	Add a No button	PegMessagePanel
MP_ABORT	Add an Abort button	PegMessagePanel
MP_RETRY	Add a Retry button	PegMessagePanel
MP_Cancel	Add a Cancel button	PegMessagePanel
SF_SNAP	Snaps a slider button to the next tick on move	PegSlider
SF_SCALE	Draw a scale on the slider's face	PegSlider
SF_VERTICAL	Orient vertically, Orients horizontally when not turned on	PegSlider
PS_SHOW_VAL	Show current value	PegProgressBar
PS_LED	Draw indicator using LED Style	PegProgressBar
PS_VERTICAL	Orient vertically, Orients horizontally when not turned on	PegProgressBar
PS_PERCENT	Draw the "%" symbol next to the current value	PegProgressBar

Style Flag	Meaning	Supportive Objects
SB_VERTICLE	Orient vertically, Orients horizontally when not turned on	PegSpinButton
MLP_SHOW_PARTIAL_ROW	Shows partial row at the top and/ or bottom	PegMLPrompt
AF_DRAW_SELECTED	Draws in a different background color when selected	PegButton, PegPrompt
AF_TRANSPARENT	Does not fill in a background frame when drawing	PegThing
AF_ENABLED	Draws enabled (normal) or disabled	PegButton

Table 11 (Valid Style Flags)

Most C/PEG objects take a style as a parameter to its create function. This is the easiest way to initially set the style of the object.

At run time, 3 functions may be used to add, remove and query styles of a PegThing and derived objects. These functions are as follows:

```
void PegStyleAdd(void *pThing, PEGUSHORT usStyle);
```

This function is used to add a style to the object. Style flags may be logically OR'd together to set several flags at once.

```
void PegStyleRemove(void *pThing, PEGUSHORT usStyle);
```

This function removes a given style from the object. Style flags may be logically OR'd together to remove several flags at one. This may also be used to clear all flags by passing `0xffff` as `usStyle`.

```
PEGBOOL PegStyleHas(void *pThing, PEGUSHORT usStyle);
```

This function queries the object to determine if a particular style is set. Style flags may be logically OR'd together to test for several styles at once. The function returns TRUE upon finding the style, FALSE if not.

In the current version of C/PEG, these functions are implemented as macros to save run time overhead.

It is strongly advised the application developer use these functions for style maintenance on an object and to not access the style member of the object directly.

Status Flags

Status Flag	Meaning	Supportive Objects
PSF_VISIBLE	Set if the object is visible, meaning that it is a member of a tree branch that can trace it's lineage to PegPresentation.	PegThing
PSF_CURRENT	Set if the object belongs to the branch on PegPresentation that currently has focus.	PegThing
PSF_SELECTABLE	Set if the object is selectable	PegThing
PSF_SIZEABLE	Set if the object may be resized by the user	PegThing
PSF_MOVEABLE	Set if the object may be moved by the user	PegThing
PSF_NONCLIENT	Set if the object is to not be considered as part of the client region of it's parent.	PegThing
PSF_ACCEPTS_FOCUS	Set if the object may receive focus	PegThing
PSF_TAB_STOP	Set if the object wishes to be on the list of child objects that may gain input focus when the user navigates through the child objects using the tab key	PegThing
PSF_OWNS_POINTER	Set when an object captures the input pointing device	PegThing
PSF_DRAWABLE	Set when the object is a member of the current branch of objects that has permission from PegPresentation to draw	PegTitle
PSF_ALWAYS_MAX_SIZE	Set for PegPanel objects that are always the maximum size of the screen	PegPanel
PSF_ALWAYS_ON_TOP	Set when an object requests to always be displayed on top of any other object	PegThing

Table 12 (Valid Status Flags)

As with the style flags, there are maintenance functions to handle adding, removing and querying the status flags of an object at run time.

Setting or removing status flags by the application code is highly discouraged. These flags are tightly maintained by the C/PEG library for each object. If the application code steps in and modifies this object member, the internal state of the object could very well be out of sync with how C/PEG is expecting to handle the object, and undefined behavior could easily show itself.

Likewise, the application code may query the status of an object at any time to determine execution paths in the application logic. The most often queried status is `PSF_DRAWABLE`. It is recommended for the application to check this status before attempting to draw. This small check may save a great deal of overhead when drawing complex objects. The PegScreen makes every attempt to stop objects that are not visible or do not have drawable status from actually doing any drawing. But, even then, there is overhead for every primitive graphics function that is unnecessarily called by the application.

Here is the form of the status query function:

```
PEGBOOL PegStatusIs(void *pThing, PEGUSHORT usStatus)
```

Again, as with the style functions, this function is implemented as a macro to save run time overhead. Status flags may be combined to check for more than one status at a time.

Signals

Signals are special types of messages which are, usually, sent from child objects to their parent in response to user interaction. The parent object will then catch these signals in its notify function providing a clear place for the application developer to introduce the logic of the system.

The following table lists the signal types used by C/PEG. As with style flags, if a signal is supported in a base object, then the objects derived from the base object will also support those signals.

Signals	Meaning	Supportive Objects
PSF_VISIBLE	Set if the object is visible, meaning that it is a member of a tree branch that can trace it's lineage to PegPresentation.	PegThing
PSF_CURRENT	Set if the object belongs to the branch on PegPresentation that currently has focus.	PegThing
PSF_SIZED	An object has been moved or resized	PegThing
PSF_FOCUS_RECEIVED	An object has received input focus	PegThing
PSF_FOCUS_LOST	An object has lost input focus	PegThing
PSF_KEY_RECEIVED	An object has received a key message that it does not support	PegThing
PSF_RIGHTCLICK	A right button mouse click occurred in the region of the screen owned by the object	PegThing
PSF_TEXT_SELECT	The user has selected all or a portion of a string	PegEditField
PSF_TEXT_EDIT	Each time the string is modified	PegEditField
PSF_TEXT_EDIT_DONE	Text modification is complete	PegEditField
PSF_CLICKED	Button selection	PegButton
PSF_CHECK_ON	Check is turned on	PegCheckBox
PSF_CHECK_OFF	Check is turned off	PegCheckBox
PSF_DOT_ON	Item is selected, or off	PegRadioButton
PSF_DOT_OFF	Item is not selected, or off	PegRadioButton
PSF_LIST_SELECT	List item is selected	PegList
PSF_SCROLL_CHANGE	Scroll position update	PegScroll
PSF_SLIDER_CHANGE	Slider position update	PegSlider
PSF_SPIN_MORE	Up or right button clicked	PegSpinButton
PSF_SPIN_LESS	Down or left button clicked	PegSpinButton

Table 13 (Signals)

As discussed above, these signals may be deciphered by the parent object using the PEG_SIGNAL macro.

There are several functions used for signal maintenance. These functions allow for adding, removing and querying signals as well as dealing with the object sending signals to its parent. These functions, the first four of which are implemented as macros, are listed below.

```
PEGBOOL PegSignalHas(void *pThing, PEGUSHORT usSignal);
```

Checks if the object supports sending a particular signal. Will return TRUE if it does and FALSE if it does not.

```
void PegSignalsSet(void *pThing, PEGUSHORT usSignal);
```

Sets the signals supported by the object. This is a bit different from simply adding a signal, in that this function does not preserve the original signal state of the object. Signals may be logically OR'd together to set several signals at once.

```
void PegSignalAdd(void *pThing, PEGUSHORT usSignal);
```

This adds a signal to the existing signal set supported by the object. Signals may be logically OR'd together to set several signals at once.

```
void PegSignalRemove(void *pThing, PEGUSHORT usSignal);
```

This function removes signals from the signal set supported by the object. Signals may be logically OR'd together to remove more than one signal at a time.

```
PEGBOOL PegSignalCheckSend(void *pThing, PEGUSHORT
       usSignal);
```

This function is used internally by the C/PEG objects to determine if a certain event should be signaled to its parent. If the application developer wishes to add new signal types to object, either stock library objects or custom objects, then this function should be called in situations where the object may wish to send a message to its parent. This function checks to make sure the object has an ID, a parent to which to send the message, if it supports the incoming signal and if its parent object is visible before sending a signal to its parent. These are very important checks and should not be by passed by application code.

```
void PegSignalSend(void *pThing, PEGUSHORT usSignal);
```

This function is called by PegSignalCheckSend when the above stated criteria for signal passing has been met by an object that wishes to send a signal to its parent. This function should not be called directly by application code, instead use PegSignalCheckSend.

Notifying Children

In dealing with messages, it is also necessary to point out when a parent object needs to inform each of its children of an event, the PegChildrenNotify function can be used. This function takes a PegMessage and notifies each child object with the message. This is useful, for example, when a parent object is moved and the `PM_PARENT_SHIFT` message needs to be sent to all of the parent object's children.

Here is the prototype for the function:

```
void PegChildrenNotify(void *pParent, const PegMessage
      *pMesg);
```

This may also be used by application code to implement custom behavior in response to C/PEG system messages or user defined messages.

Colors

Every PegThing object and derivative has four intrinsic colors that are used for drawing the objects background and text. Some objects also use these colors for determining colors such as the slider button color. The size of this color list is determined by the constant `PEG_THING_COLOR_LIST_SIZE`. The application designer is free to modify the size of the list used by PegThing objects to add more color choices for custom objects. Likewise, it is not advised for the list to be set to smaller than four, as this would break code in the C/PEG library. The constants for the indices and `PEG_THING_COLOR_LIST_SIZE` can be found in the file `pegtypes.h`.

The following table lists the indices used to access these colors and when these colors are used.

Color Index	Used To Draw
`PCI_NORMAL`	Background fill color when the object is not selected
`PCI_NTEXT`	Normal text

Color Index	Used To Draw
PCI_SELECTED	Background fill color when the object is selected. Some object optionally draw themselves in a different background color when selected.
PCI_STEXT	Text color when the object is selected. Some objects optionally support drawing their text in a different color when selected.

Table 14 (Color Indices)

There are two functions that are used to set and query which color is assigned to an index. They are as follows:

```
void PegColorSet(void *pThing, PEGUBYTE ubIndex, const
       PEGCOLORVAL c);
```

This function should be used to set the color associated with a certain index. The advantage to using this function as opposed to simply setting the value in the object in application code, is this function takes care to invalidate the screen region occupied by the object, so in subsequent drawing operations, this object is properly updated.

Alone, this function will not cause the object to be immediately redrawn. The object must have it's draw function called after the color has been set in order for the new color value to be used in drawing.

```
PEGCOLORVAL PegColorGet(void *pThing, PEGUBYTE ubIndex);
```

This function returns the color value associated with the specified index value. If the index value is out of bounds of `PEG_COLOR_LIST_SIZE`, then an invalid color value is returned.

Type

C/PEG implements a simple mechanism for determining an object's type at run time. This allows the library and application code the opportunity to discover an object's type and act on the object accordingly.

Types are broken down into two categories. The lower values are reserved for control types and the upper values are reserved for panel types.

The lower control types begin at the value of 1 and go up to `PEG_FIRST_USER_CONTROL_TYPE - 1`. These types are listed in the first table below.

Custom objects implemented by the application designer may also wish to have a unique type. This can be easily accomplished by the application designer by beginning new custom control types at `PEG_FIRST_USER_CONTROL_TYPE` and enumerating up. The ceiling for custom control types is `PEG_TYPE_PANEL - 1`, which is the boundary for the upper panel types used in C/PEG.

The upper control types are listed in the second table. The third table describes the upper and lower limits for user defined types.

Control Type	Control
PEG_TYPE_THING	PegThing
PEG_TYPE_TITLE	PegTitle
PEG_TYPE_BUTTON	PegButton
PEG_TYPE_HSCROLL	PegHorzScroll
PEG_TYPE_VSCROLL	PegVertScroll
PEG_TYPE_ICON	PegIcon
PEG_TYPE_TEXT_BUTTON	PegTextButton
PEG_TYPE_BM_BUTTON	PegBitmapButton
PEG_TYPE_TEXT_THING	PegTextThing
PEG_TYPE_RADIO_BUTTON	PegRadioButton
PEG_TYPE_CHECK_BOX	PegCheckBox
PEG_TYPE_PROMPT	PegPrompt
PEG_TYPE_VPROMPT	PegVertPrompt
PEG_TYPE_PROGRESS_BAR	PegProgressBar
PEG_TYPE_SCROLL_BUTTON	PegScrollButton
PEG_TYPE_COMBO_BOX_LIST	PegComboBoxList
PEG_TYPE_EDIT_FIELD	PegEditField
PEG_TYPE_SLIDER	PegSlider
PEG_TYPE_SPIN_BUTTON	PegSpinButton
PEG_TYPE_GROUP	PegGroup
PEG_TYPE_ML_TEXT_BUTTON	PegMLTextButton
PEG_TYPE_DECORATED_BUTTON	PegDecoratedButton
PEG_TYPE_2DPOLYGON	Peg2DPolygon
PEG_TYPE_LIST	PegList
PEG_TYPE_VLIST	PegVertList

Control Type	Control
PEG_TYPE_HLIST	PegHorzList
PEG_TYPE_COMBO_BOX	PegComboBox

Table 15 (Control Types)

Panel Type	Panel
PEG_TYPE_PANEL	PegPanel
PEG_TYPE_MESG_PANEL	PegMessagePanel
PEG_TYPE_PROGRESS_PANEL	PegProgressPanel
PEG_TYPE_HSCROLL	PegHorzScroll
PEG_TYPE_VSCROLL	PegVertScroll

Table 16 (Panel Types)

These constants should not be modified by the system developer. To do so would interfere with the PEG WindowBuilder application and how it views system objects.

Object Type	Lower Limit	Upper Limit
Control(PegThing)	PEG_FIRST_USER_CONTROL_TYPE	PEG_TYPE_PANEL - 1
Panel(PegPanel)	PEG_FIRST_USER_PANEL_TYPE	Maximum value of a system unsigned char, usually 255

Table 17 (User Type Limits)

All C/PEG objects have their type set in the object's initialization function and is part of the object creation process. Application developers should adhere to this functionality with custom objects.

There are two functions, which are implemented as macros, that deal with setting and querying an object's type. These are listed below:

```
PEGUBYTE PegTypeGet(void *pThing);
```

This function simply returns the type of the object.

```
void PegTypeSet(void *pThing, PEGBUYTE ubType);
```

This function sets the type of the object.

Traversing the Tree

It is often necessary to iterate through, or traverse, a list of PegThing objects. This feature is available due to the structure of the C/PEG object hierarchy as discussed in earlier sections.

All visible objects belong to a branch of objects that, at some point, traces it's lineage to PegPresentation. Any object type may be a parent or a child of any other object type, there is no artificial restriction on this in C/PEG. This alleviates the necessity of the application code to keep track of all of the objects it has created. If the object is visible, then PegPresentation knows about it and can return to the application code a pointer to that object.

All PegThing objects has a set of pointers that deal strictly with this list paradigm. Any PegThing object may be queried for it's parent, immediate sibling and first child by the application code. Indeed, this is how the internals of C/PEG works.

For example, consider the PegFind function, prototyped as follows:

```
PegThing *PegFind(void *pStart, PEGUSHORT usId, PEGBOOL
       bRecursive)
```

This function is used by application code to receive a pointer to a particular PegThing object who is a descendant of `pStart`, and may optionally recurs through the entire branch structure beneath `pStart`. Internally, this function uses the child and sibling pointers in the PegThing objects to search for a PegThing object whose ID matches usId. Without the basic linked list architecture of C/PEG, this would be an impossible task. But, with it, this function is very short and easily can recurs through any size tree to find the first case where the ID's match.

There are three support functions, all implemented as macros, which aid the application developer in working with these family pointers. It is suggested the application code use these functions to work with these pointers, in the event the PegThing member names may change in a future release.

```
PegThing *PegChildFirst(void *pThing);
```

Returns a pointer to the parameters first child. If the object has no children, this returns NULL.

```
PegThing *PegChildNext(void *pThing);
```

Returns a pointer to the parameters immediate sibling. If the object has no sibling, this returns NULL.

```
PegThing *PegParent(void *pThing);
```

Returns a pointer to the PegThing object's parent, or NULL if the object does not have a parent.

Example

Since this is such a fundamental part of using C/PEG, here is an example of traversing a list in order to find a particular object based on the objects type:

```
PegThing *FindObjectByType(PegThing *pStart,PEGUBYTE ubType)
{
  PegThing *pChild = PegChildFirst(pStart);
  while(pChild)
  {
    if (PegTypeGet(pChild) == ubType)
    {
      return(pChild);
    }
    pChild = PegChildNext(pChild);
  }
  return(NULL);
}
```

Text 8 (Finding an Object by Type)

For simplicity, this function does not recurs, nor does it check the validity of the starting PegThing pointer.

The important concept to see here is that to walk through a list of child objects for a given PegThing object, the code begins with getting the first child of the starting PegThing object. Subsequently, the child object is used to walk through the list of children, not the starting object. This is important because it exemplifies the chain of the list. The parent has only a pointer to its first child, which, in turn, can be used to walk through the sibling list of the child. Below is an illustration to clarify the point:

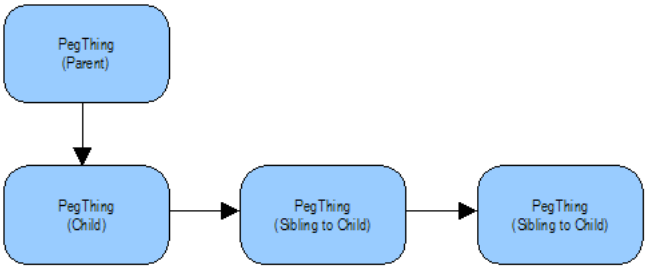

Illustration 6 (Child to Siblings)

The PegThing (Parent) object above has a pointer to its first child. This child has a pointer to its immediate sibling, who in turn has a pointer to its immediate sibling. The first child does not have a pointer to any sibling beyond the first sibling, and the parent object does not have a pointer to any child beyond its first child. But, all siblings have a pointer back to the parent object as their parent.

Finding System Objects

It is often necessary for application code to interact with the three fundamental system objects in order to get the job done. These objects are the PegPresentation, PegScreen and PegMessageQueue objects.

In any C/PEG application, there is always exactly one instance of PegPresentation and exactly one instance of PegScreen. It is highly recommended the application code use the following functions to access these objects, since these functions lock the resource before gaining a pointer to the objects.

```
PegScreen *PegScreenPtr(void);

PegPresentation *PegPresentationPtr(void);
```

These functions are fairly self explanatory. For most application code, the pointer to the PegPresentation is usually of more interest. This is because this object is always the root of the visible tree of PegThing objects, as discussed in the previous section.

For instance, given the PegFind example above, to search the entire tree of visible PegThing objects starting from the root, the application would make this call:

```
PegThing *pThing = PegFind(PegPresentationPtr(),
       ID_SOME_OBJECT,
  TRUE);
```

This call would begin the search at PegPresentation and recurs through all of it's children until it found an object with an ID or `ID_SOME_OBJECT`, or it reached the end of its children and their descendants and returned NULL.

The PegMessageQueue object is used by the application to send messages between objects and also to start up timers. Unlike the PegPresentation and PegScreen, there may be more than one PegMessageQueue object in use, although there is always exactly one system message queue.

To gain a pointer to the C/PEG system message queue, the application code would use the PegMessageQueuePtr call as prototyped below:

```
PegMessageQueue *PegMessageQueuePtr(void);
```

As mentioned, there may be more than one message queue in use by the application, but there is always only one primary system queue that is created as part of the C/PEG startup process. It is this queue that is returned by the PegMessageQueuePtr call.

For example, to start a timer for a given object, pThing, the application would make this call:

```
PegTimerStart(PegMessageQueuePtr(), pThing, ID_BUTTON, 10,
       10);
```

The details of this call are not important at this point, it is an example of using the PegMessageQueuePtr function.

CHAPTER 9

PROGRAMMING WITH C/PEG

This section introduces the basic concepts surrounding graphical application development using C/PEG. While by no means exhaustive, this section will get the average C programmer up and running with C/PEG in short order. There are also several example applications in the C/PEG distribution which demonstrate a particular object or concept which will take the basics presented here a step further.

9.0.1 C/PEG Naming Conventions

All user callable functions begin with the word 'Peg'. This serves two purposes: it prevents C/PEG function names from conflicting with that of a operating system or compiler library, and it makes C/PEG function calls self evident allowing the application developer to easily discern where C/PEG functions are being called.

The second portion or the function name may be one of two things. If the function deals directly with a specific type of object, then it is the object's name, as in PegThingCreate or PegBrushSet. If the function does not deal specifically with a certain type of object, then the action predicate makes up this section, as in PegFind or PegParent.

If the second portion of the function name is an object type, then the third portion is the action predicate that describes what is being done with the object. PegThingNotify and PegButtonDraw are examples of this.

If you can remember: “Peg” + Object Name + Action as a formula for deciding function names, then you can easily call a great majority of functions in the C/PEG library. To some developers this may seem awkward at first, but, after using the C/PEG API for a short length of time, the naming convention becomes very natural.

9.0.2 Source and Header Files

Most C/PEG objects are declared in a unique header file, with the implementation of the object is a similarly named source file. For instance, the PegThing object is declared in the `pthing.h` file and defined in the

`pthing.c` file. While this obviously makes it very easy to find where a particular object is defined and implemented, this also aids the application designer if they wish to leave out components of the library. For instance, if the PegGroup object is not used in the application, the application designer may opt to not compile the `pgroup.c` file into the library, ensuring the library and application do not include any PegGroup code.

Every C/PEG application only needs to include the header file `peg.h` to build against the C/PEG library.

9.0.3 Program Startup Review

In order for the C/PEG application to run, the PegPresentation, PegMessageQueue and PegScreen objects must first be created. This is usually completed by PegTask. There are several examples of this in the C/PEG distribution. The simplest of which is the startup code for running C/PEG in stand alone mode, perhaps with no operating system, which can be found in the `sapeg.c` source file.

After the required C/PEG objects are created, the PegAppInitialize function is called to allow the application developer to initialize the application. This segmentation was introduced in order to allow the application developer to easily move from one of the C/PEG development environments to the target platform without modifying any of the application level software.

9.0.4 Rules of Memory Ownership

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

It is often regarded as true most memory problems result from a lack of clear documentation of how and when allocated memory should be freed and by whom. There are, of course, also just bad programming practices, such as sharing pointers to globally allocated data blocks between unrelated objects, which usually leads to trouble.

For C/PEG objects, the rules are simple. When an object is added to another object which is either PegPresentation or a direct descent thereof, C/PEG can be said to own that object. The application does not have to worry about freeing that object as long as it remains a child, no matter how distant, of PegPresentation. C/PEG insures all children of PegPresentation,

along with PegPresentation itself, are freed when the object is closed or the application terminates.

Another side to this is by destroying a parent object, all of its children are subsequently destroyed. For example, if a PegPanel object is created by calling PegPanelCreate, C/PEG allocates the memory for the object and returns a pointer to the object to the application. At that point, let us say, the application code adds several child objects to the PegPanel object. At this point, all of the child objects are said to be owned by the PegPanel object. When the PegPanel object is destroyed by calling its destroy function, PegPanelDestroy being the default implementation of this function, then all of the child objects are destroyed along with it and all of the memory allocated for each object, including the PegPanel object, is freed.

Next, assume the PegPanel had been added to PegPresentation and is now visible. At this point, the application has given up all ownership of the panel and panel's child controls. C/PEG is now responsible for insuring that the panel and it's children are freed from memory when the panel is closed.

Finally, assume the application has manually removed the panel from PegPresentation, without allowing the panel to close itself in response to user input. In this case, the application owns the panel because PegPresentation no longer has any knowledge of the panels existence. However, the child controls of the panel are still owned by the panel; so, once again, when the application destroys the panel, the child objects are destroyed as well.

9.1 Creating PegThings

As stated before, all viewable C/PEG objects are derived at some point from PegThing. PegThing does not do much in terms of what is visible on the screen, but it is the common foundation that allows PegPresentation, PegScreen and PegMessageQueue to perform their tasks so effectively. This is also the underlying reason why C/PEG objects are so flexible.

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

The PegAdd function provides the mechanism for adding one object to another. When this function is called, an object is inserted into another

object's child list. If the parent object is visible, then the newly added child object becomes visible. The best way to create a complex panel is to create the panel, create all of the panel's child objects and then add the panel to PegPresentation. In this way, the panel and all of its children become visible at the same time.

PegPresentation is also a PegThing. This means that internally to C/PEG, there is no difference between adding a complex panel to PegPresentation and adding a simple text button to a panel. In both cases, it is simply a matter of adding one object to another.

A further result of the C/PEG object hierarchy is it is perfectly reasonable to create a C/PEG object which would normally be considered 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 would first display the text associated with the prompt, if any, and then allow its child objects to draw themselves. In this example, the PegButton would appear next to or over the prompt's text, depending on the dimensions of the prompt and button and the text justification flags the prompt used to draw. While this result may not appear very useful, it doesn't take much imagination to see by deriving a custom version of PegPrompt specifically for this purpose, powerful new object types can be easily created simply by combining these two C/PEG defined objects.

The following code illustrates the ease of creating and displaying a new panel using C/PEG. The panel will be created so that it occupies the entire screen and have a thick border.

```
void PegAppInitialize(PegPresentation *pPresent)
{
  PegRect r = { 10, 10, 100, 30 };
  PegPanel *pp = PegPanelCreate(NULL, ID_MY_PANEL, FF_THICK);
  PegAdd(pp, PegTextButtonCreate(&r, "OK", IDB_OK, AF_ENABLED);
  PegAdd(pPresent, pp, TRUE);
}
```

This code demonstrates the creation of a PegPanel object, adding a PegTextButton object to it, then adding the panel to PegPresentation.

It is important to note application code should never allocate C/PEG objects outside of their respective creation functions. Although it would probably not cause the application or system to fail, it yield unexpected consequences if the object is not properly allocated and initialized. The reverse of this,

destroying an object in application code by manually freeing the memory, would most certainly break the system and cause major problems.

9.2 Removing and Destroying PegThings

For some reason, freeing objects often causes more confusion and programming errors than creating them in the first place. C/PEG attempts to make removing and freeing GUI objects as painless and mistake free as possible.

The first thing to understand is removing a PegThing from its parent is not the same as destroying it. Removing the object means the application is taking that object out of the display tree. After being removed, the object no longer has a parent, and will not be visible. It is possible, even common, to later add the object back to a visible object and use it over again.

Objects may be removed from a parent by calling PegRemove and passing the parent, the child object and a flag which tells the parent object to redraw. While removing an object is useful, it is more common to want to both remove the object from it's parent and free the object from memory. There are two acceptable ways to remove and delete C/PEG objects:

1) Send a PM_DESTROY message to PegPresentation. The pSource member of the message should point to the object which is to be destroyed. This method is most often used when destroying C/PEG objects from tasks outside of C/PEG.
2) Call PegDestroy and pass a pointer to the object that is to be destroyed. Any PegThing object is able to destroy any other PegThing object in this manner. This function will first remove the object from it's parent, then release the memory associated with the object.

In no case should the application ever do something akin to free(pThing). This sort of application behavior would most certainly bring the system to a halt.

It is not necessary to manually destroy the individual children of a PegThing object, in fact, this will cause errors if attempted. If any of this discussion is not clear, please go back to the “Rules of Memory Ownership” section above.

9.3 Drawing to the Screen

Any visible C/PEG object is able to draw to the screen at any time using the C/PEG drawing functions. Drawing is most often done within the draw function of the object, but can take place within other functions and contexts as well.

The following is illustrative code for an object which draws its border, then draws two lines that form an “X” using it's own client region as line end points:

```
void MyThingDraw(void *pThing)
{
  PegBrush b;
  PegThing *pMyThing = (PegThing *)pThing;
  PegBrushSet(&b, BLACK, BLACK, CF_NONE, 1);
  PegDrawBegin(pMyThing);
  PegDrawBorder(pMyThing, PegColorGet(pMyThing, PCI_NORMAL));
  PegDrawLine(pMyThing, pMyThing->Client.sLeft,
    pMyThing->Client.sTop, MyThing->Client.sRight,
    pMyThing->Client.sBottom, &b);
  PegDrawLine(pMyThing, pMyThing->Client.sLeft,
    pMyThing->Client.sBottom, pMyThing->Client.sRight, pMyThing-
>Client.sTop, &b);
  PegThingDraw(pMyThing);
  PegDrawEnd(pMyThing);
}
```

Text 9 (Draw Example)

The important concepts here are drawing is always opened by calling the PegDrawBegin function. This informs the PegScreen object drawing operations are about to begin. Then, the actual drawing takes place. Since this object is, assumably, derived directly from PegThing, PegThingDraw is also called after the drawing operations. This gives the object's children a chance to draw. The draw operation is then concluded with a call to PegDrawEnd. This informs the PegScreen object this object has completed draw operations. If this is the last PegDrawEnd call that matches a PegDrawBegin call, and the video mode is set up to double buffer drawing operations, then the visible screen is then updated.

When C/PEG recognizes an object needs to be redrawn, it redraws the object by calling the objects draw function.

It is also possible to write functions that draw on the screen outside of the draw function. These functions must have access to a visible PegThing object to draw, since all drawing operations functions take a pointer to a PegThing derivative which is used as the drawing context.

PegScreen only allows drawing to occur to areas of the screen which have been invalidated. Areas of the screen are invalidated by calling the PegRectRegionInvalidate function. Under most circumstances, the screen invalidation is handled automatically by C/PEG as the user moves things around on the screen, or as the application code adds and removes visible objects. If all of the drawing is done from within a draw function, then the application need not worry about screen invalidation, since the draw function is called specifically because an area of the screen has been invalidated.

If the application needs to draw on the screen at random times, or for example, based on a periodic timer expiring, the application needs to invalidate the area of the screen where the drawing will take place before drawing commences, the PegRectRegionInvalidate takes any rectangle as a parameter. Most often, it is acceptable to use the client region of the object that wishes to draw. The application may wish to limit the drawing to an area smaller than this and may do so. No matter how large the invalidated rectangle on the screen, the object is never allowed to draw outside of its own borders.

The following is a function which illustrates a PegThing derived object drawing outside of the context of its draw function. 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 MyThingDrawLines(PegThing *pThing)
{
  PegBrush b;
  PEGINT yPos = pThing->Client.sTop;
  PegBrushSet(&b, RED, BLACK, CF_FILL, 0);
  PegRectRegionInvalidate(&pThing->Client);
  PegDrawBegin(pThing);
  PegDrawRectangle(pThing, &pThing->Client, &b);
  b.iWidth = 1;
  while(yPos <= pThing->Client.sBottom)
  {
    PegDrawLine(pThing, pThing->Client.sLeft,
      yPos, pThing->Client.sRight, yPos, &b);
```

```
    yPos += 4;
  }
  PegDrawEnd(pThing);
}
```

Text 10 (Drawing Outside of the Draw Function)

9.4 Determining Drawability

In order for a PegThing or derived object to draw, and to have the output of its drawing operations appear on the screen, the object must have a specific combination of status flags enabled. These flags govern, what is referred to as, drawability.

These two status flags are ultimately controlled by the PegPresentation object. When an object is made a descendant of the PegPresentation object, the object's status is modified to include the `PSF_VISIBLE` flag. The meaning of this flag is fairly self evident: The object is visible. In C/PEG, an object can not be considered to be visible unless it is a direct descendant of the PegPresentation object.

The second status flag may also seem as self evident as the first, but, there are a few more rules involved. The `PSF_DRAWABLE` status flag is set by the PegPresentation object on at least one branch of its child objects. That is to say, there is always at least one child on the PegPresentation object which has both `PSF_VISIBLE` and `PSF_DRAWABLE` status and is, therefore, drawable. This object is always the first child of the PegPresentation object, and this drawable status is always passed down the branch to every child of the drawable object.

Taking this a bit further, there also may be other child objects of the PegPresentation object who would also be considered visible, but may not be deemed as drawable. In order for a sibling of the drawable child to be drawable as well, its screen boundaries must not overlap the area occupied by the first drawable child of the PegPresentation object.

These simple rules provide for an efficient execution model whereby the application may determine object drawability by judicious use of size and placement.

For example, if the application required two sibling objects on the screen at the same time, both updating their displays based on some sort of external

input, the application designer should take care to insure these two objects do not overlap each other and are also not overlapped by a third object that may be considered to be “on top” of the first two objects.

Of course, if a single child of the PegPresentation occupies the entire screen, and is the first child of the PegPresentation object, then no other children of PegPresentation will be allowed to acquire drawable status until the first object is dismissed or resized.

The C/PEG API provides a macro that helps the application determine if a particular object is able to draw at a any given time. This macro is `PEG_CHECK_VALID_DRAW`. It is strongly suggested object drawing functions first check the return value of this macro before proceeding with any type of drawing operation. The macro is an implementation of two other macros that check the status flags of a PegThing or derived object and returns TRUE or FALSE regarding the drawability of the object. This makes for very little overhead and can potentially save a great deal of unnecessary object inspection by the low level drawing routines.

The application code should never try to cohere an object into being drawable. This could potentially have dire consequences as overlapping objects may corrupt the display of their siblings.

9.5 Object Boundaries

All PegThing derived objects have two rectangles associated with them named Real and Client. The rectangle, Real, defines the outermost limits of an object. The object and all children of the object are prevented from drawing outside the Real rectangle.

The Client rectangle defines the interior boundaries of the object. The Client rectangle is always a subset of the Real rectangle. All children of an object are clipped to the parent's Client rectangle. The exception to this is if the child has a status of `PSF_NONCLIENT`, in which case the child object is clipped to the parent object's Real rectangle.

For most objects, the Client rectangle is smaller than the Real rectangle by the width of the object's border. If the object has no border, then Client and Real are identical.

The rectangle that is passed to the objects creation function defines the outermost limits of the object, hence this rectangle becomes the Real

member rectangle. C/PEG objects initialize their Client area by calling the PegClientInit function, which reduces the Client area by the objects border width.

It is possible for the application developer to create their own non-client area decorations and add them to C/PEG objects. When this is done, the object will need to have the necessary logic to insure the Client rectangle is reduced correctly to allow space for the new non-client decorations.

9.6 Customizing Objects

Very few products developed by C/PEG developers will use every C/PEG object as it is implemented “stock, out of the box”. In this day of increasing embedded systems production, it is more important than ever for a product to distinguish itself from its competitors. There are many ways to differentiate a product. It can be the product provides functionality that is unrivaled in its field. It can be the design of the application is so appealing to customers that it is irresistible. Obviously, the Holy Grail of a product's design is when it achieves both.

For products which contain a graphical interface, the right design can help lead to both. This allows the system designers and engineers to implement advanced functionality and wrap it all up in an intuitive, user friendly package whose design immediately attracts buyers to the product and away from the competition.

To that end, a central part to the graphical user interface is a theme for the graphical objects which flawlessly implement the functionality of the product in a way which encourages the user to use the product and become comfortably familiar with the workings of the product very quickly. This leads to product loyalty and increased market share.

If the product is hard to use, then no one will use it.

C/PEG is built with this notion clearly at the forefront, easily accommodating the designer's ideas for how a GUI library should work. While the product developers are free to use the C/PEG objects as they come from our “factory”, most feel to really hit the sweet spot for the product, the C/PEG library objects need to be tweaked to give them a theme that will best represent the product in the market place.

There are two areas which are of the most importance when customizing a C/PEG object: functionality and visibility. How an object represents itself on the screen and how the object behaves, or the look and feel of an object, are the two most customized characteristics of C/PEG graphical objects.

Fortunately, C/PEG allows for this, to the point where it even expects this to occur. This means, to the application designer, implementing custom objects that override the default look and feel of the base object is a straight forward and repeatable process.

9.6.1 The Object Factory

The starting point for customization is always to find the stock C/PEG object that most closely resembles the look and feel that is intended for the custom object. Sometimes, this is a trade off between how close either criteria is met by any single object; meaning a given object may provide the functionality that is desired, but not the visibility, or the other way around. Once the application designer is familiar with the C/PEG library objects, this decision is usually a very easy one to make.

For an example, we will use a situation where an application designer wishes for an object that is able to display text and to draw it's background as a gradient between three color values. The object should also inform it's parent object when it is selected by the user, either by being clicked with the mouse, or activated with the keyboard. This may sound like a tall order, but it really is quite simple.

If we were to start at the top of the PegThing object tree, we would, of course, start with PegThing. In this case, this wouldn't be a bad place to start, but there are better options available. So, if we look at which objects are immediate derivations from PegThing, we would encounter the PegTextThing object. This better suits our needs closer because it already implements the necessary code to keep track of a NULL terminated string, but it does not have the functionality we need. But, we are getting closer.

Now, if we were to look at which objects derive from PegTextThing, which also have the text handling functionality that we need, we would see the PegButton object. Now we're on to something. The PegButton object knows how to send a `PSF_CLICKED` message up to its parent when it is selected by the user. So, we've arrived.

But, wait, PegButton really doesn't do too much in the drawing department. Sure, it has text handling capabilities it acquired from PegTextThing, and can draw its background and frame, but it doesn't know how to draw a gradient background. So, we move on to the children of PegButton. At which point we see none of the objects which derive from PegButton know how to draw a gradient background, either.

So, what do we have with PegButton? An object that supports text, but doesn't know how to draw it, can draw it's background and frame based on if it is selected or not, and knows how to send a message to it's parent when it is selected by the user. That's pretty close to what we need.

Now that we have the base C/PEG object that is pretty close to what is needed for the custom object, what do we do next?

There are two very clear paths to take from here. We'll introduce and expand on the first, which will make the second self evident. To start, here's some sample code:

```
1   void GradientObjectDraw(void *pThing);
2   /*-----------------------------------------------------*/
3   PegButton *pGradientObject = PegButtonCreate(&r, "Text",
4   IDB_GRADIENT_BUTTON, AF_ENALBED);
5   PegFuncPtrSet(pGradientObject, PFP_DRAW, GradientObjectDraw);
6   /*-----------------------------------------------------*/
7   void GradientObjectDraw(void *pThing)
8   {
9   PegDrawBegin(pThing);
10  PegButtonDraw(pThing);
11
12  /* custom gradiated draw code goes here */
13
14  /* followed by drawing the text */
15
16  PegDrawEnd(pThing);
17  }
```

Illustration 7 (Object Factory Code 1)

There are actually three distinct sections in the above code we will examine individually.

The first section is line 1. This is the declaration for the new object's draw function. This can go in a header file and be included wherever an instance of the custom object is made.

The second section goes from line 3 through to line 5. This is application code that creates a PegButton object and then assigns the object's draw method to GradientObjectDraw. This, in effect, will override the PegButton's draw function and cause the GradientObjectDraw function to be called wherever the object's instance of the draw function is called.

The third section is a snippet of code which does everything but really implement gradient drawing and text drawing (that would take up quite a bit of space and detract from the example being illustrated), and is on lines 7 through 17. The important items to note here is the code calls PegDrawBegin, then calls the PegButton's version of the draw function, then calls PegDrawEnd. If the application designer wishes to make the object completely custom in appearance, with rounded corners perhaps, then leave out the call to PegButtonDraw. This, in effect, will give the designer a clean slate without being incumbered by any style force on it by the base objects visual representation.

A note worth mentioning here is if an object implements a particular function in a way that is very similar to how the designer wishes the custom object to behave or draw, but, perhaps, the base C/PEG object does some things in the function that is undesirable, it is perfectly fine to copy and paste appropriate portions of the C/PEG implementation for the function into the custom function.

For example, in this case, to do the actual text drawing, there is no reason to re-invent the wheel. There are several objects which draw text, PegTextButton and PegPrompt are two such objects. It is usually a good idea to look at how these objects draw text and adapt this code to the custom object for text drawing.

That's a simple example of overriding a function to make a particular instance of an object do something custom. Declare a function, declare an instance of an existing C/PEG object, substitute the necessary function, implement the function to do what is necessary to fit with the design of the application, repeat.

But, what if the application needs to use this custom draw function for every button it puts on the screen? It is tedious and error prone to use the above

example in hundreds of different places in the application code. The answer to this is an object factory.

An object factory is simply a function which wraps this functionality so the implementation is invisible to the user of the object, and to partition object creation and initialization from the rest of the application.

C/PEG is full of functions like this, since every PegThing object has two of them. PegThingCreate and PegThingCreateDefault are two examples of object factories. These functions allocated memory and properly initialize the object for use by the application. It should also be noted an object factory implies an object garbage collector as well. C/PEG also has these in the form of PegThingDestroy and derived functions. In short, if the designer makes the objects, the designer should provide facilities to destroy the objects. In the above example, it would be appropriate to call PegButtonDestroy for the object since all that changed from a standard PegButton object was a function replacement.

So, let's revisit the example, this time using an object factory to see what's different.

```
1   GradientObject *GradientObjectCreate(PegRect *pRect,
2    PEGUSHORT usId);
3   void GradientObjectDestroy(GradientObject *pGradientObject);
4   void GradientObjectDraw(void *pThing);
5   /*-------------------------------------------------------------*/
6   GradientObject *GradientObjectCreate(PegRect *pRect,
7    PEGUSHORT usId)
8   {
9   GradientObject *pgo = (PegGradientObject*)PegButtonCreate
10  (pRect, usId, AF_ENABLED);
11  PegFuncPtrSet(pgo, PFP_DRAW, GradientObjectDraw);
12  }
13  /*-------------------------------------------------------------*/
14  void GradientObjectDestroy(GradientObject *pGradientObject)
15  {
16  PegButtonDestroy((PegButton *)pGradientObject);
17  }
18  /*-------------------------------------------------------------*/
19  void GradientObjectDraw(void *pThing)
20  {
21  PegDrawBegin(pThing);
22  PegButtonDraw(pThing);
23
24  /* custom gradiated draw code goes here */
25
26  /* followed by drawing the text */
27
28  PegDrawEnd(pThing);
29  }
30  /*-------------------------------------------------------------*/
31  GradientObject *pGradientObject = GradientObjectCreate(&r, ID);
32  /* work with the object */
33  GradientObjectDestroy(pGradientObject);
```

Illustration 8 (Object Factory Code 2)

It may look like a little more code, but it is definitely worth the trade off.

Lines 1 through 4 are the function declarations in a header file.

Lines 6 through 12 show the object factory itself. This insulates the caller from knowing the details of the GradientObject's implementation,

encapsulating the functionality of creating one of these object types from the rest of the application.

Lines 14 through 17 show the destroy function. In this example, we are deferring the call to the base object.

Lines 19 through 29 are pretty much the same as in the previous example.

Finally lines 31 through 33 show application code for creating a GradientObject at run time, then destroying it.

The power of this is the level of control over the object it gives the object designer. At any time, the object designer could choose to modify the GradientObject, by changing the function pointers or adding data members, and it would not directly effect any application code that uses the object and expecting the “old” version of the object.

The C/PEG library works on this same principle. The more functionality that can be encapsulated into small pieces that can be artfully assembled by the application developer, the more productive the developer becomes and the better the application is as a result.

To take this one further step to reinforce the object factory concept, we will add some data members to the GradientObject.

Let's say marketing comes back to the application engineers and want the buttons to draw text using two fonts, instead of the one font the object currently uses. Not an unreasonable request, especially from marketing.

So a member of the application engineering team draws the short straw and is tasked with implementing the change.

If the application is already using object factories, which this one is, then this is a trivial change and will not effect any other application code.

Here's some code which extends the GradientObject object by adding a new PegFont pointer data member and assigning that font to the object when the object is created.

```
1   typedef struct
2   {
3   PEG_BUTTON_DECLARE \
4   PegFont *pBigFont;
5   } GradientObject;
6
7   GradientObject *GradientObjectCreate(PegRect *pRect,
8    PEGUSHORT usId);
9   void GradientObjectDestroy(GradientObject *pGradientObject);
10  void GradientObjectDraw(void *pThing);
11  /*-------------------------------------------------------------*/
12  GradientObject *GradientObjectCreate(PegRect *pRect,
13   PEGUSHORT usId)
14  {
15  GradientObject *pgo =
16  (GradientObject*)PEG_ALLOC(sizeof(GradientObject));
17
18  PegButtonInit((PegButton *)pgo);
19  PegButtonSet((PegButton *)pgo, pRect, usId, AF_ENABLED);
20
21  PegFuncPtrSet(pgo, PFP_DRAW, GradientObjectDraw);
22  pgo->pBigFont = &NewBigFont;
23  }
24  /*-------------------------------------------------------------*/
25  void GradientObjectDestroy(GradientObject *pGradientObject)
26  {
27  PegButtonDestroy((PegButton *)pGradientObject);
28  }
29  /*-------------------------------------------------------------*/
30  void GradientObjectDraw(void *pThing)
31  {
32  PegDrawBegin(pThing);
33  PegButtonDraw(pThing);
34
35  /* custom gradiated draw code goes here */
36
37  /* followed by drawing the text */
38
39  PegDrawEnd(pThing);
40  }
41  /*-------------------------------------------------------------*/
42  GradientObject *pGradientObject = GradientObjectCreate(&r, ID);
43  /* work with the object */
44  GradientObjectDestroy(pGradientObject);
```

Illustration 9 (Object Factory Code 3)

This brings us to the final step in customization; modifying the base object itself, not just which functions the object uses for the look and feel particular to that object.

Lines 1 through 5 introduce a concrete example of deriving a custom object. Every C/PEG object is declared with a macro that consists of the object name followed by the word “DECLARE”. In this case, of course, the macro is `PEG_BUTTON_DECLARE`. For PegTextThing this macro is `PEG_TEXT_THING_DECLARE` and so forth. The macro for any object includes the data members particular to that object and all of the member variables and function pointers of all of its base members in the same order in which they are declared in the base object.

For example, the `PEG_BUTTON_CREATE` macro looks like this:

```
#define PEG_BUTTON_DECLARE \
  PEG_TEXT_THING_DECLARE
```

This may seem redundant, since PegButton does not include any data members of its own, but this allows the application designer to use the `PEG_BUTTON_DECLARE` macro without much concern if this macro changes over time or through iterations of the C/PEG library.

For instance, the PegComboBox declaration macro looks like this:

```
#define PEG_COMBOX_BOX_DECLARE \
  PEG_THING_DECLARE \
  PegComboBoxInsertFuncPtr funcInsert; \
  struct _PegComboBoxList *pList; \
  struct _PegBitmapButton *pOpenButton; \
  struct _PegBitmap *pBitmap; \
  void *pLastSelected; \
  PEGBOOL bOpen; \
  PEGSHORT sOpenHeight; \
  PEGSHORT sCloseHeight;
```

This macro could change and any application designer using a PegComboBox in their application would be unconcerned with the modifications.

This type of encapsulation is one of the great powers of C/PEG. You'll notice the only real change in the GradientObject code example was the addition of the new structure definition and changes in the object factory

function. The application code did not have to change. The object's destroy function did not have to change because the object did not allocate any new data for its new data member. If it were to add another member which did allocate memory, then the memory could be freed in the destroy function before calling the base object's destroy function.

In the object factory function, lines 12 through 23, there are some changes which are of interest. First, the function allocates its own object memory. This is in lieu of calling PegButtonCreate. It then calls two PegButton functions to initialize the object and to set its data members. It is good practice for any custom object to call its base object's initialize and set functions. Then, the new member pointer, `pBigFont` is set to point to the PegFont NewBigFont.

Of course, the draw function would need to change in order to draw the text in two different font faces, but is outside the scope of this discussion.

To conclude, the C/PEG object factory concept is a powerful mechanism for application object designers that need to extend and expand the base look and feel of any C/PEG library object.

9.6.2 Programming Examples

Please see the cpeg/examples directory for example programs which demonstrate object handling and application execution.