PEG®

Portable Embedded GUI

API Reference Manual

PEG Pro/PEG+ Library Release 2.4.0

Rev. 5

Feb. 2014

Table of Contents

FORWARD

We at Swell Software thank you for choosing PEG!

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

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

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

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

How the manuals are organized

Your documentation set includes four separate manuals:

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

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

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

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

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

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

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

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

A username and password are required to download the manuals.

What PEG IS

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

PEG is Portable

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

PEG is Embedded

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

PEG is GUI

The PEG class library provides the building blocks for a powerful and extensible graphical user interface. Extensive thought and research have gone into the design of our product to ensure that you are receiving a library that is fully capable of supporting all of the advanced GUI features you need today, while also accommodating future enhancements.

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

What PEG is NOT

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

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

List of Examples:

CHAPTER 1

BASE CLASSES

PegMessageQueue
PegPresentationManager
PegResourceManager
PegScreen
PegTextThing
PegThing
PegTimerManager

1.1 PegMessageQueue

1.1.1 Overview

PegMessageQueue is a simple encapsulated FIFO message queue with member functions for queue management. PegMessageQueue provides functions for sending and receiving PegMessage formatted messages. PegMessageQueue also performs timer maintenance and miscellaneous housekeeping duties.

PEG messages can be divided into two types. PEG system messages, Which are generated internally by PEG to control and manipulate PEG objects, and USER messages, which are defined and used by your application program. Whether a message is a system message or a user message is determined by the value of the message `Type` field. This is a 16-bit unsigned value. PEG reserves message `Type` values 1-`FIRST_USER_MESSAGE` - 1, which is currently equal to 0x4000. Message `Type` values which are greater than `FIRST_SIGNAL` (0x8000) are used for signals, which will be defined later. This leaves message types 16384 (0x4000) through 32767 (0x8000) available for user definition.

Integrated versions of PEG provide PegMessageQueue functionality based completely on the underlying RTOS message services. This implementation is invisible to the external system software, allowing PEG applications to be fully portable between development and real-time environments.

1.1.2 System Message List

The following, while not a complete list of the PEG system messages, are the system messages which would potentially be of interest in the application-level software.

Message	Description
PM_ADD	This message can be issued to add an object to another object. The message `pTarget` field should contain a pointer to the parent object, and the message `pSource` filed should contain a pointer to the child object.
PM_ADDICON	System message sent from a window to its parent to add an icon for a minimized window.

Message	Description
PM_BEGIN_MOVE	System message sent from a title bar to its parent window to begin moving the window. This message can be sent by any child object to force the parent window into 'move mode.'
PM_CLOSE	Recognized by PegWindow derived objects, and causes the recipient to remove itself from its parent and delete itself from memory.
PM_CLOSE_SIBLINGS	
PM_COPY	Sent by keyboard drivers to command a text copy operation.
PM_CURRENT	This message is sent to an object when it becomes a member of the branch of the presentation tree which has input focus.
PM_CUT	Sent by keyboard drivers to command a text cut operation.
PM_DESTROY	This message is sent to PegPresentationManager to destroy an object. The `pSource` member of the message should point to the object to be destroyed.
PM_DIALOG_APPLY	
PM_DIALOG_NOTIFY	This message is sent to the owner of a PegDialog when the dialog window is closed if the dialog window is executed non-modally. The message `Param` member will contain the ID of the button used to close the dialog window.
PM_DRAW	This message can be sent to an object to force that object to redraw itself.
PM_EXIT	This message is sent to PegPresentationManager to cause termination of the application program.
PM_GAINED_KEYBOARD	This message is sent to an object when it gains keyboard input focus. This message is only defined when `PEG_KEYBOARD_SUPPORT` is enabled.
PM_HIDE	This message is sent to an object whenever it is removed from a visible parent.
PM_HSCROLL	This message is sent from a non-client scroll bar to its parent window to effect scrolling of the window.
PM_KEY	This message is sent to the current input object when keyboard input is received. The message `Param` member contains the corresponding ASCII character code, if any, and the `ExtParams[0]` member of the message contains the keyboard scan code, if available.
PM_KEY_RELEASE	This message is sent to the current input object when keyboard input, indicating a key release, is received. The message `Param` member contains the corresponding ASCII character code, if any, and the `ExtParams[0]` member of the message contains the keyboard scan code, if available.

Message	Description
PM_LANGUAGE_CHANGE	This message is sent to all objects when the current language changes. It informs the objects that they need to update their text from the string table.
PM_LBUTTONDOWN	This message is sent to an object when the user generates mouse click input. PegPresentationManager routes mouse input directly to the lowest child object containing the click position. If the child object does not process mouse input, the message is passed up to the parent object. This process continues until an object in the active tree processes the message, or the message ends up back at PegPresentationManager. The position of the mouse click is included in the message `Point` field.
PM_LBUTTONUP	This message is sent to an object when the user releases the left mouse button. The flow of this message is identical to `PM_LBUTTONDOWN`.
PM_LOST_KEYBOARD	This message is sent to an object when it loses keyboard input focus. This message is only defined when `PEG_KEYBOARD_SUPPORT` is enabled.
PM_MAXIMIZE	This message can be sent to any PegWindow derived object. If the target window is sizeable (as determined by the `PSF_SIZEABLE` status flag), it will resize itself to fill client rectangle of its parent.
PM_MINIMIZE	Similar to `PM_MAXIMIZE`, this message can be sent to any PegWindow derived object. If the window is sizable, it will create a proxy PegIcon, add the icon to the parent window, and remove itself from its parent.
PM_MOVE_FOCUS	This message is sent by PegPresentationManager to itself when the top-level window which has input focus is closed.
PM_MWCOMPLETE	This message is sent to the owner of a PegMessageWindow when the message window is closed if the message window is executed non-modally. The message `Param` member will contain the ID of the button used to close the message window.
PM_NONCURRENT	This message is sent to an object when it loses membership in the branch of the presentation tree which has input focus.
PM_PARENTSIZED	This message is sent to all children of a PegWindow derived object if the window is resized. This makes it very easy for child windows that want to maintain a certain proportional spacing or position within their parent to catch this message and resize themselves whenever the parent window is sized.

Message	Description
PM_PASTE	Sent by keyboard drivers to command a text paste operation.
PM_POINTER_ENTER	This message is sent to an object when the mouse pointer (if any) passes over an object.
PM_POINTER_EXIT	This message is sent to an object when the mouse pointer (if any) leaves the object.
PM_POINTER_MOVE	This message is sent to an object whenever the mouse pointer moves over the object.
PM_SHOW	This message is sent to an object when it is added to a visible parent, before the object is first drawn. This allows an object to perform any necessary initialization prior to drawing itself on the screen.
PM_SIZE	This message is sent to an object to resize it. This is equivalent to calling the `Resize()` function. Note that PEG does not differentiate between moving an object and resizing an object. Both are accomplished via the Resize operation. The new size for the object is included in the message `Rect` field.
PM_MOVE	This is an obsolete message, replaced by `PM_SIZE`.
PM_RESTORE	This message can be sent to any sizable PegWindow derived object to cause that window to restore its size and position after it has been maximized or minimized.
PM_RBUTTONDOWN	This message is sent in systems that support right mouse button input. PEG objects do not process right mouse button messages.
PM_RBUTTONUP	This message is sent in systems that support right mouse button input. PEG objects do not process right mouse button messages.
PM_REMOVETHING	This message can be sent from one object to another to remove the sending object from the display. This is an indirect method of calling the `Remove()` function, and can be useful if the target object needs to know that the sender is being removed.
PM_SLIDER_DRAG	This message is sent from the slider button on a PegSlider control to the PegSlider when the slider button is operated by the user.
PM_VSCROLL	This message is sent from a non-client scroll bar to its parent window to effect scrolling of the window.
PM_TIMER	This message is sent to an object that has started a timer via the PegMessageQueue `SetTimer` function when that timer expires. The ID of the timer is included in the Param member of the message.

1.1.3 See Also

PegMessage

PegTimer

1.1.4 Derivation

PegMessageQueue is a PEG base class.

1.1.5 Constructors:

```
PegMessageQueue(void)
```

This is the only PegMessageQueue constructor. One instance of PegMessageQueue is created during program startup, usually in PegTask. This queue instance is referenced by all PEG objects.

1.1.6 Public Functions:

```
void Fold(PegMessage *pIn)

void Fold(PegMessage *pIn, PEG_QUEUE_TYPE
     pQueue)
```

This function looks for a matching message in the queue, and, if one is found, updates the existing message to contain the data values of In. If a duplicate message is not found, `Fold()` calls `Push()` to place the message at the end of the queue. Messages are determined to be equal if the `pTarget`, `Type`, and `pSource` values of the messages are equivalent.The second version of the function can be used when `PEG_MULTITHREAD` is turned on. The value `pQueue` is the message queue that is searched.

```
PEGBOOL IsEmpty(void)
```

This returns TRUE if there are no messages waiting in the queue. If there are messages in the queue it returns FALSE.

```
void Pop(PegMessage *pPut)
```

This retrieves the top message from the queue, and copies it into the PegMessage pointer `pPut`.

```
void Purge(PegThing *pDel, PEGUSHORT Type = 0,
     PEGUSHORT Id = 0)
```

This function removes messages from the queue which have a `Mesg.pTarget` field matching `pTarget`. This is used to remove messages from the queue which are destined for objects which have been deleted. The messages that get purged can further be narrowed by specifying the `Type` or `Id` fields which get compared with the `Mesg.Type` and `Mesg.Param` fields respectively.

```
PEGBOOL Push(PegMessage *)

PEGBOOL Push(PegMessage &In)
```

These functions place a new message at the end of the queue. They return TRUE if the message was successfully pushed, and FALSE otherwise.

1.1.7 Examples:

The following example creates and sends a new PegMessage. The message will cause the target object to resize.

```
void SomeObject::ResizeWindow(PegWindow *pTarget, PegRect
NewSize)
{
    PegMessage NewMessage(pTarget, NewSize);
    MessageQueue()->Push(NewMessage);
}
```

The following example window creates a periodic timer when the window is made visible, receives periodic timer messages, and destroys the timer when the window is hidden:

```
MyWindow::Message(const PegMessage &Mesg)
{
    switch(Mesg.Type)
    {
    case PM_SHOW:
        SetTimer(TIMER_1, 100, 100);
        PegWindow::Message(Mesg);
        break;

    case PM_HIDE:
        KillTimer(TIMER_1
        );
```

```
        PegWindow::Message(Mesg);
        break;

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

1.2 PegPresentationManager

1.2.1 Overview

PegPresentationManager is a transparent background window that can be thought of as the desktop window for all PEG applications. PegPresentationManager keeps track of all of the windows and sub-objects present on the display device. In addition, PegPresentationManager keeps track of which object has the input focus (i.e. which object should receive user input such as keyboard input), and which objects are on top of other objects.

There is no limit to the number of windows and/or controls that may be present on the screen at one time.

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

1.2.2 See Also

Viewports

PegWindow

1.2.3 Derivation

PegPresentationManager derives from PegWindow.

1.2.4 Constructors:

```
PegPresentationManager(PegRect &Size)
```

This constructor determines the size in pixels of the PresentationManager window. This determines the outer limits of all of your child windows and controls. PegPresentationManager is normally sized to equal the pixel dimensions of the display device.

1.2.5 Public Functions:

```
virtual void Add(PegThing *pWhat, PEGBOOL Show =
    TRUE)
```

PegPresentationManager overrides the `Add()` function to set focus to newly added top-level windows.

```
void BeginSubTaskExecute(PegThing *pWin)
```

This function creates a PegTaskInfo structure to keep track of which objects belong to which task using which PegMessageQueue. It is called from the `Execute` function. This is only available when `PEG_MULTITHREAD` is turned on.

```
virtual void CapturePointer(PegThing *pWho)
```

This function can be called to capture all mouse pointer input. This is done by modal windows, during resize and move operations, and can also be done by the application-level software at any time. The application software should call `ReleasePointer()` to end the `CapturePointer()` operation.

```
void ClearScratchPad(void)
```

This function resets the PresentationManager scratchpad buffer, and frees the associated memory.

```
virtual PEGINT DispatchMessage(PegThing *pFrom,
    PegMessage *pSend)
```

This function routes a message to the correct object. The routing follows a predetermined precedence order:

1) If the message has a non-NULL `pTarget`, route to `pTarget`.

2) If the message is a mouse pointer message, route to lowest level object that contains pointer position.

3) If the message is any other system message, route to object which has focus.

4) If the message is a User-defined message, and the `Param` member is not 0, `Find()` the object with ID matching `Param` and route to that object.

```
void DrawInvalid(void)
```

This function is responsible for drawing all objects that have been invalidated so far. It will be called automatically whenever the MessageQueue is empty.

```
void EndSubTaskExecute(PegThing *pWin)
```

This function removes the current PegTaskInfo structure from the task info list. If no other objects are using this object's message queue, the queue gets deleted. This function is only available when `PEG_MULTITHREAD` is turned on.

```
virtual PEGINT Execute(PEGBOOL AutoAdd = TRUE)
```

PegPresentationManager overrides the `PegWindow::Execute()` function to provide the main PEG message looping operation. The `AutoAdd` parameter is ignored. It is just there to maintain compatibility with `PegWindow::Execute()`.

```
PegThing *FindLowestThingContaining(PegThing
    *pStart, PegPoint Where)
```

This function determines the lowest child object that contains the point `Where`. The object must be enabled and selectable. A pointer to the child object is returned.

```
void FreeViewports(PegThing *pCaller)
```

This function is used to free all of the viewports used by the object `pCaller` as well as all of its children.

```
void GenerateViewportList(PegThing *pStart)
```

This function dissects `pStart` into different rectangular regions, called viewports, based on which viewport owners overlap it. This function is only available if `PEG_FULL_CLIPPING` is turned on.

```
PEG_QUEUE_TYPE GetCurrentMessageQueue(void)
```

This function returns the pointer to the current task's PegMessageQueue.

```
PegThing *GetCurrentThing(void)
```

This function returns a pointer to the leaf object that has input focus. This function will return NULL if no object has received focus.

```
virtual PegScrollDrawInfo *GetHScrollDrawInfo(
    void)
```

This returns a pointer to the current horizontal scrollbar drawing information structure.

```
PegThing *GetPointerOwner(void)
```

This function returns a pointer to the object that has currently captured the pointer.

```
const PEGCHAR *GetScratchPad(void)
```

This function returns a pointer to the text string that has been copied to the scratchpad, or NULL if no string is available. The PresentationManager scratchpad is used to cut, copy, and paste strings between PegEditField, and PegEditBox classes.

```
virtual PegScrollDrawInfo *GetVScrollDrawInfo(
    void)
```

This returns a pointer to the current vertical scrollbar drawing information structure.

```
PegTaskInfo *GetTaskInfo(void)
```

This returns a pointer to the current PegTaskInfo.

```
PEG_QUEUE_TYPE GetThingMessageQueue(PegThing*
    pTarget)
```

This returns the pointer to the PegMessageQueue used by `pTarget`.

```
PEGINT HScrollHeight(void)
```

This returns the height of the horizontal scrollbar.

```
void Invalidate(PegThing *pCaller, const PegRect
    &Rect)
```

This function is called to notify PegPresentationManager that a rectangular region of the screen needs to be redrawn. PegPresentationManager manages a list of all the regions that have been invalidated, and then when the MessageQueue becomes empty it redraws those regions. PEG objects normally invalidate the correct areas of the screen when they are modified via a resize, move, or other modification that requires the object to redraw itself. In some cases you will be required to invalidate the client area of your custom objects before they will be allowed to draw themselves.

```
PEGBOOL IsPointerCaptured(void)
```

Returns TRUE if the mouse pointer is captured, else FALSE.

```
PegThing *LastPointerOver(void)
```

Returns a pointer to the object the mouse pointer was last over.

```
void LastPointerOver(PegThing *pOver)
```

Sets the pointer to the object that the mouse pointer was last over.

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

PegPresentationManager overrides the Message() function to provide additional input focus adjustment and shutdown services.

```
void MoveFocusTree(PegThing *pCurrent)
```

Makes `pCurrent` the input object, and forces the branch of the display tree containing `pCurrent` to get focus. This function may be called at any time to move the input focus to any visible object.

```
virtual void ReleasePointer(PegThing *pWho)
```

Releases a captured mouse pointer. This function should always be called after the `CapturePointer()` function to return to normal operation.

```
virtual PegThing *Remove(PegThing *pWhat)
```

PresentationManager overrides the `Remove` function to check for the last object being removed, which signals PresentationManager to return from the `Execute()` function.

```
PEGBOOL RouteMessageToTask(PegMessage *pMesg)
```

This function determines where to send a message based on the `pMesg->pTarget` pointer and the `pMesg->wType` value. If it can find an object to send it to, and if that object has a message queue associated with it, then it pushed that message onto that queue. This function is only available if `PEG_MULTITHREAD` is turned on.

```
PegThing *ScreenPop(void)

PegThing *ScreenPop(PegThing *pWho, PEGBOOL
     Remove = TRUE, PEGBOOL Destroy = TRUE)
```

These functions remove an object from the screen stack and add it to the Presentation. The parameter `pWho` should be a pointer to whatever object is

already visible on the Presentation. That object can either be removed from the Presentation or destroyed.

```
PEGBOOL ScreenPush(PegThing *pOld, PegThing
     *pNew, PEGBOOL RemoveOld = TRUE, PEGBOOL
     Center = TRUE, PEGBOOL Modal = FALSE)
```

This function adds `pOld` to the screen stack and adds `pNew` to the Presentation in its place. The options are removing `pOld` from the Presentation, redrawing the screen, centering `pNew` on the Presentation, and making `pNew` execute modally.

```
void ScreenStackReset(PEGBOOL Remove = FALSE,
     PEGBOOL Destroy = TRUE, PEGBOOL Redraw =
     TRUE)
```

This function clears the screen stack by either removing or destroying the objects in it.

As stated, PresentationManager will terminate by sending a `PM_EXIT` message to itself if the last window or control is removed from the screen. When PresentationManager receives the `PM_EXIT` message it verifies no top-level windows exist, and, if not, it terminates the message loop. This operation can also be nullified by surrounding the call to `PegPresentationManager::Execute()` with a while(1) loop:

```
void PegTask(void)
{
    ...
    ...

    while(1)
    {
        Presentation->Execute();
    }
    ...
    ...
}
```

```
void SetExitCallback(void (*pCallback)())
```

This function sets a pointer to a callback function that will be called when PEG exits. This can be used to free up any loose memory that wouldn't normally be freed in an object's destructors.

```
void SetHScrollDrawInfo(PegScrollDrawInfo
     *pInfo)
```

This function assigns the horizontal scrollbar drawing information.

```
void SetScratchPad(const PEGCHAR *pText)
```

This function copies the indicated text string into the PresentationManager scratchpad buffer. This is used by user-editable text objects to support cut, copy, and paste operations.

```
void SetUserMessageHandler(PEGINT
     (*pHandler)(const PegMessage &Mesg))
```

This provides a way to catch user-defined messages in the default PresentationManager without needing to override it. A callback function pointer is passed so that it can be called whenever the PresentationManager receives a user-defined message.

```
void SetVScrollDrawInfo(PegScrollDrawInfo
     *pInfo)
```

This function assigns the vertical scrollbar drawing information.

```
void ThingDestroyed(PegThing *pCurrent)
```

This searches through the PresentationManager's list of invalid regions and removes any entries that were owned by `pCurrent`. That way it will not attempt to redraw that object after it has been destroyed.

```
void ThingRemoved(PegThing *pCurrent)
```

Resets the PresentationManager input focus status if `pCurrent` is the object that has input focus. Also resets input focus if a child of `pCurrent` object has input focus.

```
void ViewportChange(PegThing *pChanged)
```

This alerts the PresentationManager to the fact that the viewports have changed, usually because of adding/removing child objects or because of resizing.

```
PEGINT VScrollWidth(void)
```

This function returns the width of the vertical scrollbar.

1.2.6 Protected Members:

```
void AddViewport(PegThing *pTarget, PegRect
     &NewRect)
```

This function assigns a viewport to the target object. It checks to see if the viewport needs to be split because of a child object that also has viewport status partially or fully covering this object.

```
void AllocateViewportBlock()
```

This function allocates a block of viewports which get added to the list of free viewports.

```
void ChangeInputThing(PegThing *pWho)
```

This function gives the keyboard input device to the object referenced by Who. The object will receive a `PM_GAINED_KEYBOARD` message. The object, if any, that previously owned the keyboard will receive a `PM_LOST_KEYBOARD` message.

```
void ConsolidateInvalidList()
```

This function reduces the number of entries in the list of invalid regions by searching for any duplicate entries or overlapping child/parent entries. That way PEG will try not to redraw the same portion of the screen multiple times.

```
Viewport *GetFreeViewport()
```

This function returns a pointer to a viewport from the list of free viewports.

```
void InsureBranchHasFocus(PegThing *pCurrent)
```

This function ensures that `pCurrent` has input focus, and sends `PM_CURRENT` messages to the new input object if required.

```
virtual void KillFocus(PegThing *pStart)
```

Removes input focus from the specified object. Also sends `PM_NONCURRENT` messages if required.

```
virtual void SetFocus(PegThing *pStart)
```

Assigns input focus to the specified object. Also sends `PM_CURRENT` messages if required.

```
virtual void SplitView(PegThing *pTarget,
     PegRect Top, PegRect Bottom)

virtual void SplitView(PegThing *pTarget,
     PegThing *pChild, PegRect Under)
```

These functions are responsible for dividing up the viewports of an object due to the fact that it is being overlapped by another viewport owner.

```
PegThing *mpInputThing
```

Pointer to the object which has input focus, or NULL.

```
PegThing *mpLastPointerOver
```

Pointer to the object the mouse pointer was last over, or NULL.

```
PEGCHAR * mpScratchPad
```

Pointer to the current contents of the scratchpad buffer, or NULL.

```
PEGUBYTE mPointerCaptures
```

Indicates the current pointer capture nesting level. If there are no outstanding pointer captures, `mPointerCaptures` is 0.

1.3 PegResourceManager

1.3.1 Overview

PegResourceManager is a class responsible for maintaining all of the applications resources, which include its fonts, bitmaps, strings, and colors. All resources are given an ID used to reference those resources in the tables that the PegResourceManager uses. Most of the PEG library source code will use these IDs in the APIs rather than direct pointers to the resources themselves.

The main advantage to using IDs for everything is that it allows the user to switch which resources to use without needing to modify the application code. This is sometimes referred to as “skinning.” For instance, the application could be designed to allow user-selectable color schemes. To do that, the application would need to create different color resource tables for the different color schemes. Then it would simply need to call `InstallResourcesFromTable()` to start using a particular skin.

All of the functions of this class are static, so there is no need to ever create an instance of the PegResourceManager class.

1.3.2 See Also

PegBitmap

PegFont

1.3.3 Derivation

None.

1.3.4 Public Functions:

```
static void AddResource(const PegStringTablePage
     *pPage, PEGBOOL DeleteOld = FALSE)

static void AddResource(PEGUINT BitmapId,
     PegBitmap *pMap, PEGINT Flags = 0, PEGBOOL
     DeleteOld = FALSE)

static void AddResource(PEGUINT ColorId,
     PEGCOLOR Color)

static void AddResource(PEGUINT FontId, PegFont
     *pFont, PEGUINT Flags = 0, PEGBOOL
     DeleteOld = FALSE)
```

These functions are used to add resources to the PegResourceManager. Bitmaps, fonts, colors, and string table pages can all be added. The application can call these functions directly, but it is usually easier to build up a PegResourceTable structure and call the `InstallResourcesFromTable()` function. The `Flags` field is currently unused.

```
static void DestroyBitmap(PEGUINT BitmapId)
```

This function destroys the bitmap with the specified ID.

```
static void DestroyFont(PegFont *pFont)

static void DestroyFont(PEGUINT FontId)
```

This function destroys a font based on either its ID or a direct pointer to it.

```
static void DestroyAll(void)
```

This function destroys all of the tables in the PegResourceManager. It does not attempt to delete all of the actual resources because, usually, most of those are not dynamically allocated.

```
static PegBitmap *GetBitmap(PEGUINT BitmapId)
```

This function retrieves a PegBitmap pointer based on the specified ID.

```
static PEGUSHORT GetBitmapHeight(PEGUINT
     BitmapId)
```

This function returns the height of the bitmap with the specified ID.

```
static PEGUSHORT GetBitmapWidth(PEGUINT
    BitmapId)
```

This function returns the width of the bitmap with the specified ID.

```
static void GetBitmapWidthHeight(PEGUINT
    BitmapId, PEGUSHORT &Width, PEGUSHORT
    &Height)
```

This function retrieves both the width and height of a bitmap with the specified ID. This is a convenience function that does the same thing as calling both `GetBitmapHeight` and `GetBitmapWidth`.

```
static PEGCOLOR GetColor(PEGUINT ColorId)
```

This function returns the color value based on the color ID.

```
static PEGUINT GetCurrentLanguage(void)
```

This function returns the current language index. This is only available if `PEG_STRING_TABLE` is defined.

```
static char *GetLanguageName(PEGUINT Language)
```

This function returns the name of the language with the specified index. This is only available if `PEG_STRING_TABLE` is turned on.

```
static PEGUINT GetNumLanguages(void)
```

This function returns the number of languages in the string table. This is only available if `PEG_STRING_TABLE` is defined.

```
static PEGUINT GetFirstAvailableBitmapId(void)
```

This function returns the ID of the first empty slot found in the PegResourceManager's bitmap table.

```
static PEGUINT GetFirstAvailableColorId(PEGUINT
    StartVal = 1)
```

This function returns the ID of the first empty slot found in the PegResourceManager's color table.

```
static PEGUINT GetFirstAvailableFontId(void)
```

This function returns the ID of the first empty slot found in the PegResourceManager's font table.

```
static PegFont *GetFont(PEGUINT FontId)
```

This function returns a pointer to the font with the specified ID.

```
static PEGUSHORT GetStringResFontCount(void)
```

This function returns the number of fonts contained in the loaded string resource file. This is only available if `PEG_RUNTIME_RESOURCES` is defined.

```
static char *GetStringResFontName(PEGUINT Index)
```

This function returns the name of a specified font in the loaded string resource file. This is only available if `PEG_RUNTIME_RESOURCES` is defined.

```
static PEGUSHORT GetStringResStringCount(void)
```

This function returns the number of strings contained in the loaded string resource file. This is only available if `PEG_RUNTIME_RESOURCES` is defined.

```
static char *GetStringResUserString(void)
```

This function returns the user string contained in the loaded string resource file. This is only available if `PEG_RUNTIME_RESOURCES` is defined.

```
static PEGUSHORT GetStringResVersion(void)
```

This function returns the version number of the loaded string resource file. This is only available if `PEG_RUNTIME_RESOURCES` is defined.

```
static void Initialize(void)
```

This function initializes all of the tables used by the PegResourceManager by filling in the system colors, system fonts, and system bitmaps. This function should be called once at startup from the PegTask.

```
static void InstallResourcesFromTable(
     PegResourceTable *pTable, PEGBOOL DeleteOld
     = FALSE)
```

This function takes a PegResourceTable structure as input and loads the contained resources into the appropriate tables in the PegResourceManager. The PegResourceTable structure is a container for arrays of other structures containing the different types of resource data. Those structures look like this:

```
typedef struct {
    PegFont *pFont;
    PEGUSHORT FontId;
    PEGUBYTE Flags;
```

```
} PegFontTableEntry;

typedef struct {
    PegBitmap *pBitmap;
    PEGUSHORT BitmapId;
    PEGUBYTE Flags;
} PegBitmapTableEntry;

typedef struct {
    PEGCOLOR Color;
    PEGUSHORT ColorId;
} PegColorTableEntry;

struct PegStringTablePage {
    PEGUSHORT FirstSID;
    PEGUSHORT LastSID;
    PEGUSHORT NumLanguages;
    const PEGCHAR ***pTable;
    PegStringTablePage *pNext;
};

typedef struct {
    const PegStringTablePage *pStringTable;
    const PegFontTableEntry *pFontTable;
    const PegBitmapTableEntry *pBitmapTable;
    const PegColorTableEntry *pColorTable;
    void *pSpare1;
    void *pSpare2;
} PegResourceTable;
```

```
static PEGBOOL LoadResourceFile(char *pPathName,
     PEGBOOL DeleteOld = FALSE)
```

This function loads a file from the file system at runtime that contains new resource information. `pPathName` is the path and filename of the resource file. This is only available if `PEG_RUNTIME_RESOURCES` is defined.

```
static const PEGCHAR *LookupString(PEGUINT Id)
```

This function returns a string pointer based on the string ID.

```
static void SetCurrentLanguage(PEGUINT Language)
```

This function changes the current language. All objects on the screen will be sent a PM_LANGUAGE_CHANGE message to notify them to update their text from the string table. This is only available if PEG_STRING_TABLE is defined.

```
static PEGBOOL SetFontName(PEGUINT Index, char
     *pName)
```

This function assigns a name to a font with the specified ID. This is only available if PEG_RUNTIME_RESOURCES is defined.

```
static PEGBOOL SetLanguageName(PEGUINT Language,
     char *pName)
```

This function assigns a name to language with the specified ID. This is only available if PEG_STRING_TABLE is defined.

```
static void SetNumLanguages(PEGUINT Num)
```

This function resets the number of languages to Num and clears out all language names.

1.4 PegScreen

1.4.1 Overview

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

Class PegScreen is an abstract class from which the target specific PegScreen classes are derived. This hierarchy ensures that a consistent set of API functions are provided to PEG and application objects independent of the target environment.

Most often, the PegScreen member functions are used to draw directly to the video frame buffer. However, it is also possible to draw into a private PegBitmap, and then use the `PegScreen::Bitmap()` function to transfer the private PegBitmap to the video frame buffer. This technique is commonly used for displaying animation sequences, or for drawing on top of an image before displaying the image. The general sequence required for off-screen drawing is:

- Call the `CreateBitmap()` function to allocate a PegBitmap.
- Call the second form of `BeginDraw(..)`, passing in the bitmap you want to draw to.
- Call any of the standard drawing functions to draw into the private bitmap.
- Terminate the off-screen drawing by calling the second form of `EndDraw()`.

Once the above sequence is concluded, you have a PegBitmap that contains whatever custom drawing you have invoked. You can then display this PegBitmap at any location on the screen by calling the standard `PegScreen::Bitmap()` function. Once you are done with the PegBitmap and no longer need to display it, you should always use the `DestroyBitmap()` function to free all memory associated with the PegBitmap.

When running with a video controller that supports extended video memory and possibly a hardware bitblit engine, the PEG `CreateBitmap` function is optimized to create the bitmap in non-visible video memory. The PegScreen class contains a Video Memory Manager that keeps track of used and free areas of non-visible video memory. A flag is set in the PegBitmap structure to indicate that the bitmap resides in non-visible video memory. When you display the bitmap, the PegScreen class knows to use the video controller bitblit engine to display the bitmap rather than doing so via processor memory copy operations.

It is important to remember that when drawing to an off-screen PegBitmap, all drawing coordinates are relative to (0,0), which is the upper left corner position of the bitmap.

Class PegThing provides wrapper functions for the most commonly used PegScreen Drawing operations.

1.4.2 See Also

PegThing

1.4.3 Derivation

PegScreen is an abstract base class.

1.4.4 Constructors:

```
PegScreen(const PegRect &Size)
```

This constructor determines the size in pixels of the PegScreen frame buffer. This does not have to be the same dimension as PegPresentationManager, although this is the most typical case. One instance of the target-specific PegScreen-derived interface class is created during program startup. The PegScreen constructor is normally responsible for configuring the target video controller and initializing other variables used for improved drawing performance.

1.4.5 Public Functions:

```
PEGUBYTE AddPointerType(PegBitmap *pMap, PEGUINT
    XOffset, PEGUINT YOffset)
```

This function can be called to define a new pointer type, meaning a new pointer bitmap, when using a mouse or similar pointing device. The first parameter is a pointer to a PegBitmap to display when this pointer type is invoked via the `SetPointerType()` function. The x and y offset parameters specify the distance, in pixels, from the upper left corner of the bitmap to the bitmap 'hotspot.' The PEG screen drivers automatically position the pointer so that the hotspot is centered over the current pointer position.

The number of pointer types that may be defined is indicated by the definition `USER_POINTER_TYPES` in the header file pscreen.hpp. The return value of this function is the index assigned to this pointer type. The application must keep track of this index for subsequent calls to the `SetPointerType()` function. The value 0 is returned if the pointer cannot be installed.

```
virtual void Arc(PEGINT xc, PEGINT yc, PEGINT
    XRadius, PEGINT YRadius, PEGINT
    start_angle, PEGINT end_angle, PegBrush
    &Brush, PEGINT width = 1)
```

This function draws a portion of an ellipse from `start_angle` to `end_angle` (in degrees) at the indicated position and radius. The `width` parameter determines the border width, if any. This function does not use any floating point arithmetic. A pie-chart effect can be achieved by setting `Brush.Style` to `PBS_SOLID_FILL`. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual PEGINT BeginDraw(PegThing *pCaller,
    const PegRect &Invalid, PEGINT Surface)
```

This function should be called at the beginning of every `Draw()` function. This call prepares the screen for output by initializing the drawing context.

```
virtual void BeginPrint(void)
```

This method informs the screen that all subsequent drawing operations will take place in the context of printer output, as well as creating a new instance of the printer object. This function must first be called before any work on the printer object is done.

```
virtual void Bitmap(PegPoint Where, PegBitmap
    *pMap)
```

This function draws a bitmap on the screen. The parameter `Where` defines the upper left corner position of the bitmap on the screen. The parameter `pMap` points to a bitmap generated with the PegImageConvert utility.

```
virtual void Bitmap(PegPoint Where, PegBitmap
    *pMap, PegBitmap *pAlphaMap, PEGSHORT
    AlphaX, PEGSHORT AlphaY)
```

This Peg Pro only function alpha blends a bitmap on the screen using an alpha value map. The parameter `Where` defines the upper left corner position of the bitmap on the screen. The parameter `pMap` points to a bitmap generated with the PegImageConvert utility. The `pAlphaMap` pointer is the alpha value map. The parameters `AlphaX` and `AlphaY` define the starting offset within the alpha map.

```
virtual void BitmapFill(PegRect &Rect, PegBitmap
    *pMap)
```

This function fills a rectangular area of the screen with tiled copies of the bitmap pointed to by `pMap`. `Rect` defines the screen area to be filled, inclusive.

```
virtual void Capture(PegCapture *pInfo, PegRect
    &Rect)
```

This function captures a rectangular area of the screen and saves the copy in a PegCapture object referenced by pointer Info.

```
virtual void Chord(PegRect &Bound, PegPoint p1,
    PegPoint p2, PegBrush &Brush)
```

This function draws the intersection of an ellipse defined by `Bound` and the line defined by `p1` and `p2`. The `Brush` parameter determines the color and border width. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual void Circle(PEGINT xCenter, PEGINT
    yCenter, PEGINT r, PegBrush &Brush)
```

This function draws a circle at the indicated position and radius. The `Brush.Width` parameter determines the border width, if any. This function is only provided if the definition `PEG_FULL_GRAPHICS` is turned on in the file \peg\include\pconfig.hpp. This function does not use floating point math.

```
PEGBOOL ClipRect(PegRect &Rect)
```

This function takes the intersection of `Rect` with the `mpContext->Clip` rectangle. If the resulting rectangle has a positive area, then it returns TRUE; otherwise, it returns FALSE.

```
virtual PegBitmap *CreateBitmap(PEGINT Width,
     PEGINT Height, PEGINT Style = BMF_RAW|
     BMF_HAS_TRANS)
```

This function is called to create a PegBitmap at run time. The bitmap is normally formatted in the native pixel format of the active frame buffer;

however, this is transparent to the application software. A bitmap created in this fashion can be used as a 'virtual screen,' allowing drawing to the off-screen bitmap instead of to the actual video memory.

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

This function is called to create a drawing surface at run time. The surface could be used as an offscreen buffer to draw into, or it could represent a layer that gets displayed/blended on the screen.

```
virtual void DeleteFont(PegFont *pFont)
```

This function frees all memory associated with `pFont`. This function should be called after the use of a font created with `MakeFont` is complete. The `DeleteFont` function is only available if the definition `PEG_VECTOR_FONTS` is enabled when the PEG library is built. `PEG_VECTOR_FONTS` is defined in the file `\peg\include\pconfig.hpp`. The example program contained in `\peg\examples\vecfont` illustrates the use of `MakeFont` and `DeleteFont`. In this example, the font created at run time is assigned to a PegTextBox object. The font size and style can be modified freely at run time, without including additional bitmapped fonts (and the associated memory) in the application program.

```
virtual void DestroyBitmap(PegBitmap *pBitmap)
```

Deletes the bitmap associated with the `pBitmap` pointer. This method takes into account bitmaps that are using video memory directly, as opposed to bitmaps that are in system memory. This is the preferred method for deleting bitmaps that were created using the `CreateBitmap` method.

```
virtual void DestroyDrawSurface(PEGINT Surface)
```

Deletes the drawing surface associated with `Surface`. This is the preferred method for deleting surfaces that were created using the `CreateDrawSurface` method.

```
PEGUINT DrawNesting(void)
```

This function returns the current drawing nesting level.

```
virtual void DrawAliasMap(PegPoint Where, const
    PegBitmap *pMap, PegBrush &Brush)
```

This function uses the values in the bitmap `pMap` as alpha ratios to blend the color `Brush.LineColor` with whatever is on the background of the current surface.

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

This function draws the endpoints for line segments. If Brush.Style has the PBS_ROUNDED style flag set, then it draws circular bitmaps at the ends. If PBS_SIMPLE_ALIAS or PBS_TRUE_ALIAS are also set, it draws a circular alias map to make the circle appear anti-aliased.

```
void DrawRotatedText(PegPoint Center, PEGINT
    Angle, const PEGCHAR *pText, const PegFont
    *pFont, PegBrush &Brush)
```

This function draws text on the screen at a specified angle of rotation. `Center` determines the center pivot point of the rotation and `Angle` defines the angle that the text is facing.

```
virtual void DrawText(PegPoint Where, const
    PEGCHAR *pText, PegBrush &Brush, const
    PegFont *pFont, PEGINT Count = -1)
```

This function draws text on the screen. `Where` determines the upper left starting position for drawing the text. If `Count` is ≥ 0, a maximum of `Count` characters will be drawn unless the string pointed to by `pText` is less than `Count` in length. If `Count` is set to -1, the entire null terminated string is drawn at the indicated position.

This is the only text drawing function provided by the screen driver. Various font formats, including anti-aliased fonts and outlined-fonts, are drawn by the driver by testing the font type flags. The application program does not differentiate between drawing a simple binary font or drawing an anti-aliased font; this happens automatically within the screen driver itself. This enables the application program to change font types and styles simply by referencing new font pointers; no other change is required.

```
PEGUBYTE *DumpWindowsBitmap(PEGULONG *PutSize,
    PegRect &View)
```

This function creates a Microsoft Windows-compatible bitmap file in memory. The bitmap file is an exact capture of the `View` rectangle. Since

PEG assumes no file I/O capabilities, this function returns a pointer of the bitmap residing in memory. It is the caller's job to write the bitmap from memory into permanent storage such as a floppy or hard disk drive for use in a PC environment.

This function is only provided if the #define `PEG_BITMAP_WRITER` is enabled in the configuration file pconfig.hpp when the PEG library is built.

This function is used primarily for testing and/or capturing screen shots on the embedded target for use in promotional literature. This function is NOT normally used for production software since its output format is the Microsoft Windows bitmap format, not the PegBitmap format.

`DumpWindowsBitmap` returns a pointer to the bitmap image in memory. The caller can then save the image using whatever storage is available on the target system. The caller is responsible for freeing the memory associated with the memory bitmap!! The file size (i.e. memory buffer size) is returned in the location pointed to by `PutSize`, or a size of 0 to indicate failure.

```
virtual void Ellipse(const PegRect &Bound,
     PegBrush &Brush)
```

This function draws an ellipse with the provided parameters. The ellipse is bounded on all sides by `Bound`. The ellipse is filled if the `Brush.Style` value includes `PBS_SOLID_FILL`. The ellipse may have a border, drawn in `Brush.LineColor`, of `Brush.Width` pixels. If the ellipse is filled, it is filled with `Brush.FillColor`.

The `Ellipse` function is not required internally by the PEG library, and is therefore only provided if the #define `PEG_ARC_GRAPHICS` is turned on in the header file \peg\include\pconfig.hpp.

```
virtual void EndDraw()
```

This function should be called at the end of every `Draw()` function. This call notifies the PegScreen that drawing has been completed. It is used to maintain the level of draw nesting and to then call the `MemoryToScreen` function when all nesting levels are finished. On systems which utilize multiple palettes, draw nesting is used to determine when a new palette should be loaded into the palette buffer or video controller palette registers.

```
virtual void EndPrint(void)
```

This method informs the screen that all printer operations are complete and all subsequent output will be to the screen. Once this method is called, the

printer context is no longer valid and any reference from application code to the printer object will result in unspecified behavior.

```
DrawSurface *FindSurface(PEGINT SurfaceId)
```

This function searches the list of used surfaces to find one that has the ID indicated by `SurfaceId`. If one is found, a pointer to it is returned. Otherwise it returns NULL.

```
virtual PEGCOLOR GetBitmapPixel(PEGINT x, PEGINT
     y, PegBitmap *pMap)
```

Returns the `PEGCOLOR` associated with the pixel in the bitmap at coordinates x, y. This function is only provided if the #define `PEG_IMAGE_SCALING` is enabled in the configuration file `pconfig.hpp` when the PEG library is built.

```
virtual PEGUBYTE *GetPalette(PEGULONG *pPutSize)
```

Returns a pointer to the current palette, and writes the number of palette entries to `pPutSize`. There will be 3 `PEGUBYTE` values for each palette entry. These values are the Red, Green, and Blue components for each color value.

```
virtual PEGCOLOR GetPixel(PEGINT x, PEGINT y)
```

Returns the `PEGCOLOR` associated with the pixel at screen coordinates x, y.

```
PegBitmap *GetPointer(void)
```

This method returns a PegBitmap pointer to the current mouse pointer bitmap.

```
PegPoint GetPointerPos(void)
```

This method returns a PegPoint with x/y coordinates for the location of the pointer.

```
PEGUBYTE GetPointerType(void)
```

This function returns the current pointer type. The available pointer types are enumerated at the top of the file \peg\include\pscreen.hpp. The types provided with PEG include:

```
PPT_NORMAL
PPT_VSIZE
PPT_HSIZE
PPT_NWSE_SIZ
E
```

```
PPT_NESW_SIZE
PPT_IBEAM
PPT_HAND
```

Additional pointer shapes can also be defined and added using `AddPointerType()`.

```
PegBitmap *GetSurfaceBitmap(PEGINT Surface)
```

This function returns a pointer to the PegBitmap used by the surface with ID `Surface`.

```
PegBitmap *GetSurfaceBitmapAndClose(PEGINT
    Surface)
```

This function returns a pointer to the PegBitmap used by the surface with ID `Surface`. It also then destroys every part of the surface except the bitmap. The caller is then responsible for deleting the bitmap.

```
PEGINT GetXPointerOffset(void)
```

Returns the X offset of the currently used mouse pointer.

```
PEGINT GetXRes(void)
```

Returns the horizontal screen resolution, in pixels.

```
PEGINT GetYPointerOffset(void)
```

Returns the Y offset of the currently used mouse pointer.

```
PEGINT GetYRes(void)
```

Returns the vertical screen resolution, in pixels.

```
virtual PegBitmap *InitHardwareSurface(PEGINT
    Width, PEGINT Height, PEGINT xOffset,
    PEGINT yOffset, PEGINT HardLayer, PEGINT
    Alpha = 255))
```

This function, by default, just returns NULL. It is meant to be overridden in screen drivers that support hardware layers. It should return a pointer to a PegBitmap that is created using the parameters passed into it. This function is not typically called by applications directly; instead, it is called from the `CreateDrawSurface` function when the `HardLayer` parameter is greater than -1.

```
PEGBOOL IsPrinting(void)
```

This function can be used to check if the screen is currently directing output to a printer. This function is only available if `PEG_PRINTER_SUPPORT` has been defined in the library build.

```
void IsPrinting(PEGBOOL Printing)
```

This function sets PEG in printing mode if `Printing` is TRUE. This function is only available if `PEG_PRINTER_SUPPORT` is turned on.

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

This is the basic line drawing function. Optimizations are performed internally for vertical and horizontal lines. The start and end points of the line are inclusive.

```
virtual PegFont *MakeFont(PegFont *pSourceFont,
     PEGUBYTE Height, PEGBOOL Bold = FALSE,
     PEGBOOL Italic = FALSE, PEGINT ForceWidth =
     0)
```

The `MakeFont` function is used to create a new bitmapped font at the indicated point size and style. Normally, you should create fonts using PegFontCapture. `MakeFont` and `DeleteFont` (above) are used to create new fonts at run time. Because of rounding errors and limitations in the vector font format, fonts generated using `MakeFont` are generally not as visually appealing as fonts generated using PegFontCapture. The `MakeFont` function is only available if the definition `PEG_VECTOR_FONTS` is enabled when the PEG library is built. `PEG_VECTOR_FONTS` is defined in the file `\peg\include\peg.hpp`.

```
virtual void PlotPointView(PEGINT x, PEGINT y,
     PEGCOLOR c)
```

This function draws a single color value `c` at the position (`x, y`) on the screen. This is only available if `PLOTPOINTVIEW_AS_FUNCTION` is defined in your library build. Otherwise it is a macro function.

```
virtual void Polygon(PegPoint *pPoints, PEGINT
     NumPoints, PegBrush &Brush)
```

This function draws a polygon on the screen. The polygon may be concave or convex, filled or bordered. PEG objects do not use the polygon function, and it is therefore optional in derived classes.

```
virtual void PolyLine(PegPoint *pPoints, PEGUINT
    NumPoints, PegBrush &Brush)
```

This function draws a series of connected lines defined by the array of points, `pPoints`.

```
virtual void PutBitmapPixel(PEGINT xPos, PEGINT
    yPos, PegBitmap *pMap, PEGCOLOR Color)
```

This function plots a single point at the indicated x, y position in the indicated color, in the bitmap `pMap`. This function is only provided if the #define `PEG_IMAGE_SCALING` is enabled in the configuration file `pconfig.hpp` when the PEG library is built.

```
virtual void PutPixel(PEGINT xPos, PEGINT yPos,
    PEGCOLOR Color)
```

This function plots a single point at the indicated x, y position in the indicated color.

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

This function draws a rectangle on the screen. The rectangle may have any border width and may optionally be filled if the Brush.Style parameter is set to `PBS_SOLID_FILL`. The rectangle border is drawn with Brush.LineColor and, if the rectangle is filled, it is filled with Brush.FillColor.

```
void RectangleXOR(const PegRect &InRect)
```

This function inverts the pixel color values of the indicated rectangle.

```
virtual PEGBOOL RectMove(PegRect Get, PegRect
    ClipTo, PEGINT xShift, PEGINT yShift)
```

When `FAST_BLIT` is enabled, PEG objects use this function to rapidly scroll. For high-end video controllers with hardware accelerated bitblit capability, this function directly invokes the hardware pixel move operation. This function is software emulated when running with a low-end video controller.

```
virtual void ResetPalette(void)
```

This function resets the current color palette back to the default state.

```
virtual PegBitmap *ResizeImage(PegBitmap *pSrc,
    PEGINT Width, PEGINT Height)
```

This function creates a new PegBitmap from an existing PegBitmap. The new bitmap is resized to `Width` x `Height` pixels regardless of the size of the source bitmap. A PegBitmap created in this manner should be destroyed by calling the DestroyBitmap() function.

```
virtual void Restore(PegCapture *pInfo, PEGBOOL
    OnTop = FALSE)
```

This function restores a captured area of screen pixels.

```
virtual void  SetPalette(PEGINT First, PEGINT
    Num, const PEGUBYTE *pPal)
```

This function installs the palette indicated by `pPal`. The number of entries in the palette is indicated by `Num`, and the first entry to be modified is indicated by `First`. To install a custom 256 color palette, for example, you would call `SetPalette` like this:

```
SetPalette(0, 256, &PalData);
```

PEG color palettes are simple 8-8-8 RGB color values. There should be (3* `Num`) `PEGUBYTE` values in the array pointed to by `pPal`.

```
virtual void SetPointer(PegPoint Where)
```

This function sets the current position of the mouse pointer on the screen.

```
virtual void SetPointerType(PEGUBYTE Type)
```

This function is used to change the active pointer type. For example, PegWindow objects change the pointer type during resize operations as an indication to the user that the window border is being dragged. The available pointer types are defined in the header file pscreen.hpp. They include:

```
PPT_NORMAL
PPT_VSIZE
PPT_HSIZE
PPT_NWSE_SIZ
E
PPT_NESW_SIZ
E PPT_IBEAM
PPT_HAND
```

Most often an application will capture the mouse pointer before changing the pointer type, and release the pointer after restoring the pointer to normal.

```
void SetSurfaceAlpha(PEGINT Surface, PEGINT
     Alpha)
```

This function sets the alpha blend ratio for the surface with ID `Surface` to `Alpha`.

```
void SetSurfaceOffset(PEGINT Surface, PEGINT
     xDrawOffset, PEGINT yDrawOffset)
```

This function modifies the x and y offsets used by the surface with ID `Surface`.

```
void SetSurfaceSize(PEGINT Surface, PEGINT
     xSize, PEGINT ySize, PEGINT xOffset, PEGINT
     yOffset)
```

This function is used to modify an existing surface's size and offset values.

```
virtual void ShowPointer(PEGBOOL Show)
```

This function is used to show and hide the pointer bitmap. Passing FALSE to this function disables display of the pointer bitmap, and passing TRUE enables display of the pointer bitmap. The pointer bitmap is displayed by default after power up if `PEG_MOUSE_SUPPORT` is defined. Note that the pointer still operates normally, including tracking pointer position and sending click messages, even if the bitmap is not displayed.

```
void SurfaceToFront(PEGINT Surface)
```

This function adjusts the z-ordering of the visible surfaces by bringing the surface with ID `Surface` to the front.

```
virtual PEGINT TextHeight(const PegFont *pFont)

virtual PEGINT TextHeight(PEGINT FontId)
```

This function returns the height, in pixels, of the indicated font. All characters in a font are guaranteed to be the same height so there is no need to pass a text string into this function. A PegFont pointer or a font ID can be used.

```
virtual PEGINT TextWidth(const PEGCHAR *pText,
     const PegFont *pFont, PEGINT Len = -1)

virtual PEGINT TextWidth(const PEGCHAR *pText,
     PEGINT FontId, PEGINT Len = -1)
```

This function returns the width, in pixels, of the indicated string up to a maximum of `Len` characters using the indicated font. If `Len` is -1, the width of the entire null terminated string is returned. A PegFont pointer or a font ID can be used.

1.4.6 Protected Members:

```
virtual void AALine(PegThing *pCaller, PEGINT
     XStart, PEGINT YStart, PEGINT XEnd, PEGINT
     YEnd, PegBrush &Brush
```

Draws an anti-aliased (dithered) line between the start and end coordinates. This is only supported for 16 or 24 bit color depths, and only if the configuration flag `PEG_AA_LINE` is enabled in the `pconfig.hpp` configuration file.

```
virtual void AALineView(PEGINT xStart, PEGINT
     yStart, PEGINT xEnd, PEGINT yEnd, PegRect
     &View, PegBrush &Brush)
```

This function draws an anti-aliased line, clipped to the `View` rectangle, between the start and end coordinates. Applications typically do not call this directly, but call `AALine` instead. This is a pure virtual function, so it must be implemented in the screen driver. This function is only available if `PEG_AA_LINE` is turned on.

```
virtual void ArcFill(PEGINT xc, PEGINT yc,
     PEGINT XRadius, PEGINT YRadius, PEGINT
     start_angle, PEGINT end_angle, PegBrush
     &Brush, PegRect &View)
```

This function draws a filled arc shape using the specified center, x-radius and y-radius, and start and end angles. The arc is clipped to the `View` rectangle. Applications typically do no call this directly, but call `Arc` instead. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual void ArcLine(PEGINT xs, PEGINT ys,
     PEGINT xe, PEGINT ye, PegRect &View)
```

This is a line function used by the arc drawing algorithm. Applications typically do not call this directly, but call `Arc` instead. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual void ArcToView(PEGINT xc, PEGINT yc,
     PEGINT radius, PEGINT start_angle, PEGINT
     end_angle, PegBrush &Brush, PegRect &View)
```

This function clips an arc to the specified viewport. Applications typically do not call this directly, but call `Arc` instead. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual void ArcToViewSpecial(PEGINT xCen,
     PEGINT yCen, PEGINT XRadius, PEGINT
     YRadius, PEGINT StartAngle, PEGINT
     EndAngle, PegBrush &Brush, PegRect &View)
```

This function is very similar to `ArcToView`, except that it handles the special case when the start and end angles are in the same quadrant. Applications typically do not call this directly, but call `Arc` instead. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual void BitmapView(const PegPoint Where,
     const PegBitmap *pMap, const PegRect &View)
```

This function draws the bitmap `pMap` at the point `Where`, clipped to the `View` viewport rectangle. Applications typically do not call this directly, but call `Bitmap` instead. This is a pure virtual function so it must be implemented in the screen driver.

```
virtual void BitmapView(PegPoint Where, PegBitmap
     *pMap, const PegRect &View, PegBitmap
     *pAlphaMap, PEGSHORT AlphaX, PEGSHORT
     AlphaY)
```

This Peg Pro only function alpha blends a bitmap on the screen using an alpha value map. The parameter `Where` defines the upper left corner position of the bitmap on the screen and the drawing is clipped to the `View` viewport rectangle. The parameter `pMap` points to a bitmap generated with the PegImageConvert utility. The `pAlphaMap` pointer is the alpha value map. The parameters `AlphaX` and `AlphaY` define the starting offset within the alpha map.

```
virtual void BoxLine(PEGINT xs, PEGINT ys,
     PEGINT xe, PEGINT ye, PegBrush &Brush)
```

This function is used to draw a wide diagonal line between the start and end points. Applications typically do not call this directly, but call `Line` instead.

```
void BoxLineView(const PegRect &LimitRect,
     PegFixedPoint *pPoints, PegBrush &Brush)
```

This function is used to draw a wide diagonal line using the corner points in pPoints, and clipped to `LimitRect`. Applications typically do not call this directly, but call `Line` instead.

```
void CalcEllipsePoints(PEGINT Angle, PEGINT asq,
     PEGINT bsq, PEGINT &X, PEGINT &Y)
```

This function finds a specific x, y point on an ellipse at the specified angle. Applications typically do not call this directly, but call `Arc` instead. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual void ChordFill(PEGINT xCen, PEGINT yCen,
     PEGINT XRadius, PEGINT YRadius, PEGINT
     StartAngle, PEGINT EndAngle, PegBrush
     &Brush, PegRect &View)
```

This function draws the filled portion of a chord shape on the screen. The chord is defined by the center point, radii, and angles, and then clipped to the `View` rectangle. Applications typically do not call this, but call `Chord` instead. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual void ChordFillSpecial(PEGINT xCen,
     PEGINT yCen, PEGINT XRadius, PEGINT
     YRadius, PEGINT StartAngle, PEGINT
     EndAngle, PegBrush &Brush, PegRect &View)
```

This function is very similar to `ChordFill`, except that it handles the special case where the start and end angles are in the same quadrant. Applications typically do not call this, but call `Chord` instead. This function is only available if `PEG_ARC_GRAPHICS` is turned on.

```
virtual void Circle(const PegRect &LimitRect,
     PEGINT xCenter, PEGINT yCenter, PEGINT
     Radius, PegBrush &Brush)
```

This function draws a circle defined by the center coordinates and the `Radius`. It is used in the standard circle drawing algorithm. Applications typically do not call this function directly, but call the public `Circle` function instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual void CircleFast(PEGINT xCenter, PEGINT
     yCenter, PEGINT Radius, PegBrush &Brush)
```

This function draws a circle defined by the center coordinates and the `Radius`. It is used in the standard circle drawing algorithm. Applications typically do not call this function directly, but call the public `Circle` function instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual void DrawAliasMapView(PegPoint Where,
     const PegBitmap *pMap, PegBrush &Brush,
     const PegRect &View)
```

This function uses the values in the bitmap `pMap` as alpha ratios to blend the color `Brush.LineColor` with whatever is on the background of the current surface. The drawing is clipped to the `View` rectangle.

```
virtual void DrawTextView(PegPoint Put, const
     PEGCHAR *pText, PegBrush &Brush, const
     PegFont *pFont, PEGINT Len, PegRect &View)
```

This function draws the specified text `pText` at the point `Put`, using the font `pFont`, clipped to the rectangle `View`. Applications typically do not call this function directly, but call DrawText instead.

```
virtual void EllipseFast(const PegRect &Bound,
     PegBrush &Brush)
```

This function draws an ellipse defined by the rectangle `Bound`. It is used in the standard ellipse drawing algorithm. Applications typically do not call this function directly, but call `Ellipse` instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual void EllipseToView(const PegRect &Bound,
     PegBrush &Brush, PegRect View)
```

This function draws an ellipse defined by the rectangle `Bound`, and clipped to the rectangle `View`. It is used in the standard ellipse drawing algorithm. Applications typically do not call this function directly, but call `Ellipse` instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
void FastWideHLine(PEGINT xStart, PEGINT xEnd,
     PEGINT yVal, PEGINT Width, PEGCOLOR
     LineColor)
```

This function draws a solid horizontal line, clipped. Applications typically do not call this function directly, but call `Line` instead.

```
void FastWideVLine(PEGINT yStart, PEGINT yEnd,
     PEGINT xVal, PEGINT Width, PEGCOLOR
     LineColor)
```

This function draws a solid vertical line, clipped. Applications typically do not call this function directly, but call `Line` instead.

```
virtual void FillCircle(const PegRect
    &LimitRect, PEGINT xCenter, PEGINT yCenter,
    PEGINT r, PEGCOLOR FillColor)
```

This function draws a filled circle using the specified center coordinates and radius; clipping will be used. It is used in the standard circle drawing algorithm. Applications typically do not call this function directly, but call `Circle` instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual void FillCircleFast(PEGINT xCenter,
    PEGINT yCenter, PEGINT r, PEGCOLOR
    FillColor);
```

This function draws a filled circle using the specified center coordinates and radius; clipping will not be used. It is used in the standard circle drawing algorithm. Applications typically do not call this function directly, but call `Circle` instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual void FillPolygon(PegRect &Bound,
    PegPoint *pPoints, PEGINT NumPoints,
    PEGCOLOR Color)
```

This function draws a filled polygon using the specified points. It is used in the standard polygon drawing algorithm. Applications typically do not call this function directly, but call `Polygon` instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual void FillPolygonView(const PegRect
    &View, PegPoint *pPoints,PEGINT NumPoints,
    PEGCOLOR Color)
```

This function draws a filled polygon, clipped to the `View` rectangle, using the specified points. It is used in the standard polygon drawing algorithm. Applications typically do not call this function directly, but call `Polygon` instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual PEGCOLOR GetPixelView(PEGINT x, PEGINT
    y)
```

Returns the `PEGCOLOR` associated with the pixel at screen coordinates `x, y`. Applications typically do not call this function directly, but call `GetPixel` instead.

```
virtual void HidePointer(void)
```

This function is called to hide the mouse pointer.

```
virtual void HorizontalLine(PEGINT xStart,
     PEGINT xEnd, PEGINT y, PEGCOLOR Color,
     PEGINT Width)
```

This function draws a horizontal line between `xStart` and `xEnd` at the given y coordinate. Applications typically do not call this function directly, but call `Line` instead.

```
virtual void HorizontalLineXOR(PEGINT xs, PEGINT
     xe, PEGINT y)
```

This function draws an inverted horizontal line between `xs` and `xe` at the given y coordinate. This is mostly used for drawing the outline of a window when it gets resized or moved.

```
virtual void InitVidMemManager(PEGUBYTE *pStart,
     PEGUBYTE *pEnd)
```

This function is used to initialize the built-in video memory manager. This is only available if `PEG_VID_MEM_MANAGER` is turned on in `pconfig.hpp`

```
virtual void LineView(PEGINT xStart, PEGINT
     yStart, PEGINT xEnd, PEGINT yEnd,  PegRect
     &View, PegBrush &Brush)
```

This function draws a line between the points (`xStart`, `yStart`) and (`xEnd`, `yEnd`), and clipped to the `View` rectangle. Applications typically do not call this function directly, but call `Line` instead.

```
virtual void MemoryToScreen(void)
```

This function is used to transfer the local frame buffer data to the actual video memory. This is a pure virtual function, so the actual implementation is located in the screen driver.

```
PEGULONG mNumColors
```

Number of output colors. This is the number of colors supported by the output device, rather than the number of colors in the current palette.

```
PEGINT mCurXOffset
```

Offset in x axis between upper-left corner of mouse pointer bitmap and pointer hotspot.

```
PEGINT mCurYOffset
```

Offset in y axis between upper-left corner of mouse pointer bitmap and pointer hotspot.

```
PegBitmap *mpCurPointer
```

Address of current mouse pointer bitmap

```
PegPointer mpPointers[NUM_POINTER_TYPES]
```

Array of mouse pointer bitmap addresses. This array must be extended if new pointer types are defined.

```
PEGUINT mDrawNesting
```

Nesting level within `Draw` functions

```
PEGINT mHRes
```

The horizontal screen resolution in pixels.

```
PEGINT mVRes
```

The vertical screen resolution in pixels.

```
virtual void OutlineCircle(const PegRect
     &LimitRect, PEGINT xCenter, PEGINT yCenter,
     PEGINT r, PEGCOLOR Color, PEGINT Width)
```

This function draws the outline of a circle with the given center point and radius. Applciations typically do not call this directly, but call `Circle` instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual void OutlineCircleFast(PEGINT xCenter,
     PEGINT yCenter, PEGINT r, PEGCOLOR Color,
     PEGINT Width)
```

This function draws a filled circle using the specified coordinates and radius. It is used in the standard circle drawing algorithm. Applications typically do not call this function directly, but call `Circle` instead. This function is only available if `PEG_FULL_GRAPHICS` is turned on.

```
virtual void PatternFillPolygon(PegRect &Bound,
     PegPoint *pPoints, PEGINT NumPoints,
     PegBrush &Brush)
```

This function draws a filled polygon using the specified points. It is used in the standard polygon drawing algorithm. Applications typically do not call

this function directly, but call Polygon instead. This function is only available if PEG_FULL_GRAPHICS is turned on.

```
virtual void PatternFillPolygonView(const
     PegRect &View, PegPoint *pPoints, PEGINT
     NumPoints, PegBrush &Brush)
```

This function uses a bitmap to fill a polygon defined by the specified points and clipped to the View rectangle. It is used in the standard polygon drawing algorithm. Applications typically do not call this function directly, but call Polygon instead. This function is only available if PEG_FULL_GRAPHICS is turned on.

```
virtual void PatternLine(PEGINT XStart, PEGINT
     YStart, PEGINT XEnd, PEGINT YEnd, PegBrush
     &Brush)
```

This function draws a dashed line. The dash pattern is determined by the Brush.Pattern value. For example, a pattern value of 0xf0f0f0f0 would draw a line alternating between 4 pixels on and 4 pixels off. Applications typically do not call this function directly, but call Line instead. This function is only provided if the definition PEG_FULL_GRAPHICS is turned on in the file pconfig.hpp.

```
virtual void PatternLineView(PEGINT XStart,
     PEGINT YStart, PEGINT XEnd, PEGINT YEnd,
     PegRect &Rect, PegBrush &Brush)
```

This function draws a dashed line, clipped to Rect. The dash pattern is determined by the Brush.Pattern value. For example, a pattern value of 0xf0f0f0f0 would draw a line alternating between 4 pixels on and 4 pixels off. Applications typically do not call this function directly, but call Line instead. This function is only provided if the definition PEG_FULL_GRAPHICS is turned on in the file pconfig.hpp.

```
virtual void PlotEllipsePoints(PegRect &Bound)
```

This function is used by the ellipse drawing algorithm to calculate the ellipse points bounded by the Bound rectangle. Applications typically do not call this function directly, but call Ellipse instead. This function is only available if PEG_FULL_GRAPHICS is turned on in pconfig.hpp.

```
void PrepareBrush(PegBrush &Brush)
```

This function is used to create the bitmap used for endpoints when drawing lines. The bitmap could be a solid circle, or an anti-aliased circle.

```
virtual void RectangleView(const PegRect
    &InRect, PegRect &View, PegBrush &Brush)
```

This function draws a rectangle defined by `InRect` and clipped to the `View` rectangle. Applications typically do not call this function directly, but call the public version of `Rectangle` instead.

```
virtual void RectangleXORView(const PegRect
    &View)
```

This function inverts the pixel color values of the indicated rectangle. Applications typically do not call this function directly, but call `RectangleXOR` instead.

```
virtual void RectMoveView(const PegRect &View,
    const PEGINT xMove, const PEGINT yMove)
```

When `FAST_BLIT` is enabled, PEG objects use this function to rapidly scroll. For high-end video controllers with hardware accelerated bitblit capability, this function directly invokes the hardware pixel move operation. This function is software-emulated when running with a low-end video controller. Applications typically do not call this function directly, but call `RectMove` instead.

```
void SetSurfaceOffset(DrawSurface *pSurface,
    PEGINT xOffset, yOffset)
```

This function adjusts the x and y offsets of `pSurface`. This is used, for example, when a surface is moving across the screen. The offsets get modified so that the caller does not need to modify the coordinates that it uses to draw into the surface.

```
PEGINT SqRoot(PEGINT x)
```

This function returns the integer square root of `x`. This is used for the ellipse calculations. This function uses a simple lookup table, which makes it quite fast. It does not use any floating point numbers.

```
virtual void VerticalLine(PEGINT yStart, PEGINT
    yEnd, PEGINT x, PEGCOLOR Color, PEGINT
    Width)
```

This function draws a vertical line between `yStart` and `yEnd` at the specified x coordinate. Applications typically do not call this directly, but call `Line` instead.

```
virtual void VerticalLineXOR(PEGINT ys, PEGINT
    ye, PEGINT x)
```

This function draws an inverted vertical line between `ys` and `ye` at the specified x coordinate. This is mostly used for drawing the outline of a window when it gets resized or moved.

1.4.7 Examples:

Draw() Function Example

The following example installs a custom palette. This palette would normally be generated by PegImageConvert, although a palette can be created using several means.

```
PEGINT MyWindow::Message(const PegMessage &Mesg)
{
    switch(Mesg.Type)
    {
    case PM_SHOW:
        Screen()->SetupPalette(PegCustomPalette, 256);
        PegWindow::Message(Mesg);
        break;

    case PM_HIDE:
        Screen()->ResetPalette();
        PegWindow::Message(Mesg);
        break;

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

1.5 PegTextThing

1.5.1 Overview

PegTextThing serves as a base class for all PEG objects that display or manipulate text. This provides a common set of API functions for all PEG classes that display text, such as PegTitle, PegTextButton, PegPrompt, PegEditField, and others.

PegTextThing provides string storage and manipulation functionality for all PegTextThing derived classes. This insulates the PEG classes from the character encoding method, which enables PEG to support both 8-bit ASCII and 16-bit UNICODE character encoding methods without dramatic changes to any of the derived classes.

It is important to understand that, by default, PegTextThing does NOT copy text strings when a string assignment is made. PegTextThing normally copies only the pointer to the text string. If the `TT_COPY` style flag is associated with the PegTextThing derived class, PegTextThing copies the actual text string when an assignment is made.

For this reason, if you dynamically create a string that will be associated with a PegTextThing derived class, you should use the `TT_COPY` style when the class is constructed. For example, if you build up a string in an automatic character array using a function like `itoa` or `sprintf`, the storage for that character string is temporary storage, usually on the stack. After the function returns, the storage is no longer valid. If you are using a PegTextThing class in this way, the `TT_COPY` should be used.

If the string associated with an object is static, which is most often the case, the `TT_COPY` flag should not be used. For example, when a PegTitle is created like this:

```
PegTitle *pTitle = new PegTitle("Hello World");
```

the string is a string literal. The compile/linker will allocate storage space for this string and the storage space will never be deleted. In this case, it is not necessary for PegTextThing to copy the actual string data.

1.5.2 Derivation

PegTextThing is derived from PegThing. It is important to remember that all public PegThing member functions are available to objects derived from PegTextThing.

1.5.3 Style Flags

PegTextThing supports only the `TT_COPY` flag. Additional style flags are passed to the base PegThing class.

1.5.4 Constructors:

```
PegTextThing(const PegRect &Rect, PEGUINT
     StringId = 0, PEGUSHORT Id = 0, PEGULONG
     Style = FF_NONE, PEGUINT FontIndex = 0)

PegTextThing(const PEGCHAR *pText, const PegRect
     &Rect, PEGUSHORT Id = 0, PEGULONG Style =
     TT_COPY|FF_NONE, PEGUINT FontIndex = 0)

PegTextThing(PEGUINT StringId, PEGUSHORT Id,
     PEGULONG Style = FF_NONE, PEGUINT FontIndex
     = 0)

PegTextThing(const PEGCHAR *pText, PEGUSHORT Id,
     PEGULONG Style = TT_COPY|FF_NONE, PEGUINT
     FontIndex = 0)

PegTextThing(PEGULONG Style = TT_COPY|FF_NONE)
```

1.5.5 Public Functions:

```
void DataClear(void)
```

This inline function sets the object's text to NULL.

```
const PEGCHAR *DataGet(void)
```

This inline function returns a pointer to the text string associated with an object.

```
virtual void DataSet(const PEGCHAR *pText)
```

This function is called to assign the string associated with any object derived from PegTextThing.

```
virtual void DataSet(PEGUINT StringId)
```

This function is called to assign the string associated with any object derived from PegTextThing. This version uses the string table to look up the string ID and find the actual text.

```
const PegFont *GetFont(void)
```

This inline function returns the font associated with a PegTextThing-derived object.

```
PEGUINT GetStringId(void)
```

Returns the current string ID

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

PegTextThing overrides the `PegThing::Message` function in order to handle `PM_LANGUAGE_CHANGE` events.

```
void SetCopyMode(void)
```

This method allows copy mode to be set to true after the object has been constructed. Once set to true, it cannot be set back to false. This forces the object to behave as if you had passed the `TT_COPY` flag in the constructor in that it makes a copy of the text and stores it internally.

```
virtual void SetFont(PEGUINT FontIndex)
```

This function assigns the font associated with any PegTextThing-derived object.

```
PEGINT TextLength(void)
```

This inline function returns the number of characters in the string currently associated with any PegTextThing-derived object.

```
void TextThingInit(PEGULONG Style, PEGUINT
     StringId, const PEGCHAR *pText, PEGUINT
     FontIndex)
```

Common initialization code called by all constructors.

```
PEGINT GetXShadow(void)
```
This function returns the X-axis offset of a text shadow.

```
PEGINT GetYShadow(void)
```
This function returns the Y-axis offset of a text shadow.

```
PEGCOLOR GetShadowColor(void)
```
This function returns the color value of a text shadow.

```
PEGCOLOR GetShadowColorId(void)
```
This function returns the color Id of a text shadow.

```
PEGINT GetShadowBlur(void)
```
This function returns the blur value of a text shadow.

```
PEGSHORT GetShadowOpacity(void)
```
This function returns the opacity level of a text shadow.

```
PEGUSHORT GetShadowScale(void)
```
This function returns the scale of a text shadow.

```
PegShadowMode GetShadowMode(void)
```
This function returns the shadow mode of a text shadow.

```
PEGUSHORT GetTextOpacity(void)
```
This function returns the opacity of the text drawn with a text shadow.

```
void SetXShadow(PEGINT XShadow)
```
This function sets the X-axis offset of a text shadow.

```
void SetYShadow(PEGINT YShadow)
```
This function sets the Y-axis offset of a text shadow.

```
void SetShadowColor(PEGCOLOR Color)
```
This function sets the color a text shadow to the PEGCOLOR value `Color`.

```
void SetShadowColorId(PEGUINT ColorId)
```
This function sets the color of a text shadow to `ColorId`.

```
void SetShadowBlur(PEGINT Blur)
```
This function sets the amount of blur of a text shadow.

```
void SetShadowOpacity(PEGSHORT Opacity)
```
This function sets the opacity of a text shadow.

```
void SetShadowScale(PEGUSHORT Scale)
```
This function sets the amount the text shadow is scaled.

```
void SetShadowMode(PegShadowMode Mode)
```
This function sets the shadow mode of a text shadow.

```
void SetTextOpacity(PEGUSHORT Opacity)
```
This function sets the opacity of the text drawn with a text shadow.

```
virtual PegBitmap *CreateDropShadowText(PegPoint
Where, const PEGCHAR *pText, PegBrush &Brush,
const PegFont *pFont, PegShadowEffect
*pShadowEffect, PEGINT Count)
```
The `CreateDropShadowText` function returns a PegBitmap of the text string `pText` drawn with a drop shadow defined by `pShadowEffect`. Like the DrawText function of PegScreen, `Count` is the number of drawn characters if `Count` > 0 otherwise the function draws the entire string.

```
virtual PegBitmap *CreateDropShadowText(PegPoint
Where, const PEGCHAR *pText, PegBrush &Brush,
PEGUINT FontId, PegShadowEffect *pShadowEffect,
PEGINT Count = -1)
```
Similar to the above `CreateDropShadowText` function except uses the font ID FontId instead of a PegFont pointer.

1.5.6 Protected Members:

```
const PegFont *mpFont
```

Pointer to the PegFont associated with the object.

```
PEGCHAR *mpText
```

Pointer to the string associated with the object.

```
PEGUINT mFontIndex
```

ID of the PegFont associated with the object.

```
PEGINT mStrLen
```

Number of characters in the current assigned string.

```
PEGBOOL mCopy
```

TRUE/FALSE value indicating if the string copy mode is in effect.

1.5.7 Examples:

The following function creates a PegTextButton, which is a PegTextThing-derived class, and assigns a custom font to the button. The font has the ID labeled 'CUSTOM_BTN_FONT.' The button is then added to the parent window.

```
void MyWindow::AddCustomButton(const PEGCHAR *pButtonText)
{
    PegRect Rect;
    Rect.Set(10, 10, 89, 59);
    PegTextButton *pButton = new PegTextButton(pButtonText,
        Rect);
    pButton->SetFont(CUSTOM_BTN_FONT);
    Add(pButton);
}
```

The following function obtains the string associated with a prompt. The string is converted to an integer, a range check is made, and the modified value is then reassigned to the prompt. Note that in this case, the prompt should be created with the `TT_COPY` flag enabled.

```
void MyWindow::CheckPromptVal(PegPrompt *pPrompt, PEGINT Min,
```

```
PEGINT Max)
{
    const PEGCHAR *pString = pPrompt->DataGet();
    PEGBOOL Replace = FALSE;

    if (pString)
    {
        PEGINT Val = PegAtoI(pString);

        if (Val < Min)
        {
            Val = Min;
            Replace = TRUE;
        }
        if (Val > Max)
        {
            Val = Max;
            Replace = TRUE;
        }
    }
    else
    {
        Val = Min;
        Replace = TRUE;
    }

    if (Replace) // prompt is out of range?
    {
        PEGCHAR Temp[40];
        PegLtoA(Val, Temp, 10);
        pPrompt->DataSet(Temp);   // re-assign string
   }
}
```

1.6 PegThing

1.6.1 Overview

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

Understanding and remembering the functions of class PegThing is vital to using the PEG library. We encourage you to read the Programming Manual chapter on class PegThing, which contains much more information about the design and use of the PegThing class.

In addition to the true member functions, PegThing implements several inline wrapper functions. These functions provide a simplified syntax for calling member functions of the PegScreen and PegMessageQueue classes. For example, you can always draw a line by calling the PegScreen Line function directly:

```
Screen()->Line(.....)
```

However, for the most common operations PegThing provides an inline wrapper function that eliminates the need to obtain the PegScreen instance pointer. This improves the API syntax and eliminates typing effort. The above function can be invoked more easily by using the wrapper function:

```
Line(...)
```

The public member functions and wrapper functions are listed in the Members section of this reference.

PegThing objects are NOT viewport objects. If you are constructing a large container class, you may want to derive that class from PegWindow, rather than from class PegThing. For small custom gadgets, PegThing works well as a foundation upon which to build your custom class.

1.6.2 See Also

Viewports

1.6.3 Derivation

PegThing is a PEG base class.

1.6.4 Member Functions

Public Functions

Add()
AddToEnd()
AddStatus()
Center()
Constructors
CenterOf()
CheckDirectionalMove()
CheckSignal()
DefaultKeyHandler()
Destroy()
Distance()
Draw()
DrawChildren()
Find()
FindNearest()
First()
FrameStyle()
GetId()
GetColor()
GetSignals()
GetStatus()
GetStyle()
Hide()
InitClient()
Invalidate()
IsDescendentOf()
KillFocus()
Message()
MessageChildren()
MessageQueue()
MoveToFront()
Next()
NextTabLink()
Parent()
ParentShift()
Presentation()
Previous()
PrevTabLink()
Printer()
Remove()
RemoveStatus()
RemoveStyle()
Resize()
Screen()
SendSignal()
SetColor()
SetDefaultTabLinks()
SetId()
SetMessageQueuePointer()
SetPrintPtr()
SetScreenPtr()
SetSignals()
SetStyle()
SetTabLink()
SetTabOrder()
SetTimer()
SetTimeManager()
StandardBorder()
StatusIs()
Type()
UpdateChildClipping()
Version()

WrapperFunctions

BeginDraw()	KillTimer()
Bitmap()	Line()
Rectangle()	RectMove()
BitmapFill()	ReleasePointer()
Circle()	SetPointerType()
CapturePointer()	SetTimer()
DrawText()	TextHeight()
EndDraw()	TextWidth()
Invalidate()	

Public Data

mReal mClip mClient

1.6.5 Constructors:

```
PegThing(const PegRect &Rect, PEGUSHORT Id = 0,
     PEGULONG Style = FF_NONE)
```

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

```
PegThing(PEGUSHORT Id = 0, PEGULONG Style =
     FF_NONE)
```

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

The easiest way to set an object's position is to call the member function `Resize()`, which accepts a PegRect argument which should contain the

desired screen coordinates. Calling `Resize()` is the only acceptable way to set an object's size or position after the object is visible.

A more direct method of setting an object's position and size is to directly modify the object's `mReal` (the absolute bounding rectangle of an object) and `mClient` (the inside client area of an object) variables. This method must be used with caution since PEG base classes often must insure that `mClient` remains correctly positioned relative to `mReal`. Also, you should never directly modify `mReal` or `mClient` after an object is visible, since PEG clipping enforcement will likely prevent the desired result.

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

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

An object is normally drawn after being added to a visible parent. This operation can be prevented, if desired, by passing a FALSE value as the `DoShow` parameter to the Add function.

If the object `pWho` is already a member of the current object's child list, `pWho` is not added again to the list. Instead, `pWho` is simply unlinked from the child list and re-linked at the head of the child list. This action changes the order of child objects, which may be the desired operation.

Objects are added to the parent according to the status of the object, meaning that objects with `PSF_VIEWPORT` status or `PSF_ALWAYS_ON_TOP` status are always maintained ahead of child objects which do not have this status. These differences are maintained internally by PEG and are necessary to ensure proper drawing; however, they are not normally the concern of the application-level program.

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

When constructing complex windows and dialogs, it is best to first add all of the child objects to the main window or dialog, and then to add the main

window or dialog to PegPresentationManager. This is slightly more efficient than adding each child object to a window or dialog that is already visible.

```
virtual void AddStatus(PEGULONG OrVal)
```

This function can be used to modify an object's `mStatus` flags. `AddStatus` will logically OR the `OrVal` parameter with the object's `mStatus` variable. This function is often used by the PEG foundation objects to modify the state of a visible window or control, but it is rarely used by the application-level software. The system status flag list and definitions are found here.

```
virtual void AddStyle(PEGULONG OrVal)
```

This function can be used to modify an object's `mStyle` flags. `AddStyle` will logically OR the `OrVal` parameter with the object's `mStyle` variable.

```
virtual void AddToEnd(PegThing *pWho, PEGBOOL
     DoShow = TRUE)
```

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

The `AddToEnd` function works very much like the `Add()` function, except that the added object is added to the end of the linked list of child objects rather than being added to the head of the linked list. This is sometimes useful when adding objects to PegList containers in order to correct the child display order.

Like the `Add()` function, `AddToEnd()` operates within the constraints of object status, meaning that objects with `PSF_VIEWPORT` status or `PSF_ALWAYS_ON_TOP` status are always maintained ahead of child objects which do not have this status.

```
void CapturePointer(void)
```

This function acts as a wrapper function, allowing access by a PegThing to the PegPresentationManager member function of the same name. Use of this function is equivalent to `Presentation()->CapturePointer(this)`.

```
virtual void Center(PegThing *pWho)
```

This function will adjust the screen coordinates of `pWho` such that `pWho` is horizontally and vertically centered over the client area of `this`. `pWho` does not necessarily have to be a child of `this`, although this is the most

common case. The following example demonstrates centering an object on the screen:

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

Presentation()->Add(MyWin);    // make the window visible
```

```
virtual PegPoint CenterOf(PegThing *pWho)
```

This function returns the coordinates of the center of the object in a PegPoint structure.

```
virtual PEGBOOL CheckDirectionalMove(PEGINT Key)
```

Default arrow key handling. Returns TRUE if the key was processed, otherwise FALSE. This function is only provided if both #define `PEG_KEYBOARD_SUPPORT` and #define `PEG_ARROW_KEY_SUPPORT` are enabled in the configuration file pconfig.hpp.

```
PEGBOOL CheckSendSignal(PEGUBYTE Signal)
```

This function creates and sends a PegMessage with the appropriate `PEG_SIGNAL` value loaded in `Type`, object ID loaded in `Param`, `this` loaded in `pSource`, and `Parent()` loaded into the `pTarget` field of the message. The boolean value TRUE is returned after the signal is sent. FALSE is returned if the signal was not sent: either because the current object has no parent, the object ID is 0, or the specified signal bit is currently disabled.

```
virtual PEGINT DefaultKeyHandler(const
     PegMessage &InMesg)
```

This function is called by default when `PM_KEY` messages are received. It is only provided when `PEG_KEYBOARD_SUPPORT` is defined.

This function checks for keys that cause an input focus change, such as TAB and ARROW keys. If the key is not one of these keys, the function then checks to see if the object signals require that a `PSF_KEY_RECEIVED` signal be sent to the object parent. Finally, if none of these operations are performed, the key is passed up to the parent of the current object.

This function may be overridden in derived classes to perform custom key handling, although it is more common to simply catch `PM_KEY` messages in derived classes.

```
void Destroy(PegThing *pWho)
```

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

```
PEGLONG Distance(PegPoint p1, PegPoint p2)
```

This function calculates the square of the distance between two PegPoints.

```
virtual void Draw(const PegRect &Invalid)
```

This function is called by PegPresentationManager when an object has been previously invalidated. It can also be called by the application software when an object has been modified, though this should be rare. An example of overriding the `Draw()` function is provided in the PEG programming manual.

```
virtual void DrawChildren(const PegRect
    &Invalid)
```

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

An example of overriding the `Draw()` function is provided in Overriding the Draw() function.

```
virtual PegThing *Find(PEGUSHORT Id, PEGBOOL
    Recursive = TRUE)
```

This function can be used to find any object based on the object ID value. For example, you may create a PegDialog window that has many child controls. If you need to modify the status of those controls as the dialog is manipulated, you will need to keep or obtain pointers to those child

controls. There are two ways of obtaining a pointer to each child control. You could add member pointers to the dialog window that are initialized as each child control is constructed. This is faster than using the `Find()` function to locate child controls, but requires more memory to store all of the child control pointers. An alternative is to use `Find()` to obtain a pointer to a child control when the pointer is needed.

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

```
PEGINT MyDialog::Message(const PegMessage &Mesg)
{
    switch (Mesg.Type)
    {
    case PEG_SIGNAL(IDB_OK, PSF_CLICKED):
        {
        PegEditField *pEdit = (PegEditField *)
Find(IDS_MY_STRING);

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

    defalt:
        return PegDialog::Message(Mesg);
    }
    return 0;
}
```

```
   virtual PegThing *FindNearestNeighbor(PEGINT
        Key, PegThing *pStart, PegPoint CenterThis)

   virtual PegThing *FindNearestNeighbor(PegThing
        *pStart, PEGLONG *pPutDist, PEGINT Key,
        PegPoint CenterThis, PEGBOOL Loose)
```

This function is used for arrow key handling. It finds the nearest object in a specified direction.

```
PegThing *First(void) const
```

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

```
void FrameStyle(PEGULONG Style)
```

This function can be used to modify the appearance of the frame for most PegThing-derived objects. This function is provided for convenience, and is nearly identical to the `SetStyle()` function shown below with one exception: it guarantees that only the object's frame style is modified, whereas the `SetStyle()` function can modify all style flags.

The available frame styles are:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame.
`FF_RAISED`	Raised 3D Frame.
`FF_RECESSED`	Recessed 3D Frame.
`FF_THICK`	Thick 3D Frame.

```
PEGUSHORT FrameStyle(void)
```

This function returns the current frame style of an object.

```
virtual PEGINT GetColor(const PEGUBYTE Index)
```

This function returns the current color ID for the specified color index of an object. See `SetColor` for a description of color indices.

```
PEGUSHORT GetId(void)
```

Returns the value of the object's `mId` member. The `mId` value is not used by PEG directly, but it is useful to the application software for keeping track of individual controls or other objects when a window such as a complex dialog has several instances of a particular object type associated with it. By assigning IDs to each object, the application can determine precisely the source of a control notification by requesting the control's `mId` value. Object IDs are also used to send and receive signals. The message number associated with a particular signal is calculated based on the object ID and the signal being sent.

```
PEGUSHORT GetSignals(void)
```

This function returns the signals of which the object is set to notify its parent. The available signal masks and descriptions of each may be found here.

```
PEGUSHORT GetStatus(void)
```

Retrieves the current value of the mStatus member variable.

```
virtual PEGULONG GetStyle(void)
```

This function returns the current style flags for an object.

```
Viewport *GetViewportList(void)
```

This function returns the object's list of viewports. This is only available if `PEG_VIEWPORTS` is defined.

```
void Hide(PegThing *pChild)
```

This function stops the object `pChild` from being displayed on the screen. This is similar to the function `Remove`, except that `pChild` still remains a child of its parent. It is still added to the presentation tree; it is just told not to draw.

```
virtual void InitClient(void)
```

This function should be called if the frame or border style of the object is modified at run time after the object has been initialized. This function determines the new client area rectangle based on the `mReal` rectangle and frame style. Many classes override this function to do a custom calculation.

```
PEGBOOL IsDescendentOf(PegThing *pParent)
```

This function determines if the object is a child of `pParent` (or a child of its child of its child of its child, etc.). It returns TRUE if it is a descendent, and FALSE otherwise.

```
virtual void KillFocus(PegThing *pThing)
```

This function generates a `PM_NONCURRENT` message and sends it to `pThing` to tell it that it no longer has focus.

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

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

Messages can either be those defined internally by PEG, or they can be new messages defined by you. PEG system messages are recognized by

the PegMessage.Type field, which is < FIRST_USER_MESSAGE for PEG system messages. For this reason, you should always ensure that your user message types are greater than FIRST_USER_MESSSAGE. A complete list of all PEG system messages is contained in the section of this manual entitled PegMessageQueue.

The complete list of system messages can be found here.

An example of overriding the message function can be found here.

```
void MessageChildren(const PegMessage &Mesg)
```

This function passes the message on to all of its children.

```
static PegMessageQueue *MessageQueue(void)
```

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

```
virtual void MoveFocusToFirstClientChild(void);
```

This function looks through all of the current object's children until it finds one that can accept focus. Then it moves focus to that object.

```
virtual void MoveToFront(PegThing *pWho)
```

This function is used to change the z-order of a parent's child objects. This is useful when objects overlap in the parent's client area. In this case, one object may be brought 'to the front' when selected, so that it is drawn on top of its siblings. The Add() function can also be used to move an object that is already a child to the front of the child list. However, this function differs from the Add() function in that MoveToFront does not change the sequential order of objects; i.e. the tab order is not modified by calling MoveToFront. The presentation tree is simply modified such that the caller is moved to be the first child object.

Like the Add() function, MoveToFront() operates within the constraints of object status, meaning that objects with PSF_VIEWPORT status or PSF_ALWAYS_ON_TOP status are always maintained ahead of child objects which do not have this status.

```
PegThing *Next(void) const
```

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

```
PegThing *NextTabLink(void)
```

Returns a pointer to the object that is next in the tab order, if known; otherwise returns NULL. This function is only provided if PEG_KEYBOARD_SUPPORT and PEG_TAB_KEY_SUPPORT are defined. When this configuration is defined, PEG objects that accept keyboard input focus are linked together in a circular list when the parent window is displayed. The order of this circular list is defined by functions SetDefaultTabLinks() and SetTabOrder().

```
PegThing *Parent(void) const
```

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

```
virtual void ParentShift(PEGINT x, PEGINT y)
```

This function shifts this object and all of its children the specified amount in the x and y direction.

```
static PegPresentationManager
     *Presentation(void)
```

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

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

```
PegThing *Previous(void) const
```

Returns a pointer to the current object's previous sibling, or NULL if the current object is the first node of the current branch of the object tree.

```
PegThing *PrevTabLink(void)
```

Returns a pointer to the object that is previous in the tab order, if known; otherwise returns NULL. This function is only provided if PEG_KEYBOARD_SUPPORT and PEG_TAB_KEY_SUPPORT are defined. When this configuration is defined, PEG objects that accept keyboard input focus are linked together in a circular list when the parent window is displayed. The order of this circular list is defined by functions SetDefaultTabLinks() and SetTabOrder().

```
static PegPrinter *Printer()
```

This function returns a pointer to the printer driver, which is derived from PegPrinter. The current printer driver supports only HP-PCL compatible printers. This function is provided only if `PEG_PRINTER_SUPPORT` is defined in the `pconfig.hpp` configuration file. Note that if this function is called outside of the `PegScreen::BeginPrint` and `PegScreen::EndPrint` functions, this call will return NULL.

```
void ReleasePointer(void)
```

This function acts as a wrapper function allowing access by a PegThing to the PegPresentationManager member function of the same name. Use of this function is equivalent to `Presentation()->ReleasePointer(this)`.

```
virtual PegThing *Remove(PegThing *pWho)
```

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

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

```
virtual void RemoveStatus(PEGULONG AndVal)
```

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

```
virtual void RemoveStyle(PEGULONG AndVal)
```

The opposite of `AddStyle()`, `RemoveStyle()` can be used to clear individual bits or a combination of bits in an object's `mStyle` variable. This function will logically AND the complement of `AndVal` with the object's `mStyle` variable.

```
virtual void Resize(const PegRect &Rect)
```

Any PEG object can resize itself or any other object at any time by calling the `Resize()` function. The new screen coordinates for the object are

passed in the parameter `Rect`. If you maintain or find a pointer to another object, you can also resize that object by calling the same function. The following example illustrates this concept:

```
PegRect Rect(10, 10, 40, 40);

PegButton *MyButton = new PegTextButton(Rect, 0, "Hello");

.

.  // at any time, to resize MyButton:

.

Rect.Set(20, 20, 60, 60);

MyButton-

>Resize(Rect);
```

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

```
static PegScreen *Screen(void)
```

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

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

```
virtual void SendSignal(PEGUBYTE Signal)
```

This function builds a signal based on `Signal` and sends it to this object's parent.

```
virtual void SetColor(const PEGUBYTE Index,
    const PEGINT ColorId)
```

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

PCI_NORMAL: The normal client area fill color.

PCI_SELECTED: The fill color when the object is selected.

PCI_NTEXT: The normal text color for the object.

PCI_STEXT: The text color to use when the object is selected.

The default system color IDs are defined in the file presmgr.hpp. The actual color values associated with these IDs will vary depending on the color depth supported on the target system. A few PEG objects such as PegSpreadsheet have additional color indices associated with them.

```
void SetDefaultTabLinks(void)
```

This is called automatically by PegWindow derived objects when they receive the PM_SHOW message. This establishes the initial order of tabbing through the list of child objects. Note that this function is only defined when PEG_KEYBOARD_SUPPORT and PEG_TAB_KEY_SUPPORT are defined in the pconfig.hpp header file. This function is also called when objects that can receive keyboard focus are added to a visible window.

The default tab order is determined from the position of child objects with PSF_ACCEPTS_FOCUS status. Child objects are placed in the tab list in a top-to-bottom, left-to-right search order. By default, the child object that initially receives input focus is always the top most, left most child object.

The application program can change the default initial focus by first finding the child object with PSF_DEFAULT_FOCUS status and removing this status by calling pChild->RemoveStatus(PSF_DEFAULT_FOCUS). The application can then re-define the initial focus by calling pChild->SetStatus(PSF_DEFAULT_FOCUS) on the desired child object.

```
void SetId(PEGUSHORT Id)
```

Assigns the value of the object's mId member. The default value is 0. Object IDs are used by PEG signaling classes to determine the message

number associated with notification messages. For all other class types, `mId` has no effect on the internal operation of PEG, but it can be useful to the application-level software for identifying objects at run time.

```
static void SetMessageQueuePtr(PegMessageQueue
     *pq)
```

This function replaces the current message queue with `pq`. Note that you may need to delete the original message queue to avoid memory leaks.

```
static void SetPresentationManagerPtr(
     PegPresentationManager *pm)
```

This function replaces the current presentation manager with `pm`. Note that you may need to delete the original presentation manager to avoid memory leaks.

```
static void SetPrinterPtr(PegPrinter *pPrinter)
```

This function is called during program startup to initialize the static pointer to the PEG printer driver. This function is only provided if `PEG_PRINTER_SUPPORT` is defined in the `pconfig.hpp` configuration file.

```
static void SetScreenPtr(PegScreen *pScreen)
```

This function is called during program startup to initialize the static pointer to the PEG screen driver.

```
void SetSignals(PEGUSHORT Mask)
```

This function is used to identify which notification messages a signaling control should send to its parent. The mask value should be created by using the `SIGMASK` macro. This enables multiple signals to be enabled with one call to `SetSignals`, similar to the object style flags. The available signal masks and descriptions of each may be found here.

```
void SetSignals(PEGUSHORT Id, PEGUSHORT Mask)
```

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

```
virtual void SetStyle(PEGULONG Style)
```

This function is used to set the style flags for an object. Not all style flags are supported by all classes. In all cases, the desired style flags can be 'OR'ed together to form one style parameter.

As an aid in remembering the names of the style flags, the flags are grouped into different categories, and the name of each flag starts with an abbreviation of that category. For example, the frame flag names start with FF for Frame Flag, and the button flags start with BF for Button Flag. The style flags are found here.

```
void SetTabLink(PegThing *pNext)
```

This function determines which object that focus moves to when the Tab key is pressed. `PEG_KEYBOARD_SUPPORT` and `PEG_TAB_KEY_SUPPORT` must be defined.

```
void SetTabOrder(PEGUSHORT *pIds)
```

This function will override the default tab order of the PEG objects. This is only provided if both #define `PEG_KEYBOARD_SUPPORT` and #define `PEG_TAB_KEY_SUPPORT` are enabled in the configuration file `pconfig.hpp`. The parameter `pIds` should be an array of object IDs sorted in your preferred order, with the last entry set to 0. If the object of an ID doesn't exist, it ignores it and continues with the next one. Make sure that this function is called after the base object becomes visible, because it is there that the default tab order is installed. For example:

```
PEGINT MyPegClass::Message(const PegMessage &Mesg)
{
    // Terminate with 0
    PEGUSHORT Order[] = {MyID_1, MyID_2, MyID_3, 0};

    switch (Mesg.Type)
    {
    case PM_SHOW:
        // This will set the default tab order.
        PegWindow::Message(Mesg);
        // This will set user-defined tab order.
        SetTabOrder(Order);
        break;

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

```
static void SetTimerManager(PegTimerManager *pt)
```

This function replaces the current timer manager with `pt`. Note that you may need to delete the old timer manager to avoid memory leaks.

```
void Show(PegThing *pWhat)
```

This function is responsible for making an object visible on the screen. An object can theoretically be added to a parent window without becoming visible. Calling `Show()` will make it visible. Calling `Hide()` will make invisible again.

```
void StandardBorder(PEGCOLOR FillColor)
```

This function draws a standard border based on the object frame style, fill color, and `mReal` values.

```
PEGBOOL StatusIs(PEGULONG Mask)
```

This function is used to test individual bits of an object's private `mStatus` variable. This variable contains system status flags common to all PEG classes. An application program generally should never attempt to modify these flags. However, it is sometimes useful to read this value to test for certain object states. The system status flag list and definitions are found here.

```
PEGUSHORT Type(void)
```

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

```
void Type(PEGUSHORT Set)
```

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

```
void UpdateChildClipping(void)
```

This function updates the `mClip` rectangle of all of the child (and descendent) objects based on the current object's `mReal` and `mClip` rectangles.

```
const PEGCHAR *Version(void)
```

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

1.6.6 Public Data Members:

```
PegRect mClient
```

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

```
PegRect mClip
```

This rectangle defines the clipping rectangle of the object. This may be smaller than `mReal` if the object extends beyond the client area of its parent.

```
PegRect mReal
```

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

```
PEGINT mColorId[4]
```

This array defines the four basic colors that all PegThing objects use to draw themselves. The colors are indexed as `PCI_NORMAL`, `PCI_NTEXT`, `PCI_SELECTED`, and `PCI_STEXT`.

1.6.7 Inline Wrapper Functions:

```
inline void Arc(PEGINT xc, PEGINT yc, PEGINT
     XRadius, PEGINT YRadius, PEGINT
     start_angle, PEGINT end_angle, PegBrush
     &Brush, PEGINT width = 1)
```

Implementation: `Arc(xc, yc, XRadius, YRadius, start_angle, end_angle, Brush, width)`

```
inline void BeginDraw(const PegRect &Invalid)

inline void BeginDraw(const PegRect &Invalid,
     PEGINT Surface)
```

Implementation: `Screen()->BeginDraw(this, Invalid, mSurface)`

The first form of this function is used to draw into the default mSurface. The second form is used to begin drawing into some other existing surface defined in the application.

```
inline void Bitmap(PegPoint Where, PegBitmap
     *pGetmap)

inline void Bitmap(PegPoint Where, PEGUINT
     BitmapId)
```

Implementation: `Screen()->Bitmap(Where, pGetmap)`

The first form is used when a direct bitmap pointer is available. The second form is used when a bitmap's ID is available. In that case, the ID is used to find the bitmap pointer in the PegResourceManager.

```
inline void BitmapFill(PegRect Rect, PegBitmap
     *pGetmap)
```

Implementation: `Screen()->BitmapFill(Rect, Getmap)`

```
inline void Circle(PEGINT xCenter, PEGINT
     yCenter, PEGINT radius, PegBrush &Brush)
```

Implementation: `Screen()->Circle(xCenter, yCenter, radius, Brush)`

```
inline void DrawText(PegPoint Where, const
     PEGCHAR *pText, PegBrush &Brush, const
     PegFont *pFont, PEGINT Count = -1)
```

Implementation: `Screen()->DrawText(Where, pText, Brush, pFont, Count)`

```
inline void EndDraw(void)
```

Implementation: `Screen()->EndDraw()`

```
inline void Invalidate(void)
```

Implementation: `Presentation()->Invalidate(this, mReal)`

```
inline void Invalidate(const PegRect &Rect)
```

Implementation: `Presentation()->Invalidate(this, Rect)`

```
inline void KillTimer(PEGUSHORT Id)
```

Implementation: TimerManager()->KillTimer(this, Id);

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

Implementation: `Screen()->Line(XStart, YStart, XEnd, YEnd, Brush)`

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

Implementation: `Screen()->Rectangle(Rect, Brush)`

```
inline PEGBOOL RectMove(PegRect Get, PegRect
    ClipTo, PEGINT xShift, PEGINT yShift)
```

Implementation: `Screen()->RectMove(Get, ClipTo, xShift, yShift)`

```
inline void SetPointerType(PEGUBYTE Type)
```

Implementation: `Screen()->SetPointerType(Type)`

```
inline void SetTimer(PEGUSHORT Id, PEGLONG
    Count, PEGLONG Reset)
```

Implementation: `TimerManager()->SetTimer(this, Id, Count, Reset)`

```
inline PEGINT TextHeight(const PEGCHAR *pText,
    const PegFont *pFont)
```

Implementation: `Screen()->TextHeight(pText, pFont)`

```
inline PEGINT TextWidth(const PEGCHAR *pText,
    const PegFont *pFont)
```

Implementation: `Screen()->TextWidth(Text, Font)`

1.6.8 Protected Members:

```
PEGUSHORT mId
```

Object ID value.

```
PEGULONG mStatus
```

Object status flags.

```
PEGULONG mStyle
```

Object style flags.

1.7 PegTimerManager

1.7.1 Overview

PegTimerManager is where all timer functionality resides. This involves starting and stopping PegTimers owned by PegThing objects as well as keeping track of the underlying timer ticks that keep the PegTimers going.

After enough ticks have gone by to set off a PegTimer, a PM_TIMER message is sent to whatever object created the timer. If this PegTimer was created with a non-zero reset value, then it will get reset to go off again after the specified number of ticks. If the reset value was 0, or if the `KillTimer` function is called, then the PegTimer will be deleted.

The PegTimerManager class itself is entirely portable and independent of any specific RTOS implementation. It works by having either the PegTask or a separate timer task constantly call `TimerTick()` at a regular interval. This can be implemented with interrupts or with a polling loop.

Note that the PegThing class includes wrapper functions for `SetTimer` and `KillTimer` that pass the PegThing's `this` pointer as the target for the timer. This is just a convenience, since this is typically the only way those functions ever get called.

1.7.2 Derivation

PegTimerManager is a PEG base class.

1.7.3 Style Flags

None.

1.7.4 Constructors:

```
PegTimerManager(void)
```

This constructs a PegTimerManager object. There only needs to be one instance of this in an application. This instance is referenced by all PEG objects.

1.7.5 Public Functions:

```
void KillTimer(PegThing *pWho, PEGUSHORT Id)
```

This function is responsible for removing the timer that contains the ID `Id`, and is targeted for `pWho`. If `Id` is 0, then all timers targeted for `pWho` will be eliminated.

```
void SetTimer(PegThing *pWho, PEGUSHORT Id,
     PEGLONG Count, PEGLONG Reset)
```

This function starts (or restarts) a timer with the ID `Id`. The timer will initially be set to go off after `Count` number of timer ticks, and then repeat after `Reset` number of ticks. If `Reset` is 0, then it will not repeat. When the timer does go off, a `PM_TIMER` message will be sent to `pWho`.

```
void TimerTick(void)
```

This function is called by either the PegTask or by a separate timer task. It will go through every PegTimer that has been created so far and decrement their counters by one. If a counter reaches 0, that means it is time to send a `PM_TIMER` message to whatever object the timer is targeted for. If its reset value was non-zero, the counter will be reset to that value. Otherwise, the PegTimer is deleted.

1.7.6 Example:

This example shows a PegPrompt that changes color twice per second. Note that the PegThing wrapper functions are used here to simplify the code.

```
#define MY_TIMER_ID 1

PEGINT MyPrompt::Message(const PegMessage &Mesg)
{
    switch (Mesg.Type)
    {
    case PM_SHOW:
        SetTimer(MY_TIMER_ID, ONE_SECOND / 2,
                 ONE_SECOND / 2);
        PegPrompt::Message(Mesg);
        break;

    case PM_TIMER:
```

```
        SetColor(PCI_NTEXT, GetColor(PCI_NTEXT) + 1);
        Draw();
        break;

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

    return 0;
}
```

CHAPTER 2

CONTROL CLASSES

PegAnimation
PegBitmapButton
PegBitmapSlider
PegBmpProgressBar
PegButton
PegCheckBox
PegDecoratedButton
PegEditField
PegGroup
PegHelpButton
PegHScroll
PegIcon
PegIconButton
PegMenu
PegMenuBar
PegMenuButton
PegMLTextButton
PegProgressBar
PegPrompt
PegRadioButton
PegScroll
PegScrollPrompt
PegSlider
PegSpinButton
PegStatusBar
PegTextButton

PegTitle
PegToolBar
PegToolBarPanel
PegVPrompt
PegVScroll

2.1 PegAnimation

2.1.1 Overview

PegAnimation is an animated bitmap display class. It is designed to be used for the simple display of animated GIF type images; however, any list of images can be created and used with the PegAnimation class.

The PegAnimation class requires an array of PegAnimationFrame structures to define the animation images. This array is created automatically by PEG WindowBuilder, or it can be created manually if desired. The PegAnimationFrame structure is defined as follows:

```
typedef struct {
    PEGUINT  BitmapId;
    PEGINT   Delay;
    PEGINT   xOffset;
    PEGINT   yOffset;
    PEGINT   Disposal;
} PegAnimationFrame;
```

The structure fields are:

`BitmapId`: ID value of the PegBitmap to display
`Delay`: Time delay in timer ticks between frames
`xOffset`: Offset from the left edge of the PegAnimation object
`yOffset`: Offset from the top edge of the PegAnimation object
`Disposal`: Control field indicating how to erase the previous frame. Supported values are (2) = background color fill, (0/1) = do nothing, and (4) = write previous.

2.1.2 See Also

PegIcon

2.1.3 Style Flags

PegAnimation supports the standard frame styles `FF_NONE`, `FF_THIN`, `FF_THICK`, `FF_RAISED`, and `FF_RECESSED`.

2.1.4 Signals

PegAnimation class sends no signals.

2.1.5 Derivation

PegAnimation is derived from PegThing.

2.1.6 Constructors:

```
PegAnimation(const PegRect &Rect, PegAnimationFrame
     *pFrameList, PEGUSHORT Id = 0, PEGULONG Style =
     FF_NONE)

PegAnimation(const PegRect &Rect, PEGUSHORT Id = 0,
     PEGULONG Style = FF_NONE)
```

The constructor creates a PegAnimation widget. `Rect` is the object size and position. `pFrameList` is the address of an array of PegAnimationFrame structures. The array is terminated with a structure having a -1 bitmap ID.

2.1.7 Public Functions:

```
void AssignFrameList(PegAnimationFrame *pFrameList);
```

This function can be called to reassign the animation frame list. The parameter is the address of an array of PegAnimationFrame structures. The array is terminated with a structure containing a -1 bitmap ID.

```
virtual void Draw(const PegRect &Invalid)
```

PegAnimation overrides the `Draw()` function to draw the associated bitmap frames.

```
PEGINT GetCurrentFrame(void)
```

Returns the index of the current frame of animation.

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

PegAnimation overrides the `Message` function to receive timer messages for drawing the animation frames.

```
void SetCurrentFrame(PEGINT Frame)
```

Sets the current frame of animation to be `Frame`.

```
void SetMode(PEGUSHORT Mode)
```

This function is called to set the mode of operation. The following mode flags are supported, and can be OR'ed together to produce the desired animation mode:

```
PEG_AF_WRAP
```

This mode causes the animation to continuously wrap from the last frame to the first. If this mode flag is not set, the animation will run once when activated and terminate after displaying the last animation frame.

```
PEG_AF_SHOWIDLE
```

This mode flag causes the animation to display the current animation frame even when the animation is idle (i.e. it is not cycling through the list of animation frames). If this flag is not set, the animation widget displays no image when it is not running.

```
PEG_AF_AUTOSTART
```

This mode flag causes the animation to automatically start running when it is visible. If this mode flag is not set, the animation is started via program control by calling the `Start()` function.

```
void Start(PEGINT Frame = -1);
```

This function is called to start the animation display sequence, if the `PEG_AF_AUTOSTART` mode is not set. If the `Frame` parameter is -1, the animation starts at the current frame. The application can also pass a frame number >= 0 to begin the animation at any specified frame index.

```
void Stop(void);
```

This function stops the animation. The animation will display the last active frame if `PEG_AF_SHOWIDLE` mode is set; otherwise, it will display no graphic when stopped.

2.2 PegBitmapButton

2.2.1 Overview

PegBitmapButton is a PegButton class that displays a PegBitmap with no frame. There are actually three PegBitmaps associated with a PegBitmapButton. There is one for the normal state, one for the pressed state, and one for the focused state.

The button is NOT restricted to stay the same dimensions as the bitmaps, so if the bitmap is smaller than the `mReal` rectangle of the button, then the outer section of the button will essentially be transparent. Keep in mind, however, that the user will still be able to click in the region outside the bitmap if it is still within the button's `mReal`. For some applications, this might not be ideal, so it would be a good idea to keep the dimensions the same.

Also, since there is no visible frame, if the bitmap has transparency in it, then the button itself appears transparent. But it's a good idea to make all of the bitmaps that the button uses transparent in the same locations. For example, if the normal bitmap is rectangular with the exception of a rounded upper left corner, and the pressed bitmap is rectangular with the exception of a rounded lower right corner, then there are going to be artifacts left on the screen whenever the bitmap changes. The reason is because the transparent region does not get redrawn. So if there was a solid rectangle drawn there before, it won't go away if a transparent bitmap is drawn on top of it. To avoid this, the parent window can be redrawn to clear out the background.

The PegBitmapButton uses four bitmaps, which can be indexed with the following enumerations.

`PBMI_NORMAL`	Normal button state
`PBMI_PRESSED`	Pressed button state
`PBMI_FOCUS`	Focused button state
`PBMI_DISABLED`	Disabled button state

2.2.2 See Also

PegButton

PegIconButton

PegTextButton

PegRadioButton

PegCheckBox

2.2.3 Style Flags

PegBitmapButton supports the standard frame styles `FF_NONE`, `FF_THIN`, `FF_THICK`, `FF_RAISED`, and `FF_RECESSED`.

PegBitmapButton also supports the button styles `BF_REPEAT`, `BF_DOWNACTION`, `BF_EXCLUSIVE` and `BF_TOGGLE`.

2.2.4 Signals

PegBitmapButton sends the `PSF_CLICKED` signal when selected.

2.2.5 Derivation

PegBitmapButton is derived from PegButton.

2.2.6 Constructors:

```
PegBitmapButton(const PegRect &Rect, PEGUINT
     NormalBmp, PEGUINT PressedBmp, PEGUINT
     FocusedBmp = 0, PEGUSHORT Id = 0, PEGULONG Style
     = AF_ENABLED|FF_NONE)
```

The constructor creates a PegBitmapButton with a user-defined size. The `NormalBmp` and `PressedBmp` are required, but if `FocusedBmp` is 0, then the button defaults to using the `NormalBmp` when it receives focus.

2.2.7 Public Functions:

```
virtual void AssignBitmaps(PEGUINT NormalBmp, PEGUINT
     PressedBmp, PEGUINT FocusedBmp = 0, PEGUINT
     DisabledBmp = 0)
```

This function is used to assign all of the bitmaps used by the button. If `FocusedBmp` or `DisabledBmp` are equal to 0, then they are set to use `NormalBmp`.

```
virtual void Draw(const PegRect &Invalid)
```

PegIconButton overrides the `Draw()` function to draw the bitmaps.

```
virtual void SetBitmap(PEGUINT Index, PEGUINT
     BitmapId)
```

This function assigns an individual bitmap to the button using the specified `Index` and `BitmapId`.

2.2.8 Protected Members

```
PEGuINT mBmpIds[4]
```

This is the array of bitmap IDs used by the button.

2.2.9 Examples:

The following example creates a PegBitmapButton. It uses a red circle bitmap for the normal state, a green circle bitmap for the focused state, and a modified green circle bitmap for the pressed state.

```
...
...
PegRect Rect;
Rect.Set(20, 20, 99, 49);
Add(new PegBitmapButton(Rect, BID_RED_BTN, BID_GREEN_UP_BTN,
    BID_GREEN_DOWN_BTN));
...
...
```

2.3 PegBitmapSlider

2.3.1 Overview

PegBitmapSlider is an analog adjustment control. The end user adjusts the slider value by dragging the slider 'handle.' PegSlider can be either horizontal or vertical. The orientation is determined by the style flags. This class accepts bitmaps for the background and for the handle to allow a much more customized appearance.

PegBitmapSlider sends `PSF_SLIDER_CHANGE` notification signals to the slider parent when the user adjusts the slider value.

PegBitmapSlider contains two bitmaps referenced by the following enumerations.

`PBMI_SLIDER_BACKGROUND`	The background bitmap of the slider
`PBMI_SLIDER_NEEDLE`	The needle button bitmap of the slider

2.3.2 See Also

PegSlider

2.3.3 Style Flags

PegBitmapSlider defines the following styles:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame.
`FF_RAISED`	Raised 3D Frame.
`FF_RECESSED`	Recessed 3D Frame.
`SF_SNAP`	Snaps the slider handle to the exact tick positions.
`SS_ORIENTVERT`	Determines orientation. If this is used, the slider is vertically oriented. Otherwise, it is horizontal.

SS_FACELEFT	Determines the horizontal position of the slider button when SS_ORIENTVERT is also used. If this is used, the button is placed on the right side of the slider so that it can 'face left.' Otherwise, it is placed on the left side so that it faces right.
SS_FACETOP	Determines the vertical position of the slider button when SS_ORIENTVERT is not used. If this is used, the button is placed on the bottom edge of the slider so that it can 'face top.' Otherwise, it will be placed on the top edge so that it faces the bottom.

2.3.4 Signals

PegBitmapSlider sends PSF_SLIDER_CHANGE signals to the slider parent when adjusted by the end user. The message contains the following values:

Message.pSource = Pointer to slider control.
Message.Param = ID of slider control.
Message.ExtParams[0] = Current slider value.

2.3.5 Derivation

PegBitmapSlider is derived from PegSlider.

2.3.6 Constructors:

```
PegBitmapSlider(const PegRect &Rect, PEGLONG Min,
      PEGLONG Max, PEGINT BkgndBmp = 0, PEGINT
      NeedleBmp = 0, PEGUINT Id = 0, PEGULONG Style =
      FF_RAISED, PEGLONG Scale = -1)
```

The PegBitmapSlider constructor creates a slider control at the position and size specified in Rect. The Min and Max values specify the initial limits of the slider. The BkgndBmp parameter is the ID of the bitmap that will be drawn in the background. The NeedleBmp parameter is the ID of the bitmap that will be drawn for the needle or handle of the slider. If either bitmap IDs are -1, the default PegSlider appearance will be used. The slider ID, if non- zero, enables the PSF_SLIDER_CHANGE notification signal. The slider scale determines the interval between slider tickmarks.

2.3.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegBitmapSlider overrides the `Draw()` function to draw the bitmaps on the slider

```
PEGINT GetBitmap(PEGINT Index) const
```

This inline function returns the ID of the bitmap specified by `Index`.

```
void SetBitmap(PEGINT Index, PEGINT BmpId)
```

This function assigns a bitmap to the slider using the specified `Index` and `BmpId`.

```
PEGINT GetNeedleOffset(void) const
```

This inline function returns the offset of the slide button where it should actually be pointing at the current value.

2.4 PegBmpProgressBar

2.4.1 Overview

PegBmpProgressBar is a simple progress bar control used to indicate to an end user the completion status of a slow activity. PegBmpProgressBar can assume any scale value within the range of the `PEGINT` data type; however, it is most common to display a value that is a percentage of the completion status.

The style, range, and initial value of a PegBmpProgressBar object are passed to the object constructor. As the operation being monitored progresses, the application software calls the `Update()` member function to change the displayed completion value.

The bitmap progress bar control has three main styles. The default style is `PS_BITMAP`. This style displays a background bitmap and an overlay bitmap. As progress increases, more of the overlay bitmap is shown. Multiple overlay bitmaps are can be used to create an animation effect. Another style, `PS_BUTTON`, displays a gradient filled indicator, in which case the progress bar internally draws a PegButton within the specified frame style. The `PS_LED` style, on the other hand, does not use a client area button to indicate progress; instead, it invokes a custom drawing style meant to appear as a series of LEDs, with the lighted LEDs indicating the current progress.

2.4.2 See Also

PegProgressBar
PegProgressWindow

2.4.3 Style Flags

PegBmpProgressBar supports the following styles:

PS_SHOW_VAL	This style instructs the progress bar to display the current progress value in text form, in addition to the graphical presentation.
PS_RECESSED	This style draws a solid, recessed progress indicator. The default appearance is a solid raised indicator.
PS_LED	This style draws a segmented LED indicator. The default appearance is a solid raised indicator.
PS_VERTICAL	This style orients the progress control graphical indicator vertically. The default is horizontal orientation.
PS_PERCENT	This style instructs the progress bar to add the '%' indicator to the displayed text value. This flag has no effect if the PS_SHOW_VAL style is not active.

2.4.4 Signals

PegBmpProgressBar is a passive object, is not user selectable, and sends no signals.

2.4.5 Derivation

PegBmpProgressBar is derived from PegProgressBar.

2.4.6 Constructors:

```
PegBmpProgressBar(const PegRect &Rect,
     PEGULONG Style = FF_THIN|PS_SHOW_VAL|PS_PERCENT|
     PS_BUTTON, PEGUINT BakbmpID = 0,
      PEGUINT Ovlay1ID = 0, PEGUINT Ovlay2ID = 0,
     PEGINT Min = 0, PEGINT Max = 100,
     PEGINT Current = 0)
```

This constructor creates a PegBmpProgressBar. The default values construct a progress bar that displays both text and a graphical value. The graphical indicator has a raised border. The progress bar has a range of 0 to 100, and displays a '%' sign after the output value. The BkgndBmp parameter is the ID of the bitmap that will be drawn in the background. The Ovlay1ID parameter is the ID of the bitmap that will be drawn to show the

graphical representation of the current progress. The `Ovlay2ID` parameter is also a bitmap ID used to show progress. The bitmap progress bar alternates between displaying `Ovlay1ID` and displaying `Ovlay2ID` to create an animation effect.

```
PegBmpProgressBar(PegBitmap *Background, PegBitmap
     *Overlay1, PegBitmap *Overlay2,const PegRect
     &Rect, PEGULONG Style = FF_THIN|PS_SHOW_VAL|
      PS_PERCENT| PS_BUTTON, PEGINT Min = 0,
      PEGINT Max = 100, PEGINT Current = 0)
```

Similar as above constructor except this version uses PegBitmap pointers instead of bitmap IDs.

2.4.7 Public Functions:

```
void Draw(const PegRect &Invalid)
```

PegBmpProgressBar overrides the `Draw()` function to check for and draw the background bitmap and overlay bitmaps.

```
PEGINT Message(const PegMessage &Mesg)
```

The `Message()` function has been overridden to handle messages that start timers, stop timers, and switch overlay bitmaps.

```
PEGINT GetAnimationRate(void)
```

Returns the animation rate.

```
void SetAnimationRate(PEGINT Rate)
```

Sets the animation rate.

```
void SetBitmaps(PegBitmap *Background, PegBitmap
*Overlay1 = NULL, PegBitmap *Overlay2 = NULL)
```

Sets bitmaps using PegBitmap pointers.

```
void SetBitmaps(PEGINT Bkgr_ID, PEGINT Overlay1_ID,
PEGINT Overlay2_ID)
```

Sets bitmaps using PegBitmap pointers.

```
void SetBitmap(PegBitmap *Img, PEGINT TypeID)
```

This function sets the bitmap designated by `TypeID` using a PegBitmap pointer.

```
void SetBitmap(PEGINT Img_ID, PEGINT TypeID)
```

This function sets the bitmap designated by `TypeID` using an image ID.

```
void SetMode(PEGUSHORT Mode)
```

This function sets the animation mode i.e. `PEG_AF_AUTOSTART` for the overlay bitmaps.

```
void Start(void)
```

This function starts the overlay bitmap animation.

```
void Stop(void)
```

This function stops the overlay bitmap animation.

2.4 PegButton

2.4.1 Overview

PegButton serves as the base class for nearly all PEG button style objects. PegButton provides the `BF_REPEAT` timer operation, default frame drawing, and default selection `PEG_SIGNAL`s. You would not normally create an instance of PegButton in your system software; however, PegButton is very useful as a base class for your own custom button styles.

2.4.2 See Also

PegBitmapButton

PegIconButton

PegTextButton

PegRadioButton

PegCheckBox

2.4.3 Style Flags

PegButton supports the standard frame styles `FF_NONE`, `FF_THIN`, `FF_THICK`, `FF_RAISED`, and `FF_RECESSED`.

PegButton also supports the button styles `BF_REPEAT`, `BF_DOWNACTION`, `BF_EXCLUSIVE` and `BF_TOGGLE`.

2.4.4 Signals

PegButton sends the `PSF_CLICKED` signal when selected.

2.4.5 Derivation

PegButton is derived from PegTextThing.

2.4.6 Constructors:

```
PegButton(const PegRect &Rect, PEGUINT StringId = 0,
     PEGUINT Id = 0, PEGULONG Style = AF_ENABLED|
     FF_RAISED, PEGUINT FontIndex = 0))

PegButton(const PEGCHAR *pText, const PegRect &Rect,
     PEGUINT Id = 0, PEGULONG Style = TT_COPY|
     AF_ENABLED|FF_RAISED, PEGUINT FontIndex = 0)

PegButton(void)
```

The first and second constructors creates a PegButton with a defined size and position. The third constructor creates a PegButton with an undefined position. Note that PegButtons take text and a font as parameters, but by default do not draw them on the button. To display text on a button, use PegTextButton.

2.4.7 Public Functions:

```
void Disable(void)
```

This function disables the button, and prevents it from being selected.

```
virtual void Draw(const PegRect &Invalid)
```

PegButton overrides the `Draw()` function to provide the 3D appearance of the button when it is selected.

```
void Enable(void)
```

Enables the button, and allows it to be selected.

```
virtual void HandleDownClick(PEGINT Type)
```

This function handles down-click functionality. If the button has `BF_TOGGLE` or `BF_EXCLUSIVE` style, the button becomes selected. Otherwise, it draws itself in the pressed state. If it has `BF_DOWNACTION` style, a `PSF_CLICKED` signal will also be generated. The `Type` parameter determines the type of down-click that occurred (i.e. either a `PM_LBUTTONDOWN` or a `PM_KEY`).

```
virtual void HandleUpClick(const PegMessage &Mesg)
```

This function handles up-click functionality. If the button has `BF_TOGGLE`, it becomes unselected. Otherwise, it draws itself in the normal (un-depressed) state and sends a `PSF_CLICKED` signal. The `Mesg` parameter is the message that occurred that triggered an up-click (i.e. either a `PM_LBUTTONUP` or a `PM_KEY_RELEASE`).

```
virtual void Initialize(PEGULONG Style)
```

This is a common initialization function called by both constructors to set up colors and status flags.

```
virtual PEGBOOL IsSelected(void)
```

This returns TRUE if the button contains the BF_PUSHED style. Otherwise, it returns false.

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

PegButton catches PM_LBUTTONDOWN, PM_LBUTTONUP, and PM_TIMER messages.

```
virtual void SetSelected(PEGBOOL Select)
```

This function either selects or unselects the button by adding or removing the BF_PUSHED style flag. If the button is being selected and it has BF_EXCLUSIVE style, then it will also unselect all of its sibling PegButtons.

```
virtual void SetStyle(PEGULONG Style)
```

PegButton overrides the PegThing::Style function to toggle PSF_SELECTABLE and PSF_TAB_STOP status if the AF_ENABLED style flag is changed.

2.4.8 Protected Members:

```
void CheckSignal(PEGBOOL Selected)
```

This function checks to see if a signal needs to be sent to the parent object based on whether or not the button is selected, and if it has the BF_TOGGLE or BF_EXCLUSIVE style flags. The signals it could send are PSF_CLICKED, PSF_CHECK_ON, PSF_CHECK_OFF, PSF_DOT_ON or PSF_DOT_OFF.

```
void UnselectSiblings(void)
```

This function is used to make sure all other buttons that are children of this button's parent object will become unselected.

2.5 PegCheckBox

2.5.1 Overview

PegCheckBox is a PegButton class that provides a toggle operation. Any number of PegCheckBox objects with a common parent can be selected.

The PegCheckBox uses 4 bitmaps, which can be indexed with the following enumerations.

`PBMI_CHECK_ON`	Checked button state
`PBMI_CHECK_OFF`	Unchecked button state
`PBMI_CHECK_ON_DIS`	Checked and disabled button state
`PBMI_CHECK_OFF_DIS`	Unchecked and disabled button state

2.5.2 See Also

PegButton

PegTextButton

PegRadioButton

PegIconButton

PegBitmapButton

2.5.3 Style Flags

PegCheckBox supports the `BF_PUSHED`, `AF_ENABLED`, `BF_TOGGLE`, and `BF_EXCLUSIVE` styles.

2.5.4 Signals

PegCheckBox sends the `PSF_CHECK_ON` signal when selected, and the `PSF_CHECK_OFF` signal when de-selected.

2.5.5 Derivation

PegCheckBox is derived from PegButton.

2.5.6 Constructors:

```
PegCheckBox(const PegRect &Rect, PEGUINT StringId,
      PEGUSHORT Id = 0, PEGULONG Style = AF_ENABLED|
      FF_NONE|BF_TOGGLE)

PegCheckBox(const PEGCHAR *pText, const PegRect &Rect,
      PEGUSHORT Id = 0, PEGULONG Style = TT_COPY|
      AF_ENABLED|FF_NONE|BF_TOGGLE)
```

The first constructor creates a PegCheckBox by obtaining text from the string table using a string ID. The second constructor takes a pointer to a string directly.

2.5.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegCheckBox overrides the `Draw()` function to draw the checkbox appearance.

```
virtual PEGBOOL IsChecked(void)
```

Returns TRUE if the checkbox is selected, else FALSE.

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

PegCheckBox catches the `PM_LBUTTONDOWN` and `PM_LBUTTONUP` messages.

```
virtual void SetBitmap(PEGUINT Index, PEGUINT
      BitmapId)
```

This function is used to assign a bitmap to the button using the specified `Index` and `BitmapId`.

```
virtual void SetSelected(PEGBOOL Selected = TRUE)
```

This inline function can be called at any time to toggle the checkbox on or off under program control.

2.5.8 Examples:

The following is an example of PegCheckBox:

The following example creates a PegCheckBox. The checkbox will determine its own width based on the string width.

```
...
...
PegRect Rect;
Rect.Set(20, 20, 99, 39);
Add(new PegCheckBox("I like Coffee", Rect));
...
...
```

2.6 PegDecoratedButton

2.6.1 Overview

PegDecoratedButton is a PegButton class that has the ability to display both text and a PegBitmap. The location of the text relative to the bitmap can be selected using a set of extended style flags. PegDecoratedButton also supports a 'flyover' mode, where the button appears flat until the pointer is over the button.

2.6.2 See Also

PegButton

PegIconButton

PegBitmapButton

PegTextButton

PegRadioButton

PegCheckBox

2.6.3 Style Flags

PegDecoratedButton supports the standard frame styles `FF_NONE`, `FF_THIN`, `FF_THICK`, `FF_RAISED`, and `FF_RECESSED`.

PegDecoratedButton also supports the button styles `BF_REPEAT`, `BF_DOWNACTION`, `BF_TOGGLE`, and `BF_EXCLUSIVE`.

PegDecoratedButton also supports two extended styles, `BF_ORIENT_TR` and `BF_ORIENT_BR`. These style flags are independent of the PegButton-derived style flags. These flags allow the bitmap and text to be positioned relative to each other. Here is how they work together.

`!BF_ORIENT_TR && !BF_ORIENT_BR`	Bitmap is displayed to the left of the text
`BF_ORIENT_TR && !BF_ORIENT_BR`	Bitmap is displayed above the text
`!BF_ORIENT_TR && BF_ORIENT_BR`	Bitmap is displayed below the text
`BF_ORIENT_TR && BF_ORIENT_BR`	Bitmap is displayed to the right of the text

2.6.4 Signals

PegDecoratedButton sends the `PSF_CLICKED` signal when selected.

2.6.5 Derivation

PegDecoratedButton is derived from PegButton.

2.6.6 Constructors:

```
PegDecoratedButton(const PegRect &Rect, PEGINT
      StringId, PEGINT BitmapId, PEGUSHORT Id = 0,
      PEGULONG Style = AF_ENABLED, PEGBOOL FlyOver =
      FALSE)

PegDecoratedButton(const PEGCHAR *pText, const PegRect
      &Rect, PEGINT BitmapId, PEGUSHORT Id = 0,
      PEGULONG Style = AF_ENABLED, PEGBOOL FlyOver =
      FALSE)
```

The constructors use the parameter `Rect` to determine the size of the button, and can take in both a bitmap and text. Depending on which constructor is used, either a string ID or a pointer to the text is passed in.

2.6.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegDecoratedButton overrides the `Draw()` function to draw the associated text and bitmap.

```
PEGBOOL GetFlyOver(void) const
```

This inline function returns a boolean indicating whether the PegDecoratedButton flyover mode for drawing itself is enabled or disabled.

```
PEGBOOL InFlyOver(void) const
```

This inline function returns whether or not the button is currently operating in flyover mode. In other words, if flyover mode is set to TRUE, then this will return TRUE if the pointer is over the button. If the pointer is currently not over the button, then this will return FALSE.

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

PegDecoratedButton overrides the `Message()` method to catch the `PM_NONCURRENT`, `PM_POINTER_ENTER`, and `PM_POINTER_EXIT` messages.

```
virtual void SetBitmap(PegBitmap *pNewBitmap)
```

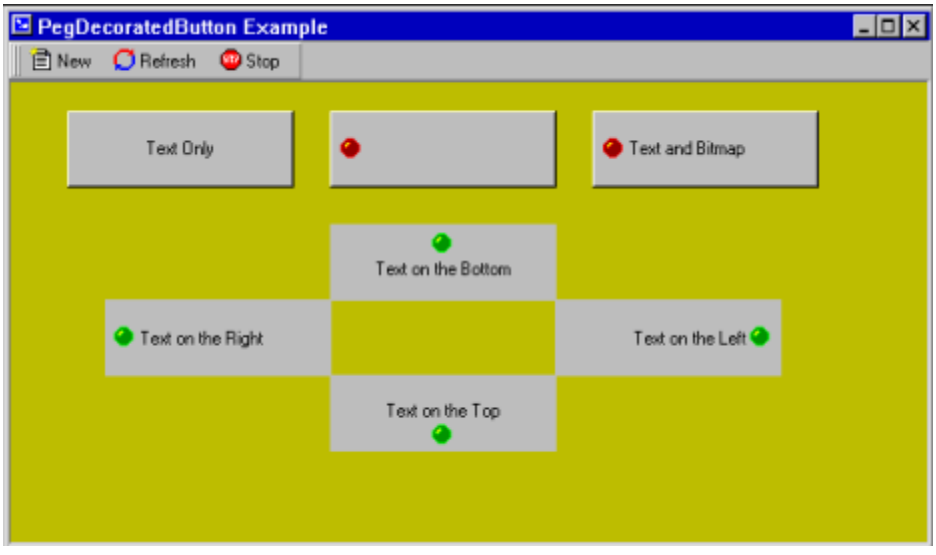
This inline function assigns the bitmap associated with the button. The bitmap may be changed at any time.

```
void SetFlyOver(PEGBOOL FlyOver)
```

This inline function enables or disables the PegDecoratedButton flyover mode for drawing itself.

2.6.8 Examples:

The following are examples of PegDecoratedButton:



The above example shows seven PegDecoratedButtons on the parent window. The first row of three does not have flyover enabled. (They are always drawn just like any other PegButton-derived object). They also demonstrate text-only, bitmap-only, and text-and-bitmap button configurations.

The other four PegDecoratedButtons in the diamond pattern have flyover enabled. They are drawn flat until the pointer 'flies over' them. At that point, the border (if any) is drawn. This style is typical of most toolbar buttons in modern GUIs. They also demonstrate how the text and bitmap may be oriented on the button.

The PegToolBar on the parent window has one PegToolBarPanel attached to it with three PegDecoratedButtons on it. These buttons are also drawn in 'flyover' style. (They do not draw their borders until the pointer is over them.) When a PegDecoratedButton is put on a PegToolBarPanel, it is best to have the bitmap situated either to the right or to the left of the text, not on the top or bottom, since the height of the button is restricted when it is on a PegToolBarPanel.

The following example creates the four PegDecoratedButtons displayed in the above image in a diamond pattern on the parent PegDecoratedWindow.

.

```
.
.
PegRect Rect;
.
.
.
Rect.Shift(-175, 75);
pButton = new PegDecoratedButton(Rect, SID_TEXT_ON_BOTTOM,
        BID_GREEN_DOT, 0, AF_ENABLED | BF_ORIENT_TR, TRUE);
pWindow->Add(pButton);

Rect.Shift(-150, 50);
pButton = new PegDecoratedButton(Rect, SID_TEXT_ON_RIGHT,
        BID_GREEN_DOT, 0, AF_ENABLED, TRUE);
pWindow->Add(pButton);

Rect.Shift(150, 50);
pButton = new PegDecoratedButton(Rect, SID_TEXT_ON_TOP,
        BID_GREEN_DOT, 0, AF_ENABLED | BF_ORIENT_BR, TRUE);
pWindow->Add(pButton);

Rect.Shift(150, -50);
pButton = new PegDecoratedButton(Rect, SID_TEXT_ON_LEFT,
        BID_GREEN_DOT, 0, AF_ENABLED | BF_ORIENT_TR |
BF_ORIENT_BR,
        TRUE);
pWindow->Add(pButton);
```

2.7 PegEditField

2.7.1 Overview

PegEditField is a user-editable graphical string object. PegEditField can be displayed with any font or color and several different border styles. Only one line of text can be displayed in a PegEditField.

PegEditField supports mark, cut, copy, and paste operations via keyboard input when the definition #define `PEG_KEYBOARD_INPUT` is enabled in the file `\peg\include\pconfig.hpp`. The length of the contained string can be limited if desired. PegEditField shifts the string left or right as it is edited if it cannot be completely displayed in the client area of the PegEditField object.

2.7.2 See Also

PegPrompt

PegTextBox

2.7.3 Style Flags

PegEditField supports the following styles:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame
`FF_RAISED`	Raised 3D Frame
`FF_RECESSED`	Recessed 3D Frame
`EF_EDIT`	When this style is applied, the user can edit the PegEditField object. If this style is applied, the PegEditField object automatically includes the `TT_COPY` style.
`TT_COPY`	Instructs the PegEditField to copy the text string assigned. This flag should be used when the text string assigned to the PegEditField is created dynamically using temporary storage.

2.7.4 Signals

In addition to the common signals defined by PegThing, PegEditField supports the following signals:

```
PSF_TEXT_SELECT  // sent when the user selects text
PSF_TEXT_EDIT // sent each time text is modified
PSF_TEXT_EDITDONE // sent when a text modification is complete
```

2.7.5 Derivation

PegEditField is derived from PegTextThing.

2.7.6 Constructors:

```
PegEditField(const PegRect &Rect, PEGUINT StringId =
     0, PEGUSHORT Id = 0, PEGULONG Style =
     FF_RECESSED|AF_ENABLED|EF_EDIT, PEGINT Len = -1)

PegEditField(const PEGCHAR *pText, const PegRect
     &Rect, PEGUSHORT Id = 0, PEGULONG Style =
     FF_RECESSED|AF_ENABLED|EF_EDIT, PEGINT Len = -1)
```

The first constructor takes a string ID as the initial text, while the second constructor takes a pointer to a string.

2.7.7 Public Functions:

```
void CopyToScratchPad(void)
```

This method copies the currently selected text to the scratch pad.

```
virtual void DataSet(const PEGCHAR *pText)
```

PegEditField overrides the `DataSet()` function to reset any in-progress string mark or edit operations. This version of the `DataSet` function is used for dynamically created strings that are not in the string table.

```
virtual void DataSet(PEGINT StringId)
```

PegEditField overrides the `DataSet()` function to reset any in-progress string mark or edit operations. This version of the `DataSet` function is used for string IDs.

```
void DeleteMarkedText(void)
```

This method deletes the marked text.

```
virtual void Draw(const PegRect &Invalid)
```

PegEditField overrides the `Draw()` function to display the string border and text.

```
virtual PEGINT GetMarkEnd(void)
```

Returns the index of the last character marked by the user.

```
virtual PEGINT GetMarkStart(void)
```

Returns the index of the first character marked by the user.

```
virtual PEGINT GetMaxLen(void)
```

Returns the maximum number of characters the object will keep.

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

PegEditField catches various mouse and keyboard messages.

```
void PasteFromScratchPad(void)
```

This method pastes the text from the scratch pad and inserts it at the current cursor position.

```
virtual void SetMark(PEGINT Start, PEGINT End)
```

This function can be used to mark all or a portion of the string text under program control. The `Start` and `End` values are the character indexes at which to begin and end marking, inclusive.

```
virtual void SetMark(PEGCHAR *pMarkStart, PEGCHAR
     *pMarkEnd)
```

This function can be used to mark all or a portion of the string text under program control. The `pMarkStart` and `pMarkEnd` values are pointers to the characters at which to begin and end marking, inclusive.

```
virtual void SetMaxLen(PEGINT Length)
```

This function assigns the maximum number of characters that the object will store.

```
virtual void SetStyle(PEGULONG Style)
```

PegEditField overrides the `Style` function to reset any in-progress mark or edit operations if/when the EF_EDIT style is removed.

2.7.8 Protected Members

```
virtual void AdvanceCursor(PEGINT New)
```

This function advances the cursor based on the width of the `New` character.

```
virtual void DrawMarked(void)
```

This function is used to draw the text when some or all of it is highlighted (marked).

```
virtual void Initialize(PEGULONG Style)
```

This function is a common initialization function called by all of the constructors to set up colors and status flags.

```
virtual PEGBOOL InsertKey(PEGINT Key)
```

This function inserts the character `Key` at the current cursor location.

```
virtual void RetardCursor(PEGINT New)
```

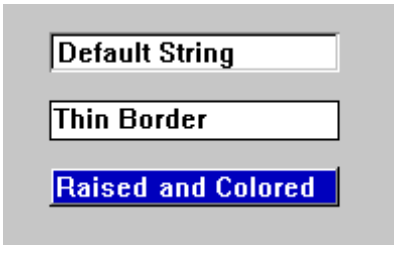
This function moves the cursor back based on the width of the `New` character.

```
virtual void SetCursorPos(PegPoint PickPoint)
```

This function finds a valid location nearest to `PickPoint` to position the cursor.

2.7.9 Examples:

The following are each different styles of PegEditField:



The following function creates a PegEditField object with a custom font:

```
void MyWindow::AddCustomString(void)
{
    PegRect ChildRect;
```

```
    ChildRect.Set(0, 0, 100, 40);
    PegEditField *pField = new PegEditField(ChildRect,
SID_FOOBAR);
    pField->SetFont(FID_CUSTOM_FONT);
    Add(pField);
}
```

2.8 PegGroup

2.8.1 Overview

PegGroup is a container class that visually groups any number of children. You can add any type of PEG object to a PegGroup, including window objects, nested groups, etc.

PegGroup is especially useful for grouping sets of PegRadioButton objects, since these objects are mutually exclusive within a common parent.

2.8.2 See Also

PegRadioButton

PegThing

PegTextThing

2.8.3 Style Flags

PegGroup supports the `AF_ENABLED` style.

2.8.4 Signals

PegGroup sends no signals. Signals sent by child objects of PegGroup are passed up to the parent of the PegGroup object.

2.8.5 Derivation

PegGroup is derived from PegTextThing.

2.8.6 Constructors:

```
PegGroup(const PegRect &Rect, PEGINT StringId,
     PEGULONG Style = AF_ENABLED)

PegGroup(const PEGCHAR *pText, const PegRect &Rect,
     PEGULONG Style = AF_ENABLED)
```

The PegGroup constructor creates a PegGroup at the indicated position with an initial text value indicated by either a string ID or a string pointer.

2.8.7 Public Functions:

```
virtual void Add(PegThing *pWho, PEGBOOL Show = TRUE)
```

PegGroup is not a Viewport object, and therefore overrides the `Add` function to prevent breaks in the Viewport tree.

```
virtual void Disable(void)
```

This function removes the `AF_ENABLED` style flag from the PegGroup and also calls the `Disable` function for any PegButton-derived child objects.

```
virtual void Draw(const PegRect &Invalid)
```

PegGroup overrides the `Draw()` function to draw the group box.

```
virtual void Enable(void)
```

This function is used to enable the group and all children of the group.

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

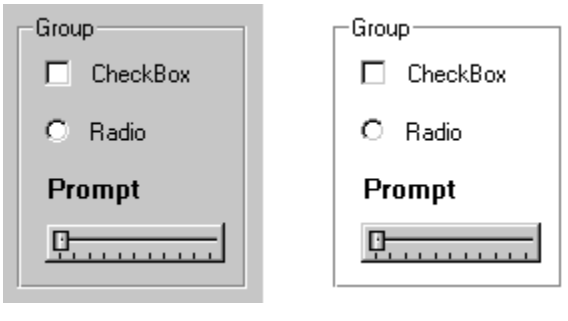
PegGroup catches the `PM_SHOW` message.

```
virtual PegThing *Remove(PegThing *pWhat)
```

Since PegGroup does not fill the client area of the group when an object is removed from it, PegGroup commands the parent object to redraw. For this reason PegGroup overrides the `Remove()` function.

2.8.8 Examples:

The following are examples of PegGroup:



The following example creates a PegGroup and adds a collection of PegRadioButton objects to the group. The group is then added to the parent object.

```
void MyWindow::AddGroup(void)
{
    PegRect GroupRect;
    GroupRect.Set(mClient.Left + 10, mClient.Top + 10,
        mClient.Right - 10, mClient.Top + 100);

    PegGroup *pGroup = new PegGroup(GroupRect,
"RadioButtons");

    PegRect ChildRect;
    ChildRect.Set(mClient.Left + 20, mClient.Top + 20,
        mClient.iLeft + 79, mClient.Top + 39);
    pGroup->Add(new PegRadioButton(ChildRect, "Button1"));
    ChildRect.Shift(0, 20);
    pGroup->Add(new PegRadioButton(ChildRect, "Button2"));
    ChildRect.Shift(0, 20);
    pGroup->Add(new PegRadioButton(ChildRect, "Button3"));

    Add(pGroup);
}
```

2.9 PegHelpButton

2.9.1 Overview

PegHelpButton is a PegIconButton class that displays a text string when the user moves the mouse over it.

2.9.2 See Also

PegButton

PegTextButton

PegIconButton

2.9.3 Style Flags

PegHelpButton supports the standard frame styles `FF_NONE`, `FF_THIN`, `FF_THICK`, `FF_RAISED`, and `FF_RECESSED`.

PegHelpButton also supports the button styles `BF_REPEAT`, `BF_DOWNACTION`, `BF_EXCLUSIVE`, and `BF_TOGGLE`.

2.9.4 Signals

PegHelpButton sends the `PSF_CLICKED` signal when selected.

2.9.5 Derivation

PegHelpButton is derived from PegIconButton.

2.9.6 Constructors:

```
PegHelpButton(const PegRect &Rect, PEGUINT BitmapId,
     PEGUINT StringId, PEGUSHORT Id, PEGULONG Style =
     AF_ENABLED)

PegHelpButton(const PEGCHAR *pText, const PegRect
     &Rect, PEGUINT BitmapId, PEGUSHORT Id, PEGULONG
     Style = AF_ENABLED)
```

The constructor creates a PegHelpButton with a user-defined size. The PegBitmap associated with the button will be displayed in the center of the button client area. The string that is passed will be displayed when the user moves the mouse over the button.

2.9.7 Public Functions:

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

PegHelpButton overrides the `Message()` function to handle the mouse messages.

2.10 PegHScroll

2.10.1 Overview

PegHScroll is a horizontal scroll bar class. The scroll bar elevator is proportional to the visible area of the object being scrolled.

PegHScroll takes two forms. The most common form is a NONCLIENT area scroll bar. In this form, PegHScroll calls the parent window `GetHScrollInfo` function to determine position, size, and limit information. An instance of this form of PegHScroll has `PSF_NONCLIENT` system status.

The second form is a client area scroll bar. This form does not have `PSF_NONCLIENT` system status. This type of scroll bar is under system software control, and does not attempt to automatically determine position and limit information.

Client area PegHScroll objects are very similar in operation to PegSlider objects. They are useful for allowing the user to update a field on the display by dragging the scroll elevator or selecting the directional scrolling buttons.

2.10.2 See Also

PegScrollInfo

PegScroll

PegVScroll

PegSlider

PegThing

How Scrolling Works

2.10.3 Style Flags

There are no styles associated with PegHScroll Objects.

2.10.4 Signals

PegHScroll sends `PSF_SCROLL_CHANGE` signals when the position of the scroll bar elevator is changed either by dragging the elevator or by selecting the directional scroll buttons. The signal message contains the following information:

`Message.Param` = ID of the PegHScroll object.
`Message.ExtParams[0]` = Current scroll position.
`Message.ExtParams[1]` = Last reported scroll position.
`Message.pSource` = Pointer to PegHScroll object.

2.10.5 Derivation

PegHScroll is derived from PegScroll.

2.10.6 Constructors:

```
PegHScroll(PegScrollDrawInfo *pDrawInfo = NULL)

PegHScroll(const PegRect &InRect, PegScrollInfo *pSi,
     PEGUINT Id = 0, PegScrollDrawInfo *pDrawInfo =
     NULL)
```

The first constructor creates a non-client area scroll bar. The scroll bar will automatically determine its position and size itself to the width of the parent window.

The second constructor creates a client area scroll bar. In this mode, a pointer to a PegScrollInfo structure is passed to set up the initial scrolling range. In this mode, the scroll bar position and size are passed to the constructor, along with the scroll bar ID, if any.

2.10.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegHScroll overrides the `Draw()` function to fill the scroll bar background area.

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

PegHScroll catches `PM_SHOW`, directional button selection, and elevator drag messages.

```
virtual void Resize(PegRect Rect)
```

PegHScroll overrides the `Resize()` function to ensure that the elevator remains proportional to the overall scroll bar size.

```
void SetThumbColor(PEGCOLOR Color)
```

This allows for the elevator (thumb) button to change color.

2.10.8 Protected Members

```
virtual void CreateButtons(void)
```

This function is responsible for creating the left and right arrow buttons, as well as the elevator button. This is a separate virtual function that can be overridden in derived classes that want to use different types of buttons.

```
virtual void ReadParentScrollInfo(void)
```

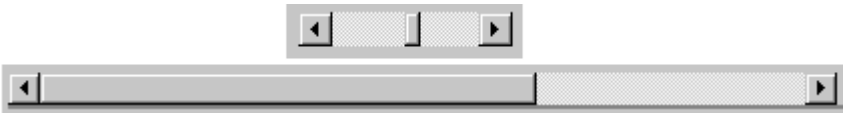
This function retrieves the PegScrollInfo data from the parent object and checks to see if it needs to be scrolled.

```
virtual void SetThumbButtonPos(void)
```

This function calculates and updates the position of the scroll button.

2.10.9 Examples:

The following are examples of PegHScroll:



The following example initializes a PegScrollInfo structure and creates a client area scroll bar. The scroll bar will report values between 0 and 200 and will initially be positioned at 100; the scroll bar elevator will be 25% as wide as the scroll bar.

```
void MyWindow::AddHScroll(void)
{
```

```
    PegScrollInfo si;

    si.Min = 0; si.Max
    = 200; si.Current =
    100; si.Step = 1;
    si.Visible = 50;
    PegRect ScrollRect;
    ScrollRect.Set(10, 10, 120, PEG_SCROLL_WIDTH + 10);

    Add(new PegHScroll(ScrollRect, &si));
}
```

2.11 PegIcon

2.11.1 Overview

PegIcon is a simple bitmap display object. PegIcon can also be used to represent another object.

PegIcon can be assigned a 'Proxy' object pointer. If this pointer is assigned, the icon will add the Proxy object to the icon parent and destroy itself when selected. This is how window icons function. PegDecoratedWindow classes will create an instance of PegIcon when they are minimized, add the icon to the window parent, and remove themselves from the parent.

A PegIcon object with no Proxy assignment is also useful for simply displaying a PegBitmap.

2.11.2 See Also

PegBitmap

PegThing

2.11.3 Style Flags

None.

2.11.4 Signals

None.

2.11.5 Derivation

PegIcon is derived from PegThing.

2.11.6 Constructors:

```
PegIcon(PegThing *pProxy, PEGINT BitmapId = 0, PEGUINT
      Id = 0, PEGULONG Style = FF_NONE)

PegIcon(const PegRect &Where, PEGINT BitmapId = 0,
      PEGUINT Id = 0, PEGULONG Style = FF_NONE)

PegIcon(PEGINT BitmapId = 0, PEGUINT Id = 0, PEGULONG
      Style = FF_NONE)
```

The first constructor creates a PegIcon that represents or serves as a proxy for another object. The second and third constructors create a PegIcon that will simply display a bitmap. The second constructor allows the caller to specify the icon size and position. The third constructor allows the icon to self determine the overall icon size to match the bitmap size. When the third constructor is used, the application software may immediately use the `Resize()` function to position the icon.

2.11.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegIcon overrides the `Draw()` function to draw the icon bitmap.

```
virtual PegBitmap *GetIcon(void)
```

This inline function returns the address of the bitmap associated with the PegIcon.

```
virtual PegThing *GetProxy(void)
```

This inline function returns the address of the object that is represented by the PegIcon.

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

PegIcon catches `PM_LBUTTONUP` messages.

```
virtual void SetIcon(PegBitmap *nbm)
```

This inline function can be used to alter the PegIcon bitmap at any time.

```
virtual void SetProxy(PegThing *pNew)
```

This inline function can be used to assign or alter the object represented by the PegIcon.

2.11.8 Examples:

The following are examples of PegIcon:

2.12 PegIconButton

2.12.1 Overview

PegIconButton is a PegButton class that displays a PegBitmap centered in the button client area.

2.12.2 See Also

PegButton

PegTextButton

PegRadioButton

PegCheckBox

2.12.3 Style Flags

PegIconButton supports the standard frame styles `FF_NONE`, `FF_THIN`, `FF_THICK`, `FF_RAISED`, and `FF_RECESSED`.

PegIconButton also supports the button styles `BF_REPEAT`, `BF_DOWNACTION`, `BF_EXCLUSIVE` and `BF_TOGGLE`.

2.12.4 Signals

PegIconButton sends the `PSF_CLICKED` signal when selected.

2.12.5 Derivation

PegIconButton is derived from PegButton.

2.12.6 Constructors:

```
PegIconButton(const PegRect &Rect, PEGUINT BmpId,
      PEGUINT Id = 0, PEGULONG Style = AF_ENABLED|
      FF_RAISED)
```

The constructor creates a PegIconButton with a user defined size. The PegBitmap associated with the button will be displayed in the center of the button client area.

2.12.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegIconButton overrides the `Draw()` function to draw the associated bitmap.

```
virtual void SetBitmap(PEGUINT BmpId)
```

This inline function assigns the bitmap associated with the button. The bitmap may be changed at any time.

2.12.8 Protected Members

```
PEGUINT mBitmapId
```

This is the ID of the bitmap drawn on the button.

2.12.9 Examples:

The following example creates a PegIconButton. The button will display the PegBitmap 'OnMap' in the button client area.

```
...
...
PegRect Rect;
Rect.Set(20, 20, 99, 49);
Add(new PegIconButton(Rect, BID_ON_MAP));
...
...
```

2.13 PegMenu

2.13.1 Overview

PegMenu is the background object used to display a list of menu items. PegMenu may contain any number of PegMenuButton objects.

PegMenu accepts a PegMenuDescription parameter. This description defines the PegMenuButton objects that will initially be created and displayed on the menu. The PegMenuDescription is most often defined statically, or defined by PegWindowBuilder, but the PegMenuDescription can also be created dynamically during program execution.

When dynamically created PegMenuDescriptions are used to create instances of PegMenu, it is important to include the `TT_COPY` flag in each PegMenuDescription style. By default, the menu descriptions are not copied.

PegMenu also provides functions for finding, adding, and removing PegMenuButton objects at any time. This allows PegMenu objects to be altered at run time under program control.

PegMenu objects are automatically created by PegMenuBar when top-level menu bar options are selected. PegMenu sizes itself to the size required to display all children. The position of the PegMenu is defined when the menu is displayed.

2.13.2 See Also

PegMenuBar

PegMenuButton

PegMenuDescription

2.13.3 Style Flags

None.

2.13.4 Signals

None.

2.13.5 Derivation

PegMenu is derived from PegThing.

2.13.6 Constructors:

```
PegMenu(PegMenuDescription *pDesc, PEGBOOL Popup =
     FALSE)
```

The PegMenu constructor creates a PegMenu object with PegMenuButton children. The children are defined by the PegMenuDescription parameter.

2.13.7 Public Functions:

```
void CloseSiblings(PegMenuButton *pNotMe)
```

This function closes all of the submenus except for `pNotMe`.

```
virtual void Draw(const PegRect &Invalid)
```

PegMenu overrides the `Draw()` function to draw the menu border and background.

`PegMenuButton *FindButton(const PEGCHAR *pWho)` This function searches for a PegMenuButton child with a matching command string. This function is in addition to the normal `Find(PEGUINT Id)` which may also be used to find specific PegMenu command buttons.

The pointer returned can be used to Enable/Disable/Remove the PegMenuButton.

```
PegRect GetMinSize(void)
```

This function examines all child objects of the PegMenu to determine the minimum size required to display all children. The resulting rectangle is returned. This result can be used to intelligently position the PegMenu.

```
virtual void MenuKeyHandler(PEGINT Key, PEGLONG Flags)
```

This is the default keyboard handler for the PegMenu. This function is only provided when #define `PEG_KEYBOARD_SUPPORT` is enabled in the configuration file pconfig.hpp.

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

PegMenu catches the `PM_SHOW` message. When this message is received, the menu positions child PegMenuButtons for proper display.

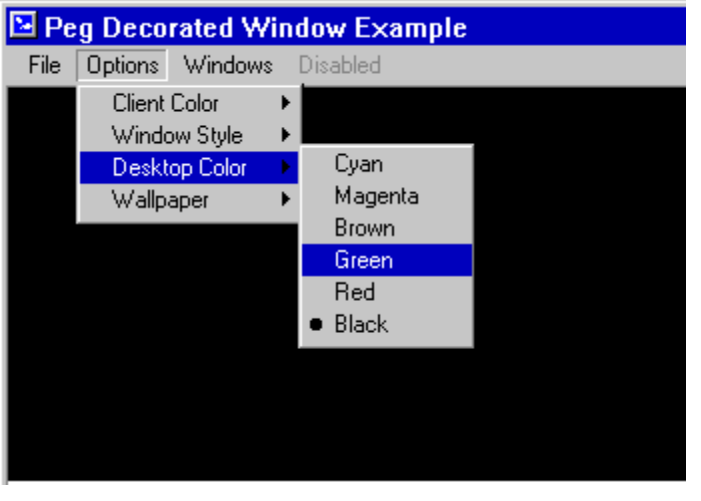
```
void SetOwner(PegThing *pWho)
```

This function assigns the owner of the menu. Menu command signals are routed to the PegMenu owner if this value is not NULL. This is often required as PegMenu objects are usually added directly to PegPresentationManager. This is done because the PegMenu may extend beyond the borders of the owner window and it is not intended that the PegMenu be clipped to the borders of the owner window. Assigning the PegMenu an owner routes menu commands to the owner window, rather than to the PegMenu parent, which in the case described would be PegPresentationManager.

2.13.8 Protected Members

2.13.9 Examples:

The following are examples of PegMenuBar, PegMenu, and PegMenuButton:



The following example creates a PegMenu from a PegMenuDescription. The menu is positioned at the top left corner of the window client area, and the menu owner is assigned. The menu is then added to PegPresentationManager:

```
static PegMenuDescription FileMenu[] =
{
    {"Exit", IDB_DEMO_EXIT, AF_ENABLED, NULL},
    {"Close", IDB_CLOSE, AF_ENABLED, NULL},
    {"Save", 0, 0, NULL},
    {"", 0, BF_SEPARATOR, NULL},
    {"Restore", IDB_RESTORE, AF_ENABLED, NULL},
    {"Minimize", IDB_MINIMIZE, AF_ENABLED, NULL},
    {"Maximize", IDB_MAXIMIZE, AF_ENABLED, NULL},
    {"", 0, 0, NULL}
};

void MyWindow::PopUpFileMenu(void)
{
    PegMenu *pMenu = new PegMenu(FileMenu);
    PegRect SizeRect = pMenu->GetMinSize();
    SizeRect.MoveTo(mClient.Left, mClient.Top);
    pMenu->Resize(SizeRect);
    pMenu->SetOwner(this);
    Presentation()-
    >Add(pMenu);
}
```

Note: In the above example, the PegMenu should be removed when a menu command is received or the owner window loses input focus.

2.14 PegMenuBar

2.14.1 Overview

PegMenuBar is a window decoration used to position and display a user command menu. PegMenuBar is designed to work in conjunction with PegDecoratedWindow objects. PegMenuBar may be added to any type of object. However, the client area of objects other than PegDecoratedWindow will not be reduced properly unless this is done in the application software.

PegMenuBar automatically positions and sizes itself to the parent window. PegMenuBar may contain any number of PegMenuButton objects.

PegMenuBar accepts a PegMenuDescription parameter. This description defines the PegMenuButton objects that will initially be created and displayed on the menu bar. The PegMenuDescription is most often defined statically, or defined by PegWindowBuilder, but the PegMenuDescription can also be created dynamically during program execution.

When dynamically created PegMenuDescriptions are used to create instances of PegMenuBar, it is important to include the `TT_COPY` flag in each PegMenuDescription style. By default, the menu descriptions are not copied.

PegMenuBar also provides functions for finding, adding, and removing PegMenuButton objects at any time. This allows PegMenu objects to be altered at run time under program control. The functions for finding and modifying menu command objects are recursive, and search the entire menu tree for the specified items.

PegMenuBar automatically creates and displays PegMenu objects when top-level menu commands are selected.

2.14.2 See Also

PegMenu

PegMenuButton

PegMenuDescription

2.14.3 Style Flags

None.

2.14.4 Signals

None.

2.14.5 Derivation

PegMenuBar is derived from PegThing.

2.14.6 Constructors:

```
PegMenuBar(PegMenuDescription *pDesc)
```

The PegMenuBar constructor creates a PegMenuBar object with PegMenuButton children. The children are defined by the PegMenuDescription parameter. PegMenuBar automatically determines its position and size.

2.14.7 Public Functions:

```
virtual void CloseSiblings(PegMenuButton *pNotMe)
```

This method closes all of the siblings of `pNotMe`.

```
virtual void Draw(const PegRect &Invalid)
```

PegMenu overrides the `Draw()` function to draw the menu bar background.

`PegMenuButton *FindButton(const PEGCHAR *pWho)` This function searches for a PegMenuButton child with a matching command string. This function is in addition to the normal `Find(PEGUINT Id)` which may also be used to find specific PegMenu command buttons.

The pointer returned can be used to Enable/Disable/Remove the PegMenuButton. This function is recursive, and will search the entire menu tree for the indicated command button.

```
virtual PEGBOOL InFlyoverMode(void)
```

Returns the state of the menu in flyover mode.

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

PegMenuBar catches `PM_PARENTSIZED` and `PM_NONCURRENT` messages.

```
virtual void SetFlyoverMode(PEGBOOL State)
```

This function puts the PegMenuBar object in flyover mode. Flyover mode means that submenu items are automatically highlighted as the mouse moves over them.

2.14.8 Protected Members

```
virtual void MenuKeyHandler(PEGINT Key, PEGLONG Flags)
```

This function handles all keystroke messages.

```
virtual void PositionButtons(void)
```

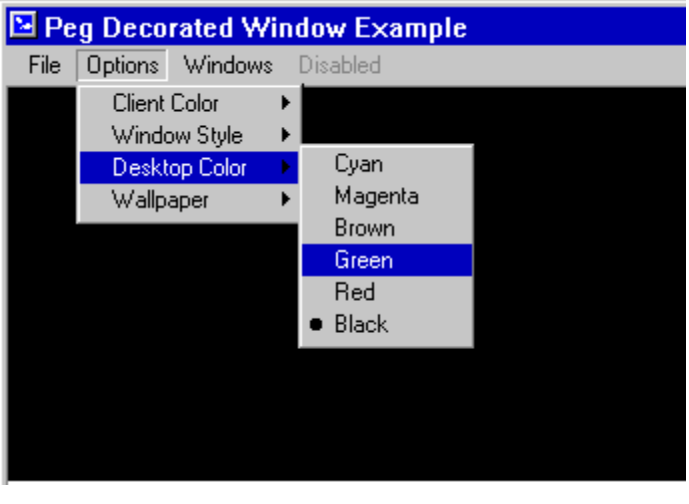
This function calculates the positions of all the buttons on the menu bar.

```
virtual void SizeToParent(void)
```

This function checks the parent's size and recalculates its own dimensions.

2.14.9 Examples:

The following are examples of PegMenuBar, PegMenu, and PegMenuButton:



The following example creates a PegMenuBar and adds the menu bar to the parent window. In this example, 'HelpMenu,' 'WindowsMenu,'

'OptionsMenu,' and 'FileMenu' are additional PegMenuDescription arrays (not shown) that define the submenus of the menu bar.

```
static PegMenuDescription MainMenu[] =
{
    {"Help",    0, AF_ENABLED, HelpMenu},
    {"Disabled", 1, 0, NULL},
    {"Windows", 0, AF_ENABLED, WindowsMenu},
    {"Options", 0, AF_ENABLED, OptionsMenu},
    {"File", 0, AF_ENABLED, FileMenu},
    {"", 0, 0, NULL}
};

void MyWindow::AddMenuBar(void)
{
    Add(new PegMenuBar(MainMenu));
}
```

2.15 PegMenuButton

2.15.1 Overview

PegMenuButton is a button class used to populate PegMenu and PegMenuBar objects. PegMenuButton sends selection signals to the owner of the PegMenu or PegMenuBar.

Custom appearance for PegMenuButton objects can be created by altering the PegMenuButton `Draw()` function or, preferably, by deriving new versions of PegMenuButton from the PegMenuButton class. Overriding the `Draw()` function of PegMenuButton allows the developer to create any desired menu button appearance.

The styles for PegMenuButton descriptions are very important. They define whether the item sends a simple command, or whether it defines additional characteristics such as 'dotable' or 'checkable.'

PegMenuButton objects are automatically created by PegMenuBar and PegMenu objects to satisfy the corresponding PegMenuDescriptions. It is NOT necessary to create individual PegMenuButton objects manually. Simply defining the PegMenuDescription is all that is required to create a fully functional command menu.

2.15.2 See Also

PegMenuBar

PegMenu

PegMenuDescription

2.15.3 Style Flags

PegMenuButton supports the following style flags:

`BF_SEPARATOR` - The menu button is a separator item. `BF_CHECKABLE` - The menu button can be 'checked' or 'unchecked.' `BF_CHECKED` - If checkable, defines initial state of the menu button. `BF_DOTABLE` - The menu button is mutually exclusive when selected. `BF_DOTTED` - If dotable, defines initial state of the menu button.

2.15.4 Signals

None.

2.15.5 Derivation

PegMenuButton is derived from class PegTextThing.

2.15.6 Constructors:

```
PegMenuButton(PegMenuDescription *pDesc)
```

The PegMenuButton constructor creates a PegMenuButton object. The description should contain either a valid button ID or the address of an additional PegMenuDescription array.

2.15.7 Public Functions:

```
virtual void CloseMenu(void)
```

This function closes the submenu associated with the PegMenuButton, if any.

```
void Disable(void)
```

This function disables a menu button so that the user cannot select it. The text also is drawn gray by default.

```
virtual void Draw(const PegRect &Invalid)
```

PegMenuButton overrides the `Draw()` function to display the menu command text string. This function may be overridden to define a custom menu button appearance.

```
void Enable(void)
```

This function enables a menu button so that the user can select it.

```
virtual PegThing *Find(PEGUSHORT Id, PEGBOOL Recursive
      = TRUE)
```

This method searches through its child objects to find an object with an ID of `Id`. If `Recursive` is true, it will search through the button's ancestry until it either finds the object or exhausts the list of child objects.

```
PegMenuButton *FindButton(const PEGCHAR *pWho)
```

This function can be used to find a particular PegMenuButton object. This function is called by PegMenu and PegMenuBar to recursively search the menu tree when the associated functions of the same name are invoked.

```
virtual PegRect GetMinSize(PEGUBYTE uType)
```

This function returns the minimum height and width required to display the PegMenuButton object. The returned value includes space for checkmarks, dot selection, and submenu indicator bitmaps.

```
PegThing *GetOwner(void)
```

Returns the owner object of the button.

```
PegMenu *GetSubMenu(void)
```

This function returns a pointer to the submenu associated with a menu button, or NULL if the button has no submenu.

```
virtual PEGBOOL IsChecked(void)
```

For `BF_CHECKABLE` PegMenuButton objects, this function returns TRUE if the item is currently selected, else FALSE.

```
virtual PEGBOOL IsDotted(void)
```

For `BF_DOTABLE` PegMenuButton objects, this function returns TRUE if the item is currently selected, else FALSE.

```
PEGBOOL IsPointerOver(void)
```

Returns a boolean stating whether the mouse pointer is over this menu button.

```
virtual PEGBOOL IsSeparator(void)
```

This function returns true if the menu button is a separator style button.

```
PEGBOOL IsSubVisible(void)
```

Returns a boolean stating whether the submenu of this button is visible.

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

PegMenuButton catches `PM_POINTERENTER`, `PM_POINTEREXIT`, `PM_LBUTTONDOWN`, and `PM_LBUTTONUP` messages.

```
virtual void SetChecked(PEGBOOL State)
```

This function can be called to check or uncheck a checkable menu item under program control, rather than through the normal method of user selection.

```
virtual void SetDotted(PEGBOOL State)
```

This function can be called to select a dotable menu item under program control, rather than through the normal method of user selection.

```
virtual void SetOwner(PegThing *pWho)
```

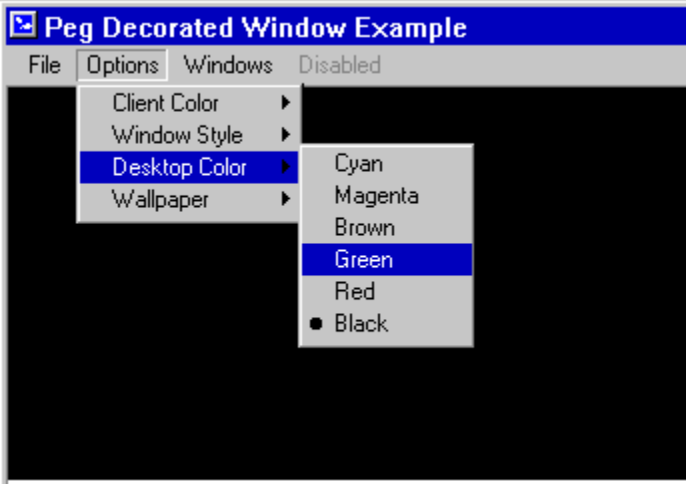
This function is called by PegMenu and PegMenuBar objects to assign the button parent. This directs the menu button command signals to the menu owner window, instead of to the menu parent window.

```
void SetSubMenu(PegMenu *pMenu)
```

This function can be called to assign at run time the submenu associated with a menu button. This is commonly used when a PegMenu is defined or modified at run time using a dynamically constructed PegMenuDescription. This function does not need to be called if the PegMenuDescription for a menu tree is statically defined.

2.15.8 Examples:

The following are examples of PegMenuBar, PegMenu, and PegMenuButton:



PegMenu example, PegMenuBar example.

2.16 PegMLTextButton

2.16.1 Overview

PegMLTextButton provides a pushbutton object with visual feedback that indicates button depress and release operation. PegMLTextButton differs from PegTextButton in that the text displayed can span ***multiple lines*** in the button client area, hence the class name PegMLTextButton.

Any custom fonts or colors may be applied to PegMLTextButton. Users often create custom derived versions of PegMLTextButton to draw modified border styles, modified border shapes, etc. PegMLTextButton is also commonly used to populate PegHorzList and PegVertList objects.

The text strings displayed on the button face are vertically centered over the button client area, and may be horizontally justified in different ways by using the different text justification styles such as `TJ_LEFT` or `TJ_CENTER`.

The text breaks for a multi-line text button are defined by the caller when the text string is passed to the button, either during object construction or with the `DataSet()` function. The character to be used as a break character is passed to the class constructor(s), and defaults to the value of the following define:

```
#define DEF_ML_MARKER  0x7c       // the '|' character
```

This allows the application engineer to easily change the character used as a line delimiter.

The break character can also be used to accomplish simple formatting, such as insertion of blank lines. If the first or last characters in the string are any number of break characters, the vertical centering will be adjusted. If the string contains two or more back-to-back break characters, blank lines will be inserted into the displayed text.

2.16.2 See Also

PegButton

PegTextButton

PegIconButton

PegBitmapButton

PegRadioButton

PegCheckBox

2.16.3 Style Flags

PegMLTextButton supports the following styles:

TJ_RIGHT	Right justified text.
TJ_LEFT	Left justified text.
TJ_CENTER	Centered text.
BF_REPEAT	This flag causes the button to send periodic PSF_CLICKED signals when held down by the user.
BF_DOWNACTION	This flag causes the button to send the PSF_CLICKED signal on the down press of the button, rather than the default action which is to signal on the button release.
AF_ENABLED	This flag allows the button to be selected.

2.16.4 Signals

PegMLTextButton sends PSF_CLICKED signals when selected.

2.16.5 Derivation

PegMLTextButton is derived from PegButton.

2.16.6 Constructors:

```
PegMLTextButton(const PegRect &Size, PEGINT StringId,
     PEGCHAR Marker = DEF_ML_MARKER, PEGUINT Id = 0,
     PEGULONG Style = TJ_CENTER|AF_ENABLED)

PegMLTextButton(const PEGCHAR *pText, const PegRect
     &Size, PEGCHAR Marker = DEF_ML_MARKER, PEGUINT
     Id = 0, PEGULONG Style = TJ_CENTER|AF_ENABLED)
```

PegMLTextButton offers two constructors, depending on how you want to specify the button text. The first constructor allows you to reference the string table with a string ID. The second constructor allows you to specify the text directly with a string pointer.

2.16.7 Public Functions:

```
virtual void DataSet(const PEGCHAR *pText)
```

PegMLTextButton overrides the `DataSet()` function to invalidate the button client area and recalculate the number of lines to display.

```
virtual void DataSet(PEGINT StringId)
```

This version of the `DataSet` function takes a string ID to retrieve the actual text from the string table.

```
virtual void Draw(const PegRect &Invalid)
```

PegMLTextButton overrides the `Draw()` function to draw the text on the button face.

```
virtual void SetFont(PEGINT FontIndex)
```

PegMLTextButton overrides the `SetFont()` function to invalidate the button client area and recalculate substring positions.

2.16.8 Examples:

The following is a default multi-line text button:

Created using the following code fragment:

```
PegRect Rect;
Rect.Set(10, 10, 49, 49);
Add(new PegMLTextButton(Rect, "Multi-Line|Text|Button"));
```

The following is a multi-line text button with a blank line, modified color, and modified font:

Created using the following code fragment:

```
Rect.Set(0, 0, 90, 72);
Rect.Shift(mClient.iLeft + 10, mClient.iTop + 10);
PegMLTextButton *pButton = new PegMLTextButton(Rect,
    "This Button|Has A||Blank Line");
pButton->SetColor(PCI_NORMAL, BLUE);
pButton->SetColor(PCI_NTEXT, WHITE);
pButton->SetFont(FID_SYSFONT);
Add(pButton);
```

2.17 PegProgressBar

2.17.1 Overview

PegProgressBar is a simple progress bar control used to indicate to an end user the completion status of a slow activity. PegProgressBar can assume any scale value within the range of the `PEGINT` data type; however, it is most common to display a value that is a percentage of the completion status.

PegProgressBar can assume any color, controlled by calling the `SetColor()` function. For this control, `PCI_NORMAL` is the color index of the progress bar background, and `PCI_SELECTED` is the color index of the progress bar foreground. `PCI_NTEXT` is the color index of the optional text value displayed in the center of the progress bar.

The style, range, and initial value of a PegProgressBar object are passed to the object constructor. As the operation being monitored progresses, the application software calls the `Update()` member function to change the displayed completion value.

The progress bar control has two main styles. The most common style is a solid indicator, in which case the progress bar internally draws a PegButton within the specified frame style. The `PS_LED` style, on the other hand, does not use a client area button to indicate progress; instead, it invokes a custom drawing style meant to appear as a series of LEDs, with the lighted LEDs indicating the current progress.

While you can create any number of PegProgessBar instances directly, it is more common to use the PegProgressWindow class, as this is a simpler method of creating and displaying a progress indicator to the end user.

2.17.2 See Also

PegProgressWindow

2.17.3 Style Flags

PegProgressBar supports the following styles:

PS_SHOW_VAL	This style instructs the progress bar to display the current progress value in text form, in addition to the graphical presentation.
PS_RECESSED	This style draws a solid, recessed progress indicator. The default appearance is a solid raised indicator.
PS_LED	This style draws a segmented LED indicator. The default appearance is a solid raised indicator.
PS_VERTICAL	This style orients the progress control graphical indicator vertically. The default is horizontal orientation.
PS_PERCENT	This style instructs the progress bar to add the '%' indicator to the displayed text value. This flag has no effect if the PS_SHOW_VAL style is not active.

2.17.4 Signals

PegProgressBar is a passive object, is not user selectable, and sends no signals.

2.17.5 Derivation

PegProgressBar is derived from PegThing.

2.17.6 Constructors:

```
PegProgressBar(const PegRect &Rect, PEGULONG Style =
     FF_THIN|PS_SHOW_VAL|PS_PERCENT, PEGINT Min = 0,
     PEGINT Max = 100, PEGINT Current = 0)
```

This constructor creates a PegProgressBar. The default values construct a progress bar that displays both text and a graphical value. The graphical indicator has a raised border. The progress bar has a range of 0 to 100, and displays a '%' sign after the output value.

2.17.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegProgressBar overrides the `Draw()` function to check for and draw the LED bar style.

```
void Reconfig(PEGULONG Style, PEGINT Min, PEGINT Max)
```

This function reconfigures the progress bar using the new style, minimum and maximum values.

```
virtual void Resize(PegRect Size)
```

PegProgressBar overrides the `PegThing::Resize` function in order to properly resize its internal members.

```
void Update(PEGINT Val)
```

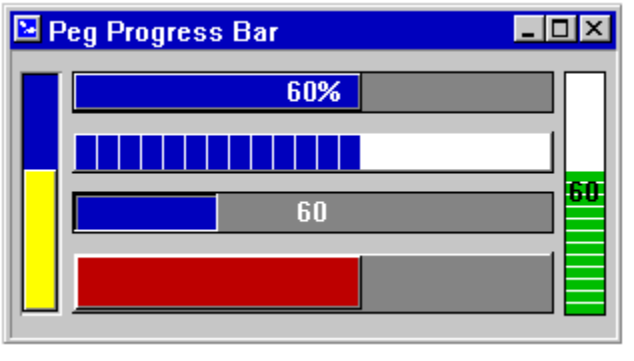
This function is called to update the progress bar completion value. The progress bar will automatically redraw to reflect the new value.

```
PEGINT Value(void)
```

This function returns the current progress bar value.

2.17.8 Examples:

The following is a PegWindow containing several styles of PegProgressBar:



The source code used to create the above example can be found in the file `\peg\examples\pegdemo\pegdemo.cpp`. The window class is named ProgBarWindow.

2.18 PegPrompt

2.18.1 Overview

PegPrompt is a text display object. PegPrompt can be drawn with several different border styles, and can be updated dynamically for interactive updates or real-time information display. PegPrompt does not support user editing.

By default, PegPrompt will send `PSF_CLICKED` signals to its parent object if the prompt ID is non-zero. By default, PegPrompt objects cannot be selected and do not send signals.

PegPrompt can also be used to populate PegComboBox and PegList objects.

The font used by PegPrompt can be changed at any time by using the `SetFont()` function, which is a PegTextThing member function. Likewise, the color used by PegPrompt can be set at any time by calling the `SetColor()` function.

2.18.2 See Also

PegEditField

2.18.3 Style Flags

PegPrompt supports the following styles:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame
`FF_RAISED`	Raised 3D Frame
`FF_RECESSED`	Recessed 3D Frame
`TJ_RIGHT`	Right justified text
`TJ_LEFT`	Left justified text
`TJ_CENTER`	Centered text

TT_COPY	Instructs the prompt to copy the string assigned. This flag should be used when the string assigned to the prompt is created dynamically using temporary storage.
AF_TRANSPARENT	Does not fill client area, assumes same color as parent.
AF_ENABLED	Prompt can be selected. The prompt will also send a PSF_FOCUS_RECEIVED signal to its parent.

2.18.4 Signals

In addition to the common signals defined by PegThing, PegPrompt supports the PSF_CLICKED and PSF_FOCUS_RECEIVED signal notifications.

2.18.5 Derivation

PegPrompt is derived from PegTextThing.

2.18.6 Constructors:

```
PegPrompt(const PegRect &Rect, PEGUINT StringId = 0,
     PEGUSHORT Id = 0, PEGULONG Style = FF_NONE|
     TJ_LEFT|AF_TRANSPARENT)

PegPrompt(const PEGCHAR *pText, const PegRect &Rect,
     PEGUSHORT Id = 0, PEGULONG Style = TT_COPY|
     FF_NONE|TJ_LEFT|AF_TRANSPARENT)

PegPrompt(const PegPoint &Put, PEGUINT StringId = 0,
     PEGUSHORT Id = 0, PEGULONG Style = FF_NONE|
     TJ_LEFT|AF_TRANSPARENT, PEGUINT FontId = 0)

PegPrompt(const PEGCHAR *pText, const PegPoint &Put,
     PEGUSHORT Id = 0, PEGULONG Style = TT_COPY|
     FF_NONE|TJ_LEFT|AF_TRANSPARENT, PEGUINT FontId =
     0)
```

There are four PegPrompt constructors. The first two allow you to fully specify the PegPrompt position and size. The third and fourth automatically determine the prompt height based on the font passed to the constructor or the default font prompt if no font is passed. The first and third constructors are used for string IDs. The second and fourth constructors are used with string pointers, for when a dynamic string is created at run time.

2.18.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

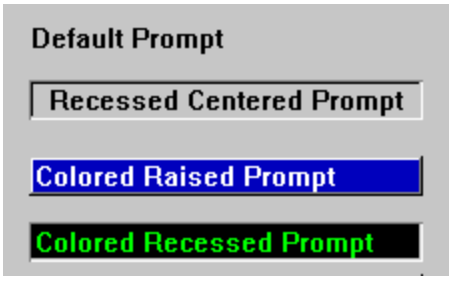
PegPrompt overrides the `Draw()` function to display the prompt text.

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

PegPrompt catches `PM_LBUTTONUP` messages.

2.18.8 Examples:

The following are each different styles of PegPrompt:



The following function creates a PegPrompt object with a custom font:

```
void MyWindow::AddCustomPrompt(void)
{
    PegRect ChildRect;
    ChildRect.Set(0, 0, 100, 40);
    PegPrompt *pPrompt = new PegPrompt(ChildRect, "FooBar");
    pPrompt->SetFont(FID_CUSTOM_FONT);
    Add(pPrompt);
}
```

2.19 PegRadioButton

2.19.1 Overview

PegRadioButton provides a mutually exclusive selection indicator. When a PegRadioButton is selected by the user, it finds all sibling radio buttons and de-selects them.

In order to allow multiple radio buttons to be selected on a single window or dialog, you must group the buttons into separate containers. PegGroup is commonly used for this purpose, although you can also use a transparent PegThing for this purpose just as well. Radio button objects that have the same parent are mutually exclusive, so by grouping the radio buttons onto different parent objects you can allow the selection of many different radio buttons at the same time.

The PegRadioButton uses four bitmaps, which can be indexed with the following enumerations:

`PBMI_RADIO_ON`	Selected radio button state
`PBMI_RADIO_OFF`	Unselected radio button state
`PBMI_RADIO_ON_DIS`	Selected and disabled radio button state
`PBMI_RADIO_OFF_DIS`	Unselected and disabled radio button state

2.19.2 See Also

PegIconButton

PegBitmapButton

PegTextButton

PegGroup

PegCheckBox

2.19.3 Style Flags

PegRadioButton supports the following styles:

AF_ENABLED	When this style is TRUE, the radio button is active and can be selected by the user. Otherwise, the button is not active and is drawn with a slightly different appearance.
BF_PUSHED	When this style is TRUE, the radio button will be drawn selected when first displayed. Only one radio button with a common parent should include this style.

2.19.4 Signals

PegRadioButton sends PSF_DOT_ON and PSF_DOT_OFF signals.

2.19.5 Derivation

PegRadioButton is derived from PegTextThing.

2.19.6 Constructors:

```
PegRadioButton(const PegRect &Rect, PEGUINT StringId =
     0, PEGUSHORT Id = 0, PEGULONG Style = AF_ENABLED|
     FF_NONE|BF_EXCLUSIVE)

PegRadioButton(const PEGCHAR *pText, const PegRect
     &Rect, PEGUSHORT Id = 0, PEGULONG Style =
     TT_COPY|AF_ENABLED|FF_NONE|BF_EXCLUSIVE)
```

The first constructor is used when the string ID for the text is known. The second constructor is used when the string is created at run time so that only a pointer is available.

2.19.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegRadioButton overrides the Draw() function to draw the radio button indicator bitmap and radio button text.

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

PegRadioButton catches the PM_LBUTTONDOWN message to toggle the radio button selection state.

```
virtual void SetBitmap(PEGUINT Index, PEGUINT
     BitmapId)
```

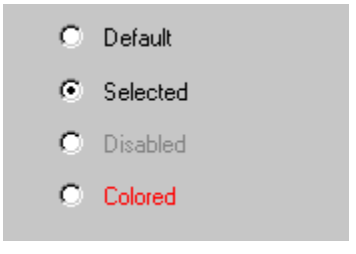
This function is used to assign a bitmap to the button using the specified `Index` and `BitmapId`.

```
virtual void SetSelected(PEGBOOL Selected = TRUE)
```

This function can be called to select or de-select a radio button at any time.

2.19.8 Examples:

The following are each different styles of PegRadioButton:



The following function creates a PegGroup and adds several radio button children to the group. The radio button 'Button1' will initially be selected:

```
void MyWindow::AddSelectGroup(void)
{
    PegRect ChildRect;
    ChildRect.Set(20, 20, 120, 100);
    PegGroup *pGroup = new PegGroup(ChildRect, "Select One");
    ChildRect.Set(30, 30, 110, 48);
    pGroup->Add(new PegRadioButton("Button1", ChildRect,
        IDR_BUTTON1,
        AF_ENABLED|FF_NONE|BF_EXCLUSIVE|BF_PUSHED))
        ;
    ChildRect.Shift(0, 20);
    pGroup->Add(new PegRadioButton("Button2", ChildRect,
        IDR_BUTTON2));
    ChildRect.Shift(0, 20);
    pGroup->Add(new PegRadioButton("Button3", ChildRect,
        IDR_BUTTON3));
    Add(pGroup);
}
```

2.20 PegScroll

2.20.1 Overview

PegScroll is an abstract base scroll bar class. It is used as the base class for both the default PegVScroll and PegHScroll scroll bar classes. Users can implement their own scroll bar classes by deriving from the PegScroll base.

This class is mainly responsible for setting and checking the PegScrollInfo member. Any derived classes must decide what to do with that information (i.e. how to draw the scroll buttons, etc.). PegVScroll and PegHScroll are the default derivatives used in PEG.

Users may provide their own custom scroll bar appearance by deriving their own scroll bar gadgets from the PegScroll base class. In order to use these custom bars, the user must also override the parent window CreateVScroll and/or CreateHScroll virtual member functions to instantiate instances of the custom scroll bar class type.

The PegScroll class uses six bitmaps, which can be indexed with the following enumerations:

`PBMI_SCROLL_FILL`	The background of the middle section of the scrollbar
`PBMI_SCROLL_UP`	The Up or Left arrow button bitmap
`PBMI_SCROLL_DN`	The Down or Right arrow button bitmap
`PBMI_THUMB_FILL`	The middle section bitmap of the thumb button
`PBMI_THUMB_UP`	The upper or left-most section bitmap of the thumb button
`PBMI_THUMB_DN`	The lower or right-most section bitmap of the thumb button

2.20.2 See Also

PegScrollInfo

PegHScroll

PegVScroll

PegSlider

How Scrolling Works

2.20.3 Style Flags

There are no styles associated with PegScroll Objects.

2.20.4 Signals

There are no signals associated with PegScroll Objects.

2.20.5 Derivation

PegScroll is derived from PegThing.

2.20.6 Constructors:

```
PegScroll(PegScrollDrawInfo *pDrawInfo = NULL, PEGBOOL
      Vertical = TRUE)

PegScroll(const PegRect &InRect, PegScrollInfo *pSi,
      PEGUINT Id = 0, PegScrollDrawInfo *pDrawInfo =
      NULL, PEGBOOL Vertical = TRUE)
```

The first constructor creates a nonclient area scroll bar. The scroll bar will automatically determine its position and size itself to the height of the parent window.

The second constructor creates a client area scroll bar. In this mode, a pointer to a PegScrollInfo structure is passed to set up the initial scrolling range. In this mode, the scroll bar position and size are passed to the constructor along with the scroll bar ID, if any.

2.20.7 Public Functions:

```
PEGUINT GetBitmap(PEGUINT Index)
```

This function returns the ID of the bitmap with the specified `Index`.

```
PEGINT GetMinOffset(void)
```

This inline function returns the minimum offset value. This is used to determine the minimum location of the elevator button.

```
PEGINT GetMaxOffset(void)
```

This inline function returns the maximum offset value. This is used to determine the maximum location of the elevator button.

```
PegScrollInfo *GetScrollInfo()
```

This inline function can be called to retrieve the current scroll bar information.

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

PegScroll overrides the `Message()` function to catch the `PM_SHOW` message.

```
void Reset()
```

```
virtual void Reset(PegScrollInfo *pSi)
```

These functions are provided so that the scroll bar position can be recalculated or reset at any time. The first form is used with nonclient scroll bars, and the second form is used with client-area scroll bars that are under program control.

```
void SetBitmap(PEGUINT Index, PEGUINT BmpId)
```

This function is used to assign a bitmap to the scroll bar using the specified `Index` and `BmpId`.

```
void SetDrawInfo(PegScrollDrawInfo *pInfo, PEGBOOL
     Size = TRUE)
```

This function assigns a PegScrollDrawInfo structure to the scroll bar. If `Size` is TRUE, then the scroll bar is resized to the width and height specified in `pInfo`.

```
void SetOffsets(PEGINT Min, PEGINT Max)
```

This function is used to set the minimum and maximum offsets of where the elevator button can be positioned.

2.20.8 Protected Members

```
virtual void CheckScrollLimits(void)
```

This function ensures that it has not exceeded its scrolling boundaries.

```
virtual void CreateButtons(void)
```

This function is intended to be overridden in derived classes so that any necessary buttons can be created. For example, the derived class PegVScroll uses it to create the up, down, and elevator buttons.

```
virtual void ReadParentScrollInfo(void)
```

This pure virtual function must be implemented in derived classes. This function is called by non-client scroll bars when the scroll bar is first shown to discover the parent window's scrolling parameters. Derived implementations should call either the parent window's `GetVScrollInfo()` or `GetHScrollInfo()` function, depending on the desired orientation of the derived class.

```
virtual void SetThumbButtonPos(void)
```

This pure virtual function must be implemented in derived classes. This function is called when the scroll bar is first shown, or when resized, or when the scroll information is changed. This function should determine the the position and size of the scroll slider button (if any). Custom implementations which do not include a slider button can implement this function as an empty function.

2.21 PegScrollPrompt

2.21.1 Overview

PegScrollPrompt is a PegPrompt-derived object that supports scrolling the text as well as assigning bitmaps to the left side, right side, and background.

PegScrollPrompt can also be used to populate PegComboBox and PegList objects.

The font used by PegScrollPrompt can be changed at any time by using the `SetFont()` function, which is a PegTextThing member function. Likewise, the color used by PegScrollPrompt can be set at any time by calling the `SetColor()` function.

PegScrollPrompt contains four bitmaps that are referenced using the following enumerations:

`PSPM_NORMAL`	Normal background bitmap
`PSPM_FOCUS`	Focused background bitmap
`PSPM_LEFT`	Left icon
`PSPM_RIGHT`	Right icon

2.21.2 See Also

PegEditField

2.21.3 Style Flags

PegPrompt supports the following styles:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame
`FF_RAISED`	Raised 3D Frame
`FF_RECESSED`	Recessed 3D Frame
`TJ_RIGHT`	Right justified text
`TJ_LEFT`	Left justified text
`TJ_CENTER`	Centered text

TT_COPY	Instructs the prompt to copy the string assigned. This flag should be used when the string assigned to the prompt is created dynamically using temporary storage.
AF_TRANSPARENT	Does not fill client area, assumes same color as parent.
AF_ENABLED	Prompt can be selected. The prompt will also send a PSF_FOCUS_RECEIVED signal to its parent.
SP_ONFOCUS	The text will scroll whenever the prompt has focus.
SP_ALWAYS	The text will always be scrolling.
SP_CONTINUOUS	The text scrolling will repeat continuously. If this is turned off, the text will stop scrolling after the first time through.
SP_DOTDOT	The prompt will append the string '...' to the text.
SP_WRAP	The text will wrap, meaning that when the scrolling reaches the end, it continues scrolling at the beginning again.

2.21.4 Signals

In addition to the common signals defined by PegThing, PegScrollPrompt supports the PSF_CLICKED, PSF_FOCUS_RECEIVED and PSF_SCROLL_COMPLETE signal notifications.

2.21.5 Derivation

PegScrollPrompt is derived from PegPrompt.

2.21.6 Constructors:

```
PegScrollPrompt(const PegRect &Rect, PEGINT StringId =
     0, PEGUSHORT Id = 0, PEGULONG Style = FF_NONE|
     TJ_LEFT|AF_TRANSPARENT|SP_ALWAYS)

PegScrollPrompt(const PEGCHAR *pText, const PegRect
     &Rect, PEGUSHORT Id = 0, PEGULONG Style =
     FF_NONE|TJ_LEFT|AF_TRANSPARENT|SP_ALWAYS)
```

There are two PegScrollPrompt constructors. The first constructor is used for string IDs. The second constructor is used with string pointers, for when a dynamic string is created at run time.

2.21.7 Public Functions:

```
virtual void DataSet(PEGINT StringId)

virtual void DataSet(const PEGCHAR *pText)
```

PegScrollPrompt overrides the `DataSet()` function because it needs to draw the new text into its scrolling bitmap.

```
virtual void Draw(const PegRect &Invalid)
```

PegScrollPrompt overrides the `Draw()` function to display the prompt text and bitmaps.

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

PegScrollPrompt catches timer and focus messages.

```
void SetBitmaps(PEGUINT NormalMap, PEGUINT
     SelectedMap, PEGUINT LeftMap, PEGUINT RightMap)
```

This function assigns bitmaps to background and the left and right sides. If the prompt gains focus, it will attempt to use `pSelectedMap` as the background. Otherwise, it will use `pNormalMap`. Any of these parameters may be 0.

```
virtual void SetFont(PEGINT FontId)
```

PegScrollPrompt overrides the `SetFont()` function because it needs to redraw the text into its scrolling bitmap.

```
void SetLeftBitmap(PEGUINT LeftMap)
```

This function assigns a bitmap to be displayed on the left side of the text.

```
void SetNormalBitmap(PEGUINT NormalMap)
```

This function assigns a bitmap to be displayed in the background of the text when the prompt does not have focus.

```
void SetRightBitmap(PEGUINT RightMap)
```

This function assigns a bitmap to be displayed on the right side of the text.

```
void SetScrollSpeed(PEGINT Amount, PEGINT Timeout)
```

This function determines how many pixels the text will scroll at a time and how fast it will make those increments.

```
void SetSelectedBitmap(PEGUINT SelectedMap)
```

This function assigns a bitmap to be displayed in the background of the text when the prompt has focus.

2.22 PegSlider

2.22.1 Overview

PegSlider is an analog adjustment control. The end user adjusts the slider value by dragging the slider 'handle.' PegSlider can be positioned horizontally or vertically. The orientation is determined by the style flags.

PegSlider draws tickmarks at specified intervals along the slider range, unless the specified interval < 1. The slider handle and tickmarks are automatically drawn proportional to the overall slider size.

PegSlider sends `PSF_SLIDER_CHANGE` notification signals to the slider parent when the user adjusts the slider value.

2.22.2 See Also

PegHScroll

PegVScroll

2.22.3 Style Flags

PegSlider defines the following styles:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame
`FF_RAISED`	Raised 3D Frame
`FF_RECESSED`	Recessed 3D Frame
`SF_SNAP`	Snaps the slider handle to the exact tick positions.
`SS_ORIENTVERT`	Determines orientation. If this is used, the slider is vertically oriented. Otherwise, it is horizontal.

SS_FACELEFT	Determines the horizontal position of the slider button when SS_ORIENTVERT is also used. If this is used, the button is placed on the right side of the slider so that it can 'face left.' Otherwise, it is placed on the left side so that it faces right.
SS_FACETOP	Determines the vertical position of the slider button when SS_ORIENTVERT is not used. If this is used, the button is placed on the bottom edge of the slider so that it can 'face top.' Otherwise, it will be placed on the top edge so that it faces the bottom.

2.22.4 Signals

PegSlider sends PSF_SLIDER_CHANGE signals to the slider parent when adjusted by the end user. The message contains the following values:

Message.pSource = Pointer to slider control.
Message.Param = ID of slider control.
Message.ExtParams[0] = Current slider value.

2.22.5 Derivation

PegSlider is derived from PegThing.

2.22.6 Constructors:

```
PegSlider(const PegRect &Rect, PEGLONG Min, PEGLONG
     Max, PEGUINT Id = 0, PEGULONG Style = FF_RAISED,
     PEGLONG Scale = -1)
```

The PegSlider constructor creates a PegSlider control at the position and size specified in Rect. The Min and Max values specify the initial limits of the slider. The slider ID, if non-zero, enables the PSF_SLIDER_CHANGE notification signal. The slider scale determines the interval between slider tickmarks.

2.22.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegSlider overrides the Draw() function to draw the slider control.

```
virtual PEGINT GetCurrentValue(void)
```

This inline function returns the current slider value.

```
virtual PEGINT GetMaxValue(void)
```

This inline function returns the slider maximum limit value.

```
virtual PEGINT GetMinValue(void)
```

This inline function returns the slider minimum limit value.

```
virtual PEGINT GetNeedleOffset(void) const
```

This inline function returns the offset of the slide button where it should actually be pointing at the current value.

```
virtual PEGINT GetScale(void)
```

This inline function returns the slider scale interval.

```
virtual PEGLONG IncrementValue(PEGLONG Val, PEGBOOL
      Redraw = TRUE)
```

This function will increment the current value by the amount specified by `Val`. A negative number can be passed to allow the value to decrement. The final value is compared to the minimum and maximum values so that it does not exceed the allowed range.

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

PegSlider catches the `PM_LBUTTONDOWN` to capture the pointer and move the slider button as it receives `PM_POINTER_MOVE` messages.

```
virtual void Reset(PEGINT Min, PEGINT Max, PEGINT New)
```

This function can be called to reset the slider limits and current slider value.

```
virtual void Resize(const PegRect &NewSize)
```

This function overrides the `PegThing::Resize` function. The position of the slider button is adjusted as it is resized.

```
virtual void SetCurrentValue(PEGINT NewVal, PEGBOOL
      Redraw = FALSE)
```

This function can be called to reset the current slider value. The slider will automatically redraw to reflect the new value if the `Redraw` value is TRUE.

```
void SetMaxTravelOffset(PEGINT Offset)
```

This function is used to set the maximum position that the slide button can travel within the slider.

```
virtual void SetMaxValue(PEGINT Max, PEGBOOL Redraw =
     TRUE)
```

This function can be called to reset the slider maximum limit value.

```
void SetMinTravelOffset(PEGINT Offset)
```

This function is used to set the minimum position that the slide button can travel within the slider.

```
virtual void SetMinValue(PEGINT iMin)
```

This function can be called to reset the slider minimum limit value.

```
void SetNeedleOffset(PEGINT Offset)
```

This function is used to set the offset within the slide button that points to the current value.

```
virtual void SetScale(PEGINT Scale)
```

This function can be called to reset the slider tickmark interval.

2.22.8 Protected Members

```
virtual void CheckSlideButton(void)
```

This function checks to see if the slide button has been created yet, and if not it creates it. This can be overridden in derived classes so that a derived version of the PegSlideButton class can be used.

```
PEGINT mMax
```

This is the maximum value for the slider.

```
PEGINT mMin
```

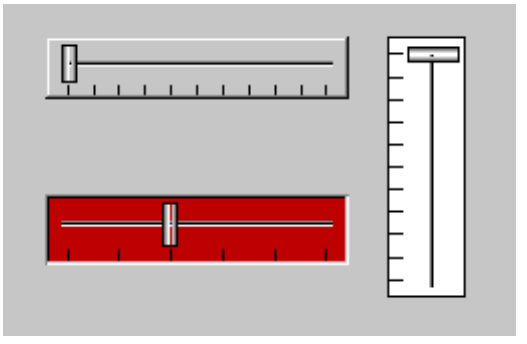
This is the minimum value for the slider.

```
PEGINT mScale
```

This is the slider tickmark interval.

2.22.9 Examples:

The following are each different styles of PegSlider:



The following function creates a PegSlider. The slider will be vertical, and will have a minimum value of 0, a maximum value of 200, and a tick mark interval of 20 (i.e. 10 tickmarks will be drawn). This initial slider value will be 50.

```
void MyWindow::AddSlider(void)
{
    PegRect SliderRect;
    SliderRect.Set(20, 20, 60, 120);
    PegSlider *pSlider = new PegSlider(SliderRect, 0, 200,
        ID_SLIDER, FF_RAISED|SS_ORIENTVERT, 20);
    pSlider->SetCurrentValue(50);
    Add(pSlider);
}
```

2.23 PegSpinButton

2.23.1 Overview

PegSpinButton is a thumbwheel style control that is normally used to adjust a numeric value displayed in an adjacent object. PegSpinButton objects can be horizontal or vertical in orientation.

There are two forms of PegSpinButton. The first form is created when the spin button has a 'buddy' object. A buddy object is a PegTextThing-derived object that is automatically updated as the spin button is manipulated by the end user. The second form of PegSpinButton has no buddy object, and therefore reports spin button selection to the parent window for application-level processing.

When a spin button has a buddy object, that object should be designed to display a numeric value. When the spin button is operated by the end user, the spin button first converts the buddy object string to an integer, then increments or decrements the integer value as required. It then converts the integer value back to a string for assignment to the buddy object.

The buddy object, if any, is required to have `TT_COPY` style. This is required because the string value assigned to the buddy object is dynamically constructed. If the buddy object does not have `TT_COPY` style, this style is added automatically by the spin button object.

2.23.2 See Also

PegSlider

PegTextThing

2.23.3 Style Flags

PegSpinButton supports the following styles:

`SB_VERTICAL`	Creates a vertical spin button. The default is to create a horizontally-oriented spin button.

2.23.4 Signals

If a PegSpinButton has a non-zero ID value, it sends `PSF_SPIN_MORE` and `PSF_SPIN_LESS` signals to the parent window as the spin button is selected by the end user.

`PSF_SPIN_MORE` and `PSF_SPIN_LESS` signal messages contain the following data:

`Message.pSource` = Pointer to spin button control.
`Message.Param` = ID of spin button control.

2.23.5 Derivation

PegSpinButton is derived from PegThing.

2.23.6 Constructors:

```
PegSpinButton(const PegRect &Rect, PEGUINT Id = 0,
     PEGULONG Style = AF_ENABLED|SB_VERTICAL)
```

This constructor is used to create a PegSpinButton that has no buddy object. This spin button will send notification signals to the parent object as the spin button is operated.

```
PegSpinButton(const PegRect &Rect, PegTextThing
     *pBuddy, PEGLONG Min, PEGLONG Max, PEGINT Step,
     PEGUINT Id = 0, PEGULONG Style = AF_ENABLED|
     SB_VERTICAL)
```

This constructor is used to create a PegSpinButton that has a buddy object. This spin button directly updates the buddy object, in addition to sending `PSF_SPIN_MORE` and `PSF_SPIN_LESS` signals to the button parent.

2.23.7 Public Functions:

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

PegSpinButton catches messages from the spin button directional selection arrows to update the buddy object or send signals to the spin button parent.

```
void SetBuddy(PegTextThing *pBuddy)
```

This inline function can be used to reset the buddy object pointer of the spin button.

```
void SetLimits(PEGLONG Min, PEGLONG Max, PEGINT Step =
     -1)
```

This function can be used to reset the spin button limits and step value. A positive step value causes the spin button to increase the value of a buddy object when the 'Up' or 'Right' spin buttons are pressed. A negative step value reverses the direction of increment/decrement.

```
void SetOutputWidth(PEGINT Width)
```

This function can be used to specify a fixed-width output format for spin buttons with a buddy object. The `Width` parameter indicates how many digits the output value should contain. The output string is left-padded with zeros if required to fill the indicated width.

2.23.8 Protected Members

```
PEGINT mStep
```

This is the amount to increment or decrement the value when a button is pressed.

```
PEGLONG mMax
```

This is the maximum value for the spinner.

```
PEGLONG mMin
```

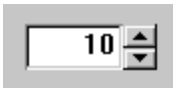
This is the minimum value for the spinner.

```
PegTextThing *mpSlave
```

This is a pointer to the text object that will display the current value.

2.23.9 Examples:

The following illustrates a vertical PegSpinButton with a PegPrompt buddy object:



The following function creates a PegPrompt for displaying a numeric value. The prompt will be updated by a vertical PegSpinButton object. The range of values will be 20 through 80, and the value will increment/decrement by 5 each time the spin button is operated.

```
void MyWindow::AddSpinPrompt(void)
{
    PegRect ChildRect;
    ChildRect.Set(20, 20, 100, 40);
    PegPrompt *pPrompt = new PegPrompt(ChildRect, "20", 0,
        FF_RECESSED|TJ_RIGHT|TT_COPY);
     Add(pPrompt);

    // set the spin button position to the right of the
prompt:

    ChildRect.Left = pPrompt->mReal.Right + 1;
    ChildRect.Right = ChildRect.Left + PEG_SCROLL_WIDTH;
    PegSpinButton *pSpin = new PegSpinButton(ChildRect,
        pPrompt, 20, 80, 5, 0, SB_VERTICAL);
    Add(pSpin);
}
```

2.24 PegStatusBar

2.24.1 Overview

PegStatusBar is a window decoration that automatically sizes and positions itself at the bottom of the client area of its parent. PegStatusBar may have any number of children. Since the PegPrompt object is the most common type of child object added to a status bar, the PegStatusBar class includes functions for easily displaying and accessing any number of PegPrompt objects.

The last object added to a PegStatusBar is sized to extend to the rightmost edge of the status bar.

A pointer to a status bar added to a PegDecoratedWindow can be obtained at any time by calling the `PegDecoratedWindow::StatusBar()` function.

2.24.2 See Also

PegDecoratedWindow

PegPrompt

2.24.3 Style Flags

None.

2.24.4 Signals

PegStatusBar does not send signals. However, children of the status bar will send the normal signals supported by the child object types. PegStatusBar will pass any signal received on to the parent window.

2.24.5 Derivation

PegStatusBar is derived from PegThing.

2.24.6 Constructors:

```
PegStatusBar(void)
```

This constructor creates a PegStatusBar. The status bar is normally added to a PegDecoratedWindow or PegDialog.

2.24.7 Public Functions:

```
virtual PegPrompt *AddTextField(PEGINT Width,
      PEGUSHORT Id, PEGUINT StringId = 0)

virtual PegPrompt *AddTextField(const PEGCHAR *pText,
      PEGINT Width, PEGUSHORT Id)
```

This function can be called to add a PegPrompt field to the status bar. Parameter `Width` indicates the desired field width, in pixels. `Id` is the ID that will be assigned to the PegPrompt, and `StringId` (or `pText`) is the initial prompt value.

PegPrompt objects added to a PegStatusBar can be updated at any time. They are located by calling `GetPrompt()` with the ID of the prompt in question.

```
virtual void Draw(const PegRect &Invalid)
```

PegStatusBar overrides the `Draw()` function to draw the status bar frame.

```
PegPrompt *GetPrompt(PEGUINT wId)
```

This function returns a pointer to the PegPrompt child of the status bar with the given ID value.

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

PegStatusBar catches the `PM_SHOW` and `PM_PARENTSIZED` messages to size itself and position the children of the status bar.

```
virtual void SetFont(PEGINT FontIndex)
```

PegStatusBar overrides the `SetFont` method so that it can resize itself and all of its children based on the size of the new font. It also alerts its parent window that it has resized.

```
virtual void SetTextField(PEGUSHORT Id, PEGINT
     StringId)

virtual void SetTextField(PEGUSHORT Id, const PEGCHAR
     *pText)
```

This function provides a shorthand method for updating a prompt that has been added to the status bar. The `Id` parameter indicates the desired prompt ID, and the `StringId` or `pText` value is the new text value to assign to the prompt.

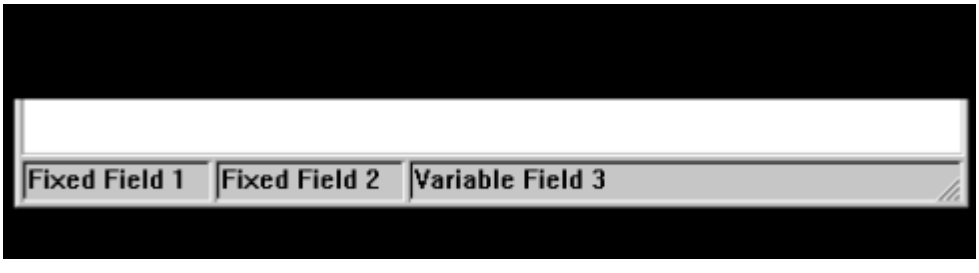
2.24.8 Protected Members

```
virtual void SizeToParent(void)
```

This function resizes the status bar based on the width of the parent window.

2.24.9 Examples:

The following is a PegStatusBar with several text fields, added to a PegDecoratedWindow:



The following function creates a PegStatusBar with three text fields, and adds the status bar to a parent window.

```
void MyWindow::AddStatusBar(void)
{
    PegStatusBar *pStat = new PegStatusBar();
    pStat->AddTextField(100, ID_FIELD1, "Fixed Field 1");
    pStat->AddTextField(100, ID_FIELD2, "Fixed Field 2");
    pStat->AddTextField(20,  ID_FIELD3, "Variable Field 3");

    Add(pStat);
}
```

Any field in the status bar created above can be updated using the following code sequence:

```
void MyWindow::UpdateSecondStatusField(PEGCHAR *pNewVal)
{
    StatusBar()->SetTextField(ID_FIELD2, pNewVal);
}
```

2.25 PegTextButton

2.25.1 Overview

PegTextButton provides a pushbutton object with visual feedback indicating to the user the button depress and release operation.

Any custom fonts or colors may be applied to PegTextButton. Users often create custom derived versions of PegTextButton to draw modified border styles, modified border shapes, etc. PegTextButton is also commonly used to populate PegHorzList and PegVertList objects.

While PegTextButton is most often a 'bottom level' object, it is possible to add children to a PegTextButton to further customize the appearance of your user interface.

The text string displayed on the button face is vertically centered over the button client area, and may be horizontally justified in different ways using the text justification style flags.

2.25.2 See Also

PegMLTextButton

PegIconButton

PegBitmapButton

PegRadioButton

PegCheckBox

2.25.3 Style Flags

PegTextButton supports the following styles:

`TJ_RIGHT`	Right justified text
`TJ_LEFT`	Left justified text
`TJ_CENTER`	Centered text

TT_COPY	Instructs the button to copy the string assigned. This flag should be used when the string assigned to the button is created dynamically using temporary storage.
BF_REPEAT	This flag causes the button to send periodic PSF_CLICKED signals when held down by the user.
BF_DOWNACTION	This flag causes the button to send the PSF_CLICKED signal on the down press of the button, rather than the default action which is to signal on the button release.
BF_TOGGLE	This flag causes the button to function much like a checkbox, in that when it is pressed it remains selected until it becomes pressed again. Any number of buttons with BF_TOGGLE style can be selected at the same time.
BF_EXCLUSIVE	This flag causes the button to function much like a radio button, in that when it is pressed, it remains selected until another sibling button is selected. It also unselects all sibling buttons when it becomes selected. Only one button within a single parent object can be selected at a time.
AF_ENABLED	This flag allows the button to be selected.

2.25.4 Signals

PegTextButton sends PSF_CLICKED signals when selected.

2.25.5 Derivation

PegTextButton is derived from PegButton and PegTextThing.

2.25.6 Constructors:

```
PegTextButton(const PegRect &Rect, PEGUINT StringId =
     0, PEGUSHORT Id = 0, PEGULONG Style = AF_ENABLED|
     FF_RAISED)

PegTextButton(const PEGCHAR *pText, const PegRect
     &Rect, PEGUSHORT Id = 0, PEGULONG Style =
     TT_COPYAF_ENABLED|FF_RAISED)
```

The first constructor is used to specify the string ID for the text. The second constructor is used when a string pointer is all that is available.

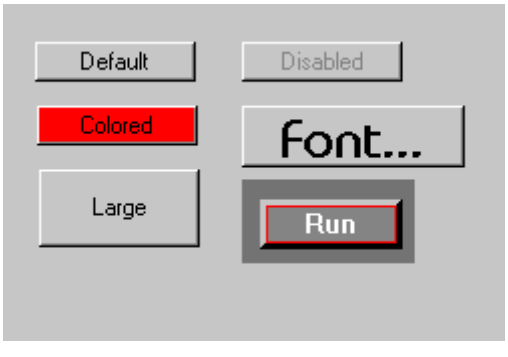
2.25.7 Public Functions:

```
virtual void Draw(const PegRect &Rect)
```

PegTextButton overrides the `Draw()` function to draw the text on the button face.

2.25.8 Examples:

The following are each different styles of PegTextButton:



The following is an example of overriding the `Draw()` function to create a custom button appearance. This example creates a button similar to the lower-right example above.

```
void CustomButton::Draw(const PegRect &Invalid)
{
    PegBrush Brush;
    BeginDraw(Invalid
    );
```

```
if (mStyle & BF_PUSHED)
{
    Brush.LineColor = CLR_BLACK;
}
else
{
    Brush.LineColor = CLR_LIGHTGRAY;
}

Brush.Width = 3;

// draw the top:
Line(mReal.Left, mReal.Top, mReal.Right, mReal.Top,
    Brush);

// draw the left:
Line(mReal.Left, mReal.Top, mReal.Left, mReal.Bottom,
    Brush);

if (mStyle & BF_PUSHED)
{
    Brush.LineColor = CLR_LIGHTGRAY;
}
else
{
    Brush.LineColor = CLR_BLACK;
}
// draw the right shadow:

Brush.Width = 1;

Line(mReal.Right, mReal.Top, mReal.Right,
    mReal.Bottom - 2, Brush);
Line(mReal.Right - 1, mReal.Top + 1, mReal.Right - 1,
    mRealiBottom - 2, Brush);
Line(mReal.Right - 2, mReal.Top + 2, mReal.Right - 2,
    mReal.Bottom - 2, Brush);

// draw the bottom shadow:

Line(mReal.Left, mReal.Bottom, mReal.Right,
    mReal.Bottom, Brush);
```

```
    Line(mReal.Left + 1, mReal.Bottom - 1, mReal.Right,
        mReal.Bottom - 1, Brush);
    Line(mReal.Left + 2, mReal.Bottom - 2, mReal.Right,
        mReal.Bottom - 1, Brush);

    // fill in the button client area:

    Brush.Set(CLR_LIGHTRED, CLR_DARKGRAY, PBS_SOLID_FILL, 1);
    Rectangle(mClient, Brush);

    // draw the text centered:

    PegPoint Put;
    Put.x = (mClient.Left + mClient.Right) >> 1;
    Put.x -= TextWidth(mpText, FID_SYSFONT) >> 1;
    Put.y = mClient.Top + 1;

    if (mStyle & BF_PUSHED)
    {
        Put.x++;
        Put.y++;
    }

    Brush.Set(CLR_WHITE, CLR_BLACK, 0, 0);
    DrawText(Put, mpText, Brush, FID_SYSFONT);
    EndDraw();
}
```

2.26 PegTitle

2.26.1 Overview

PegTitle is a window decoration that automatically sizes and positions itself at the top of the client area of its parent. PegTitle automatically adds various common control buttons to itself depending on the title style flags.

PegTitle also adds the functionality of dragging the parent window. PegTitle checks the parent window `PSF_MOVEABLE` system flag in order to provide this capability.

A pointer to the title added to a PegDecoratedWindow can be obtained at any time by calling the `PegDecoratedWindow::Title()` function.

2.26.2 See Also

PegDecoratedWindow

PegDialog

2.26.3 Style Flags

`TT_COPY`	Instructs the title to copy the string assigned. This flag should be used when the string assigned to the title is created dynamically using temporary storage.
`TF_SYSBUTTON`	This flag instructs the title to include a system button, located in the upper-left corner of the title bar. The system button activates the system menu associated with the title bar.
`TF_MINMAXBUTTON`	This flag instructs the title to include minimize and maximize buttons on the title bar.
`TF_CLOSEBUTTON`	This flag instructs the title to include a window close button on the title bar.

2.26.4 Signals

PegTitle does not send signals. However the system menu associated with the title bar will send signals to the parent window.

2.26.5 Derivation

PegTitle is derived from PegTextThing.

2.26.6 Constructors:

```
PegTitle(PEGINT StringId, PEGULONG Style =
     TF_SYSBUTTON|TF_MINMAXBUTTON|TF_CLOSEBUTTON)

PegTitle(const PEGCHAR *pText, PEGULONG Style =
     TT_COPY|TF_SYSBUTTON|TF_MINMAXBUTTON|
     TF_CLOSEBUTTON)
```

This constructor creates a PegTitle. The default style is to include all available title buttons. The second constructor adds the style TT_COPY as a default because if the text is created dynamically, it will typically need to make its own copy of it.

2.26.7 Public Functions:

```
void AssignMenu(PegMenuDescription *pDesc)
```

This function can be used to assign an alternate system menu to the PegTitle system button. The default system menu includes the commands “Close” (`PM_CLOSE`), “Maximize” (`PM_MAXIMIZE`), and “Minimize” (`PM_MINIMIZE`).

```
virtual void Draw(const PegRect &Invalid)
```

PegTitle overrides the `Draw()` function to draw the title background and title text.

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

PegTitle catches various mouse message, the title button messages, and focus alteration messages.

```
virtual void SetFont(PEGINT FontIndex)
```

PegTitle overrides the SetFont method so it can resize itself based on the size of the new font.

2.26.8 Protected Members

```
virtual void SizeToParent(void)
```

This function resizes the title bar based on the width of the parent window.

2.26.9 Examples:

The following is a PegTitle added to a PegDialog window:

The following function creates a PegTitle, assigns an alternate system menu to the title, and adds the title to the parent window.

```
static PegMenuDescription SystemMenu[] = {
    {"Exit", IDB_DEMO_EXIT, AF_ENABLED, NULL},
    {"Close", IDB_CLOSE, AF_ENABLED, NULL},
    {"Save", 0, 0, NULL},
    {"", 0, BF_SEPARATOR, NULL},
    {"Restore", IDB_RESTORE, AF_ENABLED, NULL},
    {"Minimize", IDB_MINIMIZE, AF_ENABLED, NULL},
    {"Maximize", IDB_MAXIMIZE, AF_ENABLED, NULL},
    {"", 0, 0, NULL}
};

void MyWindow::AddTitle(void)
{
    PegTitle *pTitle = new PegTitle("Hello World",
        TF_SYSBUTTON|TF_CLOSEBUTTON);
    pTitle->AssignMenu(SystemMenu);
    Add(pTitle);
}
```

2.27 PegToolBar

2.27.1 Overview

PegToolBar is a window decoration used to position and display a group of related user objects. These objects could be text or bitmap buttons, user input fields, or any other objects that represent frequently used commands or user data that does not lend itself well to a menu. PegToolBar is designed to work in conjunction with PegDecoratedWindow objects. PegToolBar may be added to any type of object. However, the client area of objects other than PegDecoratedWindow will not properly be reduced unless this is done in the application software.

PegToolBar automatically positions and sizes itself to the parent window. PegToolBar may contain any number of PegToolBarPanel objects. PegToolBar also automatically sizes to the tallest PegToolBarPanel as the panels are added. However, once the tool bar is added to a parent, it will no longer size itself in this manner. It is therefore recommended to create a PegToolBar object, add all of the PegToolBarPanels to the tool bar, then add the PegToolBar to its parent object. This will ensure that the tool bar will be the proper height. For an example of this, see the code at the bottom of this page. It is not recommended that any other objects aside from a PegToolBarPanel be added to a PegToolBar.

Any object that is placed on a PegToolBarPanel, and subsequently a PegToolBar, behaves the same way it would if it were added to any other PegThing derived object. For instance, a PegButton derived object sends its `PSF_CLICKED` message up to the PegToolBarPanel, which in turn posts it to the PegToolBar, which in turn posts the message to its parent, in this case, the PegDecoratedWindow. Therefore, the message is handled within the context of the parent PegDecoratedWindow. This model makes it very easy to handle messages from objects on the tool bar by handling them in the same message loop as all of your other command objects.

Also, for example, say you have a PegMenuButton (on the window's menu) and a PegIconButton (on the window's tool bar) that share a common ID. Whichever is selected by the end user, the same code will be executed. This makes it very easy to put frequently used commands on a tool bar, and also have them on the menu.

2.27.2 See Also

PegToolBarPanel

2.27.3 Style Flags

None.

2.27.4 Signals

None.

2.27.5 Derivation

PegToolBar is derived from PegThing.

2.27.6 Constructors:

```
PegToolBar(PEGUSHORT Id = 0)
```

The PegToolBar constructor creates a PegToolBar object. PegToolBar automatically determines its position and size.

2.27.7 Public Functions:

```
virtual void AddPanel(PegToolBarPanel *pPanel)
```

This method adds the newly created PegToolBarPanel to the PegToolBar. The panel is sized when it is created, and the PegToolBar positions the panel on the tool bar **after** any previously added panels. If the panel is already on the tool bar, this method simply does nothing and returns.

```
virtual void Draw(const PegRect &Invalid)
```

PegToolBar overrides the `Draw()` function to draw the tool bar background.

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

PegToolBar catches `PM_PARENTSIZED` and `PM_SHOW` messages.

```
virtual void PositionPanels(void)
```

This method is used internally by the PegToolBar to reposition its child panels. It is a public function so that the PegToolBarPanels can trigger all of the child panels to be repositioned.

```
virtual PegThing* RemovePanel(PegThing *pWho)
```

This method removes the panel that is pointed to by `pWho`. In doing so, it repositions the PegToolBarPanels that follow this particular panel (if there are any) to fill in the empty space on the PegToolBar left by removing the panel.

2.27.8 Protected Members

```
virtual void SizeToParent()
```

This function resizes the width of the PegToolBar based on the width of the parent window.

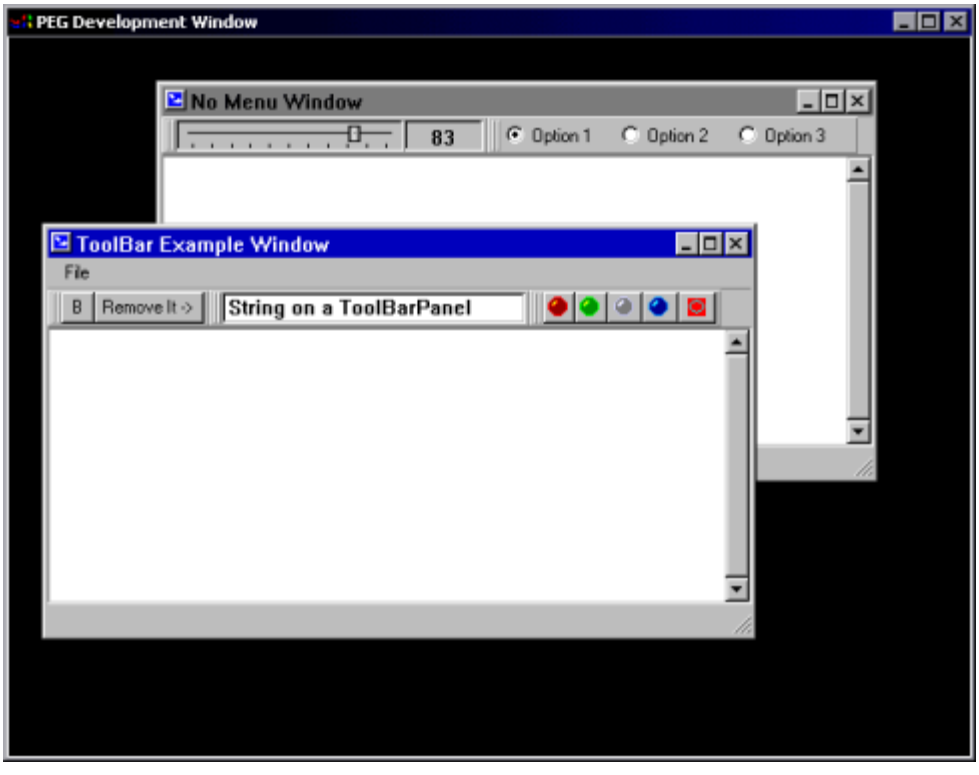
```
virtual void SizeToTallestPanel()
```

This function resizes the height of the PegToolBar based on the tallest panel contained inside it.

2.27.9 Examples:

The following is an example of two PegDecoratedWindows with PegToolBars and PegToolBarPanels. Note that there are three PegToolBarPanels on the top window and that there are two PegToolBarPanels on the second window. Note also that the PegThing-derived objects are added to the PegToolBarPanels, not to the PegToolBar itself.

PegToolBar example image:



The following example creates a PegToolBar and adds three PegToolBarPanels to the PegToolBar. This is the PegToolBar in the above top level window. Usually you would do this in the constructor of your PegDecoratedWindow derived window.

```
... some code deleted ...


    PegToolBar *pToolBar = new PegToolBar();

    PegRect Rect;
    PegToolBarPanel *pPanel = new PegToolBarPanel();
    Rect.Set(0, 0, 70, 20);
    pPanel->Add(new PegTextButton(Rect, "Remove It ->",
        IDB_ALPHA_BUTTON));
    Rect.Set(0, 0, 20, 20);
```

```
pPanel->AddToEnd(new PegTextButton(Rect, "B"));
pToolBar->AddPanel(pPanel);
Rect.Set(0, 0, 200, 20);
pPanel = new PegToolBarPanel(IDC_STRING_PANEL);
pPanel->Add(new PegEditField(Rect,
    "String on a ToolBarPanel"));
pToolBar->AddPanel(pPanel);

Rect.Set(0, 0, 19, 18);
pPanel = new PegToolBarPanel();
pPanel->Add(new PegIconButton(Rect, BID_BULLSEYE,
    IDB_BULL_BUTTON), FALSE);
pPanel->Add(new PegIconButton(Rect, BID_BLUE_DOT,
    IDB_BLUE_BUTTON), FALSE);
pPanel->Add(new PegIconButton(Rect, BID_GRAY_DOT,
    IDB_GREY_BUTTON), FALSE);
pPanel->Add(new PegIconButton(Rect, BID_GREEN_DOT,
    IDB_GREEN_BUTTON), FALSE);
pPanel->Add(new PegIconButton(Rect, BID_RED_DOT,
    IDB_RED_BUTTON), FALSE);
pToolBar->AddPanel(pPanel);
Add(pToolBar);
```

2.28 PegToolBarPanel

2.28.1 Overview

PegToolBarPanel is a container object that is used in conjunction with PegToolBar. Any PegThing-derived object may be placed on a PegToolBarPanel. As the objects are added, the panel positions the objects from left to right. The panel also resizes itself to the tallest child that is added to the panel. Objects that are smaller in height to the tallest object are placed at the top of the panel's client area.

2.28.2 See Also

PegToolBar

2.28.3 Style Flags

None.

2.28.4 Signals

None.

2.28.5 Derivation

PegToolBarPanel is derived from PegThing.

2.28.6 Constructors:

```
PegToolBarPanel(PEGUSHORT Id = 0)
```

The PegToolBarPanel constructor creates a PegToolBarPanel object. The size of the panel is determined by its child objects. Its position on the PegToolBar is determined when it is added to the PegToolBar.

2.28.7 Public Functions:

```
virtual void Add(PegThing* pWho, PEGBOOL Show = TRUE)
```

This is an overridden version of the `PegThing::Add` method. The PegToolBarPanel height will be adjusted, if necessary, to allow the

PegThing object to fit on the panel. It will also have to resize itself based on the width of the `pWho` object. It sends a message to its parent object (PegToolBar) to reposition any sibling panels based on its new size.

```
virtual void AddToEnd(PegThing* pWho, PEGBOOL Show =
     TRUE)
```

This method is identical to `Add`, but the object is added to the end of the child list.

```
virtual void Draw(const PegRect &Invalid)
```

PegToolBarPanel overrides the `Draw()` function to draw the panel background and its children.

```
virtual void Remove(PegThing* pWho)
```

This method is an override of the `PegThing::Remove` method. After passing the parameters to PegThing, we resize ourselves accordingly.

2.28.8 Protected Members

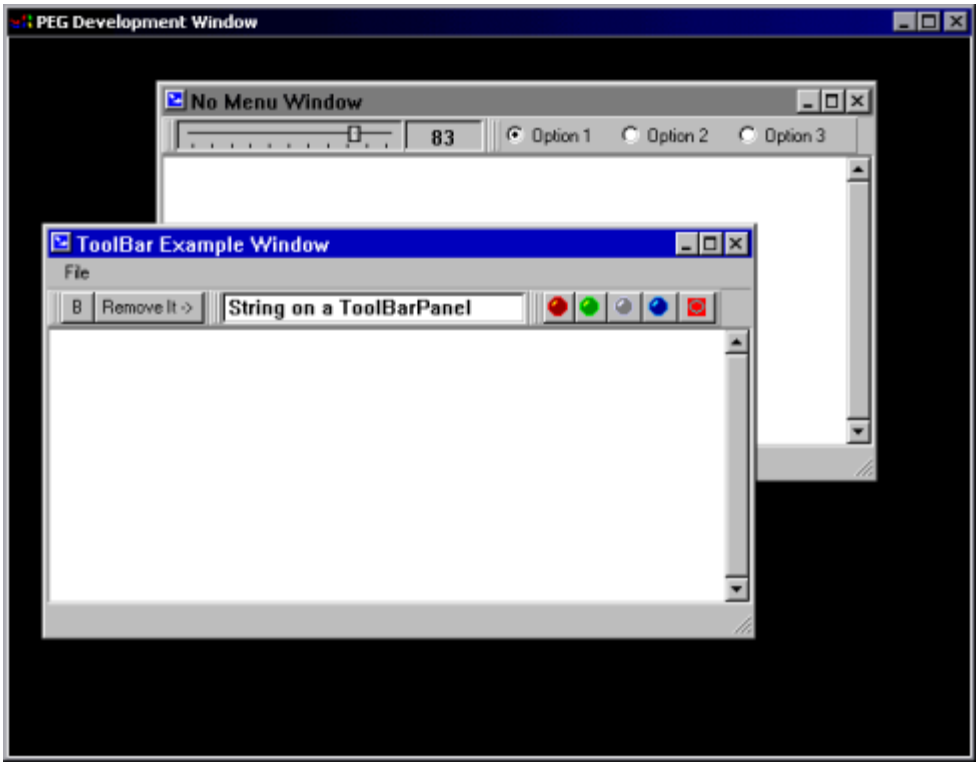
```
virtual void PositionChildren()
```

This function calculates the positions of all of its child objects.

2.28.9 Examples:

The following is an example of two PegDecoratedWindows with PegToolBars and PegToolBarPanels. Note that there are three PegToolBarPanels on the top window and that there are two PegToolBarPanels on the second window. Note also that the PegThing derived objects are added to the PegToolBarPanels, not to the PegToolBar itself.

Below is a PegToolBar example image:



The following example creates a PegToolBar and adds three PegToolBarPanels to the PegToolBar. This is the PegToolBar in the above top-level window. Usually you would do this in the constructor of your PegDecoratedWindow derived window.

```
... some code deleted ...


    PegToolBar *pToolBar = new PegToolBar();

    PegRect Rect;
    PegToolBarPanel *pPanel = new PegToolBarPanel();
    Rect.Set(0, 0, 70, 20);
    pPanel->Add(new PegTextButton(Rect, "Remove It ->",
        IDB_ALPHA_BUTTON));
    Rect.Set(0, 0, 20, 20);
```

```
pPanel->AddToEnd(new PegTextButton(Rect, "B"));
pToolBar->AddPanel(pPanel);
Rect.Set(0, 0, 200, 20);
pPanel = new PegToolBarPanel(IDC_STRING_PANEL);
pPanel->Add(new PegEditField(Rect,
    "String on a ToolBarPanel"));
pToolBar->AddPanel(pPanel);

Rect.Set(0, 0, 19, 18);
pPanel = new PegToolBarPanel();
pPanel->Add(new PegIconButton(Rect, BID_BULLSEYE,
    IDB_BULL_BUTTON), FALSE);
pPanel->Add(new PegIconButton(Rect, BID_BLUE_DOT,
    IDB_BLUE_BUTTON), FALSE);
pPanel->Add(new PegIconButton(Rect, BID_GRAY_DOT,
    IDB_GREY_BUTTON), FALSE);
pPanel->Add(new PegIconButton(Rect, BID_GREEN_DOT,
    IDB_GREEN_BUTTON), FALSE);
pPanel->Add(new PegIconButton(Rect, BID_RED_DOT,
    IDB_RED_BUTTON), FALSE);
pToolBar->AddPanel(pPanel);
Add(pToolBar);
```

2.30 PegTransIcon

2.30.1 Overview

PegIcon is a simple bitmap display object. PegIcon can also be used to represent another object.

PegIcon can be assigned a 'Proxy' object pointer. If this pointer is assigned, the icon will add the Proxy object to the icon parent and destroy itself when selected. This is how window icons function. PegDecoratedWindow classes will create an instance of PegIcon when they are minimized, add the icon to the window parent, and remove themselves from the parent.

A PegIcon object with no Proxy assignment is also useful for simply displaying a PegBitmap.

2.11.2 See Also

PegBitmap

PegThing

2.11.3 Style Flags

None.

2.11.4 Signals

None.

2.11.5 Derivation

PegIcon is derived from PegThing.

2.11.6 Constructors:

```
PegIcon(PegThing *pProxy, PEGINT BitmapId = 0, PEGUINT
      Id = 0, PEGULONG Style = FF_NONE)

PegIcon(const PegRect &Where, PEGINT BitmapId = 0,
      PEGUINT Id = 0, PEGULONG Style = FF_NONE)

PegIcon(PEGINT BitmapId = 0, PEGUINT Id = 0, PEGULONG
      Style = FF_NONE)
```

The first constructor creates a PegIcon that represents or serves as a proxy for another object. The second and third constructors create a PegIcon that will simply display a bitmap. The second constructor allows the caller to specify the icon size and position. The third constructor allows the icon to self determine the overall icon size to match the bitmap size. When the third constructor is used, the application software may immediately use the `Resize()` function to position the icon.

2.11.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegIcon overrides the `Draw()` function to draw the icon bitmap.

```
virtual PegBitmap *GetIcon(void)
```

This inline function returns the address of the bitmap associated with the PegIcon.

```
virtual PegThing *GetProxy(void)
```

This inline function returns the address of the object that is represented by the PegIcon.

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

PegIcon catches `PM_LBUTTONUP` messages.

```
virtual void SetIcon(PegBitmap *nbm)
```

This inline function can be used to alter the PegIcon bitmap at any time.

```
virtual void SetProxy(PegThing *pNew)
```

This inline function can be used to assign or alter the object represented by the PegIcon.

2.29 PegVScroll

2.29.1 Overview

PegVScroll is a vertical scroll bar class. The scroll bar elevator is proportional to the visible area of the object being scrolled.

PegVScroll takes two forms. The most common form is a NONCLIENT area scroll bar. In this form, PegVScroll calls the parent window `GetVScrollInfo` function to determine position, size, and limit information. An instance of this form of PegVScroll has `PSF_NONCLIENT` system status.

The second form is a client area scroll bar. This form does not have `PSF_NONCLIENT` system status. This type of scroll bar is under system software control, and does not attempt to automatically determine position and limit information.

Client area PegVScroll objects are very similar in operation to PegSlider objects. They are useful for allowing the user to update a field on the display by dragging the scroll elevator or selecting the directional scrolling buttons.

2.29.2 See Also

PegScrollInfo

PegHScroll

PegScroll

PegSlider

How Scrolling Works

2.29.3 Style Flags

There are no styles associated with PegVScroll Objects.

2.29.4 Signals

PegVScroll sends `PSF_SCROLL_CHANGE` signals when the position of the scroll bar elevator is changed either by dragging the elevator or by selecting the directional scroll buttons. The signal message contains the following information:

`Message.Param` = ID of the PegHScroll object.
`Message.ExtParams[0]` = Current scroll position.
`Message.ExtParams[1]` = Last reported scroll position.
`Message.pSource` = Pointer to PegHScroll object.

2.29.5 Derivation

PegVScroll is derived from PegScroll.

2.29.6 Constructors:

```
PegVScroll(PegScrollDrawInfo *pDrawInfo = NULL)

PegVScroll(const PegRect &InRect, PegScrollInfo *pSi,
     PEGUINT Id = 0, PegScrollDrawInfo *pDrawInfo =
     NULL)
```

The first constructor creates a non-client area scroll bar. The scroll bar will automatically determine its position and size itself to the height of the parent window.

The second constructor creates a client area scroll bar. In this mode, a pointer to a PegScrollInfo structure is passed to setup the initial scrolling range. In this mode, the scroll bar position and size are passed to the constructor along with the scroll bar ID, if any.

2.29.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegVScroll overrides the `Draw()` function to fill the scroll bar background area.

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

PegVScroll catches `PM_SHOW`, directional button selection, and elevator drag messages.

```
virtual void Resize(const PegRect &Rect)
```

PegVScroll overrides the `Resize()` function to ensure that the elevator remains proportional to the overall scroll bar size.

```
void SetThumbColor(PEGCOLOR Color)
```

This function modifies the color of the thumb (elevator) button.

2.29.8 Protected Members

```
virtual void CreateButtons(void)
```

This function is responsible for creating the up and down arrow buttons, as well as the elevator button. This is a separate virtual function so that it can be overridden in derived classes that want to use different types of buttons.

```
PegIconButton *mpDownButton
```

This is the down arrow button.

```
PeScrollButton *mpScrollButton
```

This is the elevator button that moves up and down inside the scroll bar.

```
PegIconButton *mpUpButton
```

This the up arrow button.

```
virtual void ReadParentScrollInfo(void)
```

This function is called by non-client scroll bars when the scroll bar is first shown to discover the parent window's scrolling parameters.

```
virtual void SetThumbButtonPos(void)
```

This function is called when the scroll bar is first shown, when it is resized, or when the scroll information is changed. This function should determine the position and size of the scroll button (if any).

2.29.9 Examples:

The following are examples of PegVScroll:

The following example initializes a PegScrollInfo structure and creates a client area scroll bar. The scroll bar will report values between 0 and 200, will initially be positioned at 100, and the scroll bar elevator will be 25% as high as the scroll bar.

```
void MyWindow::AddVScroll(void)
{
    PegScrollInfo si;

    si.Min = 0;
    si.Max = 200;
    si.Current = 100;
    si.Step = 1;
    si.Visible = 50;

    PegRect ScrollRect;
    ScrollRect.Set(10, 10, PEG_SCROLL_WIDTH + 10, 80);

    Add(new PegVScroll(ScrollRect, &si));
}
```

2.30 PegVPrompt

2.30.1 Overview

PegVPrompt (Peg Vertical Prompt) is a text display object. PegVPrompt can be drawn with several different border styles, and can be updated dynamically for interactive updates or real-time information display. PegVPrompt does not support user editing.

PegVPrompt differs from PegPrompt in that the text is displayed vertically, with the first character of the prompt string displayed at the top of the prompt object and the last character of the prompt string displayed at the bottom of the prompt object.

The PegVPrompt text is centered both horizontally and vertically within the prompt client area.

The font used by PegVPrompt can be changed at any time by using the `SetFont()` function, which is a PegTextThing member function. Likewise, the color used by PegVPrompt can be set at any time by calling the `SetColor()` function.

2.30.2 See Also

PegEditField

PegPrompt

2.30.3 Style Flags

PegVPrompt supports the same styles as PegPrompt, as shown below.

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame
`FF_RAISED`	Raised 3D Frame
`FF_RECESSED`	Recessed 3D Frame
`TJ_RIGHT`	Right justified text
`TJ_LEFT`	Left justified text
`TJ_CENTER`	Centered text

TT_COPY	Instructs the prompt to copy the string assigned. This flag should be used when the string assigned to the prompt is created dynamically using temporary storage.
AF_TRANSPARENT	Does not fill client area, assumes same color as parent.
AF_ENABLED	Prompt can be selected. The prompt will also send a PSF_FOCUS_RECEIVED signal to its parent.

2.30.4 Signals

In addition to the common signals defined by PegThing, PegVPrompt supports the PSF_CLICKED signal notification.

2.30.5 Derivation

PegVPrompt is derived from PegPrompt.

2.30.6 Constructors:

```
PegVPrompt(const PegRect &Rect, PEGUINT StringId = 0,
      PEGUSHORT Id = 0, PEGULONG Style = FF_NONE|
      AF_TRANSPARENT)

PegVPrompt(const PEGCHAR *pText, const PegRect &Rect,
      const PEGCHAR *pText, PEGUSHORT Id = 0, PEGULONG
      Style = FF_NONE|AF_TRANSPARENT)
```

The PegVPrompt constructor accepts a rectangle describing the prompt position and size, a pointer to or an ID for the initial text value, and optional object ID and style flags.

2.30.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegVPrompt overrides the Draw() function to display the prompt text.

2.30.8 Examples:

The following are examples of PegVPrompt:


ALARM
ALARM

CHAPTER 3

IMAGE CONVERSIONS

PegBmpConvert
PegGifConvert
PegImageConvert
PegJpgConvert
PegPngConvert
PegQuant

3.1 PegBmpConvert

3.1.1 Overview

PegBmpConvert is a PegImageConvert-derived class for reading and decompressing MS Windows or OS/2 BMP graphics files. Before using PegBmpConvert, be sure to read fully the PegImageConvert base class documentation.

The PegBmpConvert class is only included in the PEG library if PEG_BMP_CONVERT is defined in header file pconfig.hpp.

3.1.2 See Also

PegGifConvert

PegImageConvert

PegJpgConvert

PegPngConvert

PegQuant

3.1.3 Derivation

PegBmpConvert is derived from PegImageConvert.

3.1.4 Constructors:

```
PegBmpConvert(PEGUINT Id = 0)
```

This constructor creates a bitmap conversion object.

3.1.5 Public Functions:

```
BmpHeader *GetBmpHeader(void)
```

This function returns a pointer to the bitmap header information. This can be used to determine additional information about the decoded bitmap file. The BmpHeader structure is defined as:

```
typedef struct
{
    PEGUSHORT  id;
    PEGULONG   file_size;
    PEGUSHORT  reserved[2];
    PEGULONG   image_offset;
    PEGUSHORT  header_size;
    PEGUSHORT  not_used;
    PEGULONG   xres;
    PEGULONG   yres;
    PEGUSHORT  numplanes;
    PEGUSHORT  bits_per_pix;
    PEGULONG   compression;
    PEGULONG   bit_map_size;
    PEGULONG   hor_res;
    PEGULONG   vert_res;
    PEGULONG   number_of_colors;
    PEGULONG   num_sig_colors;
} BmpHeader;
```

```
virtual PEGBOOL GetImageInfo(PegFile *pSrc,
     PegImageInfo *pInfo)
```

This reads the header of an image file and populates a PegImageInfo structure with the height, width, and bits per pixel of the image.

```
struct PegImageInfo {
    PEGINT Width;
    PEGINT Height;
    PEGINT BitsPerPix;
};
```

The caller is responsible for allocating the structure. When the function is finished, it resets the file pointer to the beginning of the file. This function is only available if `PIC_FILE_MODE` is turned on.

```
virtual PegPixel GetULCColor(void)
```

Returns the upper-left hand corner color of the first bitmap. This is used by image conversion to add transparency to bitmap images.

```
virtual PEGBOOL ReadImage(PEGINT Count = 1)

virtual PEGBOOL ReadImage(PegFile *pFile, PEGINT Count
     = 1)
```

This function reads the BMP image. The `Count` parameter is ignored for BMP image files. The second form of the function is defined only if `PIC_FILE_MODE` is defined in the `\peg\include\pconfig.hpp` header file. The first form of the function is used to transfer source file data from an external source (such as a web browser network stack) to the image decoder.

3.1.6 Examples:

A complete working example program using the run-time image conversion classes is provided in your PEG distribution in the directory `\peg\examples\imgview`.

3.2 PegGifConvert

3.2.1 Overview

PegGifConvert is a PegImageConvert-derived class for reading and decompressing GIF input images. Before using PegGifConvert, be sure to read fully the PegImageConvert base class documentation.

The PegGifConvert class is only included in the PEG library if `PEG_GIF_CONVERT` is defined in header file `pconfig.hpp`.

PegGifConvert reads GIF image files and produces PegBitmap structures and GIF_IMAGE_INFO structures. The structures produced are NOT deleted when the PegGifConvert object is deleted, unless you first call the `DestroyImages()` function before deleting the PegGifConvert object.

Normal usage is to create the PegGifConvert object, use it to read and decompress any number of GIF images, and to retrieve the PegBitmap and GIF_IMAGE_INFO structures from the conversion object using the `GetBitmapPointer()` and `GetGifInfo()` functions. Once you have retrieved the output produced, you can delete the conversion object without losing any data. After you are done using or displaying the PegBitmaps produced, you should free the memory associated with the run-time conversion by deleting the PegBitmap and GIF_IMAGE_INFO structures with a call to the `DestroyImages()` function.

**** WARNING ****

PegGifConvert uses the LZW decompression algorithm to read and decode GIF images. The LZW algorithm is patented, and the patent is now owned by Unisys Corporation. Usage of LZW requires that the user obtain a usage license directly from Unisys. Software providers such as Swell Software are not allowed to provide a transferable usage license. It is therefore your responsibility to obtain an LZW usage license if you decide to use the PegGifConvert class in your application. By providing PEG users with a GIF decompression class, free of charge, Swell Software in no way implies that an LZW license has been obtained, and assumes no responsibility for any user who willfully violates the Unisys LZW patent by using the PegGifConvert class without obtaining an LZW license from Unisys.

At the time of this printing, LZW usage licenses are available from Unisys for a royalty charge of 0.45% (forty five one-hundredths of one percent) or

less per OEM unit shipment. Further information can be obtained from the Unisys website, www.unisys.com.

3.2.2 See Also

PegBmpConvert

PegImageConvert

PegJpgConvert

PegPngConvert

PegQuant

3.2.3 Derivation

PegGifConvert is derived from PegImageConvert.

3.2.4 Constructors:

```
PegGifConvert(PEGUINT Id = 0)
```

This constructor creates an image conversion object.

3.2.5 Public Functions:

```
void DestroyImages(void)
```

This function destroys all of the PegBitmaps that have been created as well as any other internal structures used to maintain them.

```
GIF_HEADER *GetGifHeader(void)
```

This function returns a pointer to the GIF file header information. This can be used to determine additional information about the decoded GIF. There is one GIF_HEADER structure produced for each GIF file converted. The GIF_HEADER structure is deleted when the PegGifConvert object is destroyed. The GIF header information structure is defined as:

```
typedef struct {
    PEGUINT Width;    // overall width
    PEGUINT Height;   // overall height
    PEGUINT Colors;
```

```
    PEGUBYTE BackClrIndex;
    COLORVAL Background; // background fill color
    PEGUBYTE AspectRatio;
    PEGUBYTE IsGif89;
} GIF_HEADER;
```

```
GIF_IMAGE_INFO *GetGifInfo(void)
```

This function is used to retrieve a pointer to the array of GIF information structures produced during image conversion. There will be one element in the array for each image converted, i.e. one GIF file may contain any number of images and the equivalent number of GIF_IMAGE_INFO structures will be produced. The GIF_IMAGE_INFO structures define local information for each embedded image such as size, relative offset, and delay time. The GIF_IMAGE_INFO structure is defined as:

```
typedef struct {
    PEGINT xOffset; // relative x offset
    PEGINT yOffset; // relative y offset
    PEGUINT Width;    // width in pixels
    PEGUINT Height;  // height in pixels
    PEGUINT Delay;   // delay in hundredths of a second
    PEGUBYTE HasTrans;
    PEGUBYTE TransColor;
    PEGUBYTE InputFlag; // wait for user input?
    PEGUBYTE Disposal; // image overwrite method
} GIF_IMAGE_INFO;
```

After your application has completed using the image data produced by the conversion object, you must delete the GIF_IMAGE_INFO structures with a call to the `DestroyImages()` function to avoid a memory leak.

```
virtual PEGBOOL GetImageInfo(PegFile *pSrc,
     PegImageInfo *pInfo)
```

This function fills in the PegImageInfo structure from the GIF file `pSrc`. This is only available if PIC_FILE_MODE is turned on.

```
PEGBOOL ReadFrame(void)
```

GIF images can be animated by using multiple frames. This function reads in one individual frame and converts it into a PegBitmap.

```
PEGBOOL ReadHeader()
```

This function reads information out of the GIF header, such as height, width, and number of colors.

```
PEGBOOL ReadHeader(PegFile *pSrc)
```

This version of the `ReadHeader` function is only available if `PIC_FILE_MODE` is turned on.

```
virtual PEGBOOL ReadImage(PEGINT Count = 10)

virtual PEGBOOL ReadImage(PegFile *pSrc, PEGINT Count
     = 10)
```

This function reads the GIF image. The `Count` parameter defines the maximum number of PegBitmap structures to produce in the event that the GIF file contains multiple images.

The second form of the function is defined only if `PIC_FILE_MODE` is defined in the `\peg\include\pconfig.hpp` header file. The first form of the function is used when `PIC_FILE_MODE` is not defined.

3.2.6 Examples:

A complete working example program using the run-time image conversion classes is provided in your PEG distribution in the directory `\peg\examples\imgview`.

3.3 PegImageConvert

3.3.1 Overview

PegImageConvert is the base class used to provide the image decompression and other processing used by the PEG utility program of the same name. PegImageConvert serves as the base class for PegBmpConvert, PegGifConvert, PegJpgConvert, and PegPngConvert classes. These classes provide the ability to read, color quantize, RLE encode, and generate PegBitmap-formatted data structures at program run time.

Note: Most applications do NOT use or enable run-time image conversion! Run-time image conversion can consume a large amount of dynamic memory, and should only be used if required. The better alternative to run-time image conversion in many cases is to use the ImageConvert utility program to pre-convert image files into PegBitmap source files.

The run-time image conversion classes are enabled individually with #defines in the `pconfig.hpp` header file. If any type of image conversion is enabled, the base PegImageConvert class is compiled and included in the PEG library.

The image conversion classes are designed to run either standalone using file input, or under separate low-priority execution threads. There are definitions in the file `\peg\include\pconfig.hpp` that specify how the image conversion classes are going to be used.

The definition `PIC_FILE_MODE`, when enabled, instructs the image conversion to use file I/O input methods. Alternatively, when this definition is disabled, the image conversion classes use a circular data buffer for input. This input buffer should be 'fed' by an external task. When running in this mode, the conversion object calls a user-defined callback function when the conversion object needs more input data or when the state of the conversion object changes. This facilitates usage in a multitasking environment where the input data is streaming in from an external source. The provided callback function may sleep or otherwise suspend the conversion process until additional input data is available.

If PIC_FILE_MODE is turned on, the GetImageInfo function reads the image header and populates a PegImageInfo structure. The structure contains the height, width, and bits per pixel of the image. Before the function returns, it resets the file pointer back to the beginning of the file.

One of the jobs of the image conversion class is to map the input file colors to the target system display capabilities. There are also two basic color mapping modes that can be used by the conversion object. These mapping modes are referred to as 'inline' mode and 'post-read' mode. In inline remapping mode, the converter reads and decompresses one scanline of input data to a temporary buffer and remaps this single row to the target colors or grayscale. This process continues on a line-by-line basis until the entire image has been remapped to the target system palette. The benefit of this conversion mode is that only a single-line temporary buffer is required in addition to the final output data array. As a result, this conversion mode uses less dynamic memory than post-read conversion. Please note, however, that inline conversion mode requires that the input data is scan-line oriented and not interlaced. For this reason, inline conversion mode cannot be used with interlaced PNG input files, which use an interlace format that is not scan-line oriented.

Post-read conversion means that the entire input image is read into a temporary buffer, and the image is then color mapped into the final output buffer. This conversion mode can require much more memory than inline conversion, especially for cases when high color depth (i.e. 24bpp) images are being remapped to lower color depth target displays which may require less than 8 bpp output in the final buffer. Post-read conversion has the beneficial capability of producing an optimal color palette for 8 bpp paletted systems. It is also able to handle interlaced PNG files.

The color-mapping algorithm used can be either a best-match algorithm or a dithering algorithm. For inline remapping, the algorithm is determined by the conversion mode. For post-read remapping the algorithm used is determined by which color-mapping function the application calls.

After conversion is complete, the conversion objects return a pointer to a PegBitmap structure containing the converted image ready for display. In most cases, only one converted image will be available. However, for animated GIF files there may be multiple PegBitmap structures produced. The application software can determine how many images were converted by calling the GetBitmapCount() function, and may retrieve pointers to each bitmap by calling the GetBitmapPointer() function.

The image conversion classes can also be used in combination with the PEG color quantization class, PegQuant. The image conversion classes will count the colors present in each image for use in the color quantization histogram.

The definition `PIC_QUANT`, when enabled, includes the color counting functions required for custom palette generation. When this definition is disabled, the color quantization functions are not included. This definition is disabled for most embedded targets.

The order in which the PegImageConvert member functions are called is critical for correct operation. This is detailed further in the member function descriptions. In general, the conversion process can either convert the incoming data inline to a predefined system palette, or in one pass after reading the entire input image. Inline conversion is the most memory efficient when running with a fixed system palette.

The general order of operation for inline conversion to a fixed system palette is:

- Construct the conversion object.
- Set the conversion object mode to `PIC_INLINE_REMAP` or `PIC_INLINE_DITHER`.
- Initialize the conversion object system palette.
- Set the conversion object callback function if not using a file system.
- Call the conversion object `ReadImage()` function.
- Call the conversion object `RleEncode()` function (optional).
- Retrieve the completed PegBitmap pointer using `GetBitmapPointer()`.
- Delete the conversion object.

The general order of operation for post-read conversion to a fixed system palette is:

- Construct the conversion object.
- Initialize the conversion object system palette.
- Set the conversion object callback function if not using a file system.
- Call the conversion object `ReadImage()` function.
- Call the conversion object `RemapBitmap()` or `DitherBitmap()` function to remap the image to the target system color depth.

- Call the conversion object `RleEncode()` function (optional).
- Retrieve the completed PegBitmap pointer using `GetBitmapPointer()`.
- Delete the conversion object.

This order of operation is somewhat different when the goal is to create a custom palette and perform post-read remapping to the optimal palette:

- Construct a PegQuant object.
- Construct the conversion object.
- Set the conversion object callback function if not using a file system.
- Call the conversion object `ReadImage()` function.
- Call the conversion object `CountColors()` function.
- Call the PeqQuant object `ReduceColors()` function.
- Pass palette produced by PegQuant as system palette to the conversion object.
- Call conversion object `DitherBitmap()` or `RemapBitmap()` function to remap to optimal palette.
- Retrieve the completed PegBitmap pointer(s) using `GetBitmapPointer()`.
- Delete the conversion object.

** Important Note **

The PegBitmap structures created by the conversion object are not deleted when the object is destroyed. Under normal operation, the caller retrieves the bitmaps after conversion by calling the `GetBitmapPointer()` function. The caller then owns the memory associated with the bitmaps, and must free this memory after the bitmaps are no longer needed.

If the caller does not retrieve the PegBitmap structures, they should be deleted with a call to `DestroyImages()` before deleting the conversion object. This can be useful in an application where you do not actually want to display the images, but you do need to obtain image size or other attribute information. In this case, you can construct the conversion object(s), tell them to read the image, retrieve whatever information is required, and then call `DestroyImages()` to clean up the memory associated with the conversion objects.

3.3.2 See Also

PegBmpConvert

PegGifConvert

PegJpgConvert

PegPngConvert

PegQuant

3.3.3 Derivation

PegImageConvert is a PEG base class.

3.3.4 Constructors:

```
PegImageConvert(PEGUINT Id)
```

This constructor creates an image conversion object.

3.3.5 Public Functions:

```
PEGUINT ConsumeImageInput(PEGUBYTE *pBuffer, PEGUINT
     Length)
```

This function is called by derived conversion classes to get input data. This abstracts the data source (either file or input buffer) from the derived conversion classes.

```
void CountColors(PegQuant *pQuant)
```

This function is called after the image has been decoded to count the occurrences of each possible color value. This is used for color reduction along with the PegQuant class. This function is only provided if `PIC_QUANT` is defined in the file `\peg\include\pconfig.hpp`.

```
void DestroyImages(void)
```

This function can be called to destroy all PegBitmap images created by the conversion object. This should only be done if the caller does not want to keep and use the PegBitmaps produced.

```
PEGBOOL DitherBitmap(void)
```

This function can be called after an image has been read and a system palette has been assigned with `SetSystemPalette()`. This function will dither the contained PegBitmap to the system palette.

```
PEGUINT GetBitmapCount(void)
```

This inline function can be called to learn the number of PegBitmap structures produced during the image read operation. GIF files can contain any number of individual images.

```
PegBitmap *GetBitmapPointer(PEGUINT Index)
```

This function returns a pointer to the PegBitmap structure at the specified index produced by the conversion process. Most images (JPEG, PNG, etc.) only have one bitmap, so passing an index of 0 obtains it. GIFs can have multiple bitmaps, so pass the index of the bitmap you need.

```
PEGLONG GetDataSize(PEGUINT Index = 0)
```

Returns the data size of the specified image.

```
PEGCHAR *GetErrorString(void)
```

This function returns the error string associated with a conversion failure. This value is only valid if the conversion object state variable indicates that an error has occurred.

```
virtual PEGBOOL GetImageInfo(PegFile *pSrc,
      PegImageInfo *pInfo)
```

This function reads the header of an image file and populates a PegImageInfo structure with the height, width, and bits per pixel of the image.

```
struct PegImageInfo {
    PEGINT Width;
    PEGINT Height;
    PEGINT BitsPerPix;
};
```

The caller is responsible for allocating the structure. When the function is finished, it resets the file pointer to the beginning of the file. This function is only available if `PIC_FILE_MODE` is turned on.

```
static PEGUBYTE GetImageType(PegFile *pSrc)
```

This function reads the beginning of an image file to determine what type of image it is. If it succeeds, it returns one of the following:

```
PIC_TYPE_BMP  // Bitmap file
PIC_TYPE_GIF  // GIF file
PIC_TYPE_PNG  // PNG file
PIC_TYPE_JPG  // JPEG file
```

When the function is finished, it resets the file pointer back to the beginning of the file.

```
static PEGUBYTE GetImageType(PEGUTYPE *pData)
```

This version of the `GetImageType` function is only available if `PIC_MEMORY_MODE` is turned on.

```
PEGUINT GetMode(void)
```

Returns the operating mode of the conversion object. The operating mode is determined by the caller when configuring the conversion object. The available modes are:

```
PIC_NO_CONVERT   // no inline conversion (custom palette mode)
PIC_INLINE_DITHER // dither on the fly
PIC_INLINE_REMAP  // remap to best color on the fly
PIC_RGB_TRANS  // use RGB transparency
PIC_INDEX_TRANS   // use indexed transparency
```

When operating with a fixed system palette, it is important to configure the conversion object for an inline conversion mode prior to reading the image. This saves a large amount of memory space over reading the image and then converting to a fixed palette.

```
PEGUBYTE GetOutputBitsPerPix(void)
```

This function returns the output color depth determined by the system palette assigned to the conversion object.

```
PegPixel GetPixelColor(PEGUBYTE *pGet, PEGUINT YPos,
     PEGUINT Index)
```

Returns the PegPixel value at the specified index.

```
PEGUINT GetRowsConverted(void)
```

Returns the number of rows that have been converted so far.

```
PEGUINT GetState(void)
```

Returns the state variable of the conversion object. The possible states are:

```
PIC_IDLE      // waiting for input data
PIC_ERROR     // an error has occurred
PIC_HEADER_KNOWN  // the header (width, height, type) is known
PIC_PALETTE_KNOWN // the palette is known
PIC_ONE_CONV_DONE // at least one conversion complete
PIC_COMPLETE  // all conversions are complete
```

```
PEGUINT GetStride(PEGUINT Width, PEGUINT Bits, PEGUINT
      HasAlpha = 0)
```

This function calculates the stride based on the width, bits per pixel, and whether or not it has an alpha-channel.

```
PEGUINT GetStride(PegBitmap *pMap)
```

This version of the `GetStride` function calls the other `GetStride` function using `pMap`'s width, bits-per-pixel, and alpha information.

```
PegPixel GetTransColor(void)
```

Returns the transparent color value.

```
PegPixel GetULCColor(void)
```

Returns the upper-left hand corner color of the first bitmap.

```
void Id(PEGUINT Id)
```

Sets the conversion object ID.

```
PEGUINT Id(void)
```

Returns the ID of the conversion object. This is a caller-defined value used to identify a particular converter when many conversions are occurring concurrently.

```
PEGUBYTE ImageType(void)
```

Returns the image type processed by the conversion object. The supported image types are:

```
PIC_TYPE_BMP
PIC_TYPE_GIF
PIC_TYPE_JPG
PIC_TYPE_PNG
```

```
PEGUINT InputFreeSpace(void)
```

Returns the number of bytes free in the input buffer. This function is only provided when `PIC_FILE_MODE` is not defined.

```
void KeepAlpha(PEGBOOL Keep)
```

This function determines whether the alpha information from the original image is to be kept.

```
virtual PEGBOOL ReadImage(PegFile *pFile, PEGINT Count
      = 100)

virtual PEGBOOL ReadImage(PEGINT Count = 100)
```

This function begins the conversion process. It should be called only after the converter has been configured with `SetMode()`. The first form is provided for `PIC_FILE_MODE`, and the second form for a multitasking environment. Use of the `Count` parameter depends on the `ReadImage()` implementation for a specific derived class. Currently, `Count` is used only in the GIF image conversion implementation as an upper limit on the number of GIF89a embedded images converted.

```
PEGBOOL RemapBitmap(void)
```

This function is called to do a best-color mapping of the bitmap to a fixed system palette. The `SetSystemPalette()` function must be called before `RemapBitmap()`. This function does no dithering; instead, it uses a closest-match algorithm. This is useful when an optimal palette has been generated.

```
PEGBOOL RleEncode(PEGBOOL Force = FALSE)
```

This function can be called to RLE encode a converted bitmap. The function will first determine if RLE encoding saves space before doing the encoding, so the caller cannot assume that the contained PegBitmap is encoded after calling this function, unless Force is set to TRUE. This function is only defined if `PEG_CONVERT_RLECOMP` is defined in `pconfig.hpp`.

```
void RotateImages(PEGINT Rotation, PEGBOOL FlipX =
      False, PEGBOOL Flipy = FALSE)
```

This function rotates the bitmap. Only values of 90 or 270 degrees are supported. The bitmap rotates counter-clockwise. The parameters `FlipX` and `FlipY` also tell it to mirror the image along the x or y axis, respectively.

```
PEGUINT SendData(PEGUBYTE *pGet, PEGUINT Size)
```

This function feeds input data to the converter. It is only provided when PIC_FILE_MODE is not defined. This function allows an external task to pass data to the conversion object.

```
void SetGrayscale(PEGBOOL Gray)
```

This function changes the color mapping algorithm to match colors based on brightness using a linear grayscale target palette.

```
void SetIdleCallback(PEGBOOL (*pFunc)(PEGUSHORT Id,
     PEGUSHORT State, PegImageConvert *pObject))
```

This function assigns a callback function that the conversion object will call whenever input data is needed or the state of the conversion object has changed. The callback function will receive the ID of the conversion object, the conversion object state variable, and a pointer to the conversion object. This function is only provided when PIC_FILE_MODE is not defined.

The callback function should be structured such that it tests the conversion object state variable to determine the reason for the callback. If the conversion object needs data, the callback function should provide it by calling the conversion object SendData() function. The state variable may also indicate that conversion is complete or that an error has occurred.

The State value is a bitwise OR of the status flags defined above under the GetState() function.

The main purpose for the callback structure is to facilitate running in a multitasking environment. The callback function may be structured to sleep using OS defined methods until new input data becomes available.

```
void SetInputBuffer(PEGUBYTE *pGet, PEGULONG Size)
```

This function sets the input buffer out of which the image data is read. This function is only available if PIC_MEMORY_MODE is turned on.

```
void SetMode(PEGUINT Mode)
```

This function configures the conversion object for one of several possible modes. This function should be called before the input image is actually read. The available modes are:

```
PIC_NO_CONVERT    // no inline conversion (custom palette mode)
PIC_INLINE_DITHER // dither on the fly
PIC_INLINE_REMAP  // remap to best color on the fly
```

```
PIC_RGB_TRANS // use RGB transparency
PIC_INDEX_TRANS  // use indexed transparency
```

When operating with a fixed system palette, it is important to configure the conversion object for an inline conversion mode prior to reading the image. This saves a large amount of memory space over reading the image and then converting to a fixed palette.

```
void SetPalStart(PEGUINT Index)
```

This function sets the beginning entry of the palette that it should use for the image.

```
void SetSystemPalette(PEGUBYTE *pPal, PEGUINT PalSize,
     PEGBOOL Fast = FALSE)
```

This function informs the conversion object of the working system palette. This is required for best-color remapping and dithering. The `Fast` parameter is no longer used.

```
void SetTargetSize(PEGINT Width, PEGINT Height)
```

This function sets the output size of the converted image. This does not need to be the same as the original image size.

```
void SetTransColor(PegPixel Pixel)

void SetTransColor(PEGUBYTE Index)
```

This function can be used to assign a transparent color, either by its RGB value or its palette index. This is only required for BMP and JPEG input images, since other image types embed transparency information.

3.3.6 Protected Members

```
PEGUBYTE *GetLocalPalette(void)
```

This function returns the local palette associated with the converted bitmap(s). This palette is embedded in the source image.

3.3.7 Examples:

A complete working example program using the run-time image conversion classes is provided in your PEG distribution in the directory `\peg\examples\imgview`.

3.4 PegJpgConvert

3.4.1 Overview

PegJpgConvert is a PegImageConvert-derived class for reading and decompressing JPG input images. Before using PegJpgConvert, be sure to read fully the PegImageConvert base class documentation.

The PegJpgConvert class is only included in the PEG library if `PEG_JPG_CONVERT` is defined in the header file, `pconfig.hpp`.

PegJpgConvert reads JPG image files and produces PegBitmap structures. The structures produced are NOT deleted when the PegJpgConvert object is deleted, unless you first call the `DestroyImages()` function before deleting the PegJpgConvert object.

Normal usage is to create the PegJpgConvert object, use it to read and decompress any number of JPG images, and to retrieve the PegBitmap structures from the conversion object using the `GetBitmapPointer()` function. Once you have retrieved the output produced, you can delete the conversion object without losing any data. After you are done using or displaying the PegBitmaps produced, you should free the memory associated with the run-time conversion by deleting the PegBitmap(s). You can delete PegBitmap by calling `Screen()->DestroyBitmap(pMap)`.

The PegJpgConvert header file, `\peg\include\pjpgconv.hpp`, includes several definitions specific to the JPG decoder. These definitions can be turned on or off (defined or not defined) to fine tune the memory usage and feature support of the JPG converter. **Note that run-time decompression of JPG images is CPU- and memory-intensive!** Run-time decoding of random JPG images requires a target system with at least several hundred KBytes of dynamic memory, and, possibly, virtual memory capabilities as well. For this reason, users who have the ability to design out the use of JPG are encouraged to opt instead for GIF or PNG encoded images, which use much less memory during the decode process.

`JPG_VIRTUAL_MEMORY` can be defined in your `pconfig.hpp` file. `JPG_VIRTUAL_MEMORY` must be enabled to support multiscan and progressive JPG files, but it is not required for other (i.e. single scan) JPG file types.

DEFAULT_MAX_JPEG_MEM defines the maximum amount of dynamic memory the JPG decoder may use. If JPG_VIRTUAL_MEMORY is defined, virtual memory will be used if additional memory is required during the decode process. If JPG_VIRTUAL_MEMORY is not defined, DEFAULT_MAX_JPEG_MEM must be large enough to allow the decoder to create the intermediate data objects used during the decode process. The amount of memory required is largely dependent on the type of JPG files being decoded and the file (image) size. For small (i.e. up to 200 x 200) images, 64K Bytes of dynamic memory has proven to be sufficient. For larger or more complex images, larger dynamic memory regions will be required. Trial and error is the only method for determining the best setting for your system. It is interesting to note that since JPG is an analog encode/decode process, running out of memory results in a loss of image quality. It does not usually prevent the JPG decoder from producing an image.

3.4.2 See Also

PegBmpConvert

PegGifConvert

PegImageConvert

PegPngConvert

PegQuant

3.4.3 Derivation

PegJpgConvert is derived from PegImageConvert.

3.4.4 Constructors:

```
PegJpgConvert(PEGUINT Id = 0)
```

This constructor creates an image conversion object.

3.4.5 Public Functions:

```
virtual PEGBOOL GetImageInfo(PegFile *pSrc,
      PegImageInfo *pInfo)
```

This reads the header of an image file and populates a PegImageInfo structure with the height, width, and bits per pixel of the image.

```
struct PegImageInfo {
    PEGINT Width;
    PEGINT Height;
    PEGINT BitsPerPix;
};
```

The caller is responsible for allocating the structure. When the function is finished, it resets the file pointer to the beginning of the file. This function is only available if `PIC_FILE_MODE` is turned on.

```
virtual PEGBOOL ReadImage(PEGINT Count = 1)

virtual PEGBOOL ReadImage(PegFile *pSrc, PEGINT Count
      = 1)
```

This function reads the JPG image. The `Count` parameter is ignored for JPG image files. The second form of the function is defined only if `PIC_FILE_MODE` is defined in the `\peg\include\pconfig.hpp` header file. The first form of the function is used to transfer source file data from an external source (such as a website browser network stack) to the image decoder.

3.4.6 Examples:

A complete working example program using the run-time image conversion classes is provided in your PEG distribution in the directory `\peg\examples\imgview`.

3.5 PegPngConvert

3.5.1 Overview

PegPngConvert is a PegImageConvert-derived class for reading and decompressing PNG input images. Before using PegPngConvert, be sure to read fully the PegImageConvert base class documentation.

The 'PNG' graphics format was designed as a replacement for the proprietary and patented GIF graphics format. 'PNG' is normally defined as an acronym for 'Portable Network Graphics,' but many say that PNG truly is an abbreviation for 'PNG is Not GIF.' The compression ratios, image support, and animation support of PNG are all equal to or superior to GIF.

The PegPngConvert class is only included in the PEG library if `PEG_PNG_DECODER` is defined in header file `pconfig.hpp`.

PegPngConvert reads PNG image files and produces PegBitmap structures. The structures produced are NOT deleted when the PegPngConvert object is deleted, unless you first call the `DestroyImages()` function before deleting the PegPngConvert object.

Normal usage is to create the PegPngConvert object, use it to read and decompress any number of PNG images, and to retrieve the PegBitmap structures from the conversion object using the `GetBitmapPointer()` function. Once you have retrieved the output produced, you can delete the conversion object without losing any data. After you are done using or displaying the PegBitmaps produced, you should free the memory associated with the run-time conversion by deleting the PegBitmap by calling `Screen()->DestroyBitmap(pMap)`.

3.5.2 See Also

PegBmpConvert

PegGifConvert

PegImageConvert

PegJpgConvert

PegQuant

3.5.3 Derivation

PegPngConvert is derived from PegImageConvert.

3.5.4 Constructors:

```
PegPngConvert(PEGUINT Id = 0)
```

This constructor creates an image conversion object.

3.5.5 Public Functions:

```
PEGUBYTE BitsPerChannel(void) const
```

Returns the number of bits per channel of the original image.

```
PEGULONG BytesPerRow(void) const
```

Returns the number of bytes per row of the original image.

```
PEGUBYTE ColorType(void) const
```

This returns the color type of the original image. It can be color or grayscale, palette or RGB, alpha or no alpha.

```
virtual PEGBOOL GetImageInfo(PegFile *pSrc,
      PegImageInfo *pInfo)
```

This reads the header of an image file and populates a PegImageInfo structure with the height, width, and bits per pixel of the image.

```
struct PegImageInfo {
    PEGINT Width;
    PEGINT Height;
    PEGINT BitsPerPix;
};
```

The caller is responsible for allocating the structure. When the function is finished, it resets the file pointer to the beginning of the file. This function is only available if `PIC_FILE_MODE` is turned on.

```
PEGULONG Height(void) const
```

Returns the height of the original image.

```
PEGUBYTE InterlaceType(void) const
```

This returns the type of interlacing used in the original image. Supported interlacing type is Adam7 or none.

```
PEGUBYTE NumChannels(void) const
```

Returns the number of channels in the original image.

```
virtual PEGBOOL ReadImage(PEGINT Count = 1)

virtual PEGBOOL ReadImage(PegFile *pSrc, PEGINT Count
     = 1)
```

This function reads the PNG image. The `Count` parameter defines the maximum number of PegBitmap structures to be produced in the event that the PNG file contains multiple images.

The second form of the function is defined only if `PIC_FILE_MODE` is defined in the `\peg\include\pconfig.hpp` header file. The first form of the function is used when `PIC_FILE_MODE` is not defined.

```
PEGULONG Width(void) const
```

Returns the width of original image.

3.5.6 Examples:

A complete working example program using the run-time image conversion classes is provided in your PEG distribution in the directory `\peg\examples\imgview`.

3.6 PegQuant

3.6.1 Overview

PegQuant is a run-time histogram and optimal palette producer. PegQuant implements a form of Heckbert's Median Cut color-reduction algorithm. This class is only required for applications that must determine dynamic optimal palettes. Most applications run with fixed palettes. PegQuant is passed to PegImageConvert-derived classes to create a histogram of color usage. After all included images have been added, the PeqQuant function `ReduceColors` is called to create an optimal palette for use with the scanned images.

3.6.2 See Also

`PegImageConvert`

`PegBmpConvert`

`PegGifConvert`

`PegJpgConvert`

3.6.3 Derivation

PegQuant is a PEG base class.

3.6.4 Constructors:

```
PegQuant(void)
```

Creates a PegQuant object.

3.6.5 Public Functions:

```
void AddColor(PegPixel Pixel)
```

This function adds the specified color to the histogram being created.

```
PEGUBYTE *GetPalette(void)
```

Returns a pointer to the optimal color palette produced from the image color sums.

```
PEGUINT PalSize(void)
```

This function returns the current number of colors in the palette.

```
PEGUINT ReduceColors(PEGINT Start = 16, Limit = 254,
      PEGINT StdStart = 0)
```

This function is called after all colors for all images have been counted. This function actually does the work of creating the optimal palette for use with the images counted.

3.6.6 Examples:

The following example reads several graphic files and creates an optimal palette for use in displaying the files. The palette is then installed as the new system palette. This example runs in `PIC_FILE_MODE`.

```
#define GIF1 "c:\\graphics\\tree.gif"
#define GIF2 "c:\\graphics\\house.gif"
#define BMP1 "c:\\graphics\\garage.bmp"


void MyWindow::CreatePalette(void)
{
    PeqQuant *pQuant = new PegQuant();

    CountColors(GIF1,  PIC_TYPE_GIF,  pQuant);
    CountColors(GIF2,  PIC_TYPE_GIF,  pQuant);
    CountColors(BMP1,  PIC_TYPE_BMP,  pQuant);

    PEGUINT PalSize =  pQuant->ReduceColors();
    Screen()->SetPalette(0, PalSize, pQuant->GetPalette());

    delete pQuant;
}


void MyWindow::CountColors(char *pPathName, PEGUINT Type,
    PegQuant *pQuant)
{
    PegFile Src;
    Src.Open(pPathName, PEG_FILEMODE_RO);
    PegImageConvert *pConvert;
```

```
    if (Type == PIC_TYPE_GIF)
    {
        pConvert = new PegGifConvert(0);
    }
    else
    {
        pConvert = new PegBmpConvert(0);
    }

    pConvert->ReadImage(&Src);
    pConvert-
    >CountColors(pQuant);
    pConvert->DestroyImages();
    delete pConvert;

    Src.Close();
}
```

CHAPTER 4

WINDOW CLASSES

PegAnimationWindow
PegComboBox
PegDecoratedWindow
PegDialog
PegEditBox
PegFileDialog
PegHorzList
PegList
PegMessageWindow
PegMLMessageWindow
PegNotebook
PegProgressWindow
PegRichTextBox
PegSpreadSheet
PegTable
PegTextBox
PegTreeNode
PegTreeView
PegVertList
PegVirtualVList
PegWindow

4.1 PegAnimationWindow

4.1.1 Overview

PegAnimationWindow is a window class for displaying a series of PegBitmap images. If these images are displayed in rapid sequence, the effect of smooth motion animation is produced.

PegAnimationWindow allows the programmer to specify the position and size of each displayed frame. The animation can be run automatically by PegAnimationWindow via the `Run()` function, or individual frames may be displayed by the caller.

PegAnimationWindow can be used in applications which support multiple palettes, and will install the correct palette for the animation when displayed.

The default operation of PegAnimationWindow is to render each animation frame to video memory during program execution. Custom implementations are often modified to render ALL frames into unused portions of video memory when PegAnimationWindow is initially displayed, and then to use the video hardware bitblitting functions to draw each frame on the visible screen. This yields superior performance on platforms that have advanced video controller hardware.

An x and y offset value may be specified for the animation window. This value indicates the offset from the window origin to the display of the animation frames. This allows the animation frames to be part of a larger background bitmap. Note that each animation frame must be displayed in the same relative position; i.e., there are no unique offset values for each frame.

4.1.2 See Also

PegBitmap

4.1.3 Style Flags

PegAnimationWindow supports the following style flags:

`FF_NONE`	No Frame

FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame

4.1.4 Derivation

PegAnimationWindow derives from PegWindow.

4.1.5 Constructors:

```
PegAnimationWindow(const PegRect &Rect, PEGINT BkgBmp,
      PEGINT *pFrameList, PEGUBYTE NumFrames, PEGINT
      xPos, PEGINT yPos, PEGUBYTE *pPalette = NULL,
      PEGULONG Style = FF_NONE)
```

Creates an animation window of the specified size at the specified position.

BkgBmp is the ID of the background bitmap for the animation. BkgBmp may be -1 if no background is desired.

pFrameList is the address of an array of pointers to PegBitmap images. These images are displayed in sequence when the animation window runs. The animation window will wrap back to the start frame after displaying the final frame, allowing the display of continuous animation.

NumFrames indicates the total number of frames in the frame list.

xPos and yPos indicate the upper-left corner position at which the animation frames will be displayed. This allows the animation window to be larger than, and offset relative to, the actual animation bitmaps.

pPalette is a pointer to an array of unsigned characters. This array is typically generated by PegImageConvert. The array should contain the RGB (Red, Green, Blue) values for the palette desired. The number of entries in the array should be 3 * PEG_NUM_COLORS.

Style is used to set the frame style for the animation window. The default is to display no frame, allowing the animation window to be placed within a larger parent window and provide a seamless animation appearance.

4.1.6 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegAnimationWindow overrides the `Draw()` function to display the animation bitmaps.

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

PegAnimationWindow overrides the `Message` method to set the custom palette (if there is one) when the window is displayed and to draw the correct animation frame when the timer message is received.

```
virtual void Run(PEGUINT Period, PEGUBYTE Frame = 0)
```

This function starts the animation sequence. The parameter Period is used to indicate the number of ticks that should elapse between the display of each animation frame.

```
virtual void SetAnimationList(PEGINT *pList)
```

This sets the list of animation frames that will be used in the animation. pList is an array of bitmap IDs referencing PegBitmaps.

```
virtual void SetBkgBitmap(PEGINT BkgBmp)
```

This sets the background image that is to be used on the window.

```
virtual void SetFrame(PEGUBYTE Frame = 0)
```

This function can be used to reset the animation frame, or to manually display the animation sequence.

```
virtual void SetFrameOffset(PEGINT xOffset, PEGINT
      yOffset)
```

This defines the upper-left corner position at which the animation frames will be displayed.

```
virtual void Stop(void)
```

This function stops the animation playback sequence.

4.1.7 Protected Members

```
virtual void DrawAnimationFrame(void)
```

This function draws an individual frame of the animation.

4.1.8 Examples:

An example of creating and using PegAnimationWindow can be found in your PEG distribution. The directory `\peg\examples\robot` contains a sample application that uses the PegAnimationWindow class.

4.2 PegComboBox

4.2.1 Overview

PegComboBox is similar to PegVertList. PegComboBox is a container that can have any type of object added to it. PegComboBox adds the concept of 'Opening and Closing,' which can conserve space when a large number of items are added to the combo box. A drop-down arrow is provided to open the combo box. The box closes when an item is selected or the combo box loses focus.

PegComboBox will send signal notifications to the parent window if the PegComboBox has a non-zero ID value and the selected child also has a non-zero ID value.

The LAST child added to the combo box will be displayed at the TOP of the combo box if the `Add()` function is used to add children. The order of display can be reversed by using the function `AddToEnd()` to add children to the combo box.

If PegPrompt objects are added to a PegComboBox, the style flags for the PegPrompt objects should include `FF_NONE|AF_ENABLED` for correct display. Normally PegPrompt objects are not selectable (i.e. the `AF_ENABLED` style is not used). However, when PegPrompt objects are added to a PegComboBox, the style should be set as shown above so that the prompt objects can be selected.

4.2.2 See Also

`PegVertList`

`PegHorzList`

`PegWindow`

4.2.3 Style Flags

PegComboBox supports the following style flags:

`FF_NONE`	No Frame

FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame

The styles for PegComboBox are identical to the PegThing styles. In addition, scrolling is enabled in PegComboBox in the same way as in PegWindow, by using the SetScrollMode() function.

4.2.4 Signals

PegComboBox sends PSF_LIST_SELECT signals to the parent object. This message contains:

Message.pSource = Pointer to combo box object.
Message.Param = ID of selected list item.
Message.pTarget = Pointer to combo box parent object.

In order to receive PSF_LIST_SELECT signals from a PegComboBox, you must ensure that the PegComboBox AND the list children have non-zero ID values.

4.2.5 Derivation

PegComboBox derives from PegThing.

4.2.6 Constructors:

```
PegComboBox(const PegRect &Rect, PEGUSHORT Id = 0,
      PEGULONG Style = FF_THIN)
```

This constructor creates a PegComboBox object. The Rect parameter determines the height of the combo box when open. The closed height is determined by the height of the individual combo box children.

4.2.7 Public Functions:

```
virtual void Add(PegThing *pChild, PEGBOOL Show =
      TRUE)
```

The PegComboBox overrides the PegThing::Add function so it can add pChild to its vertical list member, rather than to itself.

```
virtual void AddToEnd(PegThing *pChild, PEGBOOL Show =
     TRUE)
```

The PegComboBox overrides the `PegThing::AddToEnd` function so it can add `pChild` to its vertical list member, rather than to itself.

```
virtual PEGINT Clear(void)
```

This empties the combo box's list of all of the entries.

```
void CloseList(void)
```

This function closes the combo box list and displays the currently selected item.

```
virtual void Draw(const PegRect &Invalid)
```

PegComboBox overrides the `Draw()` function to display the combo box border.

```
virtual PegThing *Find(PEGUSHORT Id, PEGBOOL Recursive
     = TRUE)
```

The PegComboBox overrides the `PegThing::Find` function so it can search through the combo list as well.

```
PEGINT GetCloseHeight(void)
```

This returns the height of the combo box when it is closed.

```
virtual PEGINT GetIndex(PegThing *pWho)
```

This returns the index in the list where `pWho` can be found.

```
ComboList *GetListPointer(void)
```

This returns a pointer to the vertical list member.

```
PEGINT GetNumItems(void)
```

This returns the number of items in the list.

```
PEGINT GetOpenHeight(void)
```

This inline function returns the height of the combo box when open.

```
virtual PegThing *GetSelected(void)
```

This function returns a pointer to the currently-selected object.

```
virtual PEGINT GetSelectedIndex(void)
```

This function returns the index of the currently-selected object.

```
virtual PegThing *GetThing(PEGINT Index)
```

This function returns a pointer to the object specified by index `Index`.

```
virtual void Insert(PegThing *pWhat, PEGINT Where,
      PEGBOOL Select = TRUE, PEGBOOL Show = TRUE)
```

This function inserts the object `pWhat` into the list at index `Where`. If `Select` is TRUE, then the object becomes selected.

```
PEGBOOL IsOpen(void)
```

This inline function returns TRUE if the combo box is currently open, otherwise FALSE.

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

PegComboBox overrides the `Message` function to catch the drop down arrow selection message, and the `PM_SHOW` system message.

```
void OpenList(void)
```

This function is used to open the combobox by adding the list to the Presentation.

```
PegThing *PageDown(void)
```

If the list is open, this function scrolls down one page length.

```
PegThing *PageUp(void)
```

If the list is open, this function scrolls up one page length.

```
virtual PegThing *Remove(PegThing *pChild)
```

The PegComboBox overrides `PegThing::Remove` so that the objects will get removed from the list object instead of the combobox itself.

```
virtual void Resize(const PegRect &NewSize)
```

PegComboBox overrides the `Resize()` function to keep the drop-down arrow button positioned at the upper right hand corner of the combo box.

```
PegThing *SelectNext(void)
```

This function selects the next object in the list. If the currently-selected object is already at the bottom, it does nothing.

```
PegThing *SelectPrevious(void)
```

This function selects the previous object in the list. If the currently-selected object is already at the top, it does nothing.

```
void SetOpenHeight(PEGINT Height)
```

This modifies the height of the combobox when the list is open.

```
void SetScrollMode(PEGUINT Mode)
```

This sets the scroll mode of the list. It can be either `WSM_VSCROLL`, `WSM_AUTOVSCROLL`, or neither. `WSM_CONTINUOUS` can also be added to either.

```
virtual PegThing *SetSelected (PEGINT Index)
```

This function sets the object at index `Index` to be the currently selected object.

```
virtual void SetSelected(PegThing *pWho)
```

This function sets `pWho` to be the currently selected object.

```
void SetSeparation(PEGINT Sep)
```

This determines the amount of spacing used between items in the list.

4.2.8 Protected Members:

```
PEGBOOL mOpen
```

Boolean value to track the state of the combo box, open or closed.

```
PEGINT mOpenHeight
```

The height of the combo box when open.

```
PEGINT mCloseHeight
```

The height of the combo box when closed.

```
ComboList *mpList
```

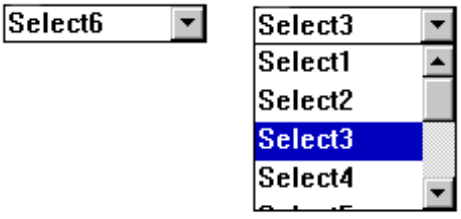
This is the list that displays all the items in the combobox.

```
PegIconButton *mpOpenButton
```

This is the button on the combobox that opens the list.

4.2.9 Examples:

The following are examples of PegComboBox:



The following example creates a PegComboBox and adds several PegPrompt objects to the combo box. The combo box will be 140 pixels tall when opened. The combo box is configured to include a vertical scroll bar. The initial item selected in the combo box will be item index 5.

```
void MyWindow::AddComboBox(void)
{
    PegRect ListRect;
    ListRect.Set(10, 10, 90, 150);
    PEGCHAR Temp[20];
    PegStrCpy(Temp, "Select");
    pList = new PegComboBox(ListRect);

    for (PEGINT Loop = 10; Loop > 0; Loop--)
    {
        PegLtoA(Loop, Temp + 6, 10);
        pList->Add(new PegPrompt(0, 0, Temp, Loop,
            FF_NONE|TJ_LEFT|AF_ENABLED|TT_COPY
            ));
    }

    pList->SetScrollMode(WSM_VSCROLL);
    pList->SetSelected(5);
    Add(pList);
}
```

4.3 PegDecoratedWindow

4.3.1 Overview

PegDecoratedWindow is a PegWindow-derived class that supports the addition of common window decorations such as PegTitle, PegMenuBar, and PegStatusBar. PegDecoratedWindow provides functions to facilitate easy access to the decorations added to a window. PegDecoratedWindow also maintains the actual client area available after the addition or removal of any of these decorations.

Like all PEG objects, PegDecoratedWindow can also have any other type of child objects added to it. The PegDecoratedWindow objects can even be nested within themselves, creating complex and interesting window types.

4.3.2 See Also

PegWindow

PegDialog

PegMessageWindow

PegMLMessageWindow

4.3.3 Style Flags

PegDecoratedWindow supports the following style flags:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame

4.3.4 Derivation

PegDecoratedWindow derives from PegWindow.

4.3.5 Constructors:

```
PegDecoratedWindow(const PegRect &Rect, PEGULONG Style
     = FF_THICK)

PegDecoratedWindow(PEGULONG Style = FF_THICK)
```

There are two constructors available for PegDecoratedWindow. The first defines the window size and position at the time the window is created. The second requires that the window size and position be determined after the window is constructed but before the window is displayed.

4.3.6 Public Functions:

```
virtual void Add(PegThing *pWhat, PEGBOOL Show = TRUE)
```

PegDecoratedWindow overrides the `Add()` function to catch the addition of non-client decorations.

```
virtual void InitClient(void)
```

PegDecoratedWindow overrides the `InitClient` method in order to resize the client area based on the presence, or absence, of a title bar, menu bar and status bar.

```
PegMenuBar *MenuBar(void)
```

This function returns a pointer to the PegMenuBar added to the window, or NULL if no menu bar is present.

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

PegDecoratedWindow catches the `PM_CURRENT` and `PM_NONCURRENT` messages.

```
virtual PegThing *Remove(PegThing *pWho)
```

PegDecoratedWindow overrides the `Remove()` function to catch the removal of non-client decorations

```
void SetTitle(PEGINT StringId)
```

This function can be called to modify the window title.

```
void SetTitle(const PEGCHAR* pText)
```

This version of the SetTitle function is used for dynamically-created strings that aren't in the string table.

```
PegStatusBar *StatusBar(void)
```

This function returns a pointer to the decorated window status bar, or NULL if no status bar is present.

```
const PEGCHAR *Title(void)
```

This function returns a pointer to the text of the decorated window title, or NULL if no title is present.

```
PegTitle *TitleObject(void)
```

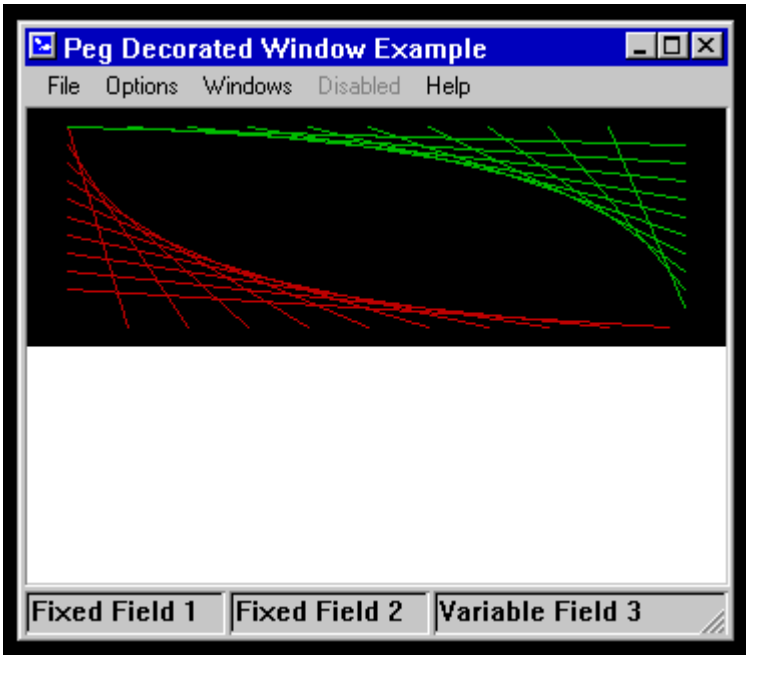
This function returns a pointer to the decorated window title, or NULL if no title is present.

```
PegToolBar *ToolBar(void)
```

This function returns a pointer to the PegToolBar added to the window, or NULL if no toolbar is present.

4.3.7 Examples:

The following is a decorated window with a title bar, status bar, and menu bar. The decorated window also contains a PegWindow child in the decorated window client area. The full source code for this window can be found in the file `\peg\examples\pegdemo\pegdemo.cpp`.



4.4 PegDialog

4.4.1 Overview

PegDialog is a PegDecoratedWindow class with added features to support modal and nonmodal dialog window execution.

Dialog windows usually draw themselves with a different frame and client color than other windows. Dialog windows may be executed as modal windows by calling their member function `Execute()`.

Dialog windows attach special significance to buttons with the following IDs:

```
IDB_CLOSE
IDB_OK
IDB_CANCE
L
IDB_ABORT
IDB_RETRY
IDB_YES
IDB_NO
IDB_APPLY
```

These button IDs are reserved by PEG and are found in the header file `pegtypes.hpp`. When a button with one of the above ID values is selected, the dialog will close. When defining a dialog window, you must ensure that at least one button added to the dialog is constructed with one of the above button ID values. Otherwise, the only way to close the dialog will be via program intervention.

When any of the buttons listed above is selected, the dialog window will send a `PM_DIALOG_NOTIFY` message to its parent window (or its 'ReportTo' window) to indicate that the dialog has been completed. The message iData member will contain the ID of the button that caused the dialog to close. In all cases except `ID_APPLY`, the dialog will close after the parent window has received the `PM_DIALOG_NOTIFY` message.

In the case of a modal dialog, `Execute()` will return when any of these messages are received, and the return value will be the ID of the button that caused the dialog to close.

In the case that the user selects the `IDB_APPLY` button, the dialog will send the `PM_DIALOG_NOTIFY` message to its parent without closing.

4.4.2 See Also

PegWindow

PegDecoratedWindow

PegMessageWindow

PegMLMessageWindow

4.4.3 Style Flags

PegDialog supports the following style flags:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame

4.4.4 Derivation

PegDialog derives from PegDecoratedWindow.

4.4.5 Constructors:

```
PegDialog(const PegRect &Rect, PEGINT TitleStringId =
      0, PegThing *pReportTo = NULL, PEGULONG Style =
      FF_RAISED)

PegDialog(const PegRect &Rect, const PEGCHAR *pText,
      PegThing *pReportTo = NULL, PEGULONG Style =
      FF_RAISED)

PegDialog(PEGINT TitleStringId = 0, PegThing
      *pReportTo = NULL, PEGULONG Style = FF_RAISED)

PegDialog(const PEGCHAR pText, PegThing *pReportTo =
      NULL, PEGULONG Style = FF_RAISED)
```

There are four constructors available for PegDialog. The first two define the window size and position at the time the window is created. The second two

require that the window size and position be determined after the window is constructed, but before the window is displayed.

4.4.6 Public Functions:

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

PegDialog overrides the `Message()` function to catch signals sent by one or more of the reserved button IDs that cause the dialog to close or report to its owner window.

4.4.7 Examples:

The following is a PegDialog window. The full source code for this window can be found in the file `\peg\examples\pegdemo\pegdemo.cpp`.

4.5 PegEditBox

4.5.1 Overview

PegEditBox is a multi-line text display control that allows full user editing via mouse and keyboard. PegEditBox is derived from PegTextBox and therefore supports all of the functionality of this base class.

PegEditBox is more complex, and therefore requires a larger code size, than PegTextBox. If you do not need user-editing capability you should use PegTextBox to display multi-line text rather than PegEditBox.

4.5.2 See Also

PegPrompt

PegEditField

PegWindow

PegTextThing

PegTextBox

4.5.3 Style Flags

PegEditBox supports the following styles:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame
EF_EDIT	When this style is applied, the user can edit the PegTextBox object. If this style is applied the PegTextBox object automatically includes the TT_COPY style.
EF_WRAP	When this style is applied, the text box will wrap long lines to prevent them from being clipped.

`TT_COPY`	Instructs the PegEditBox to copy the text string assigned. This flag should be used when the text string assigned to the PegEditBox is created dynamically using temporary storage

4.5.4 Signals

In addition to the common signals defined by PegThing, PegEditBox supports the following signals:

```
PSF_TEXT_SELECT    // sent when the user selects text
PSF_TEXT_EDIT   // sent each time text is modified
PSF_TEXT_EDITDONE   // sent when a text modification is complete
```

4.5.5 Derivation

PegEditBox is derived from PegTextBox.

4.5.6 Constructors:

```
PegEditBox(const PegRect &Rect, PEGUINT StringId = 0,
      PEGUSHORT Id = 0, PEGUSHORT Style = FF_RECESSED|
      EF_EDIT|EF_WRAP, PEGUINT MaxChars = 1000)

PegEditBox(const PEGCHAR *pText, const PegRect &Rect,
      PEGUSHORT Id = 0, PEGUSHORT Style = TT_COPY|
      FF_RECESSED|EF_EDIT|EF_WRAP, PEGUINT MaxChars =
      1000)
```

This constructor creates a PegEditBox. `MaxChars` is the maximum number of characters that the text box will be required to support.

4.5.7 Public Functions:

```
virtual void Append(PEGINT StringId)

virtual void Append(const PEGCHAR *pText)
```

This function appends the indicated text to the current text box string value. The edit cursor is automatically positioned at the end of the appended text.

```
void CopyToScratchPad(void)
```

Copies the currently-selected text to the scratch pad.

```
virtual void DataSet(PEGINT StringId)

virtual void DataSet(const PEGCHAR *pText)
```

PegEditBox overrides the `DataSet` function to reset any in-progress string mark or edit operations.

```
void DeleteMarkedText(void)
```

Deletes the currently-selected text.

```
virtual void Draw(const PegRect &Invalid)
```

PegEditBox overrides the `Draw()` function to display the text box border and text.

```
CURSOR_POS GetCursorRowCol(void)
```

This function returns a `CURSOR_POS` structure that contains the x and y coordinates of the cursor. The y-coordinate represents the line number, and the x-coordinate represents the number of characters from the beginning of the line in which the cursor resides.

```
void HomeCursor(void)
```

This function moves the edit cursor to the column 0 position on the current line.

```
PEGBOOL InEditMode(void)
```

Returns TRUE if the PegEditBox is in edit mode.

```
virtual void InsertCharAtCursor(PEGUINT Key)
```

This function inserts a single character at the cursor position, unless the maximum character limit is reached.

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

PegEditBox overrides the `PegThing::Message` function to handle mouse and keyboard events.

```
void PasteFromScratchPad(void)
```

Pastes the scratch pad text at the current cursor position.

```
void SetCursorRowCol(PEGINT Index)
```

This function positions the edit cursor at the character indicated by `Index`, which is the desired character offset from the start of the overall text string. The desired character offset can be determined by using the member

functions of PegTextBox. The text box automatically scrolls such that the indicated cursor position is visible within the editbox window client area.

4.5.8 Protected Members

```
virtual PEGBOOL AutoScrollLeft(void)
```

This function is used when the editbox scrolls horizontally. Any movement of the cursor might require the editbox to scroll to the left, so this function checks to see if that needs to happen.

```
virtual PEGBOOL AutoScrollUp(void)
```

This function is used when the editbox scrolls vertically. Moving the cursor might require the editbox to scroll upward, so this function checks to see if that needs to happen.

```
virtual PEGBOOL CheckAutoScrollDown(void)
```

This function is used when the editbox scrolls vertically. Moving the cursor or adding characters might require the editbox to scroll downward, so this function checks to see if that needs to happen.

```
virtual PEGBOOL CheckAutoScrollRight(void)
```

This function is used when the editbox scrolls horizontally. Moving the cursor or adding characters might require the editbox to scroll to the right, so this function checks to see if that needs to happen.

```
virtual PEGBOOL CheckControlKey(PEGUINT Key, PEGINT
     Ctrl)
```

This function checks to see if the Shift key was held with a directional key. If so, it marks a section of the text, depending on which direction was pressed. This function is only available if `PEG_KEYBOARD_SUPPORT` is turned on.

```
virtual void CheckMarkMove(void)
```

This function is called when a user marks some text. It checks to see if it is necessary to scroll up or down to mark/unmark a line.

```
virtual void DrawCharAtCursor(PEGINT Key)
```

This function draws a single character at the current cursor position.

```
virtual void DrawCursor(void)
```

This function draws a vertical line at the current cursor location to indicate where the user may input characters.

```
virtual void DrawMarkedText(PEGINT Line, PegPoint
     PutPoint)
```

This function draws a single line of text that has at least some of its characters marked. Note that the entire line is drawn, not just the marked section.

```
virtual void DrawOneLine(PEGINT Line)
```

This function draws a single line of text.

```
virtual void DrawSelectText(PEGINT Line, PEGINT First,
     PEGINT Last, PEGBOOL Sel)
```

This is called when the user is in the process of selecting characters. It improves performance over simply redrawing the entire line.

```
virtual void DrawTextLine(PEGINT Line, PegPoint
     PutPoint, PEGBOOL Fill = FALSE)
```

PegEditBox overrides `PegTextBox::DrawTextLine` in order to make sure that marked text is drawn correctly.

```
virtual void ExitEditMode(void)
```

This function is called when the editbox is hidden or loses focus. It takes the editbox out of edit mode and therefore removes the cursor.

```
virtual const PEGCHAR *GetCharAtCursor(void)
```

This function returns the character that is at the current cursor location.

```
virtual void GetCursorPointFromRowCol(void)
```

Calculates the x and y coordinates of the cursor from the row/column cursor point.

```
virtual void GetCursorRowColFromClick(PegPoint Where)
```

Finds the nearest valid location to place the cursor when a user clicks inside it.

```
virtual void GetMarkStartAndEnd(CURSOR_POS
     *pStartMark, CURSOR_POS *pEndMark)
```

Returns the beginning and ending positions of the marked text. The positions are in row/column format.

```
virtual PEGBOOL InsertKey(PEGUINT Key)
```

This function is the general key handler for the PegEditBox. All keys including character keys and directional keys are handled here.

```
CURSOR_POS mCursor
```

The location of the cursor in row/column format.

```
PegPoint mCursorPos
```

The location of the cursor in (x, y) format.

```
virtual void RemoveCharAtCursor(PEGBOOL BackSpace)
```

This function removes the character at the current cursor location. If `BackSpace` is TRUE, it removes the character behind the cursor. Otherwise, it removes the character in front of the cursor.

```
virtual void RemoveCursor(void)
```

This function removes the cursor from the display.

```
virtual void ReplaceMarkedText(PEGUINT Key)
```

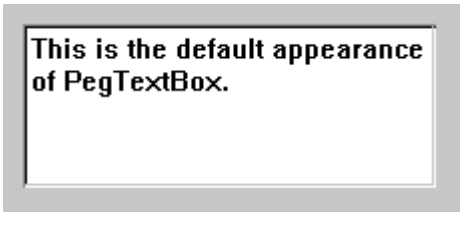
This function is called when the user presses a key after having marked some text. The marked text is completely replaced by the character that was typed.

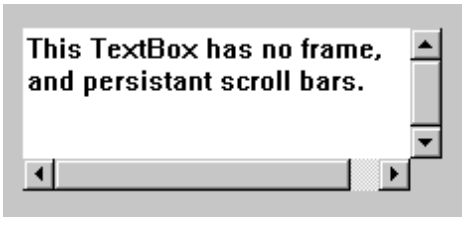
```
virtual void SetCursorRowColFromIndex(PEGINT Index)
```

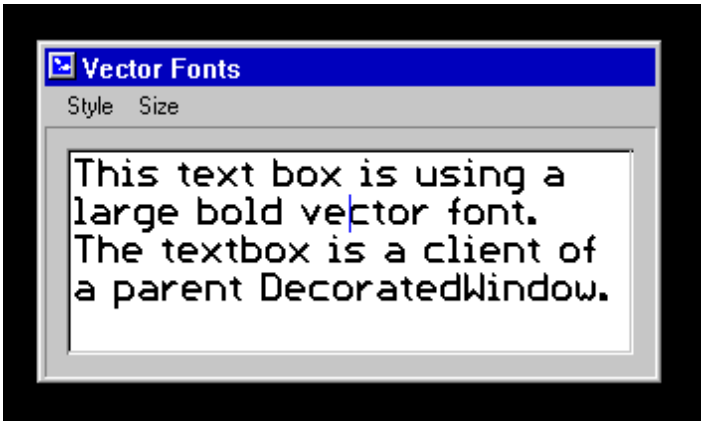
This function positions the cursor at the character indicated by `Index`, which is the desired character offset from the start of the overall text string.

4.5.9 Examples:

The following are each different styles of PegEditBox:







The complete source for the last example above can be found in the example program `\peg\examples\vecfont` distributed with your PEG release.

4.6 PegFileDialog

4.6.1 Overview

PegFileDialog is a utility class that supports the standard notion of browsing a directory structure for a file name in the 'File Open' and 'File Save As' types of scenarios. Currently, the file system routines uses the generic PegFile class (which gives it a degree of portability), as well as a few calls that are specific to the Linux API (out of necessity).

The PegFileDialog was created out of a need for the PEG utility programs (i.e. Window Builder, et al) to be able to run on the X Window System on top of Linux. Since it has proven very useful to us, we have included it in the PEG library to meet the needs of application developers who are creating applications that require such functionality.

The class has all of the basics that make it useful, but it does not, at the present time, have all of the luxuries that make it comparable to most desktop implementations of the same types of dialogs.

It is important to note that the PegFileDialog does not actually change the current working directory when the user navigates the file system. The file routines simply keep track of the current directory and use this for finding the files and directories within that directory. Therefore, you don't need to worry about the dialog changing your application's present working directory.

4.6.2 See Also

PegDialog

4.6.3 Style Flags

PegFileDialog supports the standard frame styles implemented in the PegDialog class.

4.6.4 Signals

See PegDialog for a description of which signals are sent.

4.6.5 Derivation

PegFileDialog is derived from PegDialog.

4.6.6 Constructors:

```
PegFileDialog(const PEGSTRINGID TitleId, PEGINT Left =
      -1, PEGINT Top = -1)
```

The constructor takes a constant `PEGSTRINGID` parameter that will be used in the title of the dialog. It is usually a good idea to give the dialog box a title that corresponds to the action that the user is performing (i.e. 'File Open').

The `Left` and `Top` parameters are needed if you would like to place the dialog at a specific location on the screen. If you allow these parameters to retain their default values, the dialog box will be centered on the screen. Currently, it is not a good idea to assign the dialog box a size, since the placement of the controls on the dialog expects the dialog to be a specific size.

4.6.7 Public Functions:

```
virtual PEGINT Execute(void)
```

PegFileDialog overrides the `Execute` method in order to ensure that the file name buffer has been set up properly.

```
PEGINT GetOperation(void)const
```

This function returns the type of operation being performed. This method will return `PFD_FILEOPEN` or `PFD_FILESAVEAS`.

```
PEGINT Message(const PegMessage& Mesg)
```

The `Message` method is overridden so that the dialog can catch the signals sent by the buttons on the toolbar as well as by the buttons that control directory and file name sort order.

```
void SetDefNewExt(char* pExt)
```

This function sets the default filename extension for the dialog.

```
void SetFilter(const char* pFilter)
```

This function sets a filter so that only files matching the filter will be displayed in the dialog.

```
PEGBOOL SetMode(PEGINT Operation, PEGCHAR *pBuffer,
      PEGUINT BuffSize, const char *pStartNode =
      NULL))
```

Generally, this is the method that you would call after you have constructed the object. The `Operation` parameter can be one of the following:

```
PFD_FILEOPEN
PFD_FILESAVEA
S
```

At the present time, the dialog does not behave differently based on this setting. But, for forward compatibility, it is best to specify the action you intend.

The second parameter is the node in the directory structure where you would like the dialog to start. In other words, the dialog will take this as its present working directory. If running on Linux, you may want to start the user out in his or her home directory. This can be accomplished like this:

```
passwd *pPasswd = getpwuid(getuid());
```

Then passing the

```
pPasswd->pw_dir
```

over as the `pStartNode` parameter.

The remaining parameters refer to a preallocated `PEGCHAR` buffer into which the selected file name is to be returned. If the user cancels the dialog box, the value of `pBuffer` upon return is undefined.

Once you have set these parameters, you would then call the `Execute` method and inspect the return value. If you receive a return value of `IDB_OK`, then the full qualified name of the file (assuming the buffer did not overrun) would be in `pBuffer`. If you receive a return value of `IDB_CANCEL`, then the user canceled their selection and `pBuffer` is undefined.

```
static int SortDirList(const void *, const void *)
```

This is a callback function used by qsort to sort the directory names.

```
static int SortFileList(const void *, const void *)
```

This is a callback function used by qsort to sort the file names.

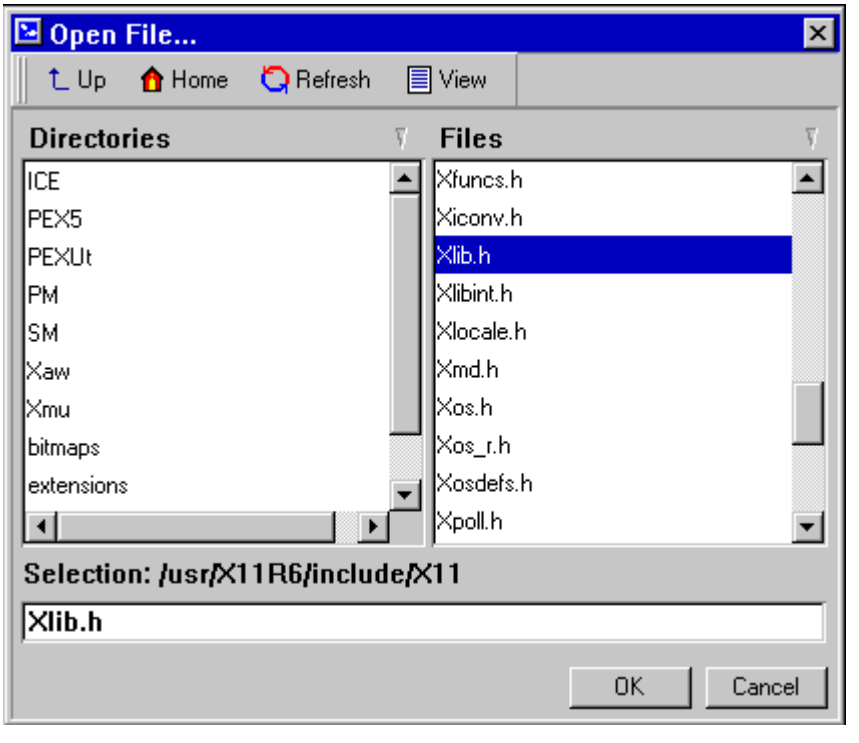
4.6.8 Examples:

The following code snippet exemplifies how to create and use a PegFileDialog object. Usage is very simple. All you need to do is create a new instance of the dialog, allocate a buffer of sufficient size, set its options, execute the dialog, and inspect the return value.

```
PEGCHAR Buffer[256];
PEGUINT BufSize = 256;

PEGINT RetVal;

PegFileDialog* pFD = new PegFileDialog("Open File...");

// Get the current user's home directory, and
// set this as the top node for passing to the
// file dialog
passwd *pPasswd = getpwuid(getuid());

if (pFD)
{
    if (pPasswd)
    {
        pFD->SetOptions(PFD_FILEOPEN, pPasswd->pw_dir,
Buffer,
            BufSize);
    }
    else
    {
        pFD->SetOptions(PFD_FILEOPEN, "/work", Buffer,
BufSize);
    }

    RetVal = pFD->Execute();

    // Check the return value from executing the dialog.
    if (RetVal == IDB_OK)
    {
        // The user selected a valid file name, so Buffer
will
        // have the full qualified path of the file the user
        // selected.
    }
```

```
    else if (RetVal == IDB_CANCEL)
    {
        // The user canceled their selection
    }
    else if(RetVal == PFD_ERROR)
    {
        // The buffer size was too small to hold the file
name
    }
}
```

Below is a screen shot of the PegFileDialog in action. You'll notice the various controls on the dialog that assist the user in selecting a file name. We'll briefly discuss these controls so that you can better understand the capabilities of the PegFileDialog class.



Starting with the toolbar, the button labeled ‘Up’ shifts the present working directory up, if possible. The ‘Home’ button changes the present working directory to the current user's home directory. The ‘Refresh’ button causes the Directory and Files lists to be updated. And the ‘View’ button toggles the

directories and files displayed in the lists. If the view is on full, then all of the hidden files and directories (i.e. the ‘dot’ files) are displayed; otherwise, they are not shown. When the dialog starts up, the view defaults to hiding these directories.

The Directories and Files lists are fairly straightforward. The lists can be independently sorted in ascending or descending order by the user clicking on the Directories or Files label. You'll note the small triangles in the far right of either label. These denote the sort order.

The ‘Selection’ label displays the current working directory. Once a file name has been selected from the Files list, the file name is entered in the edit field at the bottom. If the current working directory is changed, this file name is cleared from the edit field control. It is important to note that the PegFileDialog verifies that the file exists before it allows a file to be entered here.

The OK button closes the dialog box with a return value of `IDB_OK`. The file name is verified and put into the buffer that the application designer provided in the `SetOptions` method. If the buffer is not large enough to hold the entire file name, then `PFD_ERROR` is returned instead of `IDB_OK`. If the Cancel button is pushed, then the dialog simply exits.

4.7 PegHorzList

4.7.1 Overview

PegHorzList is a container class for displaying a scrolling list of child objects. PegHorzList automatically positions and sizes child objects. It is therefore not necessary to manually position objects before adding them to PegHorzList.

The LAST child added to the list will be displayed at the leftmost position in the list if the `Add()` function is used to add children. The order of display can be reversed by using the function `AddToEnd()` to add children to the list.

Child objects are positioned when the list receives the `PM_SHOW` message, which is a system message sent automatically when the list is first displayed. PegHorzList sets the height of each child object to fit within the horizontal list client area. The widths of child objects are not modified, and should be set as desired when each child object is constructed.

4.7.2 See Also

`PegVertList`

`PegList`

`PegWindow`

4.7.3 Style Flags

PegHorzList supports the following style flags:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame
`FF_RAISED`	Raised 3D Frame
`FF_RECESSED`	Recessed 3D Frame
`FF_THICK`	Thick 3D Frame

The styles for PegHorzList are identical to the PegWindow styles. In addition, scrolling is enabled in PegList in the same way as in PegWindow, by using the `SetScrollMode()` function.

4.7.4 Signals

PegHorzList sends `PSF_LIST_SELECT` signals to the parent object. This message contains:

`Message.pSource` = Pointer to selected object in list.
`Message.Param` = ID of selected list item.
`Message.pTarget` = Pointer to list parent object.

4.7.5 Derivation

PegHorzList derives from PegList.

4.7.6 Constructors:

```
PegHorzList(const PegRect &Rect, PEGUINT Id = 0,
     PEGULONG Style = FF_THIN)
```

This constructor creates a PegHorzList object. The `Rect` parameter determines the position and size of the list. The list children are automatically positioned by the list object.

4.7.7 Public Functions:

None.

4.7.8 Protected Members

```
virtual void PositionChildren(void)
```

This function calculates the position of all the child objects in the list.

4.7.9 Examples:

The following is a PegHorzList with PegTextButton children:

The following example creates a PegHorzList and adds several PegIconButton children. The bitmaps for the icon buttons can be generated using PegImageConvert.

```
void MyWindow::AddHList(void)
{
    PegRect Rect;
    Rect.Set(10, 10, 180, 44);

    PegHorzList *pList = new PegHorzList(Rect);

    Rect.Set(0, 0, 34, 34);
    pList->Add(new PegIconButton(Rect, BID_THUNDER));
    pList->Add(new PegIconButton(Rect, BID_LIGHT));
    pList->Add(new PegIconButton(Rect, BID_SATELLITE));
    pList->Add(new PegIconButton(Rect, BID_DYNAMITE));
    pList->Add(new PegIconButton(Rect, BID_APPLE));

    pList->SetScrollMode(WSM_HSCROLL);
    Add(pList);
}
```

4.8 PegList

4.8.1 Overview

PegList is a container class that serves as a base class for PegVertList and PegHorzList. PegList positions child objects so that they are stacked left to right or top to bottom. You would not normally create an instance of PegList in your system software. However, several of the member functions are important when working with derived PegHorzList and PegVertList classes.

The LAST child added to the list will be displayed at the leftmost or topmost position in the list if the `Add()` function is used to add children. The order of display can be reversed by using the function `AddToEnd()` to add children to the list.

Child objects are positioned when the list receives the `PM_SHOW` message, which is a system message sent automatically when the list is first displayed. The position of children added to a list object may be any value, including 0,0, as the list object will reposition objects to fit within the list client area.

For vertical lists, child object's widths are also forced to match the list client width. A child object's height is not modified, so this value is meaningful when child objects are constructed.

Likewise for horizontal lists, child object's heights are forced to fit within the list client area. Child object widths are not modified, and so this value should be set as desired when list children are constructed.

4.8.2 See Also

`PegVertList`

`PegHorzList`

`PegWindow`

4.8.3 Style Flags

PegList supports the following style flags:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame
LS_WRAP_SELECT	Wrap selection at top or bottom

The styles for PegList are identical to the PegWindow styles. In addition, scrolling is enabled in PegList in the same way as in PegWindow, by using the `SetScrollMode()` function.

4.8.4 Signals

PegList sends `PSF_LIST_SELECT` signals to the parent object. This message contains:

`Message.pSource` = Pointer to selected object in list.
`Message.Param` = ID of selected list item.
`Message.pTarget` = Pointer to list parent object.

4.8.5 Derivation

PegList derives from PegWindow.

4.8.6 Constructors:

```
PegList(const PegRect &Rect, PEGUINT Id = 0, PEGULONG
      Style = FF_THIN)
```

This constructor creates a PegList object. The `Rect` parameter determines the position and size of the list. The list children are automatically positioned by the list object.

4.8.7 Public Functions:

```
virtual void Add(PegThing *pWhat, PEGBOOL Show = TRUE)
```

This function adds `pWhat` to the head of the list. If `Show` is set to TRUE, the object will be shown after being added.

```
virtual void AddToEnd(PegThing *pWhat, PEGBOOL Show =
     TRUE)
```

This function adds `pWhat` to the end of the list. If `Show` is set to TRUE, the object will be drawn after being added.

```
virtual PEGINT Clear(void)
```

This method removes and destroys all of the child objects in the list, so use it with care. The return value is the number of child items that were removed from the list, excluding scroll bars, if present.

```
virtual PEGINT GetIndex(PegThing *pWho)
```

This function returns the index of the list item pointed to by `pWho`. If `pWho` is not a child of the list, this function returns -1.

```
PEGINT GetNumItems(void)
```

This method returns the total number of child items in the list. The return value is the total number of items, excluding scroll bars or other non-client objects, if present.

```
virtual PegThing *GetSelected(void)
```

This function returns the address of the list child that was last selected.

```
virtual PegThing *GetThing(PEGINT Index)
```

This function returns the list child with the specified index.

```
virtual PEGINT GetSelectedIndex(void)
```

This function returns the list index of the list child that was last selected.

```
virtual PegThing *GetThing(PEGINT Index)
```

This function returns a pointer to the child object with the specified index.

```
virtual void Insert(PegThing *pWhat, PEGINT Where,
     PEGBOOL Select = TRUE, PEGBOOL Show = TRUE)
```

This function inserts an object into the specified location in the list. If the `Select` flag is turned on, the inserted item will be selected and scrolled into view.

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

PegList catches the `PM_SHOW` message.

```
PegThing *PageDown(void)
```

This function scrolls the PegList child items by one full page, meaning that the first nonvisible item at the bottom or right of the list client area is scrolled into view and selected. This function returns a pointer to the newly selected item. This function is called by the `PegList::Message` function in response to the `PK_PGDN` key message.

```
PegThing *PageUp(void)
```

This function scrolls the PegList child items by one full page, meaning that the first nonvisible item at the top or left of the list client area is scrolled into view and selected. This function returns a pointer to the newly selected item. This function is called by the `PegList::Message` function in response to the `PK_PGUP` key message.

```
virtual PegThing *Remove(PegThing *pWhat)
```

This function removes `pWhat` from the list. If it is not in the list, the function doesn't do anything.

```
PegThing *SelectNext(void)
```

This function can be called to force the list to advance to the next child item. The list will automatically scroll the newly selected child into view. If `LS_WRAP_SELECT` is enabled, the list will wrap to the top item if `SelectNext()` is called when the bottom item is selected.

```
PegThing *SelectPrevious(void)
```

This function can be called to force the list to back up to the previous child item. The list will automatically scroll the newly selected child into view. If `LS_WRAP_SELECT` is enabled, the list will wrap to the bottom item if `SelectPrevious()` is called when the top item is selected.

```
virtual void SetSelected(PegThing *pWho)
```

This function can be called to force the 'last selected' member to a certain child under program control. The desired child object is pointed to by `pWho`.

```
virtual PegThing *SetSelected(PEGINT Index)
```

This function can be called to force the 'last selected' member to the child item referred to by `Index`.

```
void SetSeparation(PEGINT Val)
```

This function sets the separation amount, in pixels, between child items. The PegList-derived class will always position child items such that they are adjacent to each other, and, by default, there will be no separation between

child items. This function can be called after the list is constructed but BEFORE the list is displayed to place any amount of space between child items.

4.8.8 Protected Members

```
virtual void PositionChildren(void)
```

This pure virtual function is used by derived classes to calculate the position of its child objects.

4.9 PegMessageWindow

4.9.1 Overview

PegMessageWindow is a popup window class for display warning, error, or other status information to the user.

The PegMessageWindow class provides a quick way to display information messages. PegMessageWindow may contain a title bar, message line, bitmap, or miscellaneous buttons.

PegMessageWindow supports both modal and nonmodal execution. In addition, the signal generated when the message window is closed by the user may be directed to any top-level window.

Modal execution is achieved by calling the PegMessageWindow `Execute()` function. `Execute()` will add the message window to PegPresentationManager if the window has no parent at the time `Execute()` is called. `Execute()` will not return until the user selects one of the MessageWindow option buttons. `Execute()` will return the ID of the option button selected to close the message window.

Several button ID values are reserved by PEG for use with PegMessageWindow (and PegDialog). These ID values correlate to the common options presented on a message window. Additional options may be presented by deriving from and extending the PegMessageWindow class. The buttons included on the message window are specified by the message window style flags. There is one style flag for each of the predefined message window buttons.

4.9.2 See Also

PegWindow

PegDialog

PegMLMessageWindow

4.9.3 Style Flags

PegMessageWindow supports the following style flags:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame
MW_OK	This option displays an OK button on the message window.
MW_YES	This option displays a YES button on the message window.
MW_NO	This option displays a NO button on the message window.
MW_ABORT	This option displays an ABORT button on the message window.
MW_RETRY	This option displays a RETRY button on the message window.
MW_CANCEL	This option displays a CANCEL button on the message window.

4.9.4 Derivation

PegMessageWindow derives from PegWindow.

4.9.5 Constructors:

```
PegMessageWindow(const PegRect &Rect, PEGINT
     TitleStringId, PEGINT MessageId = 0, PEGULONG
     Style = MW_OK|FF_RAISED, PEGINT IconId = 0,
     PegThing *pOwner = NULL)

PegMessageWindow(PEGINT TitleStringId, PEGINT
     MessageId = 0, PEGULONG Style = MW_OK|FF_RAISED,
     PEGINT IconId = 0, PegThing *pOwner = NULL)

PegMessageWindow(const PEGCHAR *pTitle, const PEGCHAR
     *pMessage, PEGULONG Style = TT_COPY|MW_OK|
     FF_RAISED, PEGINT IconId = 0, PegThing *pOwner =
     NULL)
```

There are three constructors available for PegMessageWindow. The first constructor allows you to specify the overall message window size. This constructor is used in cases where you would like to add additional decorations to the message window.

When the second or third constructor is used, the message window calculates the required height and width of the window in order to fit the message text and all specified option buttons. These constructors are most commonly used.

The `IconId` parameter allows you to specify a bitmap that will be displayed to the left of the text message.

The `pOwner` pointer allows you to specify a window that should receive a `PM_MWCOMPLETE` message. This is only useful when the message window is not executed modally. When the window is executed modally, the `Execute()` function returns the ID of the button used to close the message window.

4.9.6 Public Functions:

```
virtual void DataSet(PEGINT MessageId)

virtual void DataSet(const PEGCHAR *pText)
```

PegMessageWindow overrides the `DataSet()` function so that it can resize itself and re-layout the buttons based on the size of the text.

```
virtual void Draw(const PegRect &Invalid)
```

PegMessageWindow overrides the `Draw()` function to display the message text and the bitmap, if specified, in the window client area.

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

PegMessageWindow catches option button signals to close the message window.

```
void SetTitle(PEGINT StringId)

void SetTitle(const PEGCHAR *pText)
```

This function is used to modify the text displayed on the PegTitle child object.

4.9.7 Examples:

The following is a PegMessageWindow with OK, CANCEL, and RETRY buttons.

The following function creates and modally executes the above message window:

```
void MyWindow::ModalMessage(void)
{
    PegMessageWindow *pWin = new PegMessageWindow(
        "Example Message Window",
        "This is a message window with a raised frame.",
        MW_OK|MW_CANCEL|MW_RETRY|FF_RAISED);

    Presentation()->Center(pWin);
    Presentation()->Add(pWin);
    pWin->Execute();
}
```

4.10 PegMLMessageWindow

4.10.1 Overview

PegMLMessageWindow is a popup window class for display warning, error, or other status information to the user. The basic behavior of this class is identical to the PegMessageWindow class, except that this class allows the display of multiple lines of text in the message.

The PegMLMessageWindow class provides a quick way to display information messages. PegMLMessageWindow may contain a title bar, several message lines, and miscellaneous buttons.

PegMLMessageWindow supports both modal and nonmodal execution. In addition, the signal generated when the message window is closed by the user may be directed to any top-level window.

Modal execution is achieved by calling the message window `Execute()` function. `Execute()` will add the message window to PegPresentationManager if the window has no parent at the time `Execute()` is called. `Execute()` will not return until the user selects one of the message window option buttons. `Execute()` will return the ID of the option button selected to close the message window.

Several button ID values are reserved by PEG for use with PegMLMessageWindow (and PegDialog). These ID values correlate to the common options presented on a message window. Additional options may be presented by deriving from and extending the PegMLMessageWindow class. The buttons included on the message window are specified by the message window style flags. There is one style flag for each of the predefined message window buttons.

4.10.2 See Also

PegDialog

PegMessageWindow

4.10.3 Style Flags

PegMLMessageWindow supports the following style flags:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame
MW_OK	This option displays an OK button on the message window.
MW_YES	This option displays a YES button on the message window.
MW_NO	This option displays a NO button on the message window.
MW_ABORT	This option displays an ABORT button on the message window.
MW_RETRY	This option displays a RETRY button on the message window.
MW_CANCEL	This option displays a CANCEL button on the message window.

4.10.4 Derivation

PegMLMessageWindow derives from PegWindow.

4.10.5 Constructors:

```
PegMLMessageWindow(const PegRect &Rect, PEGUINT
      TitleStringId, PEGUINT MessageId = 0, PEGULONG
      Style = MW_OK|FF_RAISED, PEGINT IconId = 0,
      PegThing *pOwner = NULL)

PegMLMessageWindow(PEGUINT TitleStringId, PEGUINT
      MessageId = 0, PEGULONG Style = MW_OK|FF_RAISED,
      PEGINT IconId = 0, PegThing *pOwner = NULL)

PegMLMessageWindow(const PEGCHAR *pTitle, const
      PEGCHAR *pMessage, PEGULONG Style = MW_OK|
      FF_RAISED, PEGINT IconId = 0, PegThing *pOwner =
      NULL)
```

There are three constructors available for PegMLMessageWindow. The first constructor allows you to specify the overall message window size. This constructor is used in cases where you would like to add additional decorations to the message window.

If the size of the window is specified with a PegRect and the message text is too large to fit in the available space, the sizing algorithm will grow the height of the rectangle until all of the text fits properly. The left, top, and right members of the specified rectangle will not change, only the bottom.

When the second or third constructor is used, the message window calculates the required height and width of the window in order to fit the message text and all specified option buttons. The second and third constructors are most commonly used.

The `IconId` parameter allows you to specify a bitmap that will be displayed to the left of the text message.

The `pOwner` pointer allows you to specify a window that should receive a `PM_MWCOMPLETE` message. This is only useful when the message window is not executed modally. When the window is executed modally, the `Execute()` function returns the ID of the button used to close the message window.

4.10.6 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegMLMessageWindow overrides the `Draw()` function to display the message text and bitmap, if specified, in the window client area.

```
PegTextBox *GetTextBox(void)
```

Returns a pointer to the text box displaying the message. This is used to edit any style parameters of the text box.

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

PegMLMessageWindow catches option button signals to close the message window.

4.10.7 Examples:

For an example of using this class, please see the example in the PegMessageWindow documentation.

4.11 PegNotebook

4.11.1 Overview

PegNotebook is a PegWindow-derived class for displaying and using a tabbed notebook-style control. The notebook can have any number of tabs, and each notebook tab is associated with a different notebook page. Each notebook page displays any user defined group of objects.

Each notebook tab can contain either simple text or any user-defined object type. Text tabs use slightly less memory, while user defined tab decorations can give the notebook control a customized appearance.

Regardless of tab type, the tabs can be displayed at the top or bottom of the notebook window.

Constructing and displaying PegNotebook requires the following steps:

- Construct the PegNotebook control, passing the number of notebook tabs and the style of the notebook tabs. For text-only tabs, include the `NS_TEXTTABS` style. For custom tabs, do not include the `NS_TEXTTABS` style.
- Populate each notebook tab with either text or custom objects. This determines what is displayed on each notebook tab.
- Populate each page of the notebook with a user-defined window or group. This determines what will be displayed on each notebook page as the tabs are selected. There can be only one child object on each notebook page. To display a group of objects, a container such as a borderless PegWindow must be created to hold the subobjects of the page. This window is then populated with the desired group of child objects, and its is set as the notebook client object.

4.11.2 See Also

`PegWindow`

4.11.3 Style Flags

PegNotebook supports the following style flags:

`FF_RAISED`	Raised 3D Frame

FF_RECESSED	Recessed 3D Frame
NS_TOPTABS	The notebook tabs appear above the notebook pages.
NS_BOTTOMTABS	The notebook tabs appear below the notebook pages.
NS_TEXTTABS	The notebook tabs are simple text, not custom objects.

4.11.4 Signals

In addition to the common signals defined by PegThing, PegNotebook sends the PSF_PAGE_SELECT signal when a new notebook page is selected. The message contains the following data:

Message.pSource = Pointer to notebook window.
Message.Param = ID of notebook window.
Message.ExtParams[0] = Page index of selected page.

4.11.5 Derivation

PegNotebook derives from PegWindow.

4.11.6 Constructors:

```
PegNotebook(const PegRect &Rect, PEGULONG Style,
     PEGUBYTE NumTabs)
```

The PegNotebook constructor accepts a PegRect defining the notebook position and size, a style value, and the number of tabs that the notebook will initially display. The number of tabs can be modified at run time.

4.11.7 Public Functions:

```
void CalculateTabPositions(void)
```

This function calculates the size and position of each of the tabs based on the font or size of the client objects.

```
virtual void Draw(const PegRect &Invalid)
```

PegNotebook overrides the Draw() function to display the notebook background, border, and tabs.

```
PEGUBYTE GetCurrentPage(void)
```

This function returns the active (displayed) page of the notebook.

```
PegThing *GetPageClient(PEGUBYTE Index)
```

This function returns the top-level client object of the current page.

```
PegThing *GetTabClient(PEGUBYTE Index)
```

This function returns the client object used to decorate the indicated notebook tab. This function returns NULL if the notebook is operating in `NS_TEXTTABS` mode or if no decoration has been defined for the requested tab.

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

PegNotebook catches mouse click messages to test for notebook tab selection. PegNotebook also catches `PM_KEY` messages to allow navigation via a keyboard.

```
PegThing *RemovePageClient(PEGUBYTE Index)
```

This function can be used to remove the client object from any notebook page. This is sometimes useful when the application desires to switch the client object displayed on a notebook page during program run time.

```
void ResetNumTabs(PEGUBYTE Num)
```

This function can be used to reset the number of tabs available on the notebook window after the notebook object has been constructed. If `Num` is greater than the current number of pages, the user should populate the new tabs and notebook client pages after this call. If `Num` is less than the current number of notebook pages, the notebook will delete any page clients for pages that are no longer required.

```
void ResetTabStyle(PEGULONG Style)
```

This function can be used to move the notebook tabs or toggle between text tabs and custom decorated tabs.

```
void SelectTab(PEGUBYTE Tab)
```

This function can be used to change the current tab selection under program control. The current tab is usually selected by the user via mouse, touch screen, or keyboard. If the indicated tab is not visible, the notebook will automatically scroll the selected tab into view.

```
virtual void SetFont(PEGINT FontId)
```

This function can be used to change the font of the text on the notebook tabs. Since the tabs are sized based on the string width of each text string in the selected font, the font can only be changed BEFORE the notebook is displayed.

```
void SetPageClient(PEGUBYTE Index, PegThing
     *pPageClient)
```

This function is called to associate an object with each notebook page. There can be only one top-level client for each notebook page. For this reason, if multiple child objects are to be associated with one or more notebook pages, a borderless container window should be constructed as the notebook page client. The child objects for the notebook page should then be added to this borderless container window.

This function should be called once for each notebook page, setting the page clients for page indexes 0 through (nTabs - 1).

If the `SetPageClient` function is called to link an object with a notebook page that already has a client object, the existing client object is destroyed and replaced with the new child object.

```
void SetTabClient(PEGUBYTE Index, PegThing
     *pTabClient)
```

This function is called to assign the custom decoration which will be displayed on each notebook tab. This function only works if the notebook is NOT configured for `NS_TEXTTABS` mode.

Custom tab decorations can be created by deriving a new custom class from PegThing or from any other PEG base class. An example of this is shown in the PegNotebook example application.

```
void SetTabString(PEGUBYTE Index, PEGINT StringId)

void SetTabString(PEGUBYTE Index, const PEGCHAR
     *pText)
```

This function sets the text string displayed on each notebook tab. This function only operates when the notebook window is configured for `NS_TEXTTABS` mode.

This function should be called once for each notebook tab, setting the text value for tab indexes 0 through (nTabs - 1).

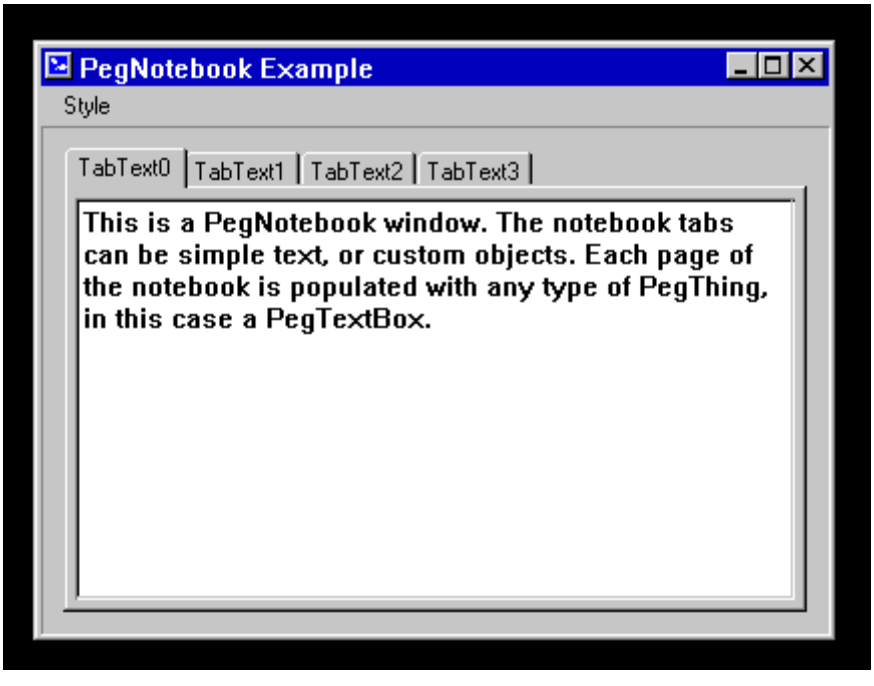
4.11.8 Protected Members

```
virtual void DrawFrame(PEGBOOL Fill = TRUE)
```

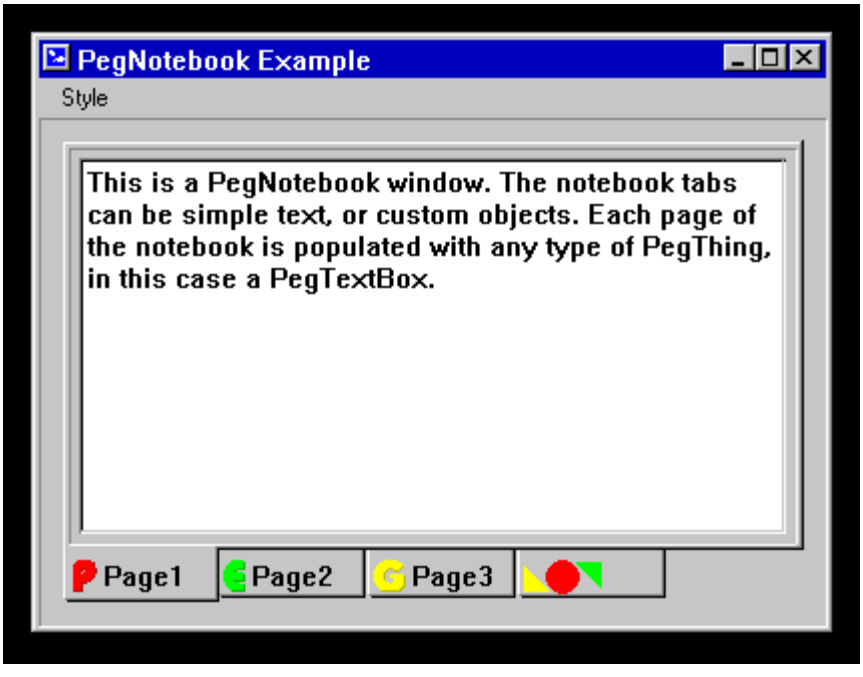
PegNotebook overrides the `PegWindow::DrawFrame` function in order to properly draw the tabs on the top or on the bottom.

4.11.9 Examples:

The following is an example of a PegNotebook window with text tabs, a raised frame, and tabs on top:



The following is an example of a PegNotebook window with custom tabs, a thick frame, and tabs on bottom:



An example application program using PegNotebook is found in the directory `\peg\examples\notebook`.

4.12 PegProgressWindow

4.12.1 Overview

PegProgressWindow is an extension of PegMessageWindow. In this case, a progress bar indicator is added to the message and optional buttons displayed on the message window. This makes it very easy to create and display a message and progress bar to the user during a long operation.

The progress bar that is a child of the progress window is directly updated by the application software. The progress window member function `Bar()` is called to retrieve a pointer to the progress bar when the application determines that the progress bar should be updated.

The progress bar added to a PegProgressWindow always has a scale of 0 to 100. It is up to the application software to prescale the input value accordingly.

The style of the progress bar displayed in the window client area is passed to the PegProgressWindow constructor.

4.12.2 See Also

`PegProgressBar`

`PegMessageWindow`

4.12.3 Style Flags

PegProgressWindow accepts both PegMessageWindow and PegProgressBar styles, and these are passed directly to each base class.

4.12.4 Signals

PegProgressWindow signals are identical to PegMessageWindow signals.

4.12.5 Derivation

PegProgressWindow is derived from PegMessageWindow.

4.12.6 Constructors:

```
PegProgressWindow(const PegRect &Rect, PEGINT
     TitleStringId, PEGINT MessageId, PEGUSHORT
     MesgStyle, PEGUSHORT ProgStyle = FF_THIN|
     PS_SHOW_VAL|PS_PERCENT, PEGINT IconId = 0,
     PegThing *pOwner = NULL)

PegProgressWindow(const PegRect &Rect, const PEGCHAR
     *pTitle, const PEGCHAR *pMessage, PEGUSHORT
     MesgStyle, PEGUSHORT ProgStyle = FF_THIN|
     PS_SHOW_VAL|PS_PERCENT, PEGINT IconId = 0,
     PegThing *pOwner = NULL)
```

The PegProgressWindow constructors are identical to the PegMessageWindow constructors, with the added style value `ProgStyle`. This additional style value is applied to the child PegProgressBar object.

4.12.7 Public Functions:

```
PegProgressBar *Bar(void)
```

This inline function returns a pointer to the child progress bar. This allows the application software to manipulate the child control directly.

4.12.8 Examples:

The following code fragment creates and displays a PegProgressWindow:

```
void MyWin::DisplayProgress(void)
{
    PegRect Rect;
    Rect.Set(10, 10, 190, 140);
    PegProgressWindow *pWin = new PegProgressWindow(Rect,
        "Working....", "Copying Information...", MW_OK|
FF_RAISED);

    Presentation()->Center(pWin);
    Presentation()->Add(pWin);
}
```

4.13 PegRichTextBox

4.13.1 Overview

The PegRichTextBox is responsible for Rich Text support in PegPro library. The PegRichTextBox with additional APIs for Rich Text Support is responsible for displaying formatted text like bold, italic, underline, etc. that are in Rich Text Format. It is responsible for displaying the RTF file.

4.13.2 Style Flags

PegRichTextBox supports the following styles:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame
`FF_RAISED`	Raised 3D Frame
`FF_RECESSED`	Recessed 3D Frame
`FF_THICK`	Thick 3D Frame

4.13.3 See Also

`PegWindow`

4.13.4 Derivation

PegRichTextBox derives from PegWindow.

4.13.5 Constructors:

PegRichTextBox(const PegRect &Rect, PEGUSHORT Id = 0,
PEGULONG Style = FF_RECESSED|EF_WRAP|TJ_LEFT)

This constructor creates a PegRichTextBox. This constructor initializes all member variables.

4.13.6 Public Functions:

virtual void Draw(const PegRect &Invalid)

PegRichTextBox overrides the Draw() function to display the text box border and rich text from the RTF file.

virtual void Resize(const PegRect &Size)
PegRichTextBox overrides the Resize function to resize object properly. This function destroys all parsed information and parses the file again with the new size of Object.

virtual void RtfParser(PEGCT *pFileName)
This function will create the PegFile object open file whose name is in pFileName. Read this file, character by character, and parse it. If there is a '\\' character, then it will call the ParseRtfKeyWord function to parse the RTF keyword and arguments associated with that keyword.

virtual void RtfParser(PEGCHAR *pRtfBuff)
This function will parse the pRtfBuff string and create a link list. If there is a '\\' character, then it will call the ParseRtfKeyWord function to parse the RTF keyword and arguments associated with that keyword.

virtual void SetNumFont(PEGUSHORT number)
This function is used to set how many fonts are used by the application. It will be called by the application. It will create an mpRichFontTable array of size 'number' to hold the font ID.

virtual void SetRichFont(PEGUINT FontId, PEGUINT FontIndex)
This function stores font ID in mpRichFontTable at the FontIndex position.

4.13.7 Protected Members Functions

virtual void AddColorToRTFColorTable(void)
This function retrieves the color ID from values of red, green, and blue. It creates the color node and adds it to the link list RtfColorTable.

virtual void AddDrawObjectNode(void)
This function creates a link list of RichTextNode. It will wrap the parsed string according to the width of the RichtextBox object. This function will fill the structure RichTextNode and form a link list. In this structure will fill text string, point, color, font id, underline flag.

virtual void ApplyPropChange(PEGINT Action, PEGINT param)
This function applies properties to the protected members of a class according to action. If there are properties related to color, it will call the AddColorToRTFColorTable function.

virtual void DrawAllRichText(const PegRect &Invalid)

This function reads link list RichTextNode node by node and draws text according to information in the link list.

virtual const PEGCHAR *FindNextLine(const PEGCHAR *pText, PEGINT MaxWidth, PEGUINT FontId, PEGINT Dir = 1)

This function looks through the pText string and returns a pointer to where the next new line will start.

virtual PEGCOLOR GetColor(PEGUBYTE Red, PEGUBYTE Green, PEGUBYTE Blue)

This function calculates color value from red, green, and blue values passed to it and returns them. This value is as per the color format used by the target.

PEGCOLOR virtual GetColorFromRTFColorTable(PEGUBYTE index)

This function searches for color ID in RtfColorTable with an index number.

virtual void ParseChar(PEGINT Action)

This function will store an action in the array and call the AddDrawObjectNode function.

virtual void ParseRtfKeyWord(void)

This function will parse RTF keywords, their associated arguments, and will call the TranslateKeyword function.

virtual void TranslateKeyword(PEGCT *ketword, PEGINT param, PEGBOOL fParam)

This function finds out keyword in the KeyWordsProp array, which holds supported keyword Information. If the keyword is not found, it will skip it. If the keyword is found in the array, it will call functions according to the keyword type. If the keyword is associated with text properties, it will call ApplayPropChange with param. If it is any special character, it will call the ParseChar function with the action that has to be taken. If the keyword has to be skipped, it will skip it.

4.13.8 Protected Members

char mText[MAX_PARSE_STRING_LEN]

This array holds parsed strings between two control Words.

char *mpFileName

Holds the pointer to the file name.

RichTextNode *mpDrawHome
Pointer to RichTextNode link list.

RtfColorTable *mpColorHome
Pointer to color table.

PEGINT *mpRichFontTable
Pointer to font ID array.

4.13.9 Examples:

The following is an example of a PegRichTextBox display RTF file.

4.14 PegSpreadSheet

4.14.1 Overview

PegSpreadSheet is a row,column matrix for displaying text or bitmaps. PegSpreadSheet is a higher-level construct than the basic PEG window and control types; therefore, there are a large number of functions available for controlling PegSpreadSheet appearance and operation.

PegSpreadSheet allows selection of individual cells, entire rows and columns, or any combination in between. PegSpreadSheet may have any number of column headings, row headings, and column footers. In addition, PegSpreadSheet supports vertical and horizontal scrolling. Specific groups of rows and/or columns may be specified as 'fixed,' in that they do not move when the spreadsheet is scrolled.

PegSpreadSheet columns automatically size themselves based on the width of the objects contained in each cell. The justification of text within each column can also be individually configured.

PegSpreadSheet cells may be configured to be selected individually or in complete rows or columns. Whenever an individual cell is selected by the user, the PegSpreadSheet will send a `PSF_CELL_SELECT` signal to the parent window assuming that notification has been enabled. The message `pSource` pointer will point to the PegSpreadSheet and the message `Point` member will contain the row and column of the selected cell, allowing the application to fully determine which cell has been selected.

When a spreadsheet column is selected, the spreadsheet will send a `PSF_COL_SELECT` signal to the parent window. The message `Param` member will contain the first column number selected, and the message `ExtParams[0]` member will contain the last column number selected. These will be the same value when only one column is selected.

When a spreadsheet row is selected, the spreadsheet will send a `PSF_ROW_SELECT` signal to the parent window. The message `Param` member will contain the first row number selected, and the message `ExtParams[0]` member will contain the last row number selected.

Each column of the spreadsheet has its own set of style flags. These flags control the appearance of each column of cells. These flags are modified through the `SetColStyle()` member function. Likewise, each row of the

spreadsheet has unique style flags, which are set with the `SetRowStyle()` member function.

PegSpreadSheet can be drawn with a flat, raised, or recessed appearance for each cell.

Optional column headers, row headers, and column footers are drawn if desired.

PegSpreadSheet automatically provides horizontal and vertical scrolling capability if the client area is not sufficient to display all cells. If the spreadsheet is resized such that the scroll bars are no longer required, they are automatically removed.

4.14.2 See Also

`PegWindow`

`PegTable`

4.14.3 Style Flags

PegSpreadSheet supports the standard frame styles. In addition, the following style flags affect the overall spreadsheet appearance. Style flags for individual rows and columns are described in the SetRowStyle and SetColumnStyle function descriptions below.

`SS_PARTIAL_COL`	Draws the last visible column, even if partially clipped.
`SS_MULTI_COL_SELECT`	Allows simultaneous selection of multiple columns.
`SS_MULTI_ROW_SELECT`	Allows simultaneous selection of multiple rows.
`SS_CELL_SELECT`	Allows individual cells to be selected.

4.14.4 Signals

In addition to the common signals defined by PegThing, PegSpreadSheet supports the following signals:

PSF_COL_SELECT	Sent when PegSpreadSheet column(s) are selected.
PSF_ROW_SELECT	Sent when PegSpreadSheet row(s) are selected.
PSF_CELL_SELECT	Sent when PegSpreadSheet cell(s) are selected.
PSF_COL_DESELECT	Sent when a PegSpreadSheet column is unselected.
PSF_ROW_DESELECT	Sent when a PegSpreadSheet row is unselected.

4.14.5 Derivation

PegSpreadSheet derives from PegWindow.

4.14.6 Constructors:

```
PegSpreadSheet(const PegRect &Rect, PEGINT Rows,
     PEGINT Cols, PEGUINT Id = 0, PEGULONG Style =
     FF_RAISED|SS_CELL_SELECT|SS_PARTIAL_COL,
     PegThing *pOwner = NULL)
```

The PegSpreadSheet constructor accepts a PegRect defining the spreadsheet position and size. The total number of spreadsheet rows is specified by `Rows`, and the total number of columns is specified in `Cols`. The spreadsheet must have a non-zero ID to send signals, and the ID can be specified in `Id`. `Style` indicates the global spreadsheet style, as each row and column style must be set individually.

The `pOwner` parameter specifies which window, if any, should receive the spreadsheet selection signals. This is required since PegSpreadSheet is often added to PegPresentationManager, rather than to the owner window.

4.14.7 Public Functions:

```
PEGINT AddColumn(PEGINT Width, const PEGCHAR *pHeader
     = NULL)
```

This function can be used to add a new column to the right of the spreadsheet after the spreadsheet has been defined and displayed. The `Width` parameter defines the new column width (column widths for the

initial columns are calculated automatically based on column data). The `pHeader` parameter, if used, defines the new column header. Only single-row column headers are supported with this parameter. If your column headers use multiple rows, you should pass NULL to this function and then assign your column headers individually for the new column.

```
PEGINT AddRow(const PEGCHAR *pHeader = NULL)
```

This function can be used to add a new row at the end of the spreadsheet after the spreadsheet has been defined and displayed. This is not the same as defining the initial spreadsheet rows, normally done before the spreadsheet is displayed, because the new row's header width will not be factored into the left margin width for spreadsheet layout. Therefore, if you anticipate adding rows to a spreadsheet that will have wider row headers than the initially-defined rows, you should pad the initial row headers to leave room for the dynamically-added row headers.

```
PEGBOOL CellSelected(const PEGINT Row, const PEGINT
     Col)
```

This function can be used to determine at any time if a spreadsheet cell is selected.

```
PEGBOOL ColSelected(const PEGINT Col)
```

This function can be used to determine at any time if a spreadsheet column is selected.

```
virtual PegScroll *CreateHorzScroll(PegRect Rect,
     PegScrollInfo *pSi, PEGUSHORT Id = 0)
```

This function is used to create a horizontal scrollbar. Derived spreadsheet classes will override this to use a custom scrollbar.

```
virtual PegScroll *CreateVertScroll(PegRect Rect,
     PegScrollInfo *pSi, PEGUSHORT Id = 0)
```

This function is used to create a vertical scrollbar. Derived spreadsheet classes will override this to use a custom scrollbar.

```
PEGINT DeleteColumn(PEGINT Col)
```

This function can be used to delete any column in the spreadsheet. All data other than the deleted column is retained. This function returns the number of columns remaining in the spreadsheet after the delete operation, or -1 on error.

```
PEGINT DeleteRow(PEGINT Row)
```

This function can be used to delete any row in the spreadsheet. All data other than the deleted row is retained. This function returns the number of rows in the spreadsheet after the delete operation, or -1 on error.

```
virtual void Draw(const PegRect &Invalid)
```

PegSpreadSheet overrides the `PegThing::Draw` function in order to properly draw the cells, headers, and footers.

```
void DrawAllCells(const PegRect &Invalid)
```

This function draws all visible cells in the spreadsheet.

```
void DrawFooters(void)
```

This function draws all visible footers in the spreadsheet.

```
virtual void DrawHeaders(void)
```

This function draws all visible headers in the spreadsheet.

```
void DrawRowHeaders(void)
```

This function draws all visible row headers in the spreadsheet.

```
void ForceColWidth(PEGINT Col, PEGINT Width)
```

This function can be used to override the default width assigned to a spreadsheet column. The default width is determined by finding the widest data string associated with a column of cells. This width is determined from cell data populated before the spreadsheet is displayed. The application may override this default width by calling the `ForceColWidth` function immediately before displaying the spreadsheet.

```
PegBitmap *GetCellBmp(const PEGINT Row, const PEGINT
     Col) const
```

Returns a pointer to the bitmap in the cell specified by `Row` and `Col`.

```
PEGCHAR *GetCellData(const PEGINT Row, const PEGINT
     Col) const
```

This function is used to retrieve the text contained in a spreadsheet cell.

```
virtual PegRect GetCellRect(const PEGINT Row, const
     PEGINT Col)
```

This function returns the PegRect coordinates defining one cell's position. This rectangle might NOT be within the client area of the spreadsheet if the spreadsheet has been scrolled.

```
PEGCHAR *GetColHeader(const PEGINT Col) const
```

This function returns the text string used for the indicated column header.

```
PEGINT GetCols(void) const
```

This inline function returns the number of columns in the spreadsheet.

```
PEGUINT GetColumnStyle(const PEGINT Col)
```

This function is used to get the style flags associated with a particular spreadsheet column.

```
PEGINT GetDispCols(void)
```

This inline function returns the number of columns that are actually visible in the spreadsheet client area.

```
PEGINT GetDispRows(void)
```

This inline function returns the number of rows that are actually visible in the spreadsheet client area.

```
PEGINT GetOptimumHeight(void)
```

This function returns the best height for the spreadsheet after all cells have been initialized. This height will allow the display of the entire spreadsheet without scrolling.

```
PEGINT GetOptimumWidth(void)
```

Returns the best width for the spreadsheet after all cells have been initialized. This width will allow the display of the entire spreadsheet without scrolling.

```
PEGCHAR *GetRowHeader(const PEGINT Row) const
```

This function returns the character string used as the row header for the indicated row.

```
PEGINT GetRows(void) const
```

This inline function returns the number of rows in the spreadsheet.

```
PEGUINT GetRowStyle(const PEGINT Row)
```

This function returns the style flags associated with each row in the spreadsheet.

```
PEGINT GetSelectedColumn(PEGINT Index = 0)
```

This function can be called at any time to inquire which column(s), if any, are selected. If the `SS_MULTI_COL_SELECT` style is not enabled, there can only be at most one selected column. If multiple-selection mode is enabled, however, there may be any number of selected columns. In this case, the caller uses the index value to step through the selected columns and retrieve all selected column numbers.

```
PEGINT GetSelectedRow(PEGINT index = 0)
```

This function can be called at any time to inquire which row(s), if any, are selected. If the `SS_MULTI_ROW_SELECT` style is not enabled, there can only be at most one selected row. If multiple-selection mode is enabled, however, there may be any number of selected rows. In this case, the caller uses the index value to step through the selected rows and retrieve all selected row numbers.

```
PEGINT InsertColumn(PEGINT Column, PEGINT Width, const
     PEGCHAR *pHeader)
```

This function adds a new column to the spreadsheet, to the left of the column specified by `Column`. The initial column width will be `Width`. The column header may be assigned by passing `pHeader` or by passing NULL and using the normal `SetColumnHeader` functions after the column has been inserted. The return value is -1 for error or the number of columns in the spreadsheet after insertion.

```
PEGINT InsertRow(PEGINT Row, const PEGCHAR *pHeader =
     NULL)
```

Similar to `InsertColumn` (above), this functions inserts a new row in the spreadsheet immediately above the indicated row. The return value is -1 for error or the number of rows in the spreadsheet after insertion.

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

PegSpreadSheet overrides the `PegThing::Message` function to handle events like scrolling or mouse clicks.

```
PEGBOOL RedrawOneCell(const PEGINT Row, const PEGINT
     Col)
```

This function invalidates and draws an individual cell.

```
virtual void Resize(const PegRect &Rect)
```

PegSpreadSheet overrides the `Resize` function to update the client area spreadsheet scroll bars.

```
PEGBOOL RowSelected(const PEGINT Row)
```

Returns TRUE if the indicated row of cells is selected, else FALSE.

```
void SelectCell(PEGINT Row, PEGINT Col, PEGBOOL Set =
     TRUE)
```

This function can be used to manually select or deselect cells independent of user interaction.

```
virtual void SelectCell(const PegPoint &Pos)
```

This function is used when a user clicks on the spreadsheet. The parameter `Pos` represents the position of the click. This function determines if it needs to select a row, column, or cell based on the click.

```
void SelectColumn(const PEGINT Col, PEGBOOL Set =
     TRUE)
```

This function can be used to manually select or deselect spreadsheet columns independent of user interaction.

```
void SelectRow(const PEGINT Row, PEGBOOL Set = TRUE)
```

This function can be used to manually select or deselect spreadsheet rows independent of user interaction.

```
virtual void SetCellBmp(const PEGINT Row, const PEGINT
     Col, const PEGINT BmpId, PEGBOOL Redraw = FALSE)
```

This function sets a bitmap to be displayed in the cell specified by Row and Col.

```
virtual void SetCellData(const PEGINT Row, const
     PEGINT Col, const PEGCHAR *Text, PEGBOOL Redraw
     = FALSE)
```

`SetCellData` is used to populate individual spreadsheet cells with program-defined data values.

```
void SetCellFont(PEGINT FontId)
```

This function is used to select the font used to display cell data. There is only one font used to display all cells.

```
virtual void SetColor(const PEGUBYTE Index, const
     PEGINT Color)
```

PegSpreadSheet overrides the `SetColor()` function to provide additional color indices. These indices include:

PCI_NORMAL	Normal (not selected) cell fill color
PCI_SELECTED	Selected cell fill color
PCI_NTEXT	Normal (not selected) cell text color
PCI_STEXT	Selected cell text color
PCI_SS_COLHEADBACK	Column header background color
PCI_SS_COLHEADTEXT	Column header text color
PCI_SS_ROWHEADBACK	Row header background color
PCI_SS_ROWHEADTEXT	Row header text color
PCI_SS_DIVIDER	PegSpreadSheet divider color
PCI_SS_BACKGROUND	PegSpreadSheet background color

```
void SetColumnFont(PEGINT Col, PEGINT FontId)
```

This function sets the font for an entire column in the spreadsheet.

```
void SetColumnStyle(const PEGINT Col, const PEGULONG
     Flags)
```

This function is used to set the style flags associated with a particular spreadsheet column. The default column style is TJ_CENTER|FF_RECESSED. The available styles are:

FF_THIN	Cells are drawn with a thin border.
FF_RAISED	Cells are drawn with a raised border.
FF_RECESSED	Cells are drawn with a recessed border.
TJ_RIGHT	Cell data is right justified.
TJ_LEFT	Cell data is left justified.
TJ_CENTER	Cell data is center justified.
SCF_SEPARATOR	The column is a separator, and cell data is not displayed.
SCF_ALLOW_SELECT	Allows selection of this column.

```
void SetFooter(const PEGINT LineNum, const PEGINT Col,
     const PEGCHAR *pText)
```

This function is used to define the column footers. Columns footers are displayed directly below each column of spreadsheet data. Column footers are optional, and are only displayed if defined.

```
void SetHeader(const PEGINT LineNum, const PEGINT Col,
     const PEGCHAR *pText)
```

This function is used to define the spreadsheet column headers. Column headers are displayed directly above each column in the spreadsheet. The number of rows of header data is determined automatically by the spreadsheet based on the number of header rows initialized by the calling object. That is, you do not need to explicitly indicate how many rows of column header data are required. You simply initialize the column headers as needed, and PegSpreadSheet determines how to display the header information. The width of the column headers is also included when determining the width of each column. Columns headers are not required, and will not be displayed if not initialized. Column headers must be defined for the spreadsheet to support column selection.

```
void SetHeaderFont(PEGINT FontId)
```

This function defines the font used to display column headers.

```
void SetRowHeader(const PEGINT Row, const PEGCHAR
     *pText)
```

This function is used to initialize the row headers. Row headers are displayed to the left of each spreadsheet row. Row headers are not required, and will not be displayed if they are not initialized. Row headers must be defined for the spreadsheet to support row selection. Row headers are not scrolled when the spreadsheet performs a horizontal scrolling operation.

```
void SetRowStyle(const PEGINT Row, const PEGULONG
     Flags)
```

This function sets the style flags associated with each row in the spreadsheet. The only row style currently supported is the `SRF_ALLOW_SELECT` style, which allows the user to select the row header to select all cells in the row. The default row style is 0.

```
void SetScrollStartCol(const PEGINT Col)
```

This function can be used to determine which columns are scrolled right and left by the horizontal scroll bar. Columns to the left of this value are not scrolled, and columns to the right are. This allow you to ‘lock’ certain columns in the display while others are scrolled.

```
void SetSize(const PEGINT Rows, const PEGINT Cols)
```

This function sets the size of the spreadsheet based on rows and columns.

```
void UnselectAll(void)
```

This function unselects all rows, columns, and cells.

```
PEGBOOL UnselectCells(void)
```

This function unselects cells. It returns TRUE if a selected cell was found, else FALSE.

```
PEGBOOL UnselectColumns(void)
```

This function unselects all columns. It returns TRUE if a selected column was found, else FALSE.

```
PEGBOOL UnselectRows(void)
```

This function unselects all rows. It returns TRUE if a selected row was found, else FALSE.

```
PEGINT UpdateColLayout(void)
```

This function forces the spreadsheet to recalculate the visible/non-visible column parameters and scroll bar settings. This function should be called if the `SS_PARTIAL_COL` style flag is changed after the spreadsheet is visible.

```
PEGINT UpdateRowLayout(PEGBOOL ForceVertical = FALSE)
```

This function forces the spreadsheet to recalculate the visible/non-visible row parameters and scroll bar settings. It should be called if the number of column header rows, cell or header font, or any other variable that affects the number of visible rows is changed after the spreadsheet is visible. Setting `ForceVertical` TRUE forces the inclusion of a vertical scroll bar. If `ForceVertical` is FALSE or omitted, a vertical scroll bar will be added only if needed.

```
virtual void UpdateScrollBars(void)
```

This function forces the spreadsheet to recalculate the size and position of the scroll bars.

```
PEGINT VScrollIntoView(const PEGINT Row)
```

This function scrolls the specified row into view.

4.14.8 Protected Members

```
virtual void CheckCellSelect(const PegPoint &Pos)
```

This function checks to see if the clicked position, `Pos`, is inside a selectable cell. If so, it selects it.

```
virtual void ClipToFace(const PegRect &Invalid)
```

This function adjusts the clipping rectangle to be contained inside the border and the scroll bar.

```
virtual void DoHorizontalScrolling(const PEGINT
     Amount)
```

This function forces a scroll horizontally by `Amount`.

```
virtual void DoVerticalScrolling (const PEGINT Amount)
```

This function forces a scroll vertically by `Amount`.

```
virtual void DrawCellBorder(PegRect &CellRect,
     PEGCOLOR BackColor, PEGINT Row, PEGINT Column)
```

This function draws a border around a single cell.

```
virtual void DrawFlat(PegRect &CellRect, PEGCOLOR
     BackColor, PEGINT Border = 0)
```

This function draws a flat (borderless) rectangle in the background of a single cell.

```
virtual void DrawRaised(PegRect &CellRect, PEGCOLOR
     BackColor, PEGINT Row, PEGINT Column)
```

This function draws a raised frame around a single cell.

```
virtual void DrawRecessed(PegRect &CellRect, PEGCOLOR
     BackColor, PEGINT Row, PEGINT Column)
```

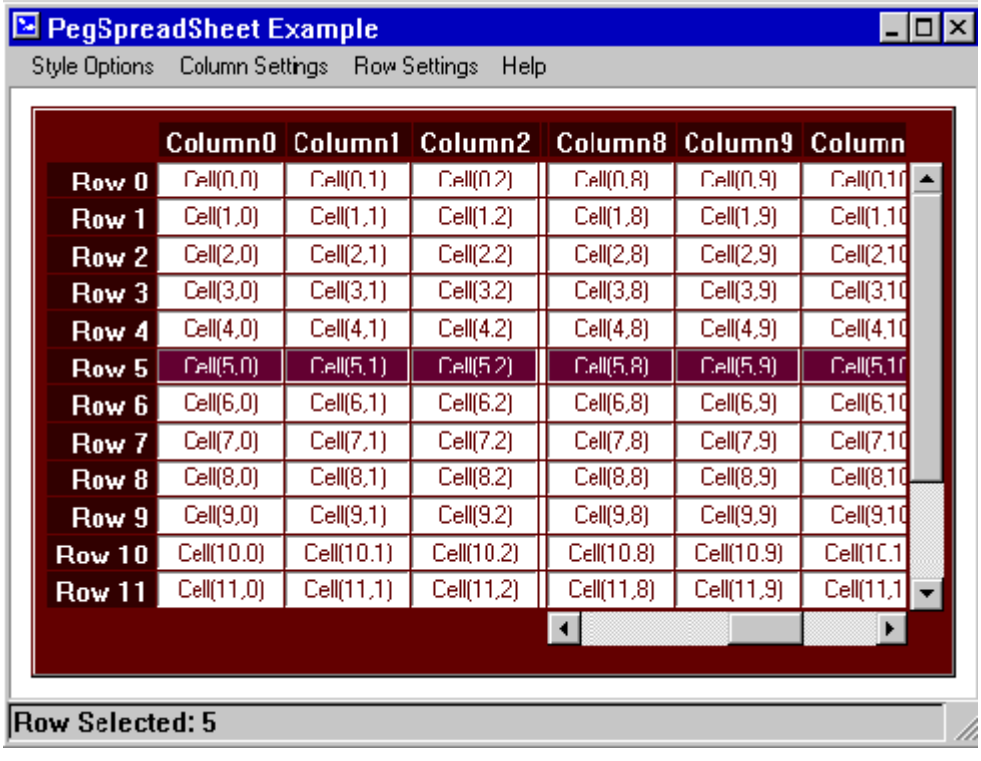
This function draws a recessed frame around a single cell.

```
virtual void UpdateCell(PegRect &CellRect, PEGCOLOR
     ForeColor, COLORVAL BackColor, PEGINT Row,
     PEGINT Col)

virtual void UpdateCell(PegRect &CellRect, const
     PEGCHAR *pText, PEGUINT Justify, PEGINT FontId,
     PEGCOLOR ForeColor, PEGCOLOR BackColor)
```

This function updates the text on a single cell using the text, font, and colors passed into it.

4.14.9 Examples:

The following is an example of a PegSpreadSheet, centered in the client area of a PegDecoratedWindow:



A complete example using PegSpreadSheet (the example used to create the screen shot above) can be found in your `\peg\examples\spread` directory.

4.15 PegTable

4.15.1 Overview

PegTable is a container object for displaying a matrix of PegThing-derived objects.

The number of table rows and columns must be passed to the PegTable constructor. The row heights and columns widths are determined dynamically as objects are added to the table.

PegTable will display a cell grid if the table grid line width is non-zero. The default grid line width is `PEG_FRAME_WIDTH`. This can be modified by calling the member function `SetGridWidth()`.

Each cell can add padding space around each child object if desired. The default padding amount is zero pixels. This can be modified by calling the table member function `SetCellPadding()`. Cell padding applies to all cells in the table.

Child objects are added to the table using the table `SetCellClient()` member function. Any PegThing-derived class may become a table cell client. As objects are added to the table, they can be set to span multiple rows and/or columns. This allows a great deal of flexibility in the final appearance of the table. When objects that span multiple rows or columns are added to the table and the `TCF_FORCEFIT` style flag is passed to the `SetCellClient()` function, the spanned rows and/or columns may be expanded to ensure that the object is fully displayed in the indicated table cells.

When constructing child objects to display in the table, it is not necessary for the application software to calculate each child object's position. This is determined by the table window. It IS necessary, however, to properly set the overall size of each child object before adding that object to the table. The table determines the row heights and column widths by examining the size of each child object in conjunction with the rowspan and columnspan attributes of each child.

After the table has been fully initialized, the `Layout()` member function should be called before the table is displayed. The `Layout` function calculates the correct overall table size, and positions each child object to fit correctly within the desired table cells.

The `Layout()` function resizes row heights and column widths if required to accommodate all children with `TCF_FORCEFIT` style. The layout algorithm works by checking the following rules in the following order:

- For each single-cell child object with a `TCF_FORCEFIT` style, ensure that the height and width of the cell occupied by the object are ≥ the object height and width.
- For each multicell child object with a `TCF_FORCEFIT`, add the total height and width of the spanned cells.
    - — If the total width of spanned cells is ≥ the object width, continue to height check.
    - — Otherwise, search for a zero-width column spanned by the object.
    - — If a zero-width column is found, increase the column width the necessary amount to display the child object.
    - — If no zero-width columns are found, increase all spanned cell widths by an equal amount to fully display the child object.
    - — If the total height of spanned cells is ≥ the object height, continue.
    - — Otherwise, search for a zero-height row spanned by the object.
    - — If a zero-height row is found, increase the row height the necessary amount to display the child object.
    - — If no zero-height rows are found, increase all spanned cell heights by an equal amount to fully display the child object.
- Position each child object to the center of the each cell.

The general order of constructing a PegTable is:

- Construct the table, passing in max rows and columns.
- Set column widths and row heights for any columns or rows with manual size settings.
- Call `SetCellClient` for each cell in the table that should contain a child object.
- Call the table member function `Layout()`.
- Add the table to the parent window or PegPresentationManager.

Each table column has a fixed width. Initially, each column has a default width of zero. The true width of each column can either be set manually by calling the `SetColWidth()` function, or can be determined automatically by

the table. When adding objects to the table, the table will automatically re-size columns (and rows) to match the size of the child object if the `TCF_FORCEFIT` style flag is passed in to `SetCellClient()`. In all cases, the column width for each column is set to the maximum width of either all children for that column or the manually-assigned column width value.

Each table row also has a fixed height, which can either be set using the `SetRowHeight()` function or determined automatically by the table.

PegTable will automatically size itself to display all children when the `Layout()` function is called. For displaying very large tables, PegTable may be added to the client area of a parent window which has scrolling enabled This will allow the table to be panned up-down and left-right.

Notifications sent from table cell clients are passed unchanged to the table parent. This allows any window containing a PegTable to receive events from the table cell client objects as if the objects were direct children of the parent window.

4.15.2 See Also

PegSpreadSheet

4.15.3 Style Flags

PegTable supports the following style flags:

`TS_SOLID_FILL`	Fills the table `mReal` rectangle with the table `PCI_NORMAL` color.
`TS_DRAW_HORZ_GRID`	Draws only the horizontal grid lines between rows of cells.
`TS_DRAW_VERT_GRID`	Draws only the vertical grid lines between columns.
`TS_DRAW_GRID`	Draws all grid lines between all cells.

PegTable also supports the following styles with regard to the cell clients. These flags are only applicable when calling the SetCellClient method.

TCF_FORCEFIT	This style bit will force the parent table to automatically adjust the row heights and column widths of the cells that are occupied by this object to ensure that the object is fully contained and displayed by the cells that it occupies
TCF_HCENTER	Horizontally centers the cell object within the bounding cell.
TCF_HLEFT	Left justifies the cell object within the bounding cell.
TCF_HRIGHT	Right justifies the cell object within the bounding cell.
TCF_VCENTER	Vertically centers the cell object within the bounding cell.
TCF_VTOP	Top justifies the cell object within the bounding cell.
TCF_VBOTTOM	Bottom justifies the cell object within the bounding cell.

4.15.4 Derivation

PegTable derives from PegWindow.

4.15.5 Constructors:

```
PegTable(PegRect &Rect, PEGINT Rows, PEGINT Cols)
```

This constructs a PegTable object, specifying the position and the number of table rows and columns.

4.15.6 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegTable overrides the `Draw()` function and breaks the procedure into two stages. First, the table itself draws the table grid lines. Then the table `mClip` variable is set to the 'client area' of each cell and individual cell clients are drawn.

```
virtual void DrawChildren(const PegRect &Invalid)
```

This function modifies the base class function of the same name to prevent child objects from drawing outside the cell area allocated to each cell client.

```
virtual void DrawGrid(void)
```

This function draws the table grid lines.

```
PegThing *GetCellClient(PEGINT Row, PEGINT Col)
```

This function returns a pointer to the object displayed at the indicated cell position.

```
PEGINT GetCellPadding(void)
```

This inline function returns the cell padding of the table.

```
void GetCellSpan(PEGINT Row, PEGINT Col, PEGINT
     *pRowSpan, PEGINT *pColSpan)
```

This function determines the row-span and column-span for the cell at the location specified by `Row` and `Col`.

```
PEGINT GetCellStyle(PEGINT Row, PEGINT Col)
```

This function returns the style flags for the specified cell.

```
PEGINT GetColumns(void)
```

This inline function returns the number of columns in the table.

```
PEGINT GetForcedColWidth(PEGINT Col)
```

This function returns the forced width of a column if it has the `TCF_FORCEFIT` style. It returns -1 if `TCF_FORCEFIT` is not enabled for that column.

```
PEGINT GetForcedRowHeight(PEGINT Row)
```

This function returns the forced height of a row if it has the `TCF_FORCEFIT` style. It returns -1 if `TCF_FORCEFIT` is not enabled for that row.

```
PEGINT GetGridWidth(void)
```

This inline function returns the table's grid width.

```
PEGBOOL GetRowCol(PegThing *pChild, PEGINT *pRow,
     PEGINT *pCol)
```

This function returns the row and column position of the child object indicated by `pChild`. If `pChild` is found in the table, `pRow` and `pCol` are written by this function to contain the row and column of the indicated child object. If the object spans multiple rows or columns, the returned row and column value will be the first, or upper left, cell position. This function returns TRUE if the object `pChild` is found in the table, else FALSE.

```
PEGINT GetRows(void)
```

This inline function returns the number of rows in the table.

```
virtual void Layout(void)
```

This function should be called whenever the list of table child objects is modified. This is normally done after the table is fully initialized but before the table is displayed. Note that child objects can be modified after the table has been displayed. The `Layout` function should be called after any modifications are made to a visible table. Note that table row heights and column widths are never decreased, but may be increased if new child objects are added which are larger than the objects previously contained in a given cell or group of cells.

```
virtual void Reconfigure(PEGINT Rows, PEGINT Cols,
     PEGINT GridWidth, PEGINT CellPadding)
```

This function recalculates row/column widths based on current entries. This should be called after changing the cell clients.

```
PegThing *RemoveCellClient(PEGINT Row, PEGINT Col)
```

This function removes the child object at the indicated cell position and returns a pointer to that object. The object is not destroyed.

```
virtual void SetCellClient(PEGINT Row, PEGINT Col,
     PegThing *pChild, PEGINT RowSpan = 1, PEGINT
     ColSpan = 1, PEGULONG Style = TCF_HCENTER|
     TCF_VCENTER)
```

This function is used to populate the children displayed in each table cell. The child objects can be any type of PegThing-derived objects, which includes all buttons, strings, prompts, text boxes, etc. The default row and column span for an object is 1, indicating that the object occupies one table cell. If the `ColSpan` or `RowSpan` values are set to a value larger than 1, this object will occupy multiple table cells.

The default `Style` flag will center the object vertically and horizontally within the bounding cell. To enable force-fitting the cell on the table, specify the `TCF_FORCEFIT` style.

See this table for a list of supported style flags for the cell objects. It is important to note that the justification styles operate only on the placement of the object within its bounding cell or cells. The style flags do not operate on the object itself. For example, a PegPrompt object has a series of styles that allow for left, center, and right justification of its text. These text

justification style flags are completely independent of the cell style flags. Therefore, it would be acceptable to right justify the text within the PegPrompt object while specifying that the PegPrompt object be positioned horizontally to the left within the cell.

```
virtual void SetCellPadding(PEGINT Pad)
```

This function is used to modify the default cell padding value, which is zero. The padding amount is in pixels.

```
virtual void SetColWidth(PEGINT Col, PEGINT Width,
     PEGBOOL Force = FALSE)
```

This function can be used to manually set the column width for any number of columns in the table. The column width cannot be decreased, but can only be set to a value larger than the current value. The initial width of all table columns is zero pixels.

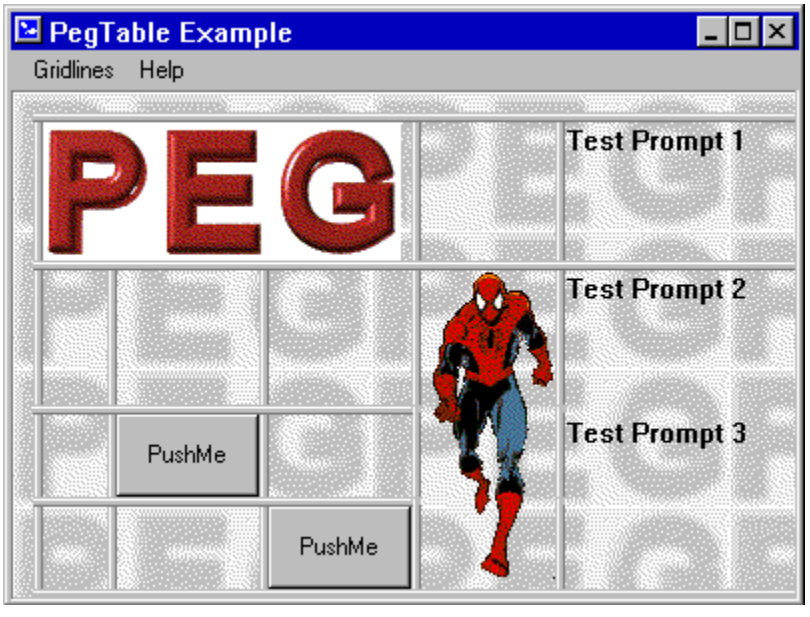
`void SetGridWidth(PEGINT Width = PEG_FRAME_WIDTH)` This function is used to set the grid line width for the grid drawn by the table. Setting the grid line width to zero disables the table grid; however, all other positioning works as normal.

```
virtual void SetRowHeight(PEGINT Row, PEGINT Height,
     PEGBOOL Force = FALSE)
```

This function can be used to manually set the row height for each table row. The row height cannot be decreased, but can only be set to a value larger than the current value. The initial height of all table rows is zero.

4.15.7 Examples:

The following is a PegTable populated with different types of cell client objects.



The full source code for this example can be found in the `\peg\examples\table` directory.

4.16 PegTextBox

4.16.1 Overview

PegTextBox is a multiline text display control. PegTextBox is derived from PegWindow.

The font, color, size, scrolling mode, and other parameters can be modified, giving PegTextbox a wide variety of appearances.

By default, PegTextBox left-justifies the displayed text. Center and right-justification is also supported.

Lines of text that are too long to fit in the client width of the text box are also wrapped by default to use two or more lines. This is controlled by the `EF_WRAP` style flag. The wrapping algorithm searches for white space, comma, or hyphen characters as logical points to break long lines. If a suitable breaking point is not found, PegTextBox simply breaks a line at the last character that fits within the client width area.

If a PegTextBox is used to display a long section of text that requires more vertical lines than are visible in the text box client area, you can use the `SetScrollMode(WSM_AUTOVSCROLL)` function to give the PegTextBox a vertical scroll bar.

Internally, PegTextBox maintains a set of `PEGUINT` offsets into the block of text displayed by the PegTextBox window. These offsets are the starting character offsets corresponding to each line of text. This allows PegTextBox to quickly display new lines of text as the text is scrolled up and down. Only a fixed maximum number of offsets are calculated at any one time (max is 100). This window of line-start offsets slides up and down as the user scrolls the text in the window. You can access these line-start offsets if needed using member functions.

In addition to scroll bars, PegTextBox catches `PK_PGUP` and `PK_PGDN` `PM_KEY` messages to scroll the displayed text up and down, page by page.

PegTextBox does not allow user editing. The text data displayed in the text box can only be modified via program control using the `DataSet()` or `Append()` member functions. For a full edit-style control, refer to PegEditBox.

4.16.2 See Also

PegPrompt

PegEditField

PegWindow

PegTextThing

PegEditBox

4.16.3 Style Flags

PegTextBox supports the following styles:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame
EF_EDIT	When this style is applied, the user can edit the PegTextBox object. If this style is applied, the PegTextBox object automatically includes the TT_COPY style.
EF_WRAP	When this style is applied, the text box will wrap long lines to prevent them from being clipped.
TT_COPY	Instructs the PegEditBox to copy the text string assigned. This flag should be used when the text string assigned to the PegEditBox is created dynamically using temporary storage.

4.16.4 Signals

PegTextBox supports no extended signals.

4.16.5 Derivation

PegTextBox is derived from PegWindow.

4.16.6 Constructors:

```
PegTextBox(const PegRect &Rect, PEGUINT StringId = 0,
      PEGUSHORT Id = 0, PEGULONG Style = FF_RECESSED|
      EF_WRAP|TJ_LEFT, PEGUINT MaxChars = 1000)

PegTextBox(const PEGCHAR *pText, const PegRect &Rect,
      PEGUSHORT Id = 0, PEGULONG Style = TT_COPY|
      FF_RECESSED|EF_WRAP|TJ_LEFT, PEGUINT MaxChars =
      1000)
```

This constructor creates a PegTextBox. `MaxChars` is the maximum number of characters that the text box will be required to support. If more than this number of characters is assigned using the `DataSet()` function, the extra characters fall off the end. If more than this number of total characters is assigned using the `Append()` function, the oldest characters fall off the top.

4.16.7 Public Functions:

```
virtual void Append(PEGINT StringId)

virtual void Append(const PEGCHAR *pText)
```

This function appends the indicated text to the current text box string value. If the total number of text box characters ≥ the maximum number of characters, PegTextBox will remove the overflow characters from the start of the text box string. This operation facilitates the creation of terminal style windows.

```
void CheckBufLen(PEGINT Len)
```

This function checks to see if the internal text buffer is large enough to hold `Len` number of characters. If not, it reallocates the buffer.

```
virtual void DataSet(PEGINT StringId)

virtual void DataSet(const PEGCHAR *pText)
```

PegTextBox overrides the DataSet function to reset any scroll bars and to recalculate line offsets.

```
virtual void Draw(const PegRect &Invalid)
```

PegTextBox overrides the `Draw()` function to display the text box border and text.

```
const PEGCHAR *FindLinePointer(PEGINT Line)
```

Finds the `PEGCHAR` at the start of the given line. If `Line` is beyond the total number of lines, then the method returns NULL.

```
virtual void GetHScrollInfo(PegScrollInfo *pPut)
```

PegTextBox overrides the `PegWindow::GetHScrollInfo` function to make the scroll bars operate relative to the contained text.

```
PEGINT GetLineIndex(PEGINT Line)
```

This function returns the starting index into the total text string at which the text for line `Line` begins.

```
const PEGCHAR *GetLineStart(PEGINT Line, PEGINT
     *pLength)
```

This function returns a pointer to the indicated line of text. Programmers must remember that the text box text lines are actually subsets of a larger text string; i.e., the returned string is not necessarily terminated at end of the requested line. If the `pLength` parameter is non-NULL, this function returns the number of characters (excluding \r or \n characters) displayed on this text line.

```
PEGUINT GetMaxChars(void)
```

This inline function returns the maximum number of characters the text box will contain.

```
PEGUINT GetTopLine(void)
```

This inline function returns the index of the top line currently displayed in the text box.

```
virtual void GetVScrollInfo(PegScrollInfo *pPut)
```

PegTextBox overrides the `PegWindow::GetVScrollInfo` function to make the scroll bars operate relative to the contained text.

```
PEGINT GetWidestLine(void)
```

This inline function returns the width (in pixels) of the widest line of currently-displayed text in the text box. The widest line may not be visible in the client area.

```
PEGINT LineCount(void)
```

This inline function returns the total number of lines contained in the text box. This is the total number of lines available, as opposed to the number of lines actually visible.

```
PEGBOOL LineDown(void)
```

This function can be called to scroll the text box down one line under program control.

```
PEGBOOL LineUp(void)
```

This function can be called to scroll the text box up one line under program control.

```
void MarkLine(PEGINT iLine)
```

This function can be used to 'mark' one line of text. The marked line of text will be displayed using the `PCI_STEXT` and `PCI_SELECTED` foreground and background colors.

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

PegTextBox catches several mouse and keyboard messages.

```
virtual void Resize(const PegRect &NewSize)
```

PegTextBox overrides the `Resize()` function to recalculate the layout of the displayed text lines.

```
void RewindDataSet(PEGINT StringId)
```

```
void RewindDataSet(const PEGCHAR *pText)
```

This function assigns the specified text string to the PegTextBox, and also resets the text box to the top line. This is useful when the text box content is changed after the user may have scrolled the text box down some number of lines. When this function is used, the new text is always displayed from the start.

```
void SetBorderWhitespace(PEGINT Space)
```

This function determines how much white space is used to pad the outside of the text, inside its border.

```
virtual void SetFont(PEGINT iFontIndex)
```

PegTextBox overrides the `SetFont()` function to recalculate the layout of the displayed text lines.

```
void SetInterlineSpace(PEGINT Space)
```

This function determines how much white space is used to pad in between lines of text.

```
void SetMaxChars(PEGINT Max)
```

This function can be used to modify the maximum number of characters the text box will allow.

```
virtual void SetScrollMode(PEGUINT Mode)
```

PegTextBox overrides `PegWindow::SetScrollMode` to make sure that the text lines get updated with it.

```
void SetTopLine(PEGINT Line)
```

This function will scroll the text box to the indicated line number under program control. If the requested line is too far down (i.e., past the last line of text) PegTextBox sets the top line such that the last line of text is visible at the bottom of the text box client area.

```
void SetTopLineToEnd(void)
```

This function will scroll the text box down such that its last line is visible.

4.16.8 Protected Members:

```
virtual void DrawAllText(const PegRect &Invalid)
```

This function draws all visible lines of text.

```
virtual void DrawTextLine(PEGINT Line, PegPoint
     PutPoint, PEGBOOL Fill = FALSE)
```

This function draws a single line of text.

```
virtual const PEGCHAR *FindNextLine(const PEGCHAR
     *pText, PEGINT MaxWidth, PEGINT Dir = 1)
```

This function looks through the `pText` string and returns a pointer to where the next new line will start.

```
PEGINT mWidestLine
```

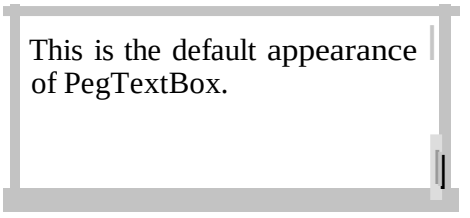
Holds the width of the widest text box line, in pixels.

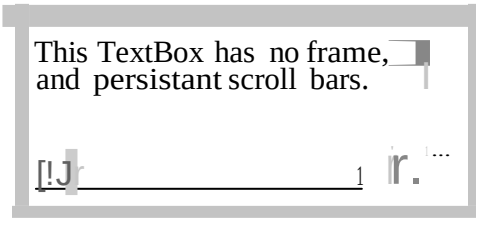
```
PEGINT mLeftOffset
```

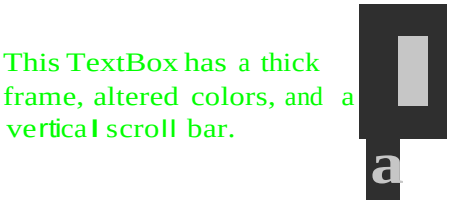
This value indicates how far the text box has scrolled from the leftmost anchor point.

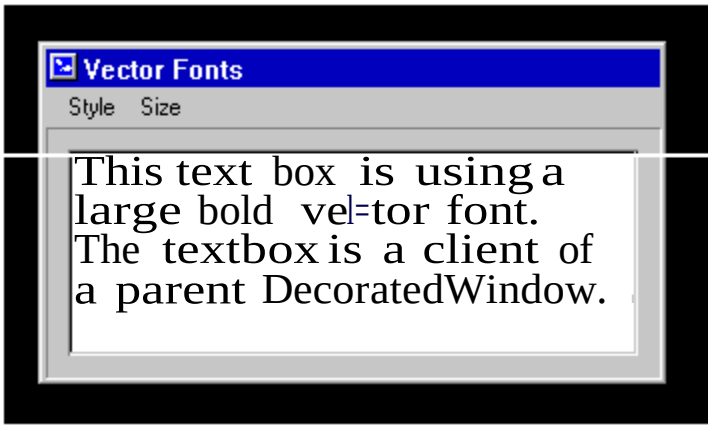
4.16.9 Examples:

The following are each different styles of PegTextBox:









The camplete source for the last example shown above can be found in the example program \peg\examples\vecfont distributed with your PEG release.

4.17 PegTreeNode

4.17.1 Overview

PegTreeNode, derived from PegTextThing, is used to populate a PegTreeView container window.

Each PegTreeNode must have an associated text string. Each node may also have a bitmap or thumbnail associated with it. If a bitmap is specified, the bitmap is displayed to the left of the text for that node. The same bitmap can be used for any number of nodes.

The bitmaps associated with each node do not all have to be the same size. The bitmaps associated with each node are displayed such that they are horizontally centered. Also, the bitmaps can be of any height and width. A pleasing display is usually created using bitmaps of between 12 x 12 and 20 x 20 pixels.

PegTreeNode objects are displayed in the order they are added to the parent node.

The model for programming PegTreeView is very similar to the general PEG programming model, since in both cases you are working with tree-structured lists of objects.

4.17.2 See Also

PegWindow

PegTreeView

4.17.3 Derivation

PegTreeNode derives from PegTextThing.

4.17.4 Constructors:

```
PegTreeNode(PEGINT StringId = 0, PEGINT BmpId = 0)

PegTreeNode(const PEGCHAR *Text, PEGINT BmpId = 0)
```

Creates a PegTreeNode object. The text string must be valid, while the bitmap is optional.

4.17.5 Public Functions:

```
virtual void Add(PegThing *pChild, PEGBOOL DoShow =
     FALSE)
```

Adds pChild to the current node.

```
PEGINT BranchHeight(void)
```

Returns the height of the current branch, factoring in all child node heights if the current branch is open.

```
PEGINT BranchWidth(PegTreeView *pParent)
```

Returns the width, in pixels, of the current branch. This includes the bitmap width (if any) and the text string width.

```
void Close(void)
```

Closes the current node. If the node has children, they are not displayed.

```
PEGINT Count(void)
```

Returns the number of children owned by the current node.

```
PegTreeNode *FirstNode(void)
```

Returns the first child node of the current node, or NULL if the current node has no children.

```
void ForceOpen(void)
```

This function forces the node to be open.

```
PEGINT GetMap(void)
```

Returns the bitmap associated with the node, or -1.

```
void Insert(PegTreeNode *pSib)
```

Inserts a new node beneath the current node. This may push other nodes down if they exist.

```
PEGBOOL IsOpen(void)
```

Returns TRUE if the current node is open and has children, else FALSE.

```
PEGBOOL IsSelected(void)
```

Returns TRUE if the current node is selected, else FALSE.

```
void MoveToTop(PegTreeNode *pChild)
```

Slides the current node to the top of its siblings, thus becoming the first child of the parent node.

```
PegTreeNode *NextNode(void)
```

Returns the following sibling of the current node, or NULL.

```
PegTreeNode *NodeAbove(void)
```

This function returns the nearest visible PegTreeNode above the current node.

```
PegTreeNode *NodeBelow(void)
```

This function returns the nearest visible PegTreeNode below the current node.

```
PegTreeNode *NodeBottom(void)
```

This function returns the bottom-most PegTreeNode on the node's subtree.

```
PEGINT NodeHeight(void)
```

Returns the height of the current node, in pixels. This function does not include the height of any child nodes.

```
PEGINT NodeWidth(void)
```

Returns the width, in pixels, of the current node. This includes the bitmap width (if any) and the text string width.

```
void Open(void)
```

Forces the node to open and display any children.

```
PegTreeNode *ParentNode(void)
```

Returns the parent of the current node, or NULL if the current node is the tree view top node.

```
void SetMap(PEGINT BitmapId)
```

Assigns the bitmap associated with a given node. This function can be called at any time, allowing the system software to change the bitmap associated with a node based on the node state.

```
void SetNext(PegThing *pNext)
```

Assigns the next node associated with the current node. This function is generally only used by PegTreeView.

```
void SetSelected(PEGBOOL Selected)
```

Forces the current node to become selected and to display itself as the selected node.

4.17.6 Examples:

Refer to PegTreeView for instance and programming examples.

4.18 PegTreeView

4.18.1 Overview

PegTreeView, derived from PegWindow, displays a hierarchical presentation of PegTreeNode objects. PegTreeView always uses automatic vertical and horizontal scrolling, so that whenever more nodes than can be displayed in the PegTreeView client area are present, scroll bars are provided for panning the node display area.

PegTreeView is actually a container object. When you create and display PegTreeView style windows, you will generally interact directly with the PegTreeNode children of the PegTreeView container. Any nesting level of PegTreeNode children can be displayed in the PegTreeView window.

PegTreeView allows node selection using the mouse or the keyboard.

The PegTreeView constructor accepts parameters for creating the first or top-level node in the tree. PegTreeView immediately creates this top node, and it is always present. Additional nodes are added by adding them to this top node.

The model for programming PegTreeView is very similar to the general PEG programming model, since, in both cases, you are working with tree-structured lists of objects.

4.18.2 See Also

PegWindow

PegTreeNode

4.18.3 Style Flags

PegTreeView supports the following style flags:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame

FF_THICK	Thick 3D Frame

4.18.4 Signals

PegTreeView sends the following signals based on user selections:

PSF_NODE_DELETE	This signal is sent when the user presses the Delete key while a node is selected. It is the responsibility of the application software to actually remove and/or delete the selected node.
PSF_NODE_SELECT	This signal is sent when a new node is selected in the PegTreeView container.
PSF_NODE_OPEN	This signal is sent when the user opens a node that has children.
PSF_NODE_CLOSE	This signal is sent when the user closes a node previously opened.

For each of the above signals, the message contains the following data:

`Message.pSource` = Pointer to PegTreeView object.
`Message.Param` = ID of the PegTreeView object.
`Message.pData` = Pointer to selected PegTreeNode.

4.18.5 Derivation

PegTreeView derives from PegWindow.

4.18.6 Constructors:

```
PegTreeView(const PegRect &Rect, PEGINT StringId,
      PEGULONG Style, PEGINT BitmapId = 0)

PegTreeView(const PegRect &Rect, const PEGCHAR *pText,
      PEGULONG Style, PEGINT BitmapId = 0)
```

The PegTreeView constructor accepts the initial tree view position, frame style, and top node text. An optional bitmap or thumbnail to be associated with the top level node may also be defined.

4.18.7 Public Functions:

```
void DestroyNode(PegTreeNode *pWho)
```

This function removes the indicated node from the tree and deletes the node object. If the indicated node has children, they are also removed and destroyed.

The `FindNode()` function is often used in conjunction with `DestroyNode()`, to remove a specified node as follows:

```
DestroyNode(FindNode(1, "Temp"));
```

```
virtual void Draw(const PegRect &Invalid)
```

PegTreeView overrides the `Draw()` function to display the tree view connecting lines and node anchors.

```
void DrawNode(PegTreeNode *pStart, PegPoint Put,
     PEGINT MaxMapWidth)
```

Draws the node pointed to by `pStart`.

```
PegTreeNode *FindNode(PEGINT Level, const PEGCHAR
     *pText)
```

This function returns a pointer to the PegTreeNode at the indicated nesting level with the matching text string. If multiple nodes at the correct nesting level have a matching text string, the first or topmost matching node pointer will be returned.

Node nesting levels start at 0. The only level 0 node is the top tree node. The first level of nodes under the top node are level 1 nodes, the next level of indented nodes are level 2 nodes, etc.

```
virtual void GetHScrollInfo(PegScrollInfo *pPut)
```

PegTreeView overrides the `GetHScrollInfo` function to calculate the tree width based on the sum of the individual node widths, and positions the horizontal scroll bar accordingly.

```
PEGINT GetIndent(void)
```

This function returns the current indent level, in pixels.

```
PegTreeNode *GetSelected(void)
```

This function returns a pointer to the selected node.

```
virtual void GetVScrollInfo(PegScrollInfo *pPut)
```

PegTreeView overrides the `GetVScrollInfo` function to calculate the tree height based on the sum of the individual node heights, and positions the vertical scroll bar accordingly.

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

PegTreeView catches mouse and keyboard messages to test for node actions.

```
PegTreeNode *RemoveNode(PegTreeNode *pWho)
```

This function removes, but does not delete, the indicated node from the tree. If the node has children, the children are also removed from the tree but remain attached to the node.

The `FindNode()` function is often used in conjunction with `RemoveNode`, to remove a specified node as follows:

```
RemoveNode(FindNode(1, "Temp"));
```

```
void Reset(PEGINT TopStringId)
```

```
void Reset(const PEGCHAR *pText)
```

This method resets the entire tree by removing and deleting all of the tree's nodes. The text for the top node is then set to the specified text.

```
void Select(PegTreeNode *pWho)
```

This function selects the indicated node via program control.

```
void SetIndent(PEGINT Val)
```

This function can be used to override the default indent level of each generation of child nodes. The indent level is specified in pixels.

```
virtual void ToggleBranch(PegTreeNode *pWho)
```

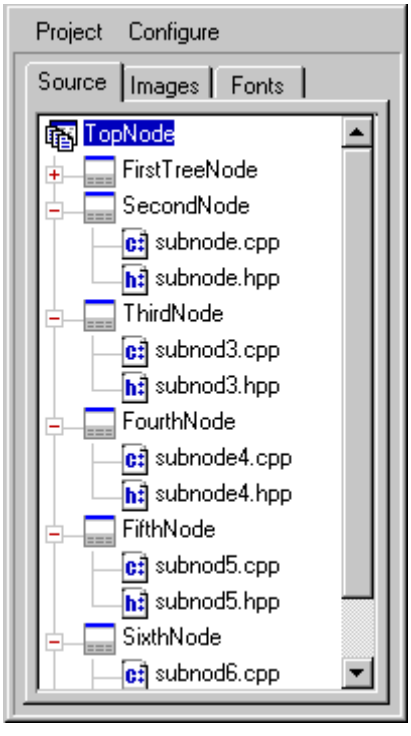
This function either opens or closes a branch of the tree, depending on its current status.

```
PegTreeNode *TopNode(void)
```

This function returns a pointer to the top tree node. Using this pointer, the application-level software can traverse the entire tree.

4.18.8 Examples:

The following is an example of a PegTreeView window populated with PegTreeNodes. In this case, the PegTreeView window is used as the client for a PegNotebook page. This example is taken from PegWindowBuilder:



The following function creates a PegTreeView control and populates the control with PegTreeNodes. The top level node is labeled ‘Hockey Teams.’ Two subnodes are created, labeled ‘Good Teams’ and ‘Bad Teams.’ To each of these nodes are added several hockey team names (no offense intended!).

```
void MyWindow::CreateTreeView(void)
{
    PegTreeView *pTree;
    pTree = new PegTreeView(mClient, "Hockey Teams",
FF_RECESSED,
        BID_HOCKEY);

    pTree->Id(IDW_HOCKEY_TREE);
```

```
    PegTreeNode *pNode = pTree->TopNode();

    pNode->Add(new PegTreeNode("Good Teams", BID_CATEGORY));
    pNode->Add(new PegTreeNode("Bad Teams", BID_CATEGORY));

    // get pointer to first sub-node:

    pNode = pNode->FirstNode();

    // add good teams to this node:

    pNode->Add(new PegTreeNode("Red Wings", BID_TEAM));
    pNode->Add(new PegTreeNode("Blues",   BID_TEAM));
    pNode->Add(new PegTreeNode("Stars",   BID_TEAM);

    // get pointer to next node:

    pNode = pNode->NextNode();

    // add bad teams to this node:

    pNode->Add(new PegTreeNode("Mighty Ducks", BID_TEAM));
    pNode->Add(new PegTreeNode("Sharks",     BID_TEAM));
    pNode->Add(new PegTreeNode("Kings",     BID_TEAM));

    Add(pTree);
}
```

4.19 PegVirtualVList

4.19.1 Overview

PegVirtualVList is very similar to a PegVertList class in that it arranges data vertically and allows the user to scroll through each element. The main difference between them is that the PegVirtualVList class does not require a PegThing object for each item in the list. It automatically makes just enough objects to fill in the visible area, and then if the user scrolls the list, only the data is shifted up and down, not the objects themselves.

By default, the PegVirtualVList class will only work with text, and it will use standard PegPrompts to display that text. However, the class is designed to be flexible to allow applications to create derived versions that could potentially display any kind of data.

Rather than creating PegPrompts and adding them to the list in the manner of a PegVertList class, the PegVirtualVList class takes an array of data as input in the `AssignVirtualList` function. The default implementation assumes the data is text, but the function's parameter is actually a `void*`, so derived versions can override this function to use other types of data. The PegVirtualVList will create just enough PegPrompts to fit in its `mClient` area, and then it will call `DataSet()` to assign text to the prompts from the list data. When the user scrolls, it is just a matter of calling `DataSet()` on each prompt again.

The main advantage here is speed. For lists of large sizes, the PegVirtualVList is much faster at scrolling then a PegVertList. The reason is that when a PegVertList scrolls its child objects, even though only a select few of them might be visible, ALL of the child objects still need to be repositioned. That means that the more objects are added to the list, the slower scrolling will become. And with lists of hundreds or thousands of items that can add up. The PegVirtualVList performs exactly the same operations no matter how much data is in the list, so it doesn't slow down.

4.19.2 See Also

`PegVertList`

`PegList`

PegWindow

4.19.3 Style Flags

PegVirtualVList supports the following style flags:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame

The styles for PegVirtualVList are identical to the PegWindow styles. In addition, scrolling is enabled in PegVirtualVList in the same way as in PegWindow, by using the SetScrollMode() function.

4.19.4 Signals

PegVirtualVList sends PSF_LIST_SELECT signals to the parent object. This message contains:

Message.pSource = Pointer to selected object.
Message.Param = ID of selected list item.
Message.pTarget = Pointer to list parent object.

4.19.5 Derivation

PegVirtualVList derives from PegWindow.

4.19.6 Constructors:

```
PegVirtualVList(const PegRect &Rect, PEGUSHORT Id = 0,
      PEGULONG Style = FF_THIN|TJ_CENTER)
```

This constructor creates a PegVirtualVList object. The Rect parameter determines the position and size of the list. The list children are automatically positioned by the list object.

4.19.7 Public Functions:

```
virtual void AssignVirtualList(void *pList, PEGINT
     Count)
```

This function is used to assign the list data to the list. The parameter `pList` is an array of data of length `Count`. `pList` is a `void*` but it is assumed to be an array of text strings. The PegVirtualVList class does not make its own copy of the `pList` data, so it is assumed that that array will be persistent.

```
virtual PEGINT GetSelectedIndex(void)
```

This function returns the index of the currently-selected item.

```
virtual void GetVScrollInfo(PegScrollInfo *pPut)
```

PegVirtualVList overrides the `GetVScrollInfo()` function to make sure it scrolls based on the total size of the data, not the size of the visible child objects.

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

PegVirtualVList overrides the `Message()` function to handle mouse, keyboard, and scrolling messages.

```
void PageDown(void)
```

This function scrolls the data down by one full page, meaning that the first non-visible data item at the bottom of the list client area is scrolled into view and selected. This function is called by the `Message` function in response to the `PK_PGDN` key message.

```
void PageUp(void)
```

This function scrolls the data up by one full page, meaning that the first non-visible data item at the top of the list client area is scrolled into view and selected. This function is called by the `Message` function in response to the `PK_PGUP` key message.

```
virtual void Resize(const PegRect &NewSize)
```

PegVirtualVList overrides the `Resize` function to check whether more or less PegPrompts will fit in the new `mClient` area.

```
void ScrollIntoView(PEGINT Index, PEGBOOL Redraw =
     TRUE)
```

This function scrolls the list up or down so that the data with the index `Index` will be visible.

```
void SelectNext(void)
```

This function selects the data item that comes after (below) the currently-selected item. If the next item is not currently visible, then the list will be scrolled.

```
void SelectPrevious(void)
```

This function selects the data item that comes before (above) the currently-selected item. If the previous item is not currently visible, then the list will be scrolled.

```
virtual void SetCellHeight(PEGINT Height)
```

This function assigns the height of all of the PegPrompts in the list. This should be set to a value that is evenly divisible by the height of the `mClient` rectangle so there won't be any empty space at the bottom. By default, the cell height is the height of the font that is used.

```
virtual void SetFont(PEGINT FontIndex)
```

PegVirtualVList overrides the `SeFont()` function so that the cell height will reflect the height of the font.

```
virtual void SetSelected(PEGINT Index)
```

This function is used to give focus to a particular element in the list. If the element at `Index` is not visible, the list will be scrolled.

```
void Unselect(PEGBOOL Redraw = TRUE)
```

This function unselects the currently selected item.

4.19.8 Protected Members

```
virtual void AssignData(PEGINT ObjIdx, PEGINT ListIdx)
```

This function is used to assign a data item to an individual PegPrompt in the list. This is called for each prompt every time the list scrolls. This is essentially the same as calling `DataSet()`, but an application may have other requirements for this in a derived version if the data is not simple text.

```
virtual void AllocObjects(PEGINT NumObjects)
```

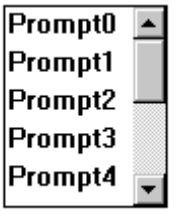
This function allocates an array of PegPrompts. This can be overridden to allocate any type of object.

```
virtual Pegthing *ConstructObject(PegRect Rect, PEGINT
      Index)
```

This function creates an individual PegPrompt object and assigns its initial text. This can be overridden to construct any type of object.

4.19.9 Examples:

The following is a PegVirtualVList:



The following example creates a PegVirtualVList:

```
PEGCHAR *gpListData[] = {
    "Prompt0",
    "Prompt1",
    "Prompt2",
    "Prompt3",
    "Prompt4",
    "Prompt5",
    "Prompt6",
    "Prompt7",
    "Prompt8",
    "Prompt9",
};

void MyWindow::AddVList(void)
{
    PegRect Rect;
    Rect.Set(10, 10, 80, 180);

    PegVirtualVList *pList = new PegVirtualVList(Rect, 0,
        FF_THIN|TJ_LEFT);
    pList->AssignVirtualList(gpListData, 10);
```

```
    pList->SetScrollMode(WSM_VSCROLL);
    Add(pList);
}
```

4.20 PegVertList

4.20.1 Overview

PegVertList is a container class for displaying a scrolling list of child objects. PegVertList automatically positions and sizes child objects. It is therefore not necessary to manually position objects before adding them to PegVertList.

The LAST child added to the list will be displayed at the topmost position in the list if the `Add()` function is used to add children. The order of display can be reversed by using the function `AddToEnd()` to add children to the list.

Child objects are positioned when the list receives the `PM_SHOW` message, which is a system message sent automatically when the list is first displayed. PegVertList forces the width of child objects to match the width of the list client area. Because the height of each child object is not modified, you should use the correct object height when constructing child objects.

4.20.2 See Also

`PegHorzList`

`PegList`

`PegWindow`

4.20.3 Style Flags

PegVertList supports the following style flags:

`FF_NONE`	No Frame
`FF_THIN`	Thin Frame
`FF_RAISED`	Raised 3D Frame
`FF_RECESSED`	Recessed 3D Frame
`FF_THICK`	Thick 3D Frame

The styles for PegVertList are identical to the PegWindow styles. In addition, scrolling is enabled in PegList in the same way as in PegWindow, by using the `SetScrollMode()` function.

4.20.4 Signals

PegVertList sends `PSF_LIST_SELECT` signals to the parent object. This message contains:

`Message.pSource` = Pointer to selected object.
`Message.Param` = ID of selected list item.
`Message.pTarget` = Pointer to list parent object.

4.20.5 Derivation

PegVertList derives from PegList.

4.20.6 Constructors:

```
PegVertList(const PegRect &Rect, PEGUINT Id = 0,
     PEGULONG Style = FF_THIN)
```

This constructor creates a PegVertList object. The `Rect` parameter determines the position and size of the list. The list children are automatically positioned by the list object.

4.20.7 Public Functions:

None.

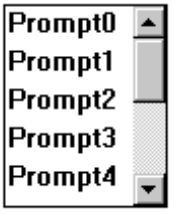
4.20.8 Protected Members

```
virtual void PositionChildren(void)
```

This function is used to calculate the positions of all the child objects.

4.20.9 Examples:

The following is a PegVertList with PegPrompt children:



The following example creates a PegVertList and adds several PegIconButton children. The bitmaps for the bitmap buttons can be generated using PegImageConvert.

```
void MyWindow::AddHList(void)
{
    PegRect Rect;
    Rect.Set(10, 10, 80, 180);

    PegVertList *pList = new PegVertList(Rect);

    Rect.Set(0, 0, 0, 34);
    pList->Add(new PegIconButton(Rect, BID_THUNDER));
    pList->Add(new PegIconButton(Rect, BID_LIGHT));
    pList->Add(new PegIconButton(Rect, BID_SATELLITE));
    pList->Add(new PegIconButton(Rect, BID_DYNAMITE));
    pList->Add(new PegIconButton(Rect, BID_APPLE));

    pList->SetScrollMode(WSM_VSCROLL);
    Add(pList);

}
```

4.21 PegWindow

4.21.1 Overview

PegWindow defines a basic rectangular area on the screen. Important functionality added by PegWindow includes the concept of scrolling a virtual client area.

PegWindow objects may be used as-is, but more commonly they serve as the base class for the more refined window classes such as PegDecoratedWindow and PegDialog. PegWindow can be resized by the user, has a virtual client area, has one of several frame styles, and controls nonclient-area scroll bars.

A PegWindow drawn with a raised border provides a blank panel that is useful for splash screens. A PegWindow with no border is useful as a container for other objects. The window can be moved to different locations or added to different parent objects, and all of the window's children will move with the window.

A simple way to create a window with a virtual scrolling client area is to nest a large window within the client area of a parent window. An example of this is included in the examples section for PegWindow.

PegWindow and PegWindow-derived classes are also Viewports by default. This means that objects underneath PegWindow are not allowed to obscure the screen area owned by the window. This is an important performance-enhancing feature of PEG, and it also provides improved visual appeal.

When a PegWindow object is resized, all children of the window receive `PM_PARENTSIZED` system messages. This message can be caught by child objects to position and size themselves relative to their parent window.

A window defaults to `PSF_MOVEABLE` and `PSF_SIZEABLE` if the `FF_THICK` frame style is used. For all other frame styles, the window will default to non-sizeable and non-moveable. This can be overridden by calling `AddStatus()` or `RemoveStatus()` after constructing the window.

A window defaults to having no scroll bars, and automatic scrolling mode is disabled. The member function `SetScrollMode()` is used to alter the scrolling mode of the window.

4.21.2 See Also

How Scrolling Works

Viewports

PegDecoratedWindow

PegDialog

PegMessageWindow

PegMLMessageWindow

4.21.3 Style Flags

PegWindow supports the following styles:

FF_NONE	No Frame
FF_THIN	Thin Frame
FF_RAISED	Raised 3D Frame
FF_RECESSED	Recessed 3D Frame
FF_THICK	Thick 3D Frame

4.21.4 Signals

None.

4.21.5 Derivation

PegWindow derives from PegTextThing.

4.21.6 Constructors:

```
PegWindow(const PegRect &Rect, PEGULONG Style =
     FF_THICK)
```

This constructor is used when the initial size and position of the window are known at the time the window is created. `Rect` defines the `mReal` position of the window.

```
PegWindow(PEGULONG Style = FF_THICK)
```

This constructor is used when the default position of the window is not known at the time the window is constructed, or when the window size and position are self-determined. When this constructor is used, the window should determine its position before or when the `PM_SHOW` message is received. This should be done in two steps:

1) Set the window `mReal` member variable.

2) Call `InitClient()`, to initialize the window's client area.

4.21.7 Public Functions:

```
virtual void Add(PegThing *pWho, PEGBOOL Show = TRUE)
```

PegWindow overrides this method to add the `pWho` object and to give this object focus, if necessary.

```
virtual void AddIcon(PegIcon *pIcon)
```

This function positions and adds a PegIcon object to the window. The window will automatically determine the correct position for the PegIcon object. It should be noted that any PegWindow-derived object can contain icons. It is sometimes assumed that only PegPresentationManager can act as a PegIcon container, which is not the case.

```
PEGUBYTE BlendMode(void)
```

Returns the window's current blend mode. This function is only available if `PEG_LAYERED_WINDOWS` is turned on.

```
PEGUBYTE BlendRatio(void)
```

Returns the window's current blend ratio. This function is only available if `PEG_LAYERED_WINDOWS` is turned on.

```
PEGBOOL CheckAutoScroll(void)
```

If either `WSM_AUTOVSCROLL` or `WSM_AUTOHSCROLL` scroll modes are set, this function is called when the window is resized to determine if scroll bars need to be updated. Derived classes also call this function at times when the scroll bars may need to be added or removed.

```
virtual PegScroll *CreateHScroll()

virtual PegScroll *CreateVScroll()
```

These functions create default PegVScroll and PegHScroll non-client scroll bars. If a derived window requires a different type of scroll bar, these functions should be overridden to create the user-defined objects.

```
PEGUBYTE CurrentMoveMode(void)
```

Returns the current move mode of the window.

```
virtual void Draw(const PegRect &Invalid)
```

The `Draw()` function of PegWindow is broken out into two sections. The function `DrawFrame()` is called first to draw the window frame, and then the child objects of the window are drawn. Classes derived from PegWindow may find it convenient to use the `DrawFrame()` function as part of an overridden `Draw()` routine.

```
virtual void DrawFrame(PEGBOOL Fill = TRUE)
```

This function draws the window frame using the current frame style flags. This function can be useful to PegWindow derived classes that have overridden the `Draw()` function.

```
virtual PEGINT Execute(PEGBOOL AutoAdd = TRUE)
```

This function can be called to execute any window modally. The `Execute()` function does not return until the window is closed. Under the MULTITHREAD model, calling execute from a task other then PegTask causes the window and any subsequent windows to execute from within the calling task's thread.

In most cases, a window should be added to PegPresentationManager before calling the `Execute()` function of the window. The exception is when MULTITHREAD operation is enabled, and the user wants to execute the window from within a secondary thread. In this case, calling `Execute()` will automatically add the window to PegPresentationManager, and the window will run from within the thread of the calling task.

```
virtual void GetHScrollInfo(PegScrollInfo *pPut)
```

This function is called by PegWindow and non-client PegHScroll children to determine the appearance and range of the window's horizontal scroll bar. The default PegWindow implementation queries the position of all client-area child objects to determine the horizontal scroll information. This

function is often overridden in derived PegWindow classes to provide custom scrolling operation.

```
virtual PegBitmap *GetIcon(void)
```

This function returns a pointer to the PegBitmap currently associated with the window Icon.

```
virtual PEGUBYTE GetScrollMode(void)
```

Returns the current window scroll mode.

```
virtual void GetVScrollInfo(PegScrollInfo *pPut)
```

This function is called by PegWindow and non-client PegVScroll children to determine the appearance and range of the window's vertical scroll bar. The default PegWindow implementation queries the position of all client-area child objects to determine the vertical scroll information. This function is often overridden in derived PegWindow classes to provide custom scrolling operation.

```
PEGINT GlobalModalExecute(void)
```

Normally a call to `Execute()` will make a window modal within its own task, while windows in other tasks can still get focus and read mouse/keyboard input. This function makes a window modal over all tasks. It is only provided if #define `PEG_MULTITHREAD` is enabled in the configuration file `pconfig.hpp`.

```
virtual void InitClient(void)
```

Initializes the client region, taking into account the presence and size of a border.

```
PEGBOOL IsMaximized(void)
```

Returns TRUE if the window is maximized, else FALSE.

```
PEGBOOL IsModal(void)
```

Returns TRUE if the window is modal, else FALSE.

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

PegWindow overrides the `Message` function, primarily to catch mouse clicks for resizing the window. PegWindow also catches `PEG_SIGNAL(IDB_CLOSE, PSF_CLICKED)` to close the window.

```
virtual void Resize(const PegRect &NewSize)
```

PegWindow overrides the `Resize()` function to send `PM_PARENTSIZED` notifications to child objects, and to update scroll bars. This function is often overridden to provide custom operation.

```
PEGBOOL SetBlendMode(PEGUBYTE Mode, PEGUBYTE Ratio =
    0)
```

This function sets the current blend mode and ratio of the window. This function is only available if `PEG_LAYERED_WINDOWS` is turned on.

```
void SetBlendRatio(PEGUBYTE Ratio)
```

This function sets the current blend ratio of the window. This function is only available if `PEG_LAYERED_WINDOWS` is turned on.

```
virtual void SetTransitionMode(PEGUBYTE EnterMode,
    PEGUBYTE ExitMode = TRANSITION_MODE_NONE,
    PEGUBYTE Steps = 8, PEGUBYTE Timer = 1, PEGBOOL
    UseSurface = 0)
```

This function allows the user to set a transition to take place when the window is first displayed and/or when the window is removed. If `UseSurface` is set to TRUE, then the window is drawn into a surface during the transition.

```
virtual void SetIcon(PEGINT BitmapId)
```

This function assigns the bitmap that will be associated with the window icon.

```
virtual void SetScrollMode(PEGUINT Mode)
```

This function sets the operation of non-client area scroll bars. Non-client area scroll bars are used to provide the appearance of a virtual-client area. The available scroll modes are:

`WSM_AUTOVSCROLL`...Add vertical scroll bar when needed
`WSM_VSCROLL`.....Add vertical scroll bar always
`WSM_AUTOHSCROLL`...Add horizontal scroll bar when needed
`WSM_HSCROLL`.....Add horizontal scroll bar always
`WSM_AUTOSCROLL`....`WSM_AUTOVSCROLL`|`WSM_AUTOHSCROLL`
`WSM_CONTINUOUS`....Continuous, smooth scrolling

Automatic scrolling relies on the values returned by the `GetVScrollInfo()` and `GetHScrollInfo()` functions to determine when each scroll bar is

required. If the PegScrollInfo.Visible value is ≥ the overall scroll range, the corresponding scroll bar is not required.

The WSM_CONTINUOUS mode can be included with any other modes. This flag causes the scroll bars to send scroll messages continuously as they are dragged by the user, rather than the default operation, which is to send a scroll message only when the scroll button is released.

Continuous scrolling requires greater hardware performance and/or hardware acceleration in the video controller to provide the best smooth scrolling. Performance-limited platforms should not use the WSM_CONTINUOUS flag.

```
virtual void SetWallpaper(PEGINT Wallpaper, PEGUBYTE
     Tile = 1)
```

This function assigns a PegBitmap that the window will draw into the client area of the window. By default, if a wallpaper is assigned it will be tiled to fill the mClient rectangle of the window. If Tile is false, the bitmap will be centered within the window client area.

4.21.8 Protected Members:

```
void BeginEnterTransition(void)
```

This function begins a transition when a window is first displayed.

```
void BeginExitTransition(void)
```

This function begins a transition when a window is removed from the display.

```
virtual PEGUBYTE BorderContains(PegPoint Point)
```

This function is called on receipt of PM_POINTERMOVE messages to determine if the mouse is over the window border. The return value is 0 if the mouse pointer is not over the window border, or a MoveMode. The MoveModes are defined in the file \peg\include\pwindow.hpp.

```
PEGUBYTE mScrollMode
```

The current window scroll mode.

```
PEGUBYTE mModal
```

TRUE if the window is executing modally.

```
PEGUBYTE mMaximized
```

TRUE if the window is currently maximized.

```
PEGUBYTE mMoveMode
```

The current move or resize mode of the window.

```
virtual void MoveClientObjects(PEGINT xShift, PEGINT
      yShift, PEGBOOL Redraw = TRUE)
```

This function shifts all child objects the specified amount in the x and y directions.

```
PegScroll *mpHScroll
```

Pointer to non-client horizontal scroll bar if present, else NULL.

```
PegScroll *mpVScroll
```

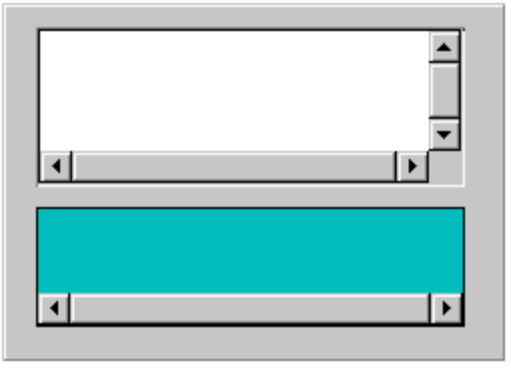
Pointer to non-client vertical scroll bar if present, else NULL.

4.21.9 Examples:

Default PegWindow:

PegWindow with `FF_RAISED` frame style:

Two PegWindow objects nested within another PegWindow. The child windows have scrolling enabled:



The following example creates a PegWindow and adds the window to PegPresentationManager. The window will have a default (thick) border, will be 190 pixels wide by 110 pixels tall, and will be centered on the screen.

```
void SomeObject::CreateWindow(void)
{
    PegRect WinSize;
    WinSize.Set(10, 10, 200, 120);
    PegWindow *pWin = new PegWindow(WinSize);
    Presentation()->Center(pWin);
    Presentation()->Add(pWin);
}
```

The following example will create a PegWindow with a recessed frame and add the window to the current object. The window will fill the client area of the current object.

```
void SomeObject::AddClientWindow(void)
{
    PegWindow *pWin = new PegWindow(mClient, FF_RECESSED);
    Add(pWin);
}
```

The following example creates two PegWindow objects. The second window will be a child of the first. The second window is also much larger than the first. The outer parent window will be configured to provide scroll bars, so that the user can pan to display all areas of the child window. The resulting parent/child window combination will be centered on the screen.

```
void SomeObject::CreateScrollingWindow(void)
{
    PegRect ParentRect, ChildRect;

    ParentRect.Set(0, 0, 200, 140);
    ChildRect.Set(0, 0, 800, 800);

    PegWindow *pOuter = new PegWindow(ParentRect);
    PegWindow *pChild = new PegWindow(ChildRect, FF_NONE);
    pOuter->Center(pChild);
    pOuter->Add(pChild);
    pOuter->SetScrollMode(WSM_AUTOSCROLL);

    Presentation()->Center(pOuter);
    Presentation()->Add(pOuter);
}
```

CHAPTER 5

CHARTING CLASSES

PegChart
PegLineChart
PegMultiLineChart
PegStripChart

5.1 PegChart

5.1.1 Overview

PegChart is the base class for all of the charts in the PEG library. PegChart is a virtual base class; therefore, it is not possible to instantiate an object of this type directly at run time. The main job of PegChart is to provide a basic framework for its derived children. It does this by keeping track of an extra style variable, over and above the one retained by PegThing. It also provides the algorithms for calculating the layout of the chart based on whether or not the chart will be drawing labels and tick marks. And, lastly, it provides drawing methods for the common elements of a chart (i.e., x and y tick marks, labels, and grid lines).

5.1.2 Style Flags

PegChart supports the following styles:

`CS_DRAWXGRID`	This flag causes grid lines to be drawn along the x axis in the chart region.
`CS_DRAWYGRID`	This flag causes grid lines to be drawn along the y axis in the chart region.
`CS_DRAWXTICS`	This flag causes tick marks to be drawn along the x axis at specified intervals.
`CS_DRAWYTICS`	This flag causes tick marks to be drawn along the y axis at specified intervals.
`CS_AUTOSIZE`	If this flag is set, the chart will always take up the entire `mClient` region of its parent. When the parent is resized or moved, the PegChart responds to the message and resizes itself appropriately.
`CS_DUALYTICS`	Setting this flag causes the chart to draw y tick marks on both the left and right side of the chart area. By default, the chart only draws tick marks on the left side.
`CS_DUALYLABELS`	Used in conjunction with `CS_DUALYTICS`, this flag forces the chart to draw y-axis labels on both the left and right sides of the chart area.

CS_XAXISONZEROY	By default, the chart will draw the x axis at the bottom of the chart region. There are occasions where it is more effective for determining data position when the x axis is, instead, placed horizontal to 0 on the y axis. Setting this flag affects this behavior. The x axis will be drawn inside of the chart region, perpendicular to the y axis, at the 0 y position.
CS_DRAWLINEFILL	This flag tells the chart to draw a filled polygon from each line segment to 0 on the y axis.
CS_DRAWXLABELS	This flag causes incremental labels to be drawn along the x axis. The label text is based on the minimum and maximum possible values of x.
CS_DRAWYLABELS	This flag causes incremental labels to be drawn along the y axis. The label text is based on the minimum and maximum possible values of y.

5.1.3 Signals

PegChart does not send any signals.

5.1.4 Derivation

PegChart is derived from PegThing.

5.1.5 Constructor:

```
PegChart(const PegRect &Rect, PEGLONG MinX, PEGLONG
     MaxX, PEGLONG MinY, PEGLONG MaxY, PEGUINT
     MajorXScale = 0, PEGUINT MajorYScale = 0)
```

The constructor is fairly straightforward. Like most PegThing-derived objects, you pass it the rectangle you wish for it to occupy. If you have CS_AUTOSIZE turned on, you may simply want to pass it the parent's mClient rectangle.

The next four PEGLONG values specify the minimum and maximum values for x and y. The last two parameters are for setting the major tick mark frequency for the x and y axis, respectively. For instance, if your minimum y value is -100 and your maximum y value is 900, and you specify a y scale of 100, if the CS_DRAWYTICS bit is set in the extended style flag, you will see tick marks on the y axis starting at -100 and incrementing 100 all the way to

900. Therefore, there will be 11 tick marks drawn on the y axis. If you were to also turn on the `CS_DRAWYGRID` bit, you would see a grid line at the same interval as the tick marks. Label scaling, or interval, is independent of the tick mark/grid line scaling. Therefore, it is possible to specify tick marks to appear at intervals of 100, while specifying labels to be drawn every 200. No matter the scaling, all drawing starts at the minimum value and works its way toward the maximum value until it meets or exceeds the maximum value. This holds true for both the x and y axis.

5.1.6 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegChart overrides the `Draw()` function to draw the background color of the chart area, and, if specified, the optional tick marks, labels, and grid lines.

```
PegRect GetChartRegion() const
```

Returns the rectangle that represents the area where the chart is actually being drawn. This rectangle's size and position is based on the position and size of its parent (if `CS_AUTOSIZE` is set), as well as any tick marks or labels that it may be drawing. In other words, this rectangle roughly corresponds to a typical `mClient` rectangle in a standard PegThing.

```
PEGUINT GetExStyle() const
```

Returns the current value of `mExStyle`. Possible values are listed here.

```
PegFont *GetFont() const
```

Returns a pointer to the current font being used to draw axis labels.

```
PEGUINT GetMajorTicSize() const
```

Returns the length, in pixels, used to draw major tics.

```
PEGUINT GetMajorXScale() const
```

Returns the tick interval for the major scale on the x axis.

```
PEGUINT GetMajorYScale() const
```

Returns the tick interval for the major scale on the y axis.

```
PEGLONG GetMaxX() const
```

Returns the maximum allowable value for x.

```
PEGLONG GetMaxY() const
```

Returns the maximum allowable value for y.

```
PEGUINT GetMinorTicSize() const
```

Returns the length, in pixels, used to draw minor ticks.

```
PEGUINT GetMinorXScale() const
```

Returns the tick interval for the minor scale on the x axis.

```
PEGUINT GetMinorYScale() const
```

Returns the tick interval for the minor scale on the y axis.

```
PEGLONG GetMinX() const
```

Returns the minimum allowable value for x

```
PEGLONG GetMinY() const
```

Returns the minimum allowable value for y.

```
PEGUINT GetXLabelHeight() const
```

Returns the height of the labels along the x axis.

```
PEGUINT GetXLabelScale() const
```

Returns the interval for drawing labels along the x axis.

```
PEGUINT GetYLabelScale() const
```

Returns the interval for drawing labels along the y axis.

```
PEGUINT GetYLabelWidth() const
```

Returns the width of the labels along the y axis.

```
virtual void MapDataToPoint(PegChartPoint *pPoint)
```

This method converts the data points held in `pPoint` to screen coordinates based on the size of the `mChartRegion` of the chart, and the minimum and maximum allowable values for x and y. It then puts the coordinates in the appropriate `pPoint` members. This method is typically not called by objects outside of the chart classes.

```
virtual void MapPointToData(PegChartPoint *pPoint)
```

This method does the opposite of `MapDataToPoint` in that it takes screen coordinates and converts them to data points based on the same criteria as

`MapDataToPoint`. This method is typically not called by objects outside of the chart classes.

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

PegChart overrides the `Message` method in order to provide appropriate layout recalculation when its parent is moved or, optionally, resized.

```
virtual void ParentShift(PEGINT xOffset, PEGINT
     yOffset)
```

PegChart overrides the `PegThing::ParentShift` function in order to make sure that the chart region rectangle also gets shifted.

```
virtual void RecalcLayout(PEGBOOL Redraw = TRUE)
```

This method calculates the region used for drawing the actual chart (represented internally by `mChartRegion`). This method takes into account the drawing of tick marks and labels. This method is typically not called by objects outside of the chart classes.

```
virtual void RecalcSize(const PegRect &NewRect,
     PEGBOOL Redraw = TRUE)
```

This method resets the chart's size to the new rectangle, then calls `RecalcLayout` and passes through Redraw. This method is typically not called by objects outside of the chart classes.

```
void SetExStyle(PEGULONG Style)
```

Sets the current value of `mExStyle`. Possible values are listed here. These values may be bitwise OR'd together.

```
void SetFont(PEGINT FontIndex)
```

Sets the font used for drawing axis labels.

```
void SetMajorTicSize(PEGUINT Size)
```

Sets the length for the major tick marks on each axis.

```
void SetMajorXScale(PEGUINT Scale)
```

Sets the tick interval for the major scale on the x axis.

```
void SetMajorYScale(PEGUINT Scale)
```

Sets the tick interval for the major scale on the y axis.

```
void SetMaxX(PEGLONG Data)
```

Sets the maximum value for x.

```
void SetMaxY(PEGLONG Data)
```
Sets the maximum value for y.

```
void SetMinorTicSize(PEGUINT Size)
```
Sets the length for the minor tick marks on each axis.

```
void SetMinorXScale(PEGUINT Scale)
```
Sets the tick interval for the minor scale on the x axis.

```
void SetMinorYScale(PEGUINT Scale)
```
Sets the tick interval for the minor scale on the y axis.

```
void SetMinX(PEGLONG Data)
```
Sets the minimum value for x.

```
void SetMinY(PEGLONG Data)
```
Sets the minimum value for y.

```
void SetXLabelHeight(PEGUINT Height)
```
Sets the height of the labels along the x axis.

```
void SetXLabelScale(PEGUINT Scale)
```
Sets the interval for drawing labels along the x axis.

```
void SetYLabelScale(PEGUINT Scale)
```
Sets the interval for drawing labels along the y axis.

```
void SetYLabelWidth(PEGUINT Width)
```
Sets the width of the labels along the y axis.

5.1.7 Protected Members

```
virtual void DrawXGrid(const PegRect &Invalid)
```
This function draws gridlines parallel with the x axis through the chart region.

```
virtual void DrawXLabels(const PegRect &Invalid)
```
This function draws numeric labels along the x axis.

```
virtual void DrawXTics(const PegRect &Invalid)
```
This function draws major and/or minor tick marks along the x axis.

```
virtual void DrawYGrid(const PegRect &Invalid)
```

This function draws gridlines parallel with the y axis through the chart region.

```
virtual void DrawYLabels(const PegRect &Invalid)
```

This function draws numeric labels along the y axis.

```
virtual void DrawYTics(const PegRect &Invalid)
```

This function draws major and/or minor tic marks along the y axis.

5.1.8 Examples:

PegChart is not directly instantiable. See PegLineChart, PegMultiLineChart or PegStripChart for an example of appropriate usage.

5.2 PegLineChart

5.2.1 Overview

PegLineChart is a simple abstract down from PegChart and supports the displaying of a single line on a given scale.

5.2.2 See Also

PegMultiLineChart

5.2.3 Style Flags

Please see PegChart for a complete description of style flags.

5.2.4 Signals

PegLineChart does not send any signals.

5.2.5 Derivation

PegLineChart is derived from PegChart.

5.2.6 Constructor:

```
PegLineChart(const PegRect &Rect, PEGLONG MinX,
     PEGLONG MaxX, PEGLONG MinY, PEGLONG MaxY,
     PEGUINT MajorXScale = 0, PEGUINT MajorYScale =
     0)
```

The constructor is fairly straightforward. Like most PegThing-derived objects, you pass it the rectangle you wish for it to occupy. If you have `CS_AUTOSIZE` turned on, you may simply want to pass it the parent's `mClient` rectangle.

The next four `PEGLONG` values specify the minimum and maximum values for x and y. The last two parameters are for setting the major tick mark frequency for the x and y axis, respectively. For instance, if your minimum y value is -100 and your maximum y value is 900, and you specify a y scale of 100, if the `CS_DRAWYTICS` bit is set in the extended style flag, you will see tick marks on the y axis starting at -100 and incrementing 100 all the way to 900. Therefore, there will be 11 tick marks drawn on the y axis. If you were

also to turn on the CS_DRAWYGRID bit, you would see a grid line at the same interval as the tick marks. Label scaling, or interval, is independent of the tick mark/grid line scaling. Therefore, it is possible to specify tick marks to appear at intervals of 100, while specifying labels to be drawn every 200. No matter the scaling, all drawing starts at the minimum value and works its way toward the maximum value until it meets or exceeds the maximum value. This holds true for both the x and y axis.

5.2.7 Public Functions:

```
PegChartPoint *AddPoint(PEGLONG X, PEGLONG Y)
```

This method adds a new point to the end of the line segment. It returns a pointer to the newly created point.

```
virtual void Draw(const PegRect &Invalid)
```

PegChart overrides the Draw() function to draw the line segments that make up the line.

```
PegChartPoint *GetFirstPoint() const
```

Returns the first point of the line segment.

```
PEGCOLOR GetLineColor() const
```

Returns the PEGCOLOR used for determining the color of the line.

```
PegChartPoint *InsertPoint(PegChartPoint *pPoint,
      PEGLONG X, PEGLONG Y)
```

This method inserts a new point after the point pointed to by pPoint. It returns a pointer to the newly created point.

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

PegChart overrides the Message method in order to provide appropriate layout recalculation when its parent is moved or, optionally, resized.

```
void RecalcLine(void)
```

This method forces a recalculation of the screen coordinates for every point on the line.

```
virtual void RecalcSize(const PegRect& NewRect,
      PEGBOOL Redraw = TRUE)
```

This method is overridden in order to ensure that the line data is up to date. In other words, if the chart is moved or resized, the screen coordinates associated with a given data point will change. In order to keep up with

these changes, this method calls the `Resize` method to update the screen coordinates of all the data points associated with the line. If the Redraw flag is TRUE, the chart is redrawn.

```
PegChartPoint *RemovePoint(PegChartPoint *pPoint)
```

This method removes the point pointed to by `pPoint`. It returns a pointer to the point preceding the deleted point, or NULL if there is none.

```
void ResetLine(void)
```

This method removes all of the points associated with the line.

```
virtual void Resize(const PegRect &NewRect)
```

The PegLineChart class overrides the `Resize()` function to adjust the line chart's location and size as specified by the rectangle `NewRect`. The function calls the PegChart base class `RecalcLayout()` function but inhibits redrawing of the chart. Function `RecalcLine()` is then used to reposition all line data points within the revised chart. This function is used privately by this chart class.

```
PEGCOLOR SetLineColor(PEGCOLOR Color)
```

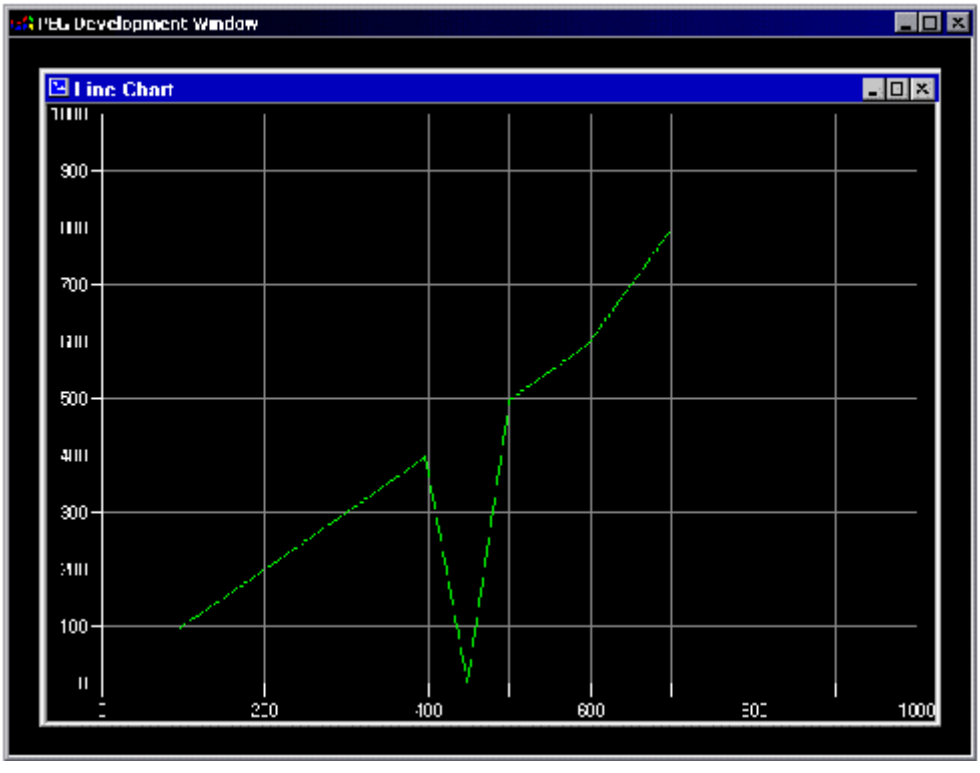
Sets the `PEGCOLOR` used for determining the color of the line. The function returns the color that was previously being used to draw chart lines.

5.2.8 Examples:

The following code snippet produces the PegLineChart pictured below.

```
    PegLineChart* pLineChart = new PegLineChart(Rect, 0,
1000, 0,
        1000, 100, 100)
    pLineChart->SetXLabelScale(200);
    pLineChart->SetExStyle(CS_DRAWXTICS | CS_DRAWYTICS |
        CS_DRAWXGRID | CS_DRAWYGRID | CS_AUTOSIZE |
        CS_DRAWYLABELS | CS_DRAWXLABELS);
    pLineChart->AddPoint(100, 100);
    pLineChart->AddPoint(200, 200);
    pLineChart->AddPoint(300, 300);
    PegChartPoint *pPoint = pLineChart->AddPoint(400, 400);
    pLineChart->AddPoint(500, 500);
    pLineChart->AddPoint(600, 600);
    pLineChart->AddPoint(700, 800);
    pLineChart->InsertPoint(pPoint, 450, 0);
```

PeglineChart example sereen shot



5.3 PegMultiLineChart

5.3.1 Overview

PegMultiLineChart supports the drawing of discrete lines that use the same x and y scaling.

5.3.2 See Also

PegLineChart

5.3.3 Style Flags

For a complete listing of styles supported by PegMultiLineChart, see PegChart.

5.3.4 Signals

PegMultiLineChart does not send any signals.

5.3.5 Derivation

PegMultiLineChart is derived from PegChart.

5.3.6 Constructor:

```
PegMultiLineChart(const PegRect &Rect, PEGLONG MinX,
     PEGLONG MaxX, PEGLONG MinY, PEGLONG MaxY,
     PEGUINT MajorXScale = 0, PEGUINT MajorYScale =
     0)
```

The constructor is fairly straightforward. Like most PegThing-derived objects, you pass it the rectangle you wish for it to occupy. If you have `CS_AUTOSIZE` turned on, you may just want to pass it the parent's `mClient` rectangle.

The next four `PEGLONG` values specify the minimum and maximum values for x and y. The last two parameters are for setting the major tick mark frequency for the x and y axis, respectively. For instance, if your minimum y value is -100 and your maximum Y value is 900, and you specify a y scale of 100, if the `CS_DRAWYTICS` bit is set in the extended style flag, you will see tick marks on the y axis starting at -100 and incrementing 100 all the way to

900. Therefore, there will be 11 tick marks drawn on the y axis. If you were to also turn on the `CS_DRAWYGRID` bit, you would see a grid line at the same interval as the tick marks. Label scaling, or interval, is independent of the tick mark/grid line scaling. Therefore, it is possible to specify tick marks to appear at intervals of 100, while specifying labels to be drawn every 200. No matter the scaling, all drawing starts at the minimum value and works its way toward the maximum value until it meets or exceeds the maximum value. This holds true for both the x and y axis.

5.3.7 Public Functions:

```
virtual PEGUBYTE AddLine(PEGCOLOR Color)
```

This method is used to add a new line to the chart. `Color` is used to draw the line in the specified color. The return value is the ID of the new line. Upon failure, the method will return 0. This chart supports up to 255 simultaneous lines.

```
PegChartPoint *AddPoint(PEGUBYTE Id, PEGLONG X,
      PEGLONG Y, PEGBOOL Redraw = TRUE)
```

This method adds a new point to the end of the line segment identified by `Id`. It returns a pointer to the newly created point.

```
virtual void Draw(const PegRect &Invalid)
```

PegMultiLineChart overrides the `Draw()` function to draw the individual lines.

```
void DrawNewLineData(PegChartLine *pLine,
      PegChartPoint *pNew, PegChartPoint *pPrevious =
      NULL)
```

This function determines the area of the chart that has changed due to a new point and then redraws that area. This method is called from the `AddPoint` and `InsertPoint` methods when new data is added to the line. If `pPrevious` is NULL, then the entire line is redrawn.

```
PegChartLine *GetFirstLine() const
```

Returns the first line segment, or NULL if there are no lines in the chart.

```
PegChartLine *GetLineFromID(PEGUBYTE Id)
```

Returns a pointer to the line segment that has the ID `Id`, or NULL if it could not find it.

```
PegChartPoint *InsertPoint(PEGUBYTE Id, PegChartPoint
     *pPoint, PEGLONG X, PEGLONG Y, PEGBOOL Redraw =
     TRUE)
```

This method inserts a new point ***after*** the point pointed to by `pPoint` for the given line segment. It returns a pointer to the newly created point.

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

PegMultiLineChart overrides the `Message` method in order to provide appropriate layout recalculation when its parent is moved or, optionally, resized.

```
void RecalcLine(PEGUBYTE Id, PEGBOOL Redraw = TRUE)
```

This method forces a recalculation of the screen coordinates for every point on the given line. If the `Redraw` flag is TRUE, the line is redrawn.

```
virtual void RecalcSize(const PegRect &NewRect,
     PEGBOOL Redraw = TRUE)
```

This method is overridden in order to ensure that the line data is up to date. In other words, if the chart is moved or resized, the screen coordinates associated with a given data point will change. In order to keep up with these changes, this method calls the `Resize` method to update the screen coordinates of all the data points associated with the line. If the `Redraw` flag is TRUE, the chart is redrawn.

```
PEGBOOL RemoveLine(PEGUBYTE Id)
```

This method is used to remove a line from the chart. It returns a boolean describing its success. Once a line has been removed from the chart, its line ID becomes free. The line ID may subsequently be reused for a new line.

```
PegChartPoint *RemovePoint(PEGUBYTE Id, PegChartPoint
     *pPoint, PEGBOOL Redraw = TRUE)
```

This function removes the point at coordinates `pPoint` from the line specified by `Id`.

```
void ResetAllLines(PEGBOOL Redraw = TRUE)
```

This method removes all of the points associated with each line in the chart. If the `Redraw` flag is TRUE, each revised line is redrawn.

```
void ResetLine(PEGUBYTE Id, PEGBOOL Redraw = TRUE)
```

This method removes all of the points associated with the given line. If the `Redraw` flag is TRUE, the revised line is redrawn.

```
virtual void Resize(const PegRect &NewRect)
```

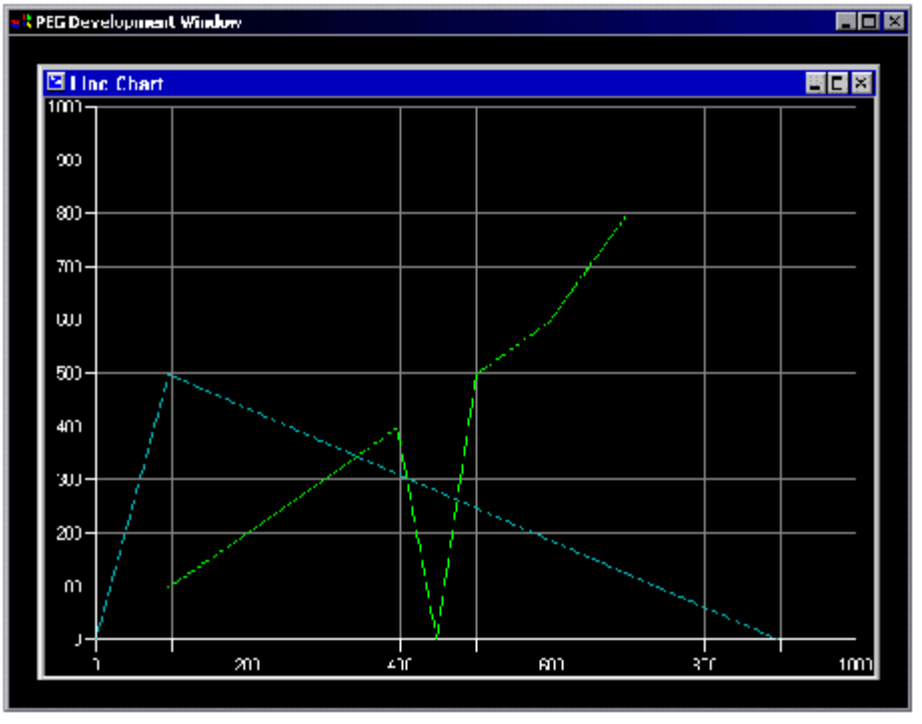
The PegMultiLineChart class overrides the `Resize` method to adjust the multiline chart's location and size as specified by the rectangle `NewRect`. The function calls the PegChart base class `RecalcLayout()` function but inhibits redrawing of the chart. Function `RecalcLine()` is then used to reposition all data points for all lines within the revised chart. This function is used privately by this chart class.

5.3.8 Examples:

The following code snippet produces the PegMultiLineChart pictured below.

```
    PegMultiLineChart *pLineChart = new
PegMultiLineChart(Rect, 0,
        1000, 0, 1000, 100
    pLineChart->SetXLabelScale(200);
    pLineChart->SetExStyle(CS_DRAWXTICS | CS_DRAWYTICS |
        CS_DRAWXGRID | CS_DRAWYGRID | CS_AUTOSIZE |
        CS_DRAWYLABELS | CS_DRAWXLABELS);

    PEGUBYTE LineID = pLineChart->AddLine(CLR_LIGHTGREEN);
    pLineChart->AddPoint(LineID, 100, 100);
    pLineChart->AddPoint(LineID, 200, 200);
    pLineChart->AddPoint(LineID, 300, 300);
    PegChartPoint *pPoint = pLineChart->AddPoint(LineID, 400,
400);
    pLineChart->AddPoint(LineID, 500, 500);
    pLineChart->AddPoint(LineID, 600, 600);
    pLineChart->AddPoint(LineID, 700, 800);
    pLineChart->InsertPoint(LineID, pPoint, 450, 0);

    PEGUBYTE LineID2 = pLineChart->AddLine(CLR_CYAN);
    pLineChart->AddPoint(LineID2, 0, 0);
    pLineChart->AddPoint(LineID2, 100, 500);
    pLineChart->AddPoint(LineID2, 900, 0);
```

PegMultiLineChart example screen shot


PEG Development Window
Line Chart
1000
900
800
700
600
500
400
300
200
100
0
0
200
400
600
800
1000

5.4 PegStripChart

5.4.1 Overview

PegStripChart supports the drawing of discrete lines plotted against the y axis, with new data samples added in series along the x axis.

5.4.2 See Also

PegChart

5.4.3 Style Flags

PegStripChart supports all of the styles described in PegChart as well as the following styles:

`CS_DRAWAGED`	If the chart is in paged mode (`CS_PAGED` is turned on), then the chart will redraw the current line in a secondary color when the data has reached the right side of the chart. New data, restarting over at the left side of the chart, will then overwrite this line. If this flag is turned off, all of the line segments are removed from the chart when the data reaches the right side of the chart.
`CS_DRAWLEADER`	This flag causes the chart to draw a vertical line indicating the position of the most recently added data and corresponding line segment.
`CS_DRAWXAXIS`	This flag draws a single horizontal line across the entire chart region. It is situated at 0 on the y scale. This line does not draw tick marks or labels. If `CS_XAXISONZEROY` is also turned on, then this flag is ignored.
`CS_PAGED`	This flag is mutually exclusive with `CS_SCROLLED`. This causes the data to be drawn along a leading edge as it is added to the line. When the edge reaches the right side of the chart region, the line wraps. At this point, the existing data becomes aged, and is drawn in a second color specified in the `AddLine` call. The behavior of the strip chart when this flag is on is somewhat akin to a heart beat monitor.

CS_SCROLLED	This flag is mutually exclusive with CS_PAGED. This causes the data to be drawn beginning from the left and moving toward the right side of the chart region. When there is enough data for the line to extend all the way across the chart region, the line scrolls itself. Thus, new data points are added on the right, and old data points are subtracted from the left. This behavior most closely resembles a seismograph.

5.4.4 Signals

PegStripChart does not send any signals.

5.4.5 Derivation

PegStripChart is derived from PegChart.

5.4.6 Constructor:

```
PegStripChart(const PegRect &Rect, PEGUINT Samples,
      PEGLONG MinY, PEGLONG MaxY, PEGUINT XScale = 0,
      PEGUINT YScale = 0)
```

The constructor is fairly straightforward. Like most PegThing-derived objects, you pass it the rectangle you wish for it to occupy. If you have CS_AUTOSIZE turned on, you may simply want to pass it the parent's mClient rectangle.

The Samples parameter specifies how many samples will fit into the chart region. If you would like to see 100 samples of data at one time, then you would set this parameter to 100.

The last two parameters are for setting the major tick mark frequency for the x and y axis, respectively. For instance, if your minimum y value is -100 and your maximum y value is 900, and you specify a y scale of 100, if the CS_DRAWYTICS bit is set in the extended style flag, you will see tick marks on the y axis starting at -100 and incrementing 100 all the way to 900. Therefore, there will be 11 tick marks drawn on the y axis. If you were to also turn on the CS_DRAWYGRID bit, you would see a grid line at the same interval as the tick marks. Label scaling, or interval, is independent of the tick mark/grid line scaling. Therefore, it is possible to specify tick marks to appear at intervals of 100, while specifying labels to be drawn every 200.

No matter the scaling, all drawing starts at the minimum value and works its way toward the maximum value until it meets or exceeds the maximum value. This holds true for both the x and y axis.

5.4.7 Public Functions:

```
PEGBOOL AddData(PEGUBYTE Id, PEGLONG RawData, PEGBOOL
     Redraw = TRUE)
```

This method adds a data point to a given line specified by `Id`. It returns TRUE if the data was added successfully, else FALSE.

```
PEGUBYTE AddLine(PEGCOLOR LineColor, PEGCOLOR
     AgedColor, PEGCOLOR FillColor = 0)
```

This adds a new line to the chart. The colors are used to draw the line when it is drawing new data, and in the case of the paged chart, the line color when it is drawing the aged data. It returns the ID of the newly added line.

If adding the line fails, a value of 0 will be returned. This chart supports up to 255 lines simultaneously.

```
void Draw(const PegRect &Invalid)
```

PegStripChart overrides the `Draw()` function to draw the individual lines.

```
PEGCOLOR GetLineAgedColor(PEGUBYTE Id)
```

Returns the `PEGCOLOR` used by line identified by `Id` to draw its historical data.

```
PEGCOLOR GetLineColor(PEGUBYTE Id)
```

Returns the `PEGCOLOR` used by line identified by `Id` to draw its current data.

```
PEGCOLOR GetLineFillColor(PEGUBYTE Id)
```

Returns the `PEGCOLOR` used by line identified by `Id` to draw the filled polygon from the current line segment to 0 on the y scale.

```
PEGCOLOR GetLineLeaderColor(PEGUBYTE Id)
```

This returns the current color of the leader. The leader is the straight vertical line drawn at the furthest point on the x axis where data is being inserted. The leader is only used when `CS_PAGED` is turned on. Every line may have its own leader color. By default, if a color is not initially specified for the leader, the normal line color is used.

```
PEGINT IndexToPointX(PEGUINT Index)
```

This method determines the screen location along the x axis given the current index. The location is based on the chart region size and the number of samples being put into that region. This method is used internally by the chart. It returns a screen point on the x axis as a `PEGINT` value.

```
PEGINT Message(const PegMessage& Mesg)
```

PegStripChart overrides the `Message` method in order to provide appropriate layout recalculation when its parent is moved or, optionally, resized.

```
virtual void RecalcSize(const PegRect &NewRect,
     PEGBOOL Redraw = TRUE)
```

This method is overridden in order to ensure that the line data is up to date. In other words, if the chart is moved or resized, the screen coordinates associated with a given data point will change. In order to keep up with these changes, this method calls the `Resize` method to update the screen coordinates of all the data points associated with each line.

```
PEGBOOL RemoveLine(PEGUBYTE Id)
```

This method removes the line with the ID of `Id`. It returns a boolean of success or failure.

```
virtual void Resize(const PegRect &NewRect)
```

This method overrides the `PegThing::Resize` method. Internally, it calls its own `RecalcLayout` method with the `NewRect` rectangle. It also calls the `PegThing::Resize` method to ensure proper layout.

```
void SetLineAgedColor(PEGUBYTE Id, PEGCOLOR Color)
```

This method sets the color used by line `Id` to draw its historical data.

```
void SetLineColor(PEGUBYTE Id, PEGCOLOR Color)
```

This method sets the color used by line identified by `Id` to draw its current data.

```
void SetLineFillColor(PEGUBYTE Id, PEGCOLOR Color)
```

This method sets the color used by line identified by `Id` to draw a filled polygon from the current line segment to 0 on the y axis.

```
void SetLineLeaderColor(PEGUBYTE Id, PEGCOLOR Color)
```

This method sets the color of the leader of the specified line denoted by the `Id` parameter.

```
PEGINT ValToPointY(PEGLONG Val)
```

This method converts a data value to a screen pixel location based on the size of the chart region and the minimum and maximum allowable values on the y axis. This method is used internally by the chart to plot data points on the screen. It returns the screen point on the y axis as a `PEGINT` value.

5.4.8 Protected Members

```
virtual void DrawInvalidChartRegion(const PegRect
     &Invalid)
```

This function draws only an invalidated section of the chart. This is called when adding new data points to the chart.

```
virtual void DrawLine(const PegRect &Invalid,
     PegStripChartLine *pLine)
```

This function draws an entire line on the chart.

```
virtual void DrawLineHistroy(const PegRect &Invalid,
     PegStripChartLine *pLine)
```

This is used for charts with the `CS_PAGED` style. After the data points reach the end of the chart, new data points start over at the beginning and all of the previous points become aged. This function draws the aged line.

```
virtual void DrawNewLineData(PegStripChartLine
     *pLine)
```

This function is responsible for drawing new data points and determining if the chart needs to be paged.

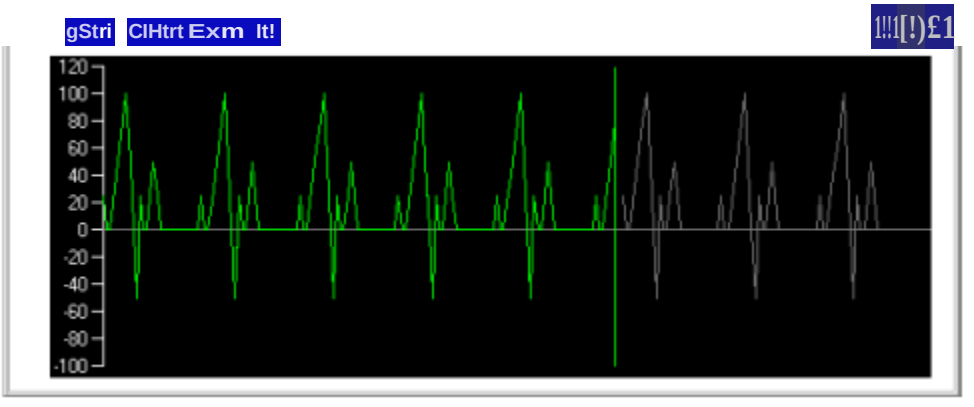
5.4.9 Examples:

The following code snippet produces the PegStripChart pictured below. This strip chart is using the `CS_PAGED` method of drawing.

```
    PegRect Rect;
    Rect.Set(40, 40, 590, 240);
    PegStripChart *pChart = new PegStripChart(Rect, 240, -
100,
        120, 0, 20);
```

```
    PEGUBYTE Lineid = pCha<t->AddLine(CLR_LIGHTGREEN,
        CLR_DARKGRAY);
    Add(pCha<t);
```

PegStripChart examp)e screen shot



CHAPTER 6

HMI CLASSES

PegDial
PegFiniteDial
PegFiniteBitmapDial
PegCircularDial
PegCircularBitmapDial

6.1 PegDial

6.1.1 Overview

PegDial is an abstract base class from which PegFiniteDial and PegFiniteBitmapDial are derived. PegDial provides the base functionality common to all the derived dial objects in the PEG library.

It is important to understand how the angle value works on the dial. Consider an xy coordinate system (Figures 1 and 2, below). A point P in the xy plane has coordinates (x,y) where x is considered *positive* along OX and *negative* along OX' while y is *positive* along OY and *negative* along OY'. The angle A described *counterclockwise* from OX is considered *positive*. If it is described *clockwise* from *OX* it is considered *negative*. We call X'OX and Y'OY the x and y axis, respectively.

The various quadrants are denoted by I, II, III, and IV called the first, second, third, and fourth quadrants, respectively. In Figure 1, for example, angle A is in the second quadrant while in Figure 2, angle A is in the third quadrant.

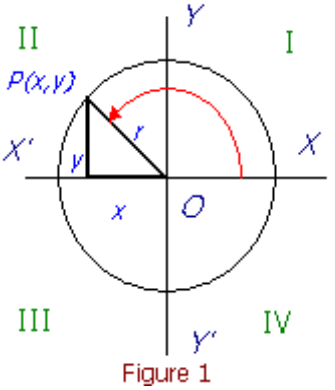


Figure 1

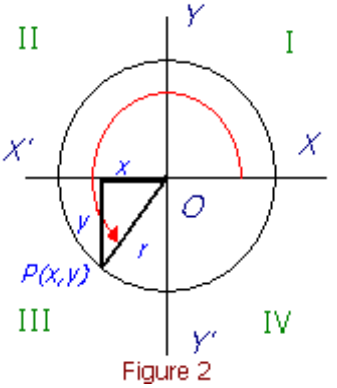


Figure 2

When working with angles with any PegDial derived class, 0 degrees is always on line OX with greater angle values going *counterclockwise*. Therefore, line OY is at 90 degrees, line OX' is at 180 degrees and line OY' is at 270 degrees.

6.1.2 See Also

PegThing

PegFiniteDial

PegFiniteBitmapDial

PegCircularDial

PegCircularBitmapDial

6.1.3 Style Flags

PegDial supports the standard frame styles `FF_NONE`, `FF_THIN`, `FF_THICK`, `FF_RAISED`, and `FF_RECESSED`.

PegDial also supports the following additional styles:

`DS_CLOCKWISE`	Sets the direction that the data will move the needle. Having this flag set causes the needle to move in a clockwise direction as the current value gets larger. Clearing this flag will cause the needle to move in a counterclockwise direction.
`DS_TICMARKS`	If this flag is set, the dial will draw tick marks at the defined intervals.
`DS_THINNEEDLE`	Will draw the needle as a thin single line.
`DS_THICKNEEDLE`	Will draw the needle as a thick single line.
`DS_POLYNEEDLE`	Will draw the needle using a filled polygon.
`DS_RECTCOR`	Rectangle Center of Rotation. Will use the center of the bounding rectangle (`mReal`) as the center for the anchor of the needle.
`DS_USERCOR`	User Center of Rotation. Allows the user to specify where the center anchor will be for the needle. This allows off-center bitmaps to be used for backgrounds on the dial.
`DS_STANDARDSTYLE`	A combination of `DS_THINNEEDLE` | `DS_CLOCKWISE` | `DS_TICMARKS` | `DS_RECTCOR`.
`AF_TRANSPARENT`	Enables the background to be drawn using the background color of the parent.

6.1.4 Signals

PegDial does not send any signals.

6.1.5 Derivation

PegDial is derived from PegThing.

6.1.6 Constructors:

```
PegDial(const PegRect &Rect, PEGULONG Style)
```

This constructor simply takes a reference to a PegRect and a `PEGULONG` for the style flags.

```
PegDial(PEGINT Left, PEGINT Top, PEGULONG Style)
```

This constructor simply takes `PEGINT` values to describe the left and top position of the dial. By default, an object created in this fashion will have a width and height of 100 pixels.

6.1.7 Public Functions:

```
PEGINT GetAnchorColor(void) const
```

This inline function returns the color ID of the anchor.

```
PEGUINT GetAnchorWidth(void) const
```

This inline function returns the width of the anchor.

```
PEGINT GetCurAngle(void) const
```

This inline function returns the current angle of the needle.

```
PEGLONG GetCurrentValue(void) const
```

This inline function returns the current value.

```
PEGINT GetDialColor(void) const
```

This inline function returns the color ID of the background of the dial.

```
PEGINT GetMaxAngle(void) const
```

This inline method returns the maximum angle supported by the dial.

```
PEGINT GetMaxValue(void) const
```

This inline method returns the maximum value supported by the dial.

```
PEGINT GetMinAngle(void) const
```

This inline method returns the minimum angle supported by the dial.

```
PEGINT GetMinValue(void) const
```

This inline method returns the minimum value supported by the dial.

```
PEGINT GetNeedleColor(void) const
```

This inline function returns the color ID of the needle.

```
PEGUINT GetNeedleLength(void) const
```

This inline function returns the length of the needle.

```
PEGLONG GetTicFreq(void) const
```

This inline function returns the frequency of tick marks.

```
PEGUINT GetTicLen(void) const
```

This inline function returns the length of individual tick marks.

```
virtual PEGLONG IncrementValue(PEGLONG Value, PEGBOOL
      Redraw = TRUE)
```

This virtual method increments the current value of the dial. By default, if the object is visible, it will redraw itself. To suppress this behavior, set `Redraw` to FALSE.

```
void SetAnchorColor(PEGINT ColorId)
```

This inline function is used to assign a color ID for the anchor point.

```
void SetAnchorWidth(PEGUINT Width)
```

This inline function is used to set the anchor width.

```
void SetDialColor(PEGINT ColorId)
```

This inline function is used to assign the color ID used to draw the dial face.

```
void SetNeedleColor(PEGINT ColorId)
```

This inline function is used to set the color of the needle.

```
void SetNeedleLength(PEGUINT Length)
```

This inline function is used to set the length of the needle. The value can be between 0 and 100 (inclusive). This value is used to calculate the length of the needle based on the lesser of the width or height of the dial. For

example, if this value is set to 80, then the length of the needle is 80% of the width or height of the dial, whichever is smaller.

```
void SetTicFreq(PEGLONG Freq)
```

This inline function is used to set the tick mark frequency. This determines the interval at which tick marks will be drawn on the dial face.

```
void SetTicLen(PEGUINT Length)
```

This inline function is used to set the tick mark length. This value works the same way as the needle length. The value is a percentage of the width or height of the dial, whichever is smaller. Therefore, if this value was set to 10, then the tick length would be 10% of the width or height of the dial, whichever is smaller.

```
virtual void SetValue(PEGLONG Value, PEGBOOL Redraw =
     TRUE)
```

This virtual method allows the current value to be set. By default, if the object is visible, it will redraw itself. To circumvent this behavior, set `Redraw` to FALSE.

6.1.8 Protected Members

```
virtual void CalcClipAndDraw(void)
```

This function calculates the needle position and, if it has moved, then it redraws.

```
virtual void CalcNeedlePos(void)
```

This pure virtual function is used to calculate the needle position.

```
virtual void CalcTicPos(PEGLONG Val, PegPoint &Pt1,
     PegPoint &Pt2)
```

This function calculates the position of the tick mark for value `Val`.

```
virtual void DrawAnchor(void)
```

This function draws the anchor, which is the pivot point of the needle. The default anchor is just a circle.

```
virtual void DrawDial(void)
```

This function draws the outer frame of the dial.

```
virtual void DrawNeedle(void)
```

This function draws the needle of the dial.

```
virtual void DrawTicMarks(void)
```

This function draws all of the tick marks on the dial.

```
virtual void EraseNeedle(void)
```

This function removes the needle from the dial.

```
virtual PEGINT ValToAngle(PEGLONG Val)
```

This function converts a numeric value to an angle based on the dial settings.

6.1.9 Examples:

See PegFiniteDial or PegFiniteBitmapDial for examples of PegDial-derived objects.

6.2 PegFiniteDial

6.2.1 Overview

PegFiniteDial is an HMI output gadget that provides an analog equivalent to a digital readout. It can be fed any integral data from any source.

The dial is categorized as finite because it takes a specific start angle and a specific end angle that map to a minimum and maximum value, respectively. So, in action, the needle on the dial moves between the start and end angles based on the current value that is between the minimum and maximum values assigned to the dial. The travel of the needle may be the entire 360 degrees of a circle, say from 0 to 359 in a counter clockwise direction, but it may not wrap.

6.2.2 See Also

PegDial

PegFiniteBitmapDial

6.2.3 Style Flags

PegFiniteDial supports the styles described in PegDial.

6.2.4 Signals

PegFiniteDial does not send any signals.

6.2.5 Derivation

PegFiniteDial is derived from PegDial.

6.2.6 Constructors:

```
PegFiniteDial(const PegRect &Rect, PEGINT MinAngle,
     PEGINT MaxAngle, PEGLONG MinValue, PEGLONG
     MaxValue, PEGULONG Style = DS_STANDARDSTYLE)
```

This constructor takes a reference to a PegRect to determine its size, a style, a minimum and maximum angle, and a minimum and maximum value.

The dial will map the minimum angle to the minimum value, and the maximum angle to the maximum value. So, the dial will behave as expected. If you set the minimum angle to be 0 and the maximum angle to be 180, and the minimum value to 0 and the maximum value to 100, then setting the value on the dial to 50 will set the needle to 90 degrees, straight up.

6.2.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegFiniteDial overrides the `Draw()` function to draw the different components of the dial.

```
virtual void SetLimits(PEGINT MinAngle, PEGINT
     MaxAngle, PEGLONG MinValue, PEGLONG MaxValue)
```

This function sets the boundaries for the min/max values and angles of the needle. It then redraws the needle if necessary.

6.2.8 Protected Members

```
virtual void CalcNeedlePos(void)
```

This function calculates the current position of the needle.

```
virtual void DrawTicMarks(void)
```

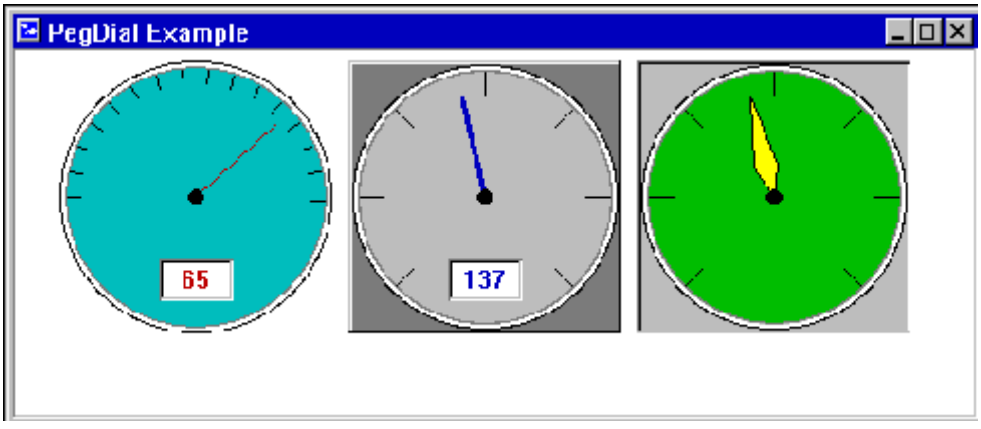
This function draws the tick marks along the outside of the dial.

```
virtual PEGINT ValToAngle(PEGLONG Val)
```

This function converts a numeric value to an angle along the dial.

6.2.9 Examples:

The following is an example of three PegFiniteDials on a PegDecoratedWindow.



The above PegFiniteDials were created by the following code snippet:

```
PegFiniteDial *mpDial1, *mpDial2, *mpDial3;
.
.
.
PegRect WinRect;

WinRect.Set(50, 50, 200, 200);
mpDial1 = new PegFiniteDial(WinRect, 180, 0, -50, 100,
    AF_TRANSPARENT | DS_STANDARDSTYLE);
mpDial1->SetTicFreq(10);
mpDial1->SetTicLen(10);
mpDial1->SetDialColor(CID_CYAN);

WinRect.Shift(160, 0);
mpDial2 = new PegFiniteDial(WinRect, 225, 315, 0, 300,
     FF_RAISED | DS_THICKNEEDLE | DS_TICMARKS | DS_RECTCOR);
mpDial2->SetColor(PCI_NORMAL, CID_DARKGRAY);
mpDial2->SetNeedleColor(CID_BLUE);
mpDial2->SetTicFreq(50);
mpDial2->SetTicLen(20);

WinRect.Shift(160, 0);
mpDial3 = new PegFiniteDial(WinRect, 225, 315, 0, 300
     FF_RECESSED | DS_POLYNEEDLE | DS_TICMARKS | DS_RECTCOR);
mpDial3->SetColor(PCI_NORMAL, CID_LIGHTGRAY);
```

```
mpDial3->SetDialColor(CID_GREEN);
mpDial3->SetNeedleColor(CID_YELLOW);
mpDial3->SetTicFreq(50);
mpDial3->SetTicLen(20);

Add(mpDial1);
WinRect.Set(105, 160, 144, 179);
PegPrompt *pPrompt = new PegPrompt(WinRect, "0", 101,
    FF_RECESSED | TJ_CENTER | TT_COPY);
pPrompt->SetColor(PCI_NTEXT, CID_RED);
mpDial1->Add(pPrompt);

Add(mpDial2);
WinRect.Set(265, 160, 304, 179);
pPrompt = new PegPrompt(WinRect, "0", 102, FF_RECESSED |
    TJ_CENTER | TT_COPY);
pPrompt->SetColor(PCI_NTEXT, CID_BLUE);
mpDial2->Add(pPrompt);

Add(mpDial3);

mpDial2->Add(pPrompt);

Add(mpDial3);
```

6.3 PegFiniteBitmapDial

6.3.1 Overview

PegFiniteBitmapDial behaves exactly the same as PegFiniteDial, from which it derives. The differences are that the user may specify a background bitmap over which the dial needle will be drawn as well as a bitmap that will be used to draw the needle anchor at the center of rotation. This allows for very customizable finite dials.

An added feature of PegFiniteBitmapDial is the ability to specify a center of rotation that is not necessarily the center of the bounding rectangle for the dial. Typically, the needle of the dial would originate at the center of the bounding rectangle. PegFiniteBitmapDial allows for the needle to originate from anywhere inside its bounding rectangle. This further adds to the custom possibilities when designing your dials.

The PegFiniteBitmapDial uses 2 bitmaps, which can be indexed with the following enumerations.

`PBMI_DIAL_BACKGROUND`	Background bitmap
`PBMI_DIAL_ANCHOR`	Anchor bitmap

6.3.2 See Also

PegDial

PegFiniteDial

6.3.3 Style Flags

PegFiniteBitmapDial supports the styles described in PegDial. PegFiniteBitmapDial does not support any of the frame styles because it uses the background bitmap for drawing in the client area of the dial.

6.3.4 Signals

PegFiniteBitmapDial does not send any signals.

6.3.5 Derivation

PegFiniteBitmapDial is derived from PegFiniteDial.

6.3.6 Constructors:

```
PegFiniteBitmapDial(const PegRect &Rect, PEGINT
     MinAngle, PEGINT MaxAngle, PEGLONG MinValue,
     PEGLONG MaxValue, PEGINT BkgBmp, PEGINT
     AnchorBmp = 0, PEGULONG Style =
     DS_STANDARDSTYLE)
```

This constructor takes a reference to a PegRect to determine its size, a minimum and maximum angle, a minimum and maximum value, a bitmap ID that will be used for drawing the background, a bitmap ID that will be used to draw the needle anchor, and style flags.

The dial will map the minimum angle to the minimum value, and the maximum angle to the maximum value. So, the dial will behave as expected. If you set the minimum angle to be 0 and the maximum angle to be 180, and the minimum value to 0 and the maximum value to 100, then setting the value on the dial to 50 will set the needle to 90 degrees, straight up.

6.3.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegFiniteBitmapDial overrides the `Draw()` function to draw the background bitmap and needle.

```
PEGINT GetBitmap(PEGINT Index) const
```

This method returns the ID of the bitmap specified by `Index`.

```
PEGINT GetCORX(void) const

PEGINT GetCORY(void) const
```

These inline functions return the respective values of the x and y center of rotation.

```
void SetBitmap(PEGINT Index, PEGINT BmpId)
```

This method sets the bitmap with the specified `Index` to `BmpId`.

```
void SetCOR(PEGINT CORX, PEGINT CORY, PEGBOOL Redraw =
     FALSE)
```

This allows for the setting of the x and y values of the center of rotation (the origin of the needle) of the dial.

```
void UseTrueCOR(PEGBOOL Use)
```

This method is shorthand for setting the style flags `DS_RECTCOR` and `DS_USERCOR` to on or off, depending on the value of `Use`.

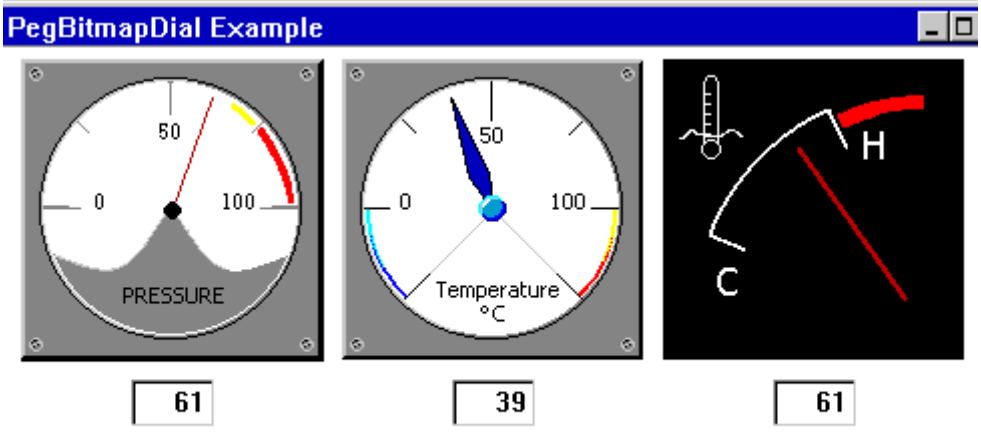
6.3.8 Protected Members

```
virtual void CalcNeedlePos(void)
```

This function calculates the current position of the needle.

6.3.9 Examples:

The following is an example of three PegFiniteBitmapDials on a PegDecoratedWindow. Notice that the third dial, on the right, has a center of rotation that is in the bottom right corner of the bounding rectangle of the dial.



The above PegFiniteBitmapDials were created by the following code snippet:

```
PegFiniteBitmapDial *pDial1, *pDial2, *pDial3;
```

```
PegRect WinRect;

WinRect.Set(50, 50, 200, 200);
pDial1 = new PegFiniteBitmapDial(WinRect, 180, 0, 0, 100,
    BID_DIAL_BKGRND1);

WinRect.Shift(160, 0);
pDial2 = new PegFiniteBitmapDial(WinRect, 225, 315, -25, 125,
    BID_DIAL_BKGRND2, BID_ANCHOR1);
pDial2->SetStyle((pDial2->GetStyle() | DS_POLYNEEDLE) &
    ~DS_THINNEEDLE);
pDial2->SetNeedleColor(CID_BLUE);

WinRect.Shift(160, 0);
pDial3 = new PegFiniteBitmapDial(WinRect, 180, 90, 0, 100,
    BID_DIAL_BKGRND3);
pDial3->SetStyle(DS_THICKNEEDLE | DS_USERCOR);
pDial3->SetCOR(123, 123);
pDial3->SetNeedleLength(65);

Add(pDial1);
Add(pDial2);
Add(pDial3);
```

6.4 PegCircularDial

6.4.1 Overview

PegCircularDial is an HMI output gadget that provides an analog equivalent to a digital readout. It can be fed any integral data from any source.

The dial is categorized as circular because it allows for multiple revolutions of the needle. Each lap of the needle around the circumference of the dial adds a specific value to the accumulated value. To achieve this, the user must specify a reference angle (the angle on the dial where the lap begins) as well as a value per revolution. Coupled with the minimum and maximum values supported by the dial, this allows for predictable circular behavior.

As an example, if the minimum value of the dial were 0 and the maximum were 900, and the value per revolution were 300 and the reference angle were 90, then setting the current value of the dial to 0 would force the needle to draw at 90 degrees. Incrementing the value up to 300 would make the needle do one complete revolution around the dial. When the current value reached 900, the needle would be at 90 degrees, having traveled around the circumference of the dial three times.

The direction of needle travel is determined by the minimum and maximum values of the measurement value. If the maximum value is greater than the minimum value, the needle will rotate clockwise. Otherwise, the needle will rotate counterclockwise.

6.4.2 See Also

PegDial

PegCircularBitmapDials6_PegFiniteDial

6.4.3 Style Flags

PegCircularDial supports all of the styles described in PegDial except for style `DS_CLOCKWISE`.

6.4.4 Signals

PegCircularDial does not send any signals.

6.4.5 Derivation

PegCircularDial is derived from PegDial.

6.4.6 Constructors:

```
PegCircularDial(const PegRect &Rect, PEGINT RefAngle,
     PEGLONG ValuePerRev, PEGLONG MinValue, PEGLONG
     MaxValue, PEGULONG Style = DS_STANDARDSTYLE)
```

This constructor takes a reference to a PegRect to determine its size. It then takes a `PEGINT` value to denote the reference angle. The reference angle is the point on the dial at which the minimum value that the dial supports will be mapped, as well as where a complete revolution is counted. `ValuePerRev` tells the dial how much to increment its internal current value for each revolution of the needle around the circumference of the dial. It then takes `PEGLONGs` for the minimum and maximum value, and a style `PEGUSHORT`.

6.4.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegCircularDial overrides the `Draw()` function.

```
PEGINT GetRefAngle(void) const
```

This inline function returns the internal reference angle.

```
PEGLONG GetValuePerRev(void) const
```

This inline function returns the internal value per revolution.

```
virtual void SetLimits(PEGINT RefAngle, PEGLONG
     ValuePerRev, PEGLONG MinValue, PEGLONG MaxValue)
```

This function sets limits on the angle and the min/max value of the dial. The needle position is recalculated. The dial is not redrawn.

6.4.8 Protected Members

```
virtual void CalcNeedlePos(void)
```

This function calculates the current position of the needle.

```
virtual void DrawTicMarks(void)
```

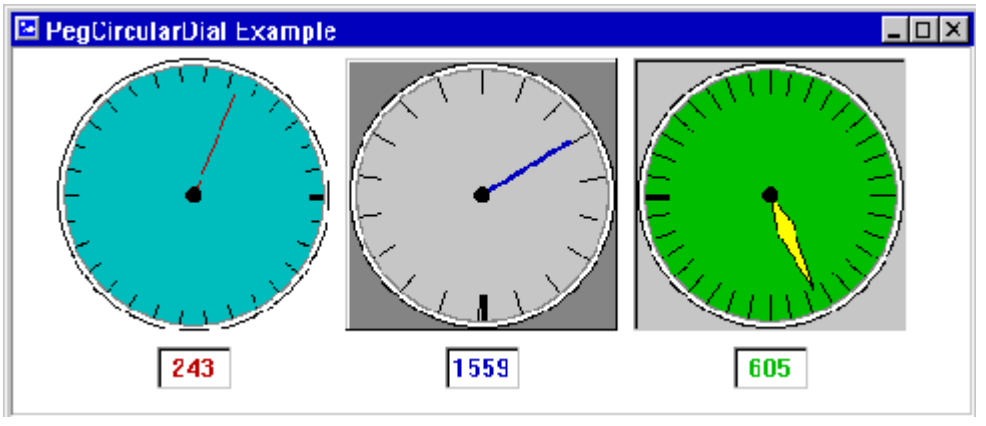
This function draws all of the tick marks along the outside of the dial.

```
virtual PEGINT ValToAngle(PEGLONG Val)
```

This function converts a numeric value to an angle along the dial.

6.4.9 Examples:

The following is an example of three PegCircularDials on a PegDecoratedWindow.



The above PegCircularDials were created by the following code snippet:

```
PegCircularDial *pDial1, *pDial2, *pDial3;
PegRect WinRect;

WinRect.Set(50, 50, 200, 200);
pDial1 = new PegCircularDial(WinRect, 0, 300, 0, 900,
    AF_TRANSPARENT | DS_STANDARDSTYLE);
pDial1->SetTicFreq(10);
pDial1->SetTicLen(10);
pDial1->SetDialColor(CID_CYAN);
pDial1->RemoveStatus(PSF_VIEWPORT);

WinRect.Shift(160, 0);
pDial2 = new PegCircularDial(WinRect, 270, 180, 0, 1800,
FF_RAISED);
pDial2->SetStyle(pDial2->GetStyle() | DS_THICKNEEDLE |
    DS_TICMARKS | DS_RECTCOR);
pDial2->SetColor(PCI_NORMAL, CID_DARKGRAY);
pDial2->SetNeedleColor(CID_BLUE);
pDial2->SetTicFreq(10);
```

```
pDial2->SetTicLen(20);

WinRect.Shift(160, 0);
pDial3 = new PegCircularDial(WinRect, 180, 360, 0, 720,
    FF_RECESSED);
pDial3->SetStyle(mpDial3->GetStyle() | DS_POLYNEEDLE |
    DS_TICMARKS | DS_RECTCOR);
pDial3->SetColor(PCI_NORMAL, CID_LIGHTGRAY);
pDial3->SetDialColor(CID_GREEN);
pDial3->SetNeedleColor(CID_YELLOW);
pDial3->SetTicFreq(10);
pDial3->SetTicLen(20);

Add(pDial1);
WinRect.Set(105, 210, 144, 229);
PegPrompt* pPrompt = new PegPrompt(WinRect, "0", 101,
    FF_RECESSED | TJ_CENTER | TT_COPY);
pPrompt->SetColor(PCI_NTEXT, CID_RED);
Add(pPrompt);

Add(pDial2);
WinRect.Set(265, 210, 304, 229);
pPrompt = new PegPrompt(WinRect, "0", 102, FF_RECESSED |
    TJ_CENTER | TT_COPY);
pPrompt->SetColor(PCI_NTEXT, CID_BLUE);
Add(pPrompt);

WinRect.Set(425, 210, 464, 229);
pPrompt = new PegPrompt(WinRect, "0", 103, FF_RECESSED |
    TJ_CENTER | TT_COPY);
pPrompt->SetColor(PCI_NTEXT, CID_GREEN);
Add(pPrompt);
Add(pDial3);
```

6.5 PegCircularBitmapDial

6.5.1 Overview

PegCircularBitmapDial behaves exactly the same as PegCircularDial, from which it derives. The differences are that the user may specify a background bitmap over which the dial needle will be drawn as well as a bitmap that will be used to draw the needle anchor at the center of rotation. This allows for very customizable circular dials.

An added feature of PegCircularBitmapDial is the ability to specify a center of rotation that is not necessarily the center of the bounding rectangle for the dial. Typically, the needle of the dial would originate at the center of the bounding rectangle. PegCircularBitmapDial allows the needle to originate from anywhere inside its bounding rectangle. This further adds to the custom possibilities when designing your dials.

The PegCircularBitmapDial uses two bitmaps, which can be indexed with the following enumerations.

`PBMI_DIAL_BACKGROUND`	Background bitmap
`PBMI_DIAL_ANCHOR`	Anchor bitmap

6.5.2 See Also

PegDial

PegCircularDial

6.5.3 Style Flags

PegCircularBitmapDial supports all of the styles described in PegDial except for style `DS_CLOCKWISE`. PegCircularBitmapDial does not support any of the frame styles because it uses the background bitmap for drawing in the client area of the dial.

6.5.4 Signals

PegCircularBitmapDial does not send any signals.

6.5.5 Derivation

PegCircularBitmapDial is derived from PegCircularDial.

6.5.6 Constructors:

```
PegCircularBitmapDial(const PegRect &Rect, PEGINT
     RefAngle, PEGLONG ValPerRev, PEGLONG MinValue,
     PEGLONG MaxValue, PEGINT BkgBmp, PEGINT
     AnchorBmp = 0, PEGULONG Style =
     DS_STANDARDSTYLE)
```

This constructor takes a reference to a PegRect to determine its size, a `PEGINT` value for the reference angle, a `PEGLONG` value for the value per revolution, a minimum and maximum value, a bitmap ID that will be used for drawing the background, a bitmap ID that will be used to draw the needle anchor, and a style `PEGULONG`.

6.5.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

PegCircularBitmapDial overrides the `Draw()` function to draw the background bitmap and needle.

```
PEGINT GetBitmap(PEGINT Index) const
```

This method returns the ID of the bitmap specified by the parameter `Index`.

```
PEGINT GetCORX(void) const

PEGINT GetCORY(void) const
```

These inline functions return the respective values of the x and y center of rotation.

```
virtual void Resize(const PegRect &Size)
```

PegCircularBitmapDial overrides the `Resize()` function so that it can adjust its private drawing surface, if needed.

```
void SetBitmap(PEGINT BmpId)
```

This method sets the bitmap using the specified `Index` and `BmpId`.

```
void SetCOR(PEGINT CORX, PEGINT CORY, PEGBOOL Redraw =
     FALSE)
```

This allows for the setting of the x and y values of the center of rotation (the origin of the needle) of the dial.

```
void UseTrueCOR(PEGBOOL Use)
```

This method is shorthand for setting the style flags `DS_RECTCOR` and `DS_USERCOR` to on or off, depending on the value of `Use`.

6.5.8 Protected Members

```
virtual void CalcNeedlePos(void)
```

This function calculates the current position of the needle.

CHAPTER 7

MISCELLANEOUS

7.1 Peg2DPolygon

7.1.1 Overview

Peg2DPolygon is an advanced drawing class that extends and encapsulates the functionality implemented in the PegScreen's `Polygon` method into a true sprite-oriented, two-dimensional polygon.

Historically, in PEG, to draw a polygon one would allocate any number of PegPoint structures, assign values to the x and y members, create a PegBrush structure, and assign its color properties accordingly. One would then call the `PegScreen::Polygon` method and pass it over the list of PegPoints and the PegBrush structure. This would effectively draw the given polygon at the coordinates designated in the PegPoint array. Peg2DPolygon takes care of this work for you, and provides easy ways to move and rotate the polygon within its bounding rectangle.

Peg2DPolygon simplifies the process of drawing 2D wireframes and filled polygons. Instead of normalizing one's desired polygon coordinates to screen coordinates, the polygon can be described in relation to origin 0 x and 0 y as the top-left corner of the bounding rectangle of the polygon. Once these coordinates are given to the Peg2DPolygon object, the object will translate them from their 0,0 base to the left and top of its bounding rectangle as it is located on the screen. And, throughout the lifetime of the object, it updates this translation every time it is moved or resized.

Another great feature is the ability to rotate the polygon to any given angle without have to translate the points yourself. The following is a discussion of how the current angle value is used in determining the translation of the points of the polygon.

It is important to understand how the angle value works in the Peg2DPolygon object. Consider an *xy* coordinate system (Figures 1 and 2, below). A point P in the *xy* plane has coordinates (x,y) where x is considered positive along OX and *negative* along *OX*' while y is *positive* along OY and *negative* along OY'. The angle A described *counterclockwise* from OX is considered *positive*. If it is described *clockwise* from OX it is considered *negative*. We call X'OX and Y'OY the x and y axis, respectively.

The various quadrants are denoted by I, II, III, and IV called the first, second, third, and fourth quadrants, respectively. In Figure 1, for example,

angle A is in the second quadrant, while in Figure 2 angle A is in the third quadrant.

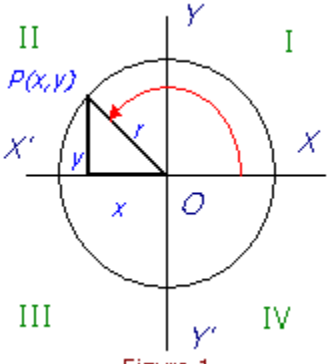


Figure 1

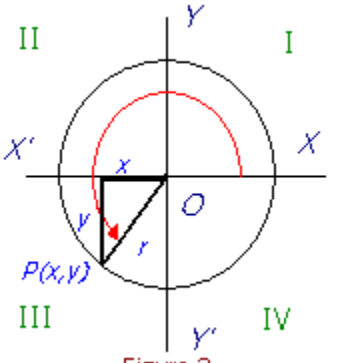


Figure 2

When working with angles with the Peg2DPolygon class, 0 degrees is always on line OX with greater angle values going *counterclockwise*. Therefore, line OY is at 90 degrees, line *OX'* is at 180 degrees, and line *OY'* is at 270 degrees.

It is also important to note that the polygon is rotated relative to its bounding rectangle. In other words, if you were to have a polygon that was bounded by a rectangle 48 pixels high and 48 pixels wide, the center of rotation for the polygon would be at relative 23 x and 23 y (24 pixels from the top left corner of the bounding rectangle). This can provide some very interesting rotational effects. The one aspect that does require some care in rotating the polygon is the point's distance from the center of the bounding rectangle. If a point lies outside of the radius of the largest concentric circle that would fit within the perimeter of the bounding rectangle, then the point may become clipped as the polygon is rotated.

7.1.2 See Also

PegThing

7.1.3 Style Flags

Peg2DPolygon supports the standard frame styles `FF_NONE`, `FF_THIN`, `FF_THICK`, `FF_RAISED`, and `FF_RECESSED`. `FF_NONE` is the default.

Peg2DPolygon supports the AF_TRANSPARENT flag. This flag can be dangerous if you are rotating the polygon and redrawing, as it may not correctly erase the relic polygon. The way to alleviate this is to rotate the polygon, then have the parent of the Peg2DPolygon object draw itself. (This assumes that you have not set the PSF_VIEWPORT status on the Peg2DPolygon object.)

Peg2DPolygon also supports the TT_COPY style flag. If set, this causes Peg2DPolygon to copy the array of PegPoints sent over in the constructor's parameter list. If you have allocated the points on the heap and would not like the data copied, do not set this flag. By default, this flag is not set.

7.1.4 Signals

Peg2DPolygon does not send any signals.

7.1.5 Derivation

Peg2DPolygon is derived from PegThing.

7.1.6 Constructors:

```
Peg2DPolygon(const PegRect &Rect, PegPoint *pPoints,
      PEGUINT NumPoints, PEGUINT Id = 0, PEGULONG
      Style = FF_NONE)
```

The constructor takes a reference to a PegRect that describes the bounding rectangle for the polygon, a pointer to a PegPoint structure, the number of points that describe the polygon, a PEGUINT for the ID of the object, and a PEGULONG for the style flags.

It is necessary for the PegPoints to be allocated as an array of points, with the pPoints parameter pointing to the first element in the array.

7.1.7 Public Functions:

```
virtual void Draw(const PegRect &Invalid)
```

Peg2DPolygon overrides the PegThing::Draw method in order to correctly draw the polygon.

```
PEGINT GetCurAngle(void) const
```

This inline function returns the current angle used to rotate the polygon.

```
PEGBOOL GetFill() const
```

This function retrieves the fill flag used to determine whether the polygon should fill its interior or only draw its outline.

```
PEGINT GetLineWidth() const
```

This function retrieves the line width used when drawing the polygon.

```
PEGUINT GetNumPoints() const
```

Returns the number of points in the polygon.

```
virtual void ParentShift(PEGINT XOffset, PEGINT
     YOffset)
```

Peg2DPolygon overrides the `PegThing::ParentShift` method in order to correctly map the polygon coordinates to the new location on the screen. This member is called internally by the library and should not be called from a user application.

```
virtual void Resize(const PegRect &Rect)
```

Peg2DPolygon overrides the `PegThing::Resize` method in order to correctly shift the polygon to the new location described by the new bounding rectangle.

```
void SetCurAngle(PEGINT Theta)
```

This function is used to set the current angle used to rotate the polygon.

```
void SetFill(PEGBOOL Fill)
```

This function sets the fill flag used to determine if the polygon should fill its interior or only draw its outline.

```
void SetLineWidth(PEGINT Width)
```

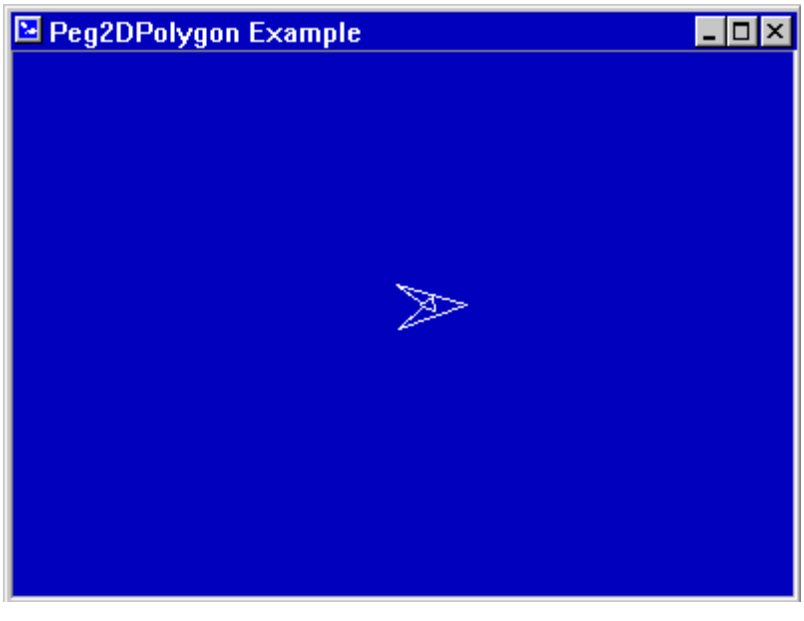
This function sets the line width used when drawing the polygon.

7.1.8 Examples:

The following is a code snippet that creates the Peg2DPolygon displayed below. You'll notice that since we globally allocated the PegPoints that described the polygon, we don't set the `TT_COPY` flag in the constructor to the Peg2DPolygon object.

```
// Global Polygon point data
static PegPoint gtPolyPoints[] = {
```

```
    {47, 24}, {12, 12}, {24, 22},
    {24, 23}, {31, 19}, {31, 27},
    {24, 25}, {12, 35} };

PolygonWindow::PolygonWindow() : PegDecoratedWindow()
{
    mReal.Set(0, 0, 400, 300);
    InitClient();

    SetColor(PCI_NORMAL, CID_BLUE);

    Add(new PegTitle("Peg2DPolygon Example"));

    PegRect Rect;
    Rect.Set(mClient.Left + 10, mClient.Top + 10,
        mClient.Left + 57, mClient.Top + 57);
    mpPolygon = new Peg2DPolygon(Rect, &gtPolyPoints[0], 8,
                                 101, FF_NONE);
    mpPolygon->SetColor(PCI_NORMAL, CID_BLUE);
    mpPolygon->SetColor(PCI_NTEXT, CID_WHITE);
    Add(mpPolygon);
}
```



7.2 PegBitmap

The PegBitmap structure contains format and data address information for PEG-compatible bitmaps.

The PegBitmap structure is defined as:

```
#define  BMF_RAW          0x00 // bitmap is not RLE encoded
#define  BMF_RLE          0x01 // bitmap is RLE encoded
#define  BMF_NATIVE       0x02 // bitmap is in native video
format
#define  BMF_ROTATED      0x04 // bitmap is rotated 90 or
270
                               //   degrees
#define  BMF_GRAYSCALE    0x08 // 1-8 bpp bitmap uses
grayscale
                               //   palette
#define  BMF_ARGB_FORMAT  0x08 // 16-bpp uses ARGB 4444
format
#define  BMF_HAS_TRANS    0x10 // bitmap uses transparency
#define  BMF_SPRITE       0x20 // bitmap resides in video
memory
#define  BMF_RGB          0x40 // 24-bit bitmap is in RGB
(not
                               //   BGR) order
#define  BMF_555_FORMAT   0x40 // 16-bit bitmap is in 555
(not
                               //   565) format
#define  BMF_332_FORMAT   0x40 // 8-bit bitmap is in 3:3:2
(not
                               //   palette) format
#define  BMF_ALPHA        0x80 // bitmap has alpha channel
(16 &
                               //   24bpp support only)


struct PegBitmap
{
    PEGUBYTE Flags;          // combination of flags above
    PEGUBYTE BitsPix;        // 1, 2, 4, 8, 16, or 24
    PEGUSHORT Width;         // in pixels
    PEGUSHORT Height;        // in pixels
    PEGULONG TransColor;     // transparent color for > 8bpp
                             //    bitmaps
    PEGUBYTE PEGFAR *pStart; // bitmap data pointer
};
```

7.3 PegBitmapRotator

7.3.1 Overview

The PegBitmapRotator class is a utility class that takes a PegBitmap as input and returns a rotated version of that PegBitmap. It uses all fixed-point math to do the calculations.

It is important to understand how the angle value works in the PegBitmapRotator object. Consider an xy coordinate system (Figures 1 and 2, below). A point P in the xy plane has coordinates (x,y) where x is considered *positive* along OX and *negative* along *OX'* while y is *positive* along OY and *negative* along *OY'*. The angle A described *counterclockwise* from OX is considered *positive*. If it is described *clockwise* from OX, it is considered *negative*. We call X'OX and Y'OY the x and y axis, respectively.

The various quadrants are denoted by I, II, III, and IV called the first, second, third and fourth quadrants, respectively. In Figure 1, for example, angle A is in the second quadrant while, in Figure 2, angle A is in the third quadrant.

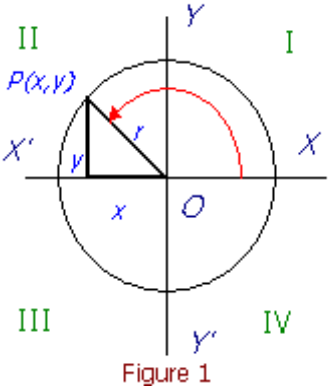


Figure 1

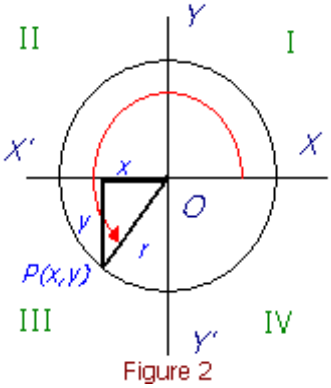


Figure 2

When working with angles with the PegBitmapRotator class, 0 degrees is always on line OX with greater angle values going *counterclockwise*. Therefore, line OY is at 90 degrees, line *OX'* is at 180 degrees, and line *OY'* is at 270 degrees.

7.3.2 See Also

PegBitmap

7.3.3 Public Functions:

```
PegBitmap *RotateBitmap(PegBitmap *pSrc, PEGINT
     Rotation)
```

This function takes a PegBitmap `pSrc` and rotates it by the specified number of degrees. It returns a pointer to the newly created rotated bitmap.

7.4 PegBrush

The PegBrush class is used to pass information to the PegScreen drawing functions. PegBrush contains foreground and background colors, pattern, width, and style flag information.

The PegBrush class is defined as:

```
class PegBrush
{
    friend class PegScreen;

    public:

        PegBrush()

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

        ~PegBrush();

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

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

    private:
        PegBitmap *pSysmap;
};
```

The brush styles are:

`PBS_SOLID_FILL`: This flag is used to fill rectangles, polygons, and text with a solid color. For text functions, `PBS_SOLID_FILL` will cause the text background area to be filled with the background color, while turning that flag off will cause only the text foreground to be drawn.

`PBS_BMP_FILL`: This flag is used to fill rectangles and polygons with a bitmap. The `Brush.pBitmap` field is used to obtain the bitmap.

`PBS_NO_ALIAS`: This flag is used to draw lines or text with no anti-aliasing. This is currently the default behavior, so it's not really necessary to set this explicitly.

`PBS_SIMPLE_ALIAS`: This flag is used to draw lines or text with 'simple anti-aliasing.' This means that, at the edges of the line, the `Brush.LineColor` will get blended with `Brush.FillColor` to produce a smoother looking line. For lines drawn on top of solid-colored surfaces, this is often sufficient.

`PBS_TRUE_ALIAS`: This flag is used to draw lines or text with 'true anti-aliasing.' This means that, at the edges of the line, the `Brush.LineColor` will get blended with whatever pixels are already found on the screen. This results in a smooth looking line even on multicolored backgrounds. Performance-wise, this takes more processing time than `PBS_SIMPLE_ALIAS`.

`PBS_UNDERLINE`: This flag is used to draw an underline underneath text when calling the `DrawText` function.

`PBS_ROUNDED`: This flag is used to draw rounded endpoints for wide lines. This can be useful when drawing a series of connected wide lines because the intersections will look smooth and round, rather than having sharp corners sticking out.

`PBS_CENTER_LINE`: By default, when drawing a thick line, the coordinates passed into the `Line` function represent the top and left coordinates of the line. In other words, if the user draws a line of width 10, the drawing would start at the specified coordinates and fill in 10 pixels in the bottom and right direction. By turning on `PBS_CENTER_LINE`, the width of the line is centered on original coordinates. So in the same example, the line would now be drawn with 5 pixels in the top and left direction and 5 pixels in the bottom and right direction.

PBS_PATTERN: This flag informs the PegScreen class that the line that is being drawn will not be a solid line. Instead it will use the Brush.Pattern value to decide which pixels to turn on and which to turn off.

PBS_ALPHA: This flag informs certain drawing functions in the PegScreen class that the drawing primitive is to be blended into the background using a specific alpha value. The drawing primitive will then use the Brush.Alpha value to blend the entire primitive.

7.5 PegCapture

PegCapture is a utility class used to capture and restore regions of the screen.

The PegCapture class is defined as:

```
class PegCapture
{
    public:
        PegCapture(voi
        d)
        {
            mRect.Set(0, 0, 0, 0);
            mBitmap.pStart = 0
            mValid = FALSE;
            mDataSize = 0;
        }

        ~PegCapture()
        {
            if (mBitmap.pStart)
            {
                delete mBitmap.pStart;
            }
        }

        PegRect &Pos(void) {return mRect;}
        PegPoint Point(void);
        PEGLONG DataSize(void) {return mDataSize;}

        void SetPos(PegRect &Rect)
        {
            mRect = Rect;
            mBitmap.Width = (PEGUSHORT) Rect.Width();
            mBitmap.Height = (PEGUSHORT) Rect.Height();
        }

        PEGBOOL IsValid(void) const {return mValid;}
        void SetValid(PEGBOOL Valid) {mValid = Valid;}
        void Realloc(PEGLONG Size);
        void Reset(void);
```

```
        void MoveTo(PEGINT Left, PEGINT Top);
        void Shift(PEGINT xShift, PEGINT yShift)
            {mRect.Shift(xShift, yShift);}
        PegBitmap *Bitmap(void) {return &mBitmap;}
};
```

7.6 PegFont

The PegFont structure contains format and data address information for PEG compatible fonts.

The PegFont structure is defined as:

```
struct PegFont
{
    PEGUBYTE Type; PEGUBYTE
    Ascent; PEGUBYTE
    Descent; PEGUBYTE
    CharHeight; PEGUBYTE
    PreSpace; PEGUBYTE
    PostSpace; PEGUBYTE
    LineHeight; PEGUSHORT
    BytesPerLine; PEGUSHORT
    FirstChar; PEGUSHORT
    LastChar; PEGUSHORT
    *pOffsets; PegFont
    *pNext;
    PEGUBYTE *pData;
};
```

7.7 PegGradient

The PegGradient class is used to draw a progression from one color to another. It can draw that progression in a number of different directions, and it can include any number of specified colors at any point along that progression, not just two colors at the ends.

To create a gradient, first create an instance of the PegGradient class, and add the colors you want to use along with their position within the gradient. The position is a value between 0x00 and 0xff. The position value is not a specific pixel location, but a ratio used to determine the location once the length of the gradient has been set. Although it does not matter in what order you add the colors, it makes sense to start with the first color at position '0.'

Once the colors have been set, the application can create a gradient in any style or direction. The PegGradient APIs can be used to draw a gradient directly to the screen, or it can draw it into a PegBitmap so that the gradient doesn't need to be recalculated every time the application wants to redraw it. Drawing into a bitmap can greatly improve performance, though it does so at the cost of requiring more memory.

The gradient can be drawn using a straight linear progression, or it could progress along a cosine curve. Using a cosine curve creates a gradient that looks a little different because the endpoint colors are better highlighted.

7.7.1 Constructors:

```
PegGradient(void)
```

This creates an instance of the PegGradient class. The gradient is empty at this point so it would be unable to draw anything.

7.7.2 Public Functions:

```
void AddColor(PEGCOLOR Color, PEGINT Position)
```

This function is responsible for adding a new entry to the list of colors that the gradient contains. `Position` is a ratio from 0x00 to 0xff that determines the position within the gradient that `Color` should appear.

```
void ClearColors(void)
```

This function removes all colors from the gradient. This is necessary if the application requires that the same PegGradient instance be used to draw multiple gradients.

```
void DrawHorizGradient(PegThing *pCaller, PegRect
     Rect, PEGBOOL Linear = TRUE, PEGBOOL Reverse =
     FALSE)
```

This function draws a horizontal gradient, meaning that it progresses from left to right. `pCaller` refers to the object that called this function. The region defined by `Rect` is the screen coordinates that the gradient will draw into. Also, the width of `Rect` is used to determine the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve. If `Reverse` is TRUE, the gradient is essentially drawn from right to left instead of left to right.

```
void DrawLLtoURGradient(PegThing *pCaller, PegRect
     Rect, PEGBOOL Linear = TRUE, PEGBOOL Reverse =
     FALSE)
```

This function draws a diagonal gradient from the lower left corner to the upper right corner. `pCaller` refers to the object that called this function. The region defined by `Rect` is the screen coordinates that the gradient will draw into. `Rect` is also used to determine the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve. If `Reverse` is TRUE, the gradient is essentially drawn from the upper right to the lower left instead of from the lower left to the upper right.

```
void DrawRadialGradient(PegThing *pCaller, PegRect
     Rect, PEGINT CenterX, PEGINT CenterY, PEGINT
     Radius, PEGBOOL Linear = TRUE)
```

This function draws a radial gradient, meaning that it starts from the point specified by `CenterX` and `CenterY` and progresses outward radially. `pCaller` refers to the object that called this function. The region defined by `Rect` is the screen coordinates that the gradient will draw into. `Radius` serves as the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve.

```
void DrawULtoLRGradient(PegThing *pCaller, PegRect
     Rect, PEGBOOL Linear = TRUE, PEGBOOL Reverse =
     FALSE)
```

This function draws a diagonal gradient, meaning that it progresses from the upper left corner to the lower right corner. `pCaller` refers to the object

that called this function. The region defined by `Rect` is the screen coordinates that the gradient will draw into. Also, `Rect` is used to determine the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve. If `Reverse` is TRUE, the gradient is essentially drawn from lower right to upper left instead of upper left to lower right.

```
void DrawVertGradient(PegThing *pCaller, PegRect Rect,
      PEGBOOL Linear = TRUE, PEGBOOL Reverse = FALSE)
```

This function draws a vertical gradient, meaning that it progresses from top to bottom. `pCaller` refers to the object that called this function. The region defined by `Rect` is the screen coordinates that the gradient will draw into. Also, the height of `Rect` is used to determine the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve. If `Reverse` is TRUE, the gradient is essentially drawn from bottom to top instead of top to bottom.

```
PEGCOLOR Get(PEGINT Index)
```

This function retrieves a particular color value from the progression of generated colors. `Index` is used to determine where in the progression to retrieve the color. `Index` should not exceed the length of the gradient.

```
PegBitmap *GetHorizGradient(PegThing *pCaller, PEGINT
      Width, PEGINT Height, PEGBOOL Linear = TRUE)
```

This function draws a horizontal gradient into a bitmap, meaning that it progresses from left to right. `pCaller` refers to the object that called this function. `Width` and `Height` define the dimensions of the bitmap that will be generated. Also, `Width` is used to determine the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve.

```
PegBitmap *GetLLtoURGradient(PegThing *pCaller,
      PEGINT Width, PEGINT Height, PEGBOOL Linear =
      TRUE)
```

This function draws a diagonal gradient from the lower left corner to the upper right corner into a bitmap. `pCaller` refers to the object that called this function. `Width` and `Height` define the dimensions of the bitmap that will be generated. They also are used to determine the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise it will progress along a cosine curve.

```
PegBitmap *GetRadialGradient(PegThing *pCaller,
      PEGINT Width, PEGINT Height, PEGINT CenterX,
```

```
PEGINT CenterY, PEGINT Radius, PEGBOOL Linear =
TRUE)
```

This function draws a radial gradient into a bitmap, meaning that it starts at the point defined by `CenterX` and `CenterY` and progresses outward radially. `pCaller` refers to the object that called this function. `Width` and `Height` define the dimensions of the bitmap that will be generated. `Radius` serves as the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve.

```
PegBitmap *GetULtoLRGradient(PegThing *pCaller,
    PEGINT Width, PEGINT Height, PEGBOOL Linear =
    TRUE)
```

This function draws a diagonal gradient from the upper left corner to the lower right corner into a bitmap. `pCaller` refers to the object that called this function. `Width` and `Height` define the dimensions of the bitmap that will be generated. They are also used to determine the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve.

```
PegBitmap *GetVertGradient(PegThing *pCaller, PEGINT
    Width, PEGINT Height, PEGBOOL Linear = TRUE)
```

This function draws a vertical gradient into a bitmap, meaning that it progresses from top to bottom. `pCaller` refers to the object that called this function. `Width` and `Height` define the dimensions of the bitmap that will be generated. Also, `Height` is used to determine the length of the gradient. If `Linear` is TRUE, the gradient will use a linear progression. Otherwise, it will progress along a cosine curve.

```
void Set(PEGINT Len, PEGCOLOR First, PEGCOLOR Last)
```

This function is used to generate all of the colors for the gradient segment between `First` and `Last`. It is typically not necessary to call this from the application code, but rather to call the other version of `Set`, or one of the `DrawXXXGradient` or `GetXXXGradient` APIs.

```
void Set(PEGINT Len)
```

This function is used to generate all of the colors for the gradient and store them in an internal array. The direction of the gradient is not relevant here because this function is only concerned with calculating the necessary colors, rather than determining where to draw them. If one of the `DrawXXXGradient` or `GetXXXGradient` APIs is going to be used, then the application does not need to call this. This is only needed if the application intends to use the `Get()` function to draw the gradient manually.

7.7.3 Examples

This example demonstrates how to make a gradient-filled button. The gradient will start with dark gray, then move to light gray, and then end with black. When the user presses on the button a different gradient will be used.

Note that this example creates the gradients as PegBitmaps in the constructor of the button and stores them in member variables. Then the `Draw()` function only needs to draw those bitmaps. This is an efficient way to do this because the gradients are only calculated once. A more complicated example would be required if the buttons needed to be resizeable after they were constructed. In that case, the gradient would need to be recalculated to fit the new dimensions.

```
MyButton::MyButton(PegRect Rect, PEGUSHORT Id,
                   PEGULONG Style)
    : PegButton(Rect, PEG_NULL_STRING, Id, Style)
{
    PegGradient grad;
    grad.AddColor(CLR_DARKGRAY, 0);
    grad.AddColor(CLR_LIGHTGRAY, 0x7f);
    grad.AddColor(CLR_BLACK, 0xff);
    mpBitmap = grad.GetVertGradient(this, mClient.Width(),
        mClient.Height(), FALSE);

    grad.ClearColors();
    grad.AddColor(CLR_LIGHTGRAY, 0);
    grad.AddColor(CLR_DARKGRAY, 0x7f);
    grad.AddColor(CLR_WHITE, 0xff);
    mpSelectedBitmap = grad.GetVertGradient(this,
        mClient.Width(), mClient.Height(), FALSE);
}

void MyButton::Draw(const PegRect &Invalid)
{
    BeginDraw(Invalid);
    PegButton::Draw(Invali
    d);

    PegPoint p;
    p.Set(mClient.Left, mClient.Top);
```

```
    if (mStyle & BF_PUSHED)
    {
        Bitmap(p, mpSelectedBitmap);
    }
    else
    {
        Bitmap(p, mpBitmap);
    }

    EndDraw();
}
```

7.8 PegMenuDescription

PegMenuDescption is a structure used to define PegMenuButton objects. Arrays of PegMenuDescriptions are used to define any number of nested menus and submenus. PegMenuDescription arrays are terminated with an entry filled with 0 or NULL values.

The PegMenuDescription structure is defined as:

```
struct PegMenuDescription
{
    const PEGCHAR *pText;
    PEGUSHORT Id;
    PEGULONG Style;
    PegMenuDescription *pSubMenu;
};
```

7.8.1 See Also

PegMenu

PegMenuButton

PegMenuBar

7.9 PegMessage

PegMessage is a data structure used to send and receive messages. PegMessageQueue is the coordinator of message transports in your PEG application.

The PegMessage structure is defined as:

```
struct PegMessage
{
    public:
        PegMessage(
        )
        {
            pNext = NULL;
            pTarget = NULL;
            pSource = NULL;
        }

        PegMessage(PEGUSHORT Val)
        {
            pNext = NULL;
            pTarget = NULL;
            pSource = NULL;
            Type = Val;
        }

        PegMessage(PegThing *pTo, PEGUSHORT Val)
        {
            pNext = NULL;
            pTarget = pTo;
            pSource = NULL;
            Type = Val;
        }

        PegThing *pSource;
        PegThing *pTarget;
        PEGUSHORT Type;
        PEGUSHORT Param;

        union
        {
```

```
            PegRect Rect;
            PegPoint Point;
            PEGLONG ExtParams[2];
            void *pData;
            PEGLONG UserLong[2];
            PEGULONG UserULONG[2];
            PEGSHORT UserShoft[4];
            PEGUSHORT UserUShoft[4];
            PEGUBYTE UserUByte[8];
        };

    private:
        PegMessage *pNext;
};
```

For system messages, the message data fields are predefined. For user defined messages, all fields except `Type` and `pTarget` are available for user definition and can be used to send application-defined data values.

7.10 PegPoint

The PegPoint structure contains a single pixel address.

The PegPoint structure is defined as:

struct PegPoint

```
{
    PEGSHORT x;
    PEGSHORT y;
    PEGBOOL operator != (const PegPoint &InPoint)const
    PEGBOOL operator == (const PegPoint &InPoint)const
    PegPoint operator +(const PegPoint &Point)const
    void Set(PEGINT InX, PEGINT InY);
};
```

7.11 PegRect

The PegRect structure contains a rectangular screen dimension along with a large number of operators, making it convenient to manipulate rectangular areas of the display screen.

The PegRect structure is defined as:

```
struct PegRect
{
    void Set(PEGINT x1, PEGINT y1, PEGINT x2, PEGINT y2)
    void Set(PegPoint ul, PegPoint br) PEGBOOL
    Contains(PegPoint Test) const; PEGBOOL
    Contains(PEGINT x, PEGINT y) const; PEGBOOL
    Contains(const PegRect &Rect) const; PEGBOOL
    Overlap(const PegRect &Rect) const; void
    MoveTo(PEGINT x, PEGINT y);
    void Shift(PEGINT xShift, PEGINT yShift);
    PegRect operator &=(const PegRect &Other);
    PegRect operator |=(const PegRect &Other);
    PegRect operator &(const PegRect &Rect) const;
    PegRect operator ^=(const PegRect &Rect);
    PegRect operator +(const PegPoint &Point) const;
    PegRect operator ++(int);
    PegRect operator +=(int x);
    PegRect operator --(int);
    PegRect operator -=(int x);
    PEGBOOL operator !=(const PegRect &Rect) const;
    PEGBOOL operator ==(const PegRect &Rect) const;

    PEGINT Width(void) const
    PEGINT Height(void) const

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

7.12 PegScrollInfo

The PegScrollInfo structure is used to pass scrolling data to and from PegHScroll and PegVScroll scroll bar classes.

The PegScrollInfo structure is defined as:

```
struct PegScrollInfo
{
    PEGINT Min;
    PEGINT Max;
    PEGINT Current;
    PEGINT Step;
    PEGINT Visible;
};
```

7.13 PegTimer

The PegTimer structure contains information about an active PegTimer. The PegTimer structure is private to PegTimerManager and is never referenced directly by external software, however we document this structure here for completeness.

The PegTimer structure is defined as:

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

    PegTimer(PegTimer *pNext, PegThing *pWho, PEGUSHORT Id,
             PEGLONG Cnt, PEGLONG Res)
    {
        mpNext = pNext;
        mpTarget = pWho;
        Count = Cnt;
        Reset = Res;
        TimerId = Id;
    }

    PegTimer *mpNext;
    PegThing *mpTarget;
    PEGLONG Count;
    PEGLONG Reset;
    PEGUSHORT TimerId;
};
```

7.14 PegZip - PegUnzip

The PEG library optionally includes functions for compressing and decompressing arbitrary data blocks. These functions can be used by your application any time the compression or decompression of large data objects is required. The most common use is to compress PegBitmap data, and decompress the PegBitmap data only when required for display.

The compression/decompression algorithm used by these functions is the LZ78 algorithm published by Jacob Ziv and Abraham Lempel, ***'Compression of Individual Sequences via Variable-Rate Coding, IEEE Transactions on Information Theory' (September 1978)***. This algorithm is highly effective and is used, in a slightly modified form, in all popular ZIP and GZIP utilities available as desktop applications. The implementation leans heavily on the open-source software published by Mark Nelson and Jean-Loup Gailly in the open source project known as 'ZLIB.' The original software has been heavily modified to improve portability, readability, and usage within the PEG software environment.

The PegZip compression function is included in the library if the configuration flag `PEG_ZIP` is defined.

The PegUnzip decompression function is included in the library if the configuration flag `PEG_UNZIP` is defined. Note that the run-time PNG decoder module uses the PegUnzip methods; therefore, PegUnzip is always required if run-time PNG file decoding is utilized.

The operation of the PegZip function is straightforward. The application program passes to the PegZip function a pointer to the data block to compress and the size of the data block. The compressed data pointer and size are returned in variables passed by address to the PegZip function.

The PegUnzip function can operate in two modes. The most efficient mode requires that the application program has preknowledge of the uncompressed data size. In this case, the application program allocates a buffer for storing the uncompressed data and passes this buffer address and size to the PegUnzip function. In this mode, the PegUnzip function is completed in one step, returning the uncompressed data.

A second, less efficient mode allows the caller to pass a NULL buffer pointer and zero (0) buffer size. In this mode, the PegUnzip function unzips the compressed data but discards the output, returning only the

uncompressed data size to the caller. The caller must then allocate the storage buffer to hold the uncompressed data and call PegUnzip a second time to retrieve the output data. This mode is not efficient and is not recommended.

In order to unzip compressed data in one operation, it is recommended that the application program save the uncompressed data size in nonvolatile memory so that decompression can occur with a single call to PegUnzip. For example, if the compressed data is to be saved to a file, the application program should save the original file size at the head of the output file. To decompress the file, the application program can first read the decompressed data size, then decompress the data in one operation.

The example program in the directory `\peg\examples\zip` demonstrates both modes of compress/decompress operation.

The PegZip function prototype is:

```
PEGINT PegZip(PEGUBYTE **pDest, PEGULONG *pDestLen,
        const PEGUBYTE *pSource, PEGULONG SourceLen);
```

The `pDest` parameter is the address of a pointer variable. The PegZip function will allocate the compressed data output buffer automatically, and return the address of this buffer in `pDest`. Your application will use this pointer to save the compressed data.

The `pDestLen` parameter is the address of a `PEGULONG` variable. The PegZip function will return the size of the compressed data buffer in this variable upon successful completion.

The `pSource` parameter is a pointer to the buffer holding the data to be compressed.

The `SourceLen` parameter is the size of the buffer to compress. This must be passed by your application to the PegZip function.

The return value of PegZip is 0 if compression is successful. A negative return value indicates that some error occurred during compression. The most common error is `PZIP_MEM_ERROR`, which indicates that there wasn't enough heap space to perform the compression operation. The possible error return codes are defined in the `pzip.hpp` header file.

The PegZip operation requires a large amount of heap space. 48K Bytes of heap data is required for data lookahead and tree structure maintenance, in addition to the actual data output buffer. Normally, applications do not require run-time compression, but instead only need to decompress previously-compressed data files.

The PegUnzip function prototype is:

```
PEGINT PegUnzip(PEGUBYTE *pDest, PEGULONG *pDestLen,
                const PEGUBYTE *pSource, PEGULONG SourceLen);
```

The `pDest` parameter is a pointer to the buffer allocated by your application for storing the uncompressed data. You may pass NULL for `pDest`, in which case the PegUnzip function discards the uncompressed data and returns the required buffer size. If `pDest` is NULL, the variable pointed to by `pDestLen` should also be 0.

The `pDestLen` parameter is the address of a variable holding the size of the output buffer allocated by the caller. If this value is 0, the PegUnzip function discards the uncompressed data but returns the size of the required output buffer in this variable. This can be useful if the uncompressed data size is not known by the application.

The `pSource` parameter is a pointer to the buffer holding the data to be uncompressed.

The `SourceLen` parameter is the size of the buffer to decompress. This must be passed by your application to the PegUnzip function.

The return value of PegUnzip is 0 if decompression is successful. A negative return value indicates that some error occurred during decompression. The most common error is `PZIP_MEM_ERROR`, which indicates that not enough heap space was available to perform the decompression operation. `PZIP_BUF_ERROR` indicates the output buffer passed to PegUnzip is not large enough to hold the decompressed data. A complete list of error return codes can be found in the `pzip.hpp` header file.

If decompression is successful, the `pDestLen` variable is updated to hold the size of the decompressed data. If `pDest` is NULL, the output is discarded; otherwise, the decompressed data is stored in the buffer allocated by the caller at the `pDest` memory address.

CHAPTER 8

PRINTER CLASSES

PegPrinter
PCLPrinter

PEG Style Flags

PEG System Status Flags

PEG Signals

Viewports

How Scrolling Works

8.1 PegPrinter

8.1.1 Overview

PegPrinter is a virtual base class that supports printing in PEG.

Much like the screen drivers, the PegPrinter provides the framework necessary for objects to draw themselves to a printer instead of to the output display.

PegPrinter-derived classes provide the necessary functionality to do the actual communications with a printer device to take the output from a particular object and render a printed page. It is important to note that printer output is not clipped to drawing areas in the same way that drawing to a screen is clipped. Any object, even if it is not visible on the screen, is allowed to produce output on a printed page. For this reason, it is often a good idea to derive an object from PegThing that only does printing, and, possibly, only certain types of printing (reports, charts, bitmaps, etc.). This gives the application developer great control over the printed output, because any object is allowed to draw anywhere on the page, not only within its own rectangular region, as is the case when drawing to the screen.

The PegPrinter API supports a document-centric model. Once a document has been started, any number of pages may then be printed using the same printer settings. The API also allows for multiple PEG objects to print to the same page, if needed. This is implemented by causing a draw operation to take place within the context of an offscreen bitmap. The bitmap is of a size necessary to include all of the drawing based on the selected width and height of the page as well as the resolution, or dots per inch. Obviously, this can lead to very large off screen bitmaps if a large page at high resolution is selected. The size of this bitmap is also affected by the color depth of the screen driver in use.

For example, if your target system is running in monochrome and you are printing to letter-sized paper at 100 dots per inch, the size of the bitmap needed to hold an entire page would be 109,438 bytes, or just under 107 KB. On the other hand, if your target system is running in 16 bit per pixel color and on the same size paper you are printing at 300 dots per inch, the bitmap would be 1.581e+07 bytes, or 15.439 MB. So, it is important to take these factors into consideration when designing printer output.

Printing is begun by calling the `Screen()->BeginPrint` function. It is important that this method be called before any reference to the printer object is performed by application code. Until this method is called and returns successfully, the static `PegThing::Printer` method will return a NULL pointer. Once this is done, the printer may then be configured and documents may be printed. When printing has completed, the application code must call `Screen()->EndPrint` to inform the screen driver that output will no longer be going to the printer, but to the screen. This effectively destroys the printer object and, therefore, once this function is called, the application code must not try to reference the printer object.

See the Examples section below for a code listing on the steps necessary to properly print.

8.1.2 See Also

PCL Printer

PegScreen

8.1.3 Style Flags

None.

8.1.4 Signals

None.

8.1.5 Derivation

None.

8.1.6 Public Functions:

```
PegPrinter *CreatePegPrinter(void)
```

This function is called by the PegScreen object when the user calls the screen's `BeginPrint` method. User application code should never need to call this function directly. This function is filled in by the installed printer driver to create an instance of the appropriate printer object.

8.1.7 Constructors:

```
PegPrinter(void)
```

The PegPrinter constructor creates the PegPrinter object and sets the object's internal members to reasonable defaults.

8.1.8 Public Functions:

```
virtual PEGINT BeginDoc(const PegPrintSetup
     *pSetup = NULL)
```

This function begins a new document. A document is defined as a print job where all of the pages print with the same printer settings. This function must be called before any other printer functions are called.

Optionally, a PegPrintSetup structure may be passed to this method. If the printer's `Setup` method was previously called, then the parameters used in that call will be used for the document.

The method returns either `PPRINT_ERR_SUCCESS` or `PPRINT_ERR_ERROR`. If an error is returned, you should check the value of the error member of the printer by calling `GetError`.

See the explanation for the `Setup` method for details on using the PegPrintSetup structure.

```
virtual PEGINT BeginPage(void)
```

This begins a new page in the document. `BeginDoc` must be called before this, or this call will fail.

```
virtual PEGINT Cancel(void)
```

Cancels the current print job.

```
virtual PEGINT EndDoc(void)
```

Complimentary to the `BeginDoc` call. This ends the current document.

```
virtual PEGINT EndPage(void)
```

Ends the current page. This is complimentary to `BeginPage`.

```
virtual PEGINT GetColorMode(void) const
```

Returns the color mode in use by the printer object. This can be one of the following:

- PPRINT_MODE_MONOCHROME
- PPRINT_MODE_GRAYSCALE
- PPRINT_MODE_COLOR_SIMPLE
- PPRINT_MODE_COLOR

```
virtual PEGINT GetError(void) const
```

The PegPrinter keeps track of errors that it has encountered when dealing with the actual printer device. The most recent error code may be retrieved using this method. The following is a list of error codes.

- PPRINT_ERR_ERROR - General error value returned in method calls
- PPRINT_ERR_SUCCESS - Successful method call
- PPRINT_ERR_SETUP - A setup item is invalid
- PPRINT_ERR_UNSUPPORTED - Unsupported capability of the printer driver
- PPRINT_ERR_RES - Selected resolution is unsupported by the printer
- PPRINT_ERR_COLOR_MODE - selected color mode is unsupported by the printer
- PPRINT_ERR_ORIENT - Selected paper orientation is unsupported by the printer
- PPRINT_ERR_PAPER_SIZE - Selected paper size is unsupported by the printer
- PPRINT_ERR_WIDTH - Selected width is too big based on the selected paper size and resolution
- PPRINT_ERR_HEIGHT - Selected height is too big based on the selected paper size and resolution
- PPRINT_ERR_COMM_INIT - Could not open the printer device
- PPRINT_ERR_COMM_DATA - Could not send data to the printer device
- PPRINT_ERR_ALLOC - Could not allocate the temporary bitmap used for drawing
- PEGUBYTE PPRINT_ERR_DRIVER - The printer driver is invalid
- PPRINT_ERR_NO_DOC - There is no current document started
- PPRINT_ERR_NO_PAGE - There is no current page started
- PPRINT_ERR_CUR_DOC - There is already a document active
- PPRINT_ERR_CUR_PAGE - There is already a page active
- PPRINT_ERR_NO_PRINTER - The printer could not be open

- PPRINT_ERR_NO_CONTEXT - There is no valid printer context
- PPRINT_ERR_SUSPENDED - Printing has been suspended
- PPRINT_ERR_NONE - No current error

```
virtual PEGINT GetOrientation(void) const
```

Returns the orientation in use by the printer driver. The return value may be one of the following:

- PPRINT_ORIENT_LANDSCAPE
- PPRINT_ORIENT_PORTRAIT

```
virtual PEGINT GetPaperSize(void) const
```

Returns a constant that denotes the paper size currently in use. Currently, the following paper sizes are supported:

- PPRINT_PS_LETTER
- PPRINT_PS_LEGAL_LIST_1
- PPRINT_PS_LEDGER
- PPRINT_PS_EXECUTIVE
- PPRINT_PS_A4
- PPRINT_PS_A3
- PPRINT_PS_COM_10
- PPRINT_PS_MONARCH
- PPRINT_PS_C5
- PPRINT_PS_B5
- PPRINT_PS_DL

By default, the printer driver will use the paper size that is in the default tray of the printer. Internally, the paper size is used to calculate the printable area of the page.

```
virtual PegRect GetPrintRect(void) const
```

This method returns the rectangle used for the printed image boundaries. If a paper size has not been selected, or if there is not an active document, the rectangle will have all of its coordinate members set to -1.

```
virtual PEGINT GetResolution(void) const
```

Returns the resolution currently in use by the driver. The return value may be one of the following:

- PPRINT_RES_75 - 75 dpi
- PPRINT_RES_100 - 100 dpi
- PPRINT_RES_150 - 150 dpi
- PPRINT_RES_300 - 300 dpi
- PPRINT_RES_600 - 600 dpi

It is important to note that not all printers support all of these resolutions. Some types of printers—thermal bar code printers, for example—only support one resolution that is hard wired in the device. But most types of desktop-type printers support a resolution of 300 dots per inch.

```
virtual PEGBOOL IsOpen(void) const
```

Returns TRUE if the printer device was successfully opened and the printer driver is able to communicate with the printer. Otherwise, it returns FALSE.

```
virtual PEGINT Reset(void)
```

Clears the setup and resets the printer.

```
virtual PEGINT Resume(void)
```

This function works in tandem with Suspend to toggle sending draw output to the printer or to the screen.

```
virtual PEGINT SetColorMode(PEGINT ColorMode)
```

This method sets the color mode used to print a subsequent document. Upon success, PPRINT_ERR_SUCCESS is returned. If there is a document currently active, PPRINT_ERR_ERROR will be returned and the internal error flag will be set to PPRINT_ERR_CUR_DOC.

See GetColorMode for a list of valid color modes.

```
virtual PEGINT SetOrientation(PEGINT
     Orientation)
```

This method sets the orientation used to printing a subsequent document. Upon success, PPRINT_ERR_SUCCESS is returned. If there is a document currently active, PPRINT_ERR_ERROR will be returned and the internal error flag will be set to PPRINT_ERR_CUR_DOC.

See GetOrientation for a list of valid orientation codes.

```
PEGINT SetPaperSize(PEGINT PaperSize)
```

This method sets the paper size used to print a subsequent document. Upon success, `PPRINT_ERR_SUCCESS` is returned. If there is a document currently active, `PPRINT_ERR_ERROR` will be returned and the internal error flag will be set to `PPRINT_ERR_CUR_DOC`.

See `GetPaperSize` for a list of valid paper sizes.

```
virtual PEGINT SetResolution(PEGINT Resolution)
```

This method sets the paper size used to print a subsequent document. Upon success, `PPRINT_ERR_SUCCESS` is returned. If there is a document currently active, `PPRINT_ERR_ERROR` will be returned and the internal error flag will be set to `PPRINT_ERR_CUR_DOC`.

See `GetResolution` for a list of valid resolution codes.

```
PEGINT Setup(const PegPrintSetup *pSetup)
```

Before any pages of a document are printed, the document first has to be set up. There are two ways of doing this, and both involve filling in a PegPrintSetup structure that details how the driver configures the printer.

The PegPrintSetup structure is defined as follows:

```
struct PegPrintSetup
{
    PEGINT ColorMode;
    PEGINT Orient;
    PEGINT Res;
    PEGINT PaperSize;
    PEGINT Width;
    PEGINT Height;

    PegPrintSetup(void)
    {
        ColorMode = PPRINT_MODE_MONOCHROME;
        Orient = PPRINT_ORIENT_PORTRAIT;
        Res = PPRINT_RES_75;
        PaperSize = PPRINT_PS_LETTER;
        Width = PPRINT_DEF_WIDTH;
        Height = PPRINT_DEF_HEIGHT;
    }
```

```
};
```

The structure has a default constructor that sets the structure members to reasonable defaults that would typically be supported by any printer.

The `Width` and `Height` members are set to `PPRINT_DEF_WIDTH` and `PPRINT_DEF_HEIGHT`, respectively. These defines instruct the driver to create a printing surface the maximum width and height supported as indicated by the size of the paper in use. The driver version of this method calculates these values and enforces the valid minimum and maximum values.

The remaining members may be set to any value as described above for their respective indicators.

This method may be optionally bypassed if a PegPrintSetup structure is passed to the `BeginDoc` method. The `Setup` function will return an error if there is already a document in progress.

```
virtual PEGINT Suspend(void)
```

This function allows the suspension of drawing output to the printer and redirects all drawing output back to the screen. This function does not actually suspend the printing of the current page in the printer.

A useful example of this method is updating a status bar that shows the current print job progress. Using this method, the first page may be printed and printing then suspended. While printing is suspended, the screen may then be updated, informing the user that page one has completed and page two is being readied. At that point, the `Resume` may be called to resume printing.

8.1.9 Protected Members:

```
virtual void BitmapToPrinter(void)
```

This pure virtual function should be overridden in the printer driver class to write the raster bitmap out to the printer.

```
virtual void ClearBitmap(COLORVAL FillColor =
    TRANSPARENCY)
```

This function fills the raster bitmap with a solid color. By default, this fills the bitmap with the `TRANSPARENCY` color.

```
virtual void CloseDevice(void)
```

This function closes the printer device.

```
virtual PEGBOOL OpenDevice(void)
```

This function attempts to open the printer device. If it succeeds, it returns TRUE. Otherwise, it returns FALSE.

```
virtual void SendData(const void *pData, PEGUINT
     DataLen)
```

This function sends the raster data to the printer.

8.1.10 Examples:

The following is a code listing that details the steps necessary to properly direct drawing output to a printer. It is analogous to drawing into an off-screen bitmap, so the interface should be familiar.

The example below shows the printing of two separate documents, each having any number of pages. Note that the first document is printed in monochrome and the second in simple color.

```
void MyThing::PrintReport(void)
{
    PEGINT ret;
    PegPrintSetup setup;

    // the setup structure has a default constructor which
    // places the minimally supported values in the data
    // members, so all we really need to change for
    // the first document is the resolution, or dots per
inch.
    setup.Res = PPRINT_RES_150;

    // tell the screen driver that we are now directing all
    // drawing operations to the printer.
    Screen()->BeginPrint();

    // here we are telling the printer driver how we wish to
    // setup the printer.
    ret = Printer()->Setup(&setup);

    // this will tell the printer driver to start a new
```

```
    // document based on the settings in setup
    ret = Printer()->BeginDoc();

    // in normal operation, you would want to check the
    // value of ret for PPRINT_ERR_ERROR or
PPRINT_ERR_SUCCESS.
    // If you receive the former value, then you may wish
    // to call Printer()->GetError() to receive the error
    // code that caused the problem. In the remainder of
    // this example, we will bypass this since it obfuscates
    // the portions of the code that focus on using
    // the printer object.

    // start the first page
    ret = Printer()->BeginPage();

    // call some functions that print a header and a report
    // on the first page.
    PrintHeader();
    PrintPage1();

    // end the page. This will also eject the paper from the
    // printer.
    ret = Printer()->EndPage();

    // now, in the same document, print another page.
    ret = Printer()->BeginPage();
    PrintHeader();
    PrintPage2();

    // end the second page.
    ret = Printer()->EndPage();

    // end the first document
    ret = Printer()->EndDoc();

    // change the setup to print in simple color and at
    // a resolution of 300 dots per inch.
    setup.Res = PPRINT_RES_300;
    setup.ColorMode = PPRINT_MODE_SIMPLE_COLOR;

    // this time, we are passing the setup structure in the
    // call to BeginDoc.
```

```
    ret = Printer()->BeginDoc(&setup);

    ret = Printer()->BeginPage();

    PrintBitmapChart();

    ret = Printer()->EndPage();

    // now, print a copy of this object as it appears on
    // the screen. This is how to do a screen dump of
    // a PEG object.
    ret = Printer()->BeginPage();

    Invalidate(mReal);
    Draw();

    ret = Printer()->EndPage();

    ret = Printer()->EndDoc();

    // Now that we have completed printing, tell the screen
    // the we will now be drawing to the screen
    Screen()->EndPrint();
}
```

The printer object is very flexible in what type of output can be produced. Basically, any type of drawing that you could normally do into a bitmap, you can do into a printer. This allows you to create any type of output.

8.2 PCLPrinter

8.2.1 Overview

The PCLPrinter class is a printer driver for printers that support the Hewlett-Packard (HP) Printer Control Language (PCL) level 3 or above. Most modern HP LaserJet and InkJet printers, including HP Photo printers, support this language. If you are in doubt regarding a specific printer, please consult the printer specifications that came with your printer or on HP's website.

This class is a straight derivation of the PegPrinter class and does not implement any new public functions.

8.2.2 See Also

PegPrinter

8.2.3 Style Flags

None.

8.2.4 Signals

None.

8.2.5 Derivation

PegPrinter

8.2.6 Constructors:

```
PCLPrinter(void)
```

The constructor creates an instance of a PCLPrinter object. This function should not be called directly by the application code. Application code should always call the `PegScreen::BeginPrint` method to begin printing.

8.2.7 Public Functions:

```
virtual PEGINT BeginDoc(const PegPrintSetup
    *pSetup = NULL)
```

PCLPrinter overrides the `PegPrinter::BeginDoc` function to send the appropriate control commands to the printer. If `pSetup` is NULL, then it uses the existing setup to start a new document.

```
virtual PEGINT BeginPage(void)
```

PCLPrinter overrides the `PegPrinter::BeginPage` function to send the appropriate control commands to the printer to start a new page.

```
virtual PEGINT Cancel(void)
```

PCLPrinter overrides the `PegPrinter::Cancel` function but it is not currently implemented.

```
virtual PEGINT EndDoc(void)
```

PCLPrinter overrides the `PegPrinter::EndDoc` function to end the current document. It also resets the printer and sends the appropriate exit codes.

```
virtual PEGINT EndPage(void)
```

PCLPrinter overrides the `PegPrinter::EndPage` function to end the current page. It informs the printer that the page is done and instructs the printer to eject the current page.

```
virtual PEGINT Reset(void)
```

PCLPrinter overrides the `PegPrinter::Reset` function but it is not currently implemented.

```
virtual PEGINT Setup(const PegPrintSetup
    *pSetup)
```

PCLPrinter overrides the `PegPrinter::Setup` function to set up the printer control parameters based on `pSetup`.

8.2.8 Protected Members:

```
virtual void BitmapToPrinter(void)
```

PCLPrinter overrides the `PegPrinter::BitmapToPrinter` function to send the raster data to the printer.

8.3 PEG Style Flags

8.3.1 Overview

The PEG style flags are used to control the default appearance and operation of many PEG classes. The style flags can be logically OR'ed together to create the style parameter passed to an object constructor. The style of an object can also be updated using the PegThing member function `SetStyle()`.

8.3.2 Frame styles:

```
FF_NONE
FF_THIN
FF_RAISED
FF_RECESSED
FF_THICK
FF_MASK
```

8.3.3 Text Justification Style:

```
TJ_RIGHT
TJ_LEFT
TJ_CENTER
TJ_MASK
```

8.3.4 Title Style:

```
TF_NONE
TF_SYSBUTTON
TF_MINMAXBUTTON
TF_CLOSEBUTTON
```

8.3.5 Text Thing Style

```
TT_COPY
```

8.3.6 Button Style:

```
BF_REPEAT
BF_PUSHED
BF_DOWNACTION
BF_TOGGLE
BF_EXCLUSIVE
BF_FLYOVER
```

8.3.7 Menu Button Style:

```
BF_SEPARATOR
BF_CHECKABLE
BF_CHECKED
BF_DOTABLE
BF_DOTTED
```

8.3.8 Decorated Button Style:

```
BF_ORIENT_TR
BF_ORIENT_BR
```

8.3.9 List Style:

```
LS_WRAP_SELECT
```

8.3.10 Edit Style:

```
EF_EDIT
EF_PARTIALROW
EF_WRAP
EF_FULL_SELECT
```

8.3.11 Message Window Style:

```
MW_OK
MW_YES
MW_NO
MW_ABORT
MW_RETRY
MW_CANCEL
```

8.3.12 Table Style:

```
TS_SOLID_FILL
TS_PARTIAL_COL
TS_PARTIAL_ROW
TS_DRAW_HORZ_GRID
TS_DRAW_VERT_GRID
TS_DRAW_GRID
```

8.3.13 Table Cell Style:

```
TCF_FORCEFIT
TCF_HCENTER
TCF_HLEFT
TCF_HRIGHT
TCF_VCENTER
TCF_VTOP
TCF_VBOTTOM
```

8.3.14 Spreadsheet Style:

```
SS_CELL_SELECT
SS_PARTIAL_COL
SS_MULTI_ROW_SELECT
SS_MULTI_COL_SELECT
```

8.3.15 Spreadsheet Column Style:

```
SCF_ALLOW_SELECT
SCF_SELECTED
SCF_SEPARATOR
SCF_CELL_SELECT
```

8.3.16 Spreadsheet Row Style:

```
SRF_ALLOW_SELECT
SRF_SELECTED
SRF_SEPARATOR
SRF_CELL_SELECT
```

8.3.17 Notebook Style:

```
NS_TOPTABS
NS_BOTTOMTABS
NS_TEXTTABS
```

8.3.18 Slider Style:

```
SF_SNAP
SF_SCALE
SS_FACELEFT
SS_FACETOP
SS_BOTTOMTOTOP
SS_LEFTTORIGHT
SS_ORIENTVERT
SS_TICMARKS
SS_USERTRACK
SS_USERTRAVEL
SS_THINNEEDLE
SS_THICKNEEDLE
SS_POLYNEEDLE
SS_FACERIGHT
SS_FACEBOTTOM
```

8.3.19 Spin Button Style:

```
SB_VERTICAL
```

8.3.20 Scroll Prompt Style:

```
SP_ONFOCUS
SP_ALWAYS
SP_CONTINUOUS
SP_DOTDOT
SP_WRAP
SP_ON_MASK
SP_MASK
```

8.3.21 Miscellaneous Appearance Style:

```
AF_SIZEABLE
AF_MOVEABLE
AF_DRAW_SELECTED
AF_TRANSPARENT
AF_ENABLED
```

8.3.22 Dial Style:

If PEG_HMI_GADGETS has been defined, then these styles are available.

```
DS_CLOCKWISE
DS_TICMARKS
DS_THINNEEDLE
DS_THICKNEEDL
DS_POLYNEEDLE
DS_RECTCOR
DS_USERCOR

DS_STANDARDSTYLE = DS_THINNEEDLE|DS_RECTCOR|DS_TICMARKS| DS_CLOCKWISE
```

8.3.23 Chart Style:

if PEG_CHARTING has been defined, then the styles below are available.

```
CS_DRAWXGRID
CS_DRAWYGRID
CS_DRAWXTICS
CS_DRAWYTICS
CS_AUTOSIZE
CS_SCROLLED
CS_PAGED
CS_DRAWXAXIS
CS_DRAWAGED
CS_DRAWLEADER
CS_DUALYTICS
CS_DUALYLABELS
CS_XAXISONZEROY
CS_DRAWLINEFILL
CS_DRAWXLABELS
CS_DRAWYLABELS
```

8.4 PEG System Status Flags

PSF_ACCEPTS_FOCUS	This flag indicates that the object will become the receiver of input events when selected. The application level software can modify this flag, but normally this is not advised. If this flag is modified for a particular object, it is important for correct operation that 'breaks' in the tree of objects accepting focus are avoided. In other words, if a parent window cannot accept focus, then neither should any of the window's child objects be allowed to accept focus.
PSF_ALWAYS_ON_TOP	This flag ensures that the object is always on top of its siblings. The application level software can modify this flag.
PSF_BUTTON_DERIVED	This flag indicates that the object is either a PegButton or derived from a PegButton. It is added to objects in the PegButton constructor and should not need to be modified by application code.
PSF_CONTINUOUS_SCROLL	This flag is used on objects that use a scroll bar. If it is turned on, the user will be able to watch the contents of the window scroll up or down as he or she drags the scroll bar's elevator button. If it is turned off, then the user only sees the window move when the elevator button is released. The application level software can modify this flag.
PSF_CURRENT	This flag indicates that the object is in the current branch of the display tree. If the object is a leaf object (i.e. it has no children) and it is current, then it is the object that will receive keyboard input messages.
PSF_KEEPS_CHILD_FOCUS	This flag indicates that when the object gets focus, it does not attempt to give focus to any of its child objects. This is for objects like the PegList that internally maintain which child object should be focused.
PSF_DEFAULT_FOCUS	This flag is given to objects when they receive the PM_CURRENT message indicating that they have focus. They only lose this flag when a different sibling object receives focus. This is used to maintain which object had focus last, even if the parent object loses focus.

PSF_HIDDEN	This flag indicates that the object has been added to a parent object, yet it is not currently displayed on the screen. This should not be modified by the application.
PSF_MOVEABLE	This flag determines whether or not an object can be moved. The application-level software can modify this flag.
PSF_NONCLIENT	This flag, when set, allows a child object to draw outside the client area of its parent. The application-level software can modify this flag after the object is constructed but before the object is displayed.
PSF_OWNS_POINTER	This flag indicates that the object has captured the pointer, meaning all mouse or touch events will be sent to this object, even if they aren't located within its `mReal`. Application software should not modify this directly, but call the `CapturePointer` and `ReleasePointer` functions instead.
PSF_SELECTABLE	This flag is tested by PegPresentationManager to determine if an object is enabled and allowed to receive input messages. The application level software can modify this flag.
PSF_SIZEABLE	This flag determines whether or not an object can be resized. The application level software can modify this flag.
PSF_TAB_STOP	This flag determines whether or not an object receives input focus when the 'tab' key is received. This flag is enabled by default for PegButton objects. Also enabled for editable PegEditField and PegTextBox objects. Enabled for PegPrompt objects that can receive focus. The application software can modify this flag.
PSF_VIEWPORT	This flag, when set, instructs PegPresentationManager that the object should be given a private screen viewport. Objects that have a viewport are drawn differently than objects that do not have a viewport. In general, large objects or objects that have a very complex drawing routine should be given viewports, while small or simple objects should not. By default, all PegWindow derived objects receive viewports, and all other objects do not. This flag should not be changed except immediately after the object is constructed.

PSF_VISIBLE	The object is visible on the screen. This flag should not be modified by the application-level software. Clearing or setting this flag will not have the effect of removing or displaying the object. The PegThing member functions Add and Remove are used for that purpose.
PSF_WINDOW_DERIVED	This flag indicates that the object is either a PegWindow or derived from a PegWindow. It is added to objects in the PegWindow constructor so it should not need to be modified by application code.

8.5 PEG Signal Definitions

8.5.1 PEG Base Signals

PSF_SIZED	Sent when the object is moved or sized.
PSF_FOCUS_RECEIVED	Sent when the object receives input focus.
PSF_FOCUS_LOST	Sent when the object loses input focus.
PSF_KEY_RECEIVED	Sent when an input key that is not supported is received.
PSF_CLICKED	Default left-click notification.
PSF_RIGHTCLICK	Sent when a right-click message is received by the object.

8.5.2 PEG Button Signals

PSF_CHECK_ON	Sent by check box and menu button when checked.
PSF_CHECK_OFF	Sent by check box and menu button when unchecked.
PSF_DOT_ON	Sent by radio button and menu button when selected.
PSF_DOT_OFF	Sent by radio button and menu button when unselected.
PSF_LIST_SELECT	Sent by PegList derived objects, and also PegComboBox.

8.5.3 PEG Notebook Signals

PSF_PAGE_SELECT	Sent when a new page is selected.

8.5.4 PEG Scroll Signals

PSF_SCROLL_CHANGE	Sent by non-client PegScroll-derived objects.

PSF_SLIDER_CHANGE	Sent by PegSlider-derived objects.

8.5.5 PEG Spin Signals

PSF_SPIN_MORE	Sent by PegSpinButton on down or right.
PSF_SPIN_LESS	Sent by PegSpinButton on up or left.

8.5.6 PEG Spreadsheet Signals

PSF_COL_SELECT	Sent when PegSpreadSheet column(s) are selected.
PSF_ROW_SELECT	Sent when PegSpreadSheet row(s) are selected.
PSF_CELL_SELECT	Sent when PegSpreadSheet cell(s) are selected.
PSF_COL_DESELECT	Sent when a PegSpreadSheet column is deselected.
PSF_ROW_DESELECT	Sent when a PegSpreadSheet row is deselected.

8.5.7 PEG Text Signals

PSF_TEXT_SELECT	Sent when the user selects all or a portion of a text object.
PSF_TEXT_EDIT	Sent each time a text object string is modified.
PSF_TEXT_EDITDONE	Sent when a text object modification is complete.
PSF_SCROLL_COMPLETE	Sent when a PegScrollPrompt has completed scrolling in 1-shot mode.

8.5.8 PEG Tree Signals

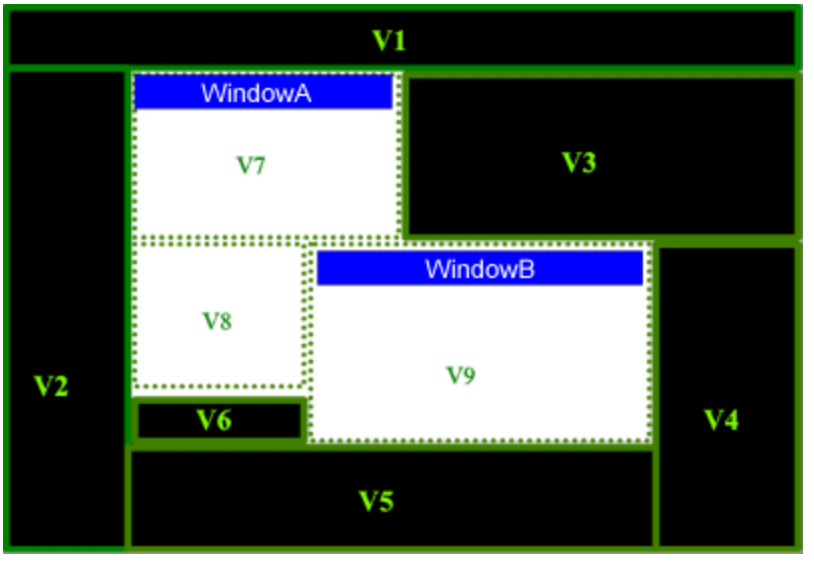
PSF_NODE_SELECT	Sent when a new node is selected.
PSF_NODE_DELETE	Sent when delete key pressed over node.
PSF_NODE_OPEN	Sent when node is opened.
PSF_NODE_CLOSE	Sent when node is closed.

PSF_NODE_RCLICK	Sent when the user right-clicks on the selected node.

8.6 Viewports

PEG uses the concept of viewports to improve drawing efficiency and to allow background drawing operations to occur without overwriting foreground graphics.

Viewports are rectangular areas of the screen owned by certain objects. Each viewport has only one owner, while one object may own several viewports. The diagram below should clarify this concept:



In the diagram above, a typical run-time screen is shown. The black area is the screen background, covered by PegPresentationManager. The two white areas are PEG windows, named WindowA and WindowB. WindowB is on top and partially covering WindowA. In this diagram, the black rectangles with solid green outlines depict the viewports owned by PegPresentationManager. In this case, PresentationManager owns viewports V1-V6. WindowA is divided into two viewports, V7 and V8. Finally, WindowB is on top and has one viewport, V9.

PEG maintains the screen viewports, and you do not ordinarily have to concern yourself with how they work. There is one exception, however, that you may need to be aware of. Normally, only PegWindow-derived objects have viewport status. That means that other smaller objects like PegButton

and PegIcon do not own viewports and simply inherit the viewport(s) of their parent window.

The viewport management algorithm employed by PEG does not allow there to be breaks in the viewport tree. That is, an object that owns viewports (i.e. a PegWindow-derived object) should only be added to another object that owns viewports. This does not mean that you cannot add PegWindow-derived objects to objects that are not derived from PegWindow, because you can. However, when you do this you should set the `PSF_VIEWPORT` status flag of the parent object to make it a viewport owner.

An example should clarify this concept. Suppose you want to create a simple object container class. This container class will simply serve as a parent for a group of lists, windows, and other controls. This is a common thing to do, as it allows you to add and remove the entire group of objects at any time simply by adding or removing the container. Since the container class does not need to actually draw anything, you decide to derive it from PegThing, the most basic PEG class. Since at least some of the children of the PegThing container are PegWindow-derived objects, you will need to make the PegThing container class a viewport owner. If you don't do this, the PegWindow-derived children of the container class won't show up on the screen. You can make the PegThing container class a viewport owner simply by adding the `PSF_VIEWPORT` system status in the container class constructor:

```
AddStatus(PSF_VIEWPORT);
```

Now your container class will work correctly, and both PegWindow-derived children and simple children will be displayed when the parent container class is displayed.

8.6.1 How Scrolling Works

PegWindow provides the capability of adding scroll bars, and using those scroll bars to pan or move the client area of the window. Scroll bars are added by calling the `SetScrollMode()` member function of PegWindow.

The scroll bars added to the window make use of two virtual PegWindow functions: `GetHScrollInfo` and `GetVScrollInfo`. When a scroll bar needs to update itself, it calls these parent window member functions to learn the scroll bar limit, current setting, and percentage visible data. `GetHScrollInfo()` and `GetVScrollInfo()` receive a pointer to a

PegScrollInfo structure. It is the job of these functions to fill in the PegScrollInfo `Min`, `Max`, `Current`, `Step`, and `Visible` values so that the scroll bar is correctly positioned.

The PegWindow class provides default implementations of `GetHScrollInfo` and `GetVScrollInfo`. These implementations examine all client-area children of the window to determine the outer limits that the scroll bars should allow scrolling to. This default implementation also uses the window client area width and height as the scroll bar 'visible' value.

The default implementation works well in most cases, and makes it very easy to create scrolling client areas. All you need to do is add a child window to a scrolling parent that is much larger than the parent client area. The default implementation will adjust the scroll bars such that the entire child window can be viewed by moving the horizontal and/or vertical scroll bars.

In some cases, the default operation does not provide the required function. In these cases, you can override the `GetHScrollInfo` and `GetVScrollInfo` functions to return custom scrolling information. For example, suppose you want to create a continuous time plot of data values with a horizontal scroll bar to move back and forth in the time period displayed. In this case, you would create a derived PegWindow class in order to draw the chart data in the window client area. You would also provide a overridden version of the `GetHScrollInfo` function to make the horizontal scrollbar reflect the accumulated time values. In this case, the PegScrollInfo minimum value might be the starting time of data recording, the maximum value would be the current time, and the visible amount would be the time period visible in the window client area.