ClassPad 300 SDK Programming Guide

Table of Contents:

Introduction

About this Document

The purpose of this document is to provide you with a reference guide while programming on the ClassPad 300. This document is not meant to be a tutorial, or a complete list of functions contained in the SDK. Please Refer to **SDK Programming Tutorial** for a tutorial on working from a "Hello World" to a scribble program on the ClassPad. And refer to the **SDK Reference Guide** for a complete list of functions and classes in the SDK.

About the SDK API

The ClassPad 300 SDK's application programming interface is C++. If you are unfamiliar with C++, there are several tutorials on the Internet that can assist you. Bruce Eckel provides digital copies of his book *Thinking in C++* for free on his web site: http://www.mindview.net/.

Keep in mind that the ClassPad is an embedded system and therefore does not support all of the C/C++ standard library functions. If you are new to C++ it is suggested that you first spend some time understanding the basics of the language before trying to write ClassPad add-in applications.

Portable Embedded GUI – PEG

The ClassPad's user interface classes are all based on the Portable Embedded GUI system, or PEG. In this section we will give a broad overview of PEG. This will include a detailed look at the PegThing, the class on which all viewable objects are based. We will also look at how PEG uses the PegPresentationManager to store these viewable components in memory. Finally, we will go over some fundamental data types that are used in PEG, but not based on the PegThing.

Static PEG Objects

There are three global static objects in PEG that are very important in understanding how all of PEG is connected. These three objects are:

```
static PegThing::PegPresentationManager *Presentation();
static PegThing::PegMessageQueue *MessageQueue();
static PegThing::PegScreen *Screen();
```

We discuss each of these objects in more detail below.

The PegPresentationManager

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

The PegMessageQueue

When a control such as a button or menu is pressed, it creates an event that places a message in the PegMessageQueue. The PegMessageQueue is a simple encapsulated FIFO message queue with member functions for queue management. The messages placed in PegMessageQueue are the driving force behind the graphical interface. These messages contain notifications and commands that cause the graphical elements to redraw themselves, remove themselves from the screen, resize themselves, or perform any number of various other tasks. Messages can also be user-defined, allowing you to send and receive a nearly unlimited number of messages whose meaning is defined by you. The PegMessageQueue is discussed in detail in the Messages portion of this document.

The PegScreen

PegScreen is the PEG class that provides the drawing primitives used by the individual PEG objects to draw themselves on the display device. PEG 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 insure that PEG is easily portable to any target environment.

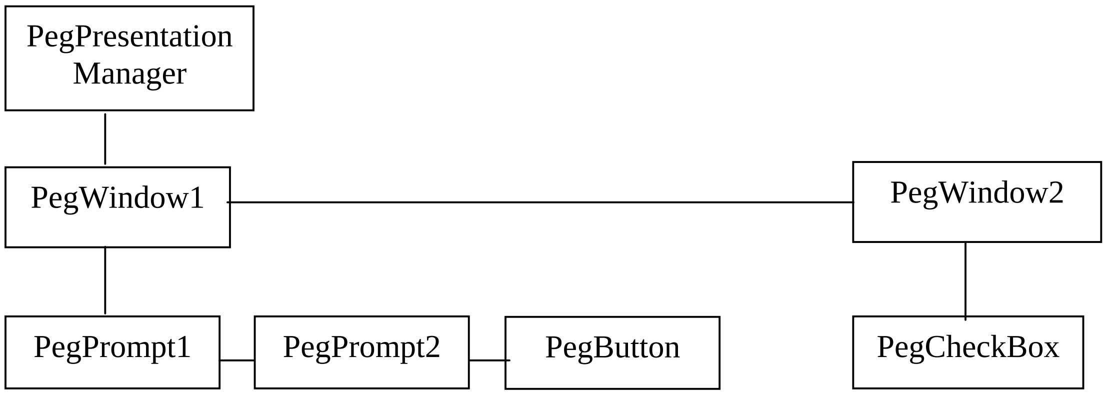
The PegThing

The most important and fundamental class in the PEG library is the PegThing. 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.

Because of the importance of these public functions, we will go though most of them in this section. We will also provide brief explanations of how and when to use the function.

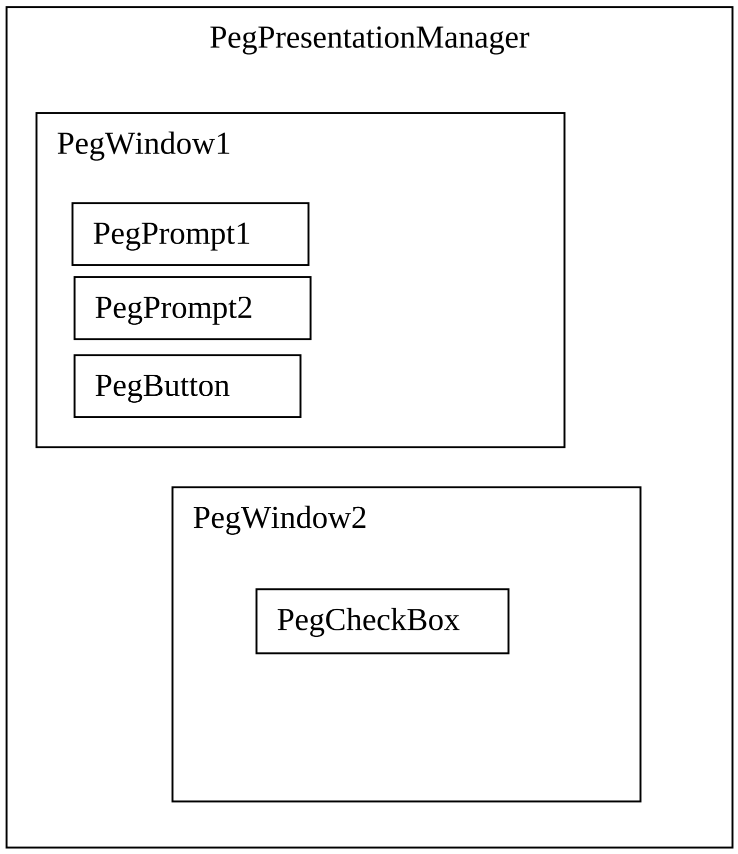
Traversing the Presentation Tree of PegThings

When you add a new PegThing to the PegPresentationManager, you begin creating a tree of all viewable objects. As more objects are added, the tree begins to take shape with relationships of parent, child and siblings. Take this possible example:



If we take the role of PegWindow1, our parent would be PegPresentationManager, our sibling would be PegWindow2 and our children would be PegPrompt1, PegPrompt2 and PegButton.

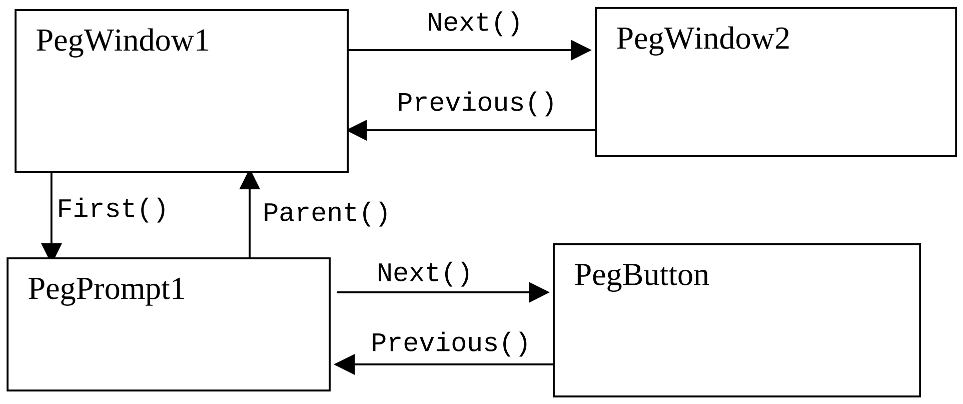
Given this arrangement of PegThings, you could expect an arrangement something like the following to be drawn to the screen:



PEG provides the following functions to access and traverse this tree structure:

```
PegThing* PegThing::Parent(void);
PegThing* PegThing::First(void);
PegThing* PegThing::Next(void);
PegThing* PegThing::Previous(void);
```

Their use can best be explained using a close up view of a portion of the previous tree diagram:



Again, let’s take the role of PegWindow1. **First()** would return the first child in our linked list of children, PegPrompt1. **Parent()**, although not drawn, would return the

PegPresentationManager. **Next()** would return PegWindow2. Since all lists are terminated with a NULL, **Previous()** would return NULL. So, if we wanted traverse all of the children of an object, we could use the following code:

```
PegThing *pTest = Parent()->First(); // first child of my parent
int iSiblings = 0;

// Since all lists are NULL terminated,
// we can loop on while ptest!=NULL
while(pTest)
{
      if (pTest != this)
      {
            iSiblings++;
      }
      pTest = pTest->Next();
}
```

Adding to and Removing from the Tree

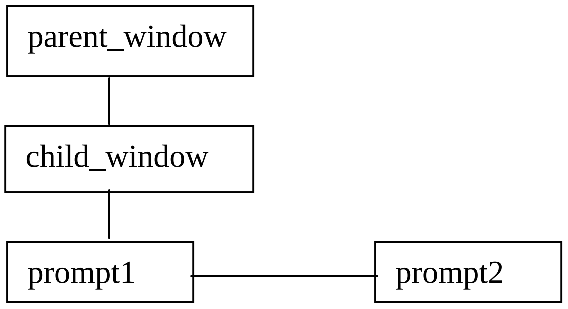
PEG provides two functions to add PegThings to the presentation tree:

```
void PegThing::Add(PegThing *Who, BOOL bDraw = TRUE);
void PegThing::AddToEnd(PegThing *Who, BOOL bDraw = TRUE);
```

Add() always adds the child to the beginning of the linked list of children. If you would like to add to the end, use **AddToEnd()**.

Let’s look at some example code and see what the tree it creates will look like:

```
PegRect Rect(10, 10, 40, 40);
PegWindow *child_window = new PegWindow(Rect);
PegWindow *parent_window = new PegWindow(Rect + 50);

PegPrompt *prompt1 = new PegPrompt(0, 0, “Prompt1”);
PegPrompt *prompt2 = new PegPrompt(0,30, “”Prompt2”);

prent_window->Add(child_window);
child_window->Add(prompt1);
child_window->AddToEnd(prompt2);
```



There are also two functions to remove an object from the tree:

```
PegThing* PegThing::Remove(PegThing *Who, BOOL bDraw = TRUE);
void Destroy(PegThing *Who);
```

The PegThing member function **Remove()** is used to detach an object from the object's parent. This removes the object from the tree, but does not remove the object from memory. The PegThing member function **Destroy()** is similar to **Remove()**, although **Destroy()** both removes the object from the tree and deletes the object from memory.

As long as items belong to the PegPresentationManager, all memory will be freed automatically. However, once you use **Remove()** to remove an object from the tree **you are in charge of deleting its memory.**

Changing a PegThing's Size or Location

PegThing has two member functions that deal with resizing or relocating itself. These functions are:

```
virtual void Resize(PegRect Rect);
virtual void Center(PegThing *Who);
```

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

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

If an object is visible when it is resized, it will automatically perform the necessary invalidation and drawing. It is perfectly acceptable to resize an object that is not visible, in fact in many cases this is the best time to do it. Note that passing a rectangle of the same size as your PegThing, but at different location will cause the **Resize()** function to move your PegThing's without changing its size.

Center() will adjust the screen coordinates of *Who* such that *Who* is horizontally and vertically centered over the client area of *this*. *Who* does not necessarily have to be a child of *this*, although that 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
```

PegThing Type and Attributes

PegThing Type

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

```
UCHAR Type(void) { return muType; }
void  Type(UCHAR uSet) {muType = uSet;}
```

Type() called with no arguments will return that PegThing's type, whereas **Type()** called with a UCHAR will set the object's type.

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

```
PegThing *pTest = First(); // get pointer to first child object
while(pTest) // search to the end of list if necessary
{
      if (pTest->Type() == TYPE_STATUS_BAR)
      {
            PegStatusBar *pStatBar = (PegStatusBar *) pTest;
           //use pStatBar to call member functions or change attributes
            break; // found the status bar, exit the loop
      }
      pTest = pTest->Next(); // continue down the list of children
}
```

PegThing Object IDs

Another way to find a specific PegThing is with its **Object ID.** You can assign each PegThing a unique object ID value that can then be used to identify the object. When an object sends a notification signal to a parent window, the object ID is contained in the iData member of the notification message. If you do not give an object ID to a PegThing, then that PegThing will not send notification signals.

To get and set an object's ID, use the following functions:

```
WORD Id(void) {return mwId;}
void Id(WORD wId) {mwId = wId;}
```

A few object ID values are reserved by PEG for proper operation of dialog boxes and message windows. Therefore you should always begin your private control enumeration with a value of 1, so as not to overlap the reserved ID values. Valid user object IDs are in the range between 1 and 999.

You can locate a child object at any time using the object's ID with the **Find()** function. **Find()** will search the child list of the current object for an object with an ID value matching the passed in value. An example of setting an Object's ID and then using **Find()** to retrieve it follows:

```
Window1::Window1(….) : PegWindow(…)
{
      Id(ID_WINDOW1);
}
PegWindow *Window2::FindWindow1(void)
{
      return Presentation()->Find(ID_WINDOW1);
}
```

PegThing Signals

All PEG objects support a basic set of **signals**. PegThing provides storage for the object ID, the signal mask, and member functions for modifying the signal mask. The signal mask determines which signals a PegThing will recognize. The mask can be changed with the following functions:

```
void SetSignals(WORD wMask);
void SetSignals(WORD wId, WORD wMask)
```

The first 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 second function, with the extra argument, is used to both assign an object's ID and the associated signal mask. Remember that an object without an object ID, or an object with an ID of 0, will not send signals.

You can use the following functions to determine what signals are set for a given PegThing:

```
WORD GetSignals(void) {return mwSignalMask;}
BOOL CheckSendSignal(UCHAR uSignal)
```

GetSingals() will return the entire signal mask for a PegThing, whereas **CheckSendSignal()** will check a PegThing for a specific signal.

The following is a list of all available signals:

PSF_CLICKED	Default button select notification
PSF_FOCUS_RECEIVED	Sent when the object receives input focus
PSF_FOCUS_LOST	Sent when the object loses input focus
PSF_TEXT_SELECT	Sent when the user selects all or a portion of a text object
PSF_TEXT_EDIT	Sent each time text object string is modified
PSF_TEXT_EDITDONE	Sent when a text object modification is complete
PSF_CHECK_ON	Sent by check box and menu button when checked
PSF_CHECK_OFF	Sent by check box and menu button when unchecked
PSF_DOT_ON	Sent by radio button and menu button when selected
PSF_DOT_OFF	Sent by radio button and menu button when unselected
PSF_SCROLL_CHANGE	Sent by non-client PegScroll derived objects
PSF_SLIDER_CHANGE	Sent by PegSlider derived objects
PSF_SPIN_MORE	Sent by PegSpinButton when up or right arrow is selected
PSF_SPIN_LESS	Sent by PegSpinButton when down or left arrow selected
PSF_LIST_SELECT	Sent by PegList derived objects, including PegComboBox
PSF_PAGE_SELECT	Sent by PegNotebook when a new page is selected
PSF_KEY_RECEIVED	Sent when an input key that is not supported is received
PSF_SIZED	Sent when the object is moved or sized

PegThing Status Flags

All PEG objects have certain system status flags associated with them. The system status flags are important to the correct operation of the library, but are generally not often needed by the application software. PegThing maintains an object's system status flags, and provides public functions that allow you to examine and/or modify the system status flags for an object. These functions are:

```
BOOL  StatusIs(WORD wMask)
virtual void  AddStatus(WORD wOrVal);
virtual void  RemoveStatus(WORD wAndVal);
WORD  GetStatus(void);
```

StautsIs() is used to test if a PegThing has a specific system flag set. The list of valid system flags follows.

PSF_VISIBLE	The object is visible on the screen. This flag should not be modified by the application level software.
PSF_CURRENT	This flag indicates that the object is in the current branch of the display tree. If the object is a leaf object (i.e. it has no children) and it is current, then it is the object which will receive keyboard input messages.
PSF_SELECTABLE	This flag is tested by PegPresentationManager to determine if an object is enabled and allowed to receive input messages. The application level software can modify this flag.

PSF_SIZEABLE	This flag determines whether or not an object can be resized. The application level software can modify this flag.
PSF_MOVEABLE	This flag determines whether or not an object can be moved. The application level software can modify this flag.
PSF_NONCLIENT	This flag, when set, allows a child object to draw outside the client area of its parent. The application level software can modify this flag after the object is constructed but before the object is displayed.
PSF_ALWAYS_ON_TOP	This flag insures that the object is always on top of its siblings. The application level software can modify this flag.
PSF_ACCEPTS_FOCUS	This flag indicates that the object will become the receiver of input events when selected. The application level software can modify this flag, but normally this is not advised.
PSF_VIEWPORT	This flag, when set, instructs PegPresentationManager that the object should be given a private screen viewport.

GetStatus() returns the status flag of the current PegThing.

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

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

PegThing Style

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

```
void  FrameStyle(WORD wStyle);
WORD FrameStyle(void);
virtual WORD Style(void);
virtual void Style(WORD wStyle);
```

The **FrameStyle()** functions can be used to get or set the appearance of the frame for most PegThing derived objects. The available styles are:

FF_NONE	No frame
FF_THIN	Thin black frame

FF_THICK	Thick Frame

The **Style()** function is used to get or set the style flags for an object. **Not all style flags are supported by all classes**.

The following is a list of general categories of PegThings, and what style flags they support:

All PegThing Styles	FF_NONE FF_THIN FF_THICK
Text Control Styles	TJ_RIGHT TJ_LEFT TJ_CENTER
Button Styles	BF_REPEAT BF_SELECTED BF_DOWNACTION BF_FULLBORDER
Menu Styles	BF_SEPARATOR BF_CHECKABLE BF_CHECKED BF_DOTABLE BF_DOTTED
Edit Styles	EF_EDIT EF_WRAP EF_FULL_SELECT EF_CHARWRAP
Message Window Styles	MW_OK MW_YES MW_NO MW_ABORT MW_RETRY MW_CANCEL
Notebook Style	NS_TEXTTABS
Progress Bar Styles	PS_SHOW_VAL PS_RECESSED PS_LED PS_VERTICAL PS_PERCENT
Spin Button Style	SB_VERTICAL

Refer to the ClassPad 300 SDK Reference Guide for more details on these styles. In all cases, the desired style flags can be logically ORed together to form one style parameter.

Current Focus

All PegThing based classes support the idea of gaining and losing focus. If a PegThing has current focus, it means all key input is sent to that object. The user typically changes the input focus by pressing the pen on an object. When an object gets the input focus it gets a PM_CURRENT message and its PSF_CURRENT status is set. All its parents up the tree also become current. You can detect if an object is a member of the input focus branch of the presentation tree at any time by testing the PSF_CURRENT system status flag:

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

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

Setting Focus

You can override the user's input selection and manually command PegPresentationManager to move the input focus at any time by calling:

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

When focus is moved from one PegThing to another, PM_NONCURRENT messages are sent to objects that are no longer members of the input focus branch, and PM_CURRENT messages are sent to objects that are members of the new input focus branch. The effect is that non-directed input messages will be sent to the newly designated input object.

When children are added to the presentation list, the newest child is always placed at the beginning of an object's children list. By default, the first child in a parent's list is the child that has focus. This means that **last** child added to a PegThing will always have default focus (**Add()** always adds a child to the front of the sibling list). Since this may not be the desired result, there are three functions that explicitly assign or test which child has default focus:

```
virtual BOOL HasDefaultFocus(void);
virtual PegThing* GetDefaultFocus(void);
virtual void SetDefaultFocus(PegThing* pThing);
```

HasDefaultFocus() tests if an object has default focus. **GetDefaultFocus()** returns the PegThing that has default focus. **SetDefaultFocus()** assigns a PegThing default focus.

Capture and Release of the Pointer

When an object gains focus, it also gains focus of the pointer. Once you gain focus of the pointer you may wish to continue to receive events based on the pointer even if your object no longer has focus. The following two functions allow you to accomplish this:

```
void CapturePointer(void);
void ReleasePointer(void);
```

These functions can be useful when dragging an object on the screen. You could, for example, call **CapturePointer()** on pen down and **ReleasePointer()** on pen up of a certain object. In this case, even if the pen leaves this object's boundaries (and goes into a different module window, for example) the pointer messages will continue to go to the object. When the user lifts up the pen, the pointer will be released and the object will no longer own the pointer. Be extremely careful that when you do capture the pointer that you are certain to release it. Otherwise user input will be stuck on the object that captured and didn't release the pointer.

PEG Data Types

In section we will look at some important data types that are used in PEG and throughout the ClassPad, but are not based on the PegThing.

Fundamental Data Types

The following simple data types are used by PEG instead of the intrinsic data types defined by the compiler to avoid conflicts when running on CPUs with differing basic word length and data manipulation capabilities. The comment next to each data type describes the storage requirements PEG requires for each type:

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

PegPoint

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

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

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

PegRect

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

```
struct PegRect
{
      void Set(SIGNED x1, SIGNED y1, SIGNED x2, SIGNED y2)
      {
            wLeft = x1;
            wTop = y1;
            wRight = x2;
            wBottom = y2;
      }

      void Set(PegPoint ul, PegPoint br)
      {
            wLeft = ul.x;
            wTop = ul.y;
            wRight = br.x;
            wBottom = br.y;
      }

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

There is more information about using these functions in the *Window and Screen Drawing* section of this document.

PegMessage

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

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

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

For user-defined messages, all but the wType and pTarget message fields can be used in any way desired. The iUserData, wUserData, and pData fields are intended to allow you to easily pass any type of data in your user defined messages. Refer to the section on *Messages and Message Handling* for more information on PegMessage.

CPString

The CPString class encapsulates the memory allocation necessary for string handling, while still providing access to a raw char*. For more information on the member functions of CPString, refer to the section *Strings and String Handling In the ClassPad* in this document.

CPArray

The CPArray class manages a variable sized array of void's, encapsulating the memory management. The CPArray class provides an easy interface for managing objects in memory.

The following is a list of all public member functions as well as a comment describing what each function does:

```
// Return the current size of the array
int  GetSize () const
```

```
// Get the value at the given index
void *  GetAt (int nIndex)

// Set the value at the given index
void  SetAt (int nIndex, void *pElement)

// Resize the array.  Note that any objects that fall of the end of the
// array are the programmer’s responsibility
void  SetSize (int iNewSize, int iGrowBy=-1)

// Add an item to the end of an array
int  Append (const CPArray &array)

// Copy the array
void  Copy (const CPArray &array)

// Free unused memory above the current upper bound.
void  FreeExtra ()

// Set the array to the given index.  If the index is out of the bounds
// of the array, grow the array to include this index
void  SetAtGrow (int nIndex, void *pElement)

// Insert element nCount times at the specified index
void  InsertAt (int nIndex, void *pElement, int nCount=1)

// Insert elements from another CPArray starting from the given index
void  InsertAt (int nIndex, CPArray *array)

// Removes nCount elements starting at specified index
void  RemoveAt (int nIndex, int nCount=1)

// Add an element to the end of the CPArray
void  Add (void *pElement)

// Removes all objects from the CPArray.  The removed objects are not
// deleted
void  RemoveAll ()
```

The **GetAt()** method is memory safe, and will return NULL if the index is outside the array. Likewise, **SetAtGrow()** will resize the array if it is too small. CPArray does not manage the memory of the objects. It only manages the memory of the array structure. It is your responsibility to delete all objects from memory.

CPList

The CPList is a singly linked circular list of void*. CPList has several member functions that allow for navigation through the list, as well as retrieving items from the list. The following is a list of all public functions and a brief description of what they do (Note: ent is typedef’d to void*):

```
/// Returns the object pointer at the begining (head) of the list
ent Head();
```

```
/// Returns the object pointer at the end (tail) of the list
ent Tail();

/// Inserts ent at the head of the list
virtual ent Insert(ent a);

/// Inserts ent b before ent a in the list
void InsertBefore(ent a, ent b);

/// Inserts ent b after slink g
void InsertAfter(slink* g, ent b);

/// Inserts ent b after ent a in the list
bool InsertAfter(ent a, ent b);

/// Replaces ent a with ent b
bool Replace(ent a, ent b);

/// Appends ent @a a to the tail of the list
int Append(ent a);

/// Removes the last element from the list
ent Pop();

/// gets the first element of the list without removing it
ent Get();

/// clears list, does not delete objects
void Clear();

/// Removes the link referring to ent a from the list. a is not deleted
void Remove(ent a);

/// returns the number of elements in the list
int Count() const;

/// returns TRUE if the list contains ent a
int Contains(ent a) const;

/// Index operator gets the i'th element if a list
ent operator[](int i) const;
```

Messages and Message Handling

The driving force behind the graphical interface on the ClassPad comes from events from the input devices and other PEG objects. All of these events all sent as messages in the PegMessageQueue. The PegMessageQueue is an encapsulated FIFO message queue with member functions for queue management. It also performs timer maintenance and miscellaneous housekeeping.

The messages placed in the queue contain notifications and commands that cause the graphical elements to redraw themselves, remove themselves from the screen, resize themselves, or perform any number of various other tasks. Messages can also be user-defined, allowing you to send and receive a nearly unlimited number of messages whose meaning is defined by you. This section will discuss these messages' structure, as well as how the messages are handled and used.

PegMessages

Definition

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

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

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

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

Peg System Messages

PEG messages can be divided into two types: PEG system messages and USER messages. As we just mentioned, whether a message is a system message or a user message is determined by the value of the message wType field.

PEG uses system messages internally to command objects to perform certain operations. For example, before an object is drawn PEG sends the internal message PM_SHOW. PEG knows that any preparation to that needs to be done before drawing can be called when that object's PM_SHOW message is received.

It is very common to want to receive and process system messages within your application. This is sometimes called 'intercepting' a message, because you can catch a message that PEG has sent to an object and change the interpretation of the message, or even cause the object to ignore the message entirely.

While at first you may want to avoid intercepting system messages, as your confidence in working with the library grows you will find that this is often the most convenient way to accomplish many tasks. Some of the common Peg System Messages are listed below. For a complete list, refer to pmessage.hpp**.**

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_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_SIZE	This message is sent to an object to cause it to re-size. This is equivalent to calling the **Resize()** function. Note that PEG does not differentiate between moving an object and resizing an object. Both are accomplished via the Resize operation. The new size for the object is included in the message Rect field.
PM_CLOSE	This message is recognized by PegWindow derived objects, and causes the recipient to remove itself from its parent and delete itself from memory.
PM_HIDE	This message is sent to an object whenever it is removed from a visible parent.
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_POINTER_MOVE	This message is sent to an object whenever the pen moves over the object.
PM_LBUTTONDOWN	This message is sent to an object when the user generates a pen down event. PegPresentationManager routes pen input directly to the lowest child object containing the click position. If the child object does not process pen 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 pen is included in the message Point field.
PM_LBUTTONUP	This message is sent to an object when the user releases the pen. The flow of this message is identical to PM_LBUTTONDOWN.
PM_DRAW	This message can be sent to an object to force that object to redraw itself.
PM_REDRAW	Like draw, but only updates the area that is marked as invalid.
PM_CURRENT	Sent to an object when it becomes a member of the branch of the presentation tree that has input focus.
PM_NONCURRENT	Sent when the object is no longer part of the focus tree.
PM_POINTER_ENTER	Sent when the pen enters a PegThing's bounding box.
PM_POINTER_EXIT	Sent when the pen exits a PegThing's bounding box.
PM_EXIT	This message is sent to PegPresentationManager to cause termination of the application program.
PM_VSCROLL	Sent by a scrollbar to signal vertical scrolling.
PM_HSCROLL	Sent by a scrollbar to signal horizontal scrolling.
PM_TIMER	This message is sent to an object that has started a timer via the PegMessageQueue TimerSet function when that timer expires. The ID of the timer is included in the iData member of the message.
PM_KEY	This message is sent to the current input object when keyboard input is received. The message iData member contains the corresponding ASCII character code, if any, and the lData member of the message contains the keyboard scan code, if available.
PM_CUT	User requested to cut data from the current object to the CPClipboard.
PM_COPY	User requested to copy data from the current object to the CPClipboard.
PM_PASTE	User requested to paste data to the current object from the CPClipboard.
PM_DIALOG_NOTIFY	This message is sent to the owner of a PegDialog when the dialog window is closed if the dialog window is executed non-modally. The message iData member will contain the

	ID of the button used to close the dialog window.
HM_SYS_ZOOM	Zoom the active CPModuleWindow to full screen.
HM_SYS_SWAP	Swap the two CPModuleWindows in the CPMainFrame.
HM_SYS_CLOSE	The application is closing. You must save your state when you get this message.
HM_SYS_RESUME	The ClassPad is powering off. Save your state so you can resume later.
HM_SYS_KEYBOARD	Turn on the keypad.
HM_SYS_CLEAR	The CLEAR key was pressed.
PM_LOSING_FOCUS	Sent just before target loses focus
PM_GAINING_FOCUS	Sent after target gains focus
PM_GET_INPUT_STATE	Sent to request the current input state of what has focus. Used by dialogs to restore the selection, cursor, scroll, etc. The pointer parameter MUST point to a PegInputStateContainer object. When receiving this message, any input control should save its state and give it to the container with the SetInputState member.
PM_FIRST_START	Sent to the current focus after PegAppInitialize is finished but before the presentation manager's Execute loop
PM_VALIDATE	Sent to a notify a dialog control to validate it's data

User Defined Messages

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

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

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

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

Peg Signals

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

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

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

A simple example of using SIGNALs is detecting a button click. To send a signal, your button must be created with an Object ID greater than 0. For example, here is the creation on the save button taken from the AddressBook example that came with the SDK:

```
// Create Save Contact button
// The button gets created with ID SAVE_ID
b = new PegBitmapButton(rr, &gbsaveBitmap, SAVE_ID,AF_ENABLED|TT_COPY);
```

When this button is clicked, a unique message will be sent to the window's message processing function that is formed by combining the Message Id PSF_CLICKED and the object ID SAVE_ID. In the next section we will continue this example by showing how this signal will be processed by the button's parent window.

Handling Messages

Any add-in that you write that must respond to user input will have to process PegMessages and Signals. For a PegThing to respond to a message it must override the following function:

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

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

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

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

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

```
SIGNED MyClass::Message(const PegMessage &Mesg)
{
      switch (Mesg.wType)
      {
      case UIM_SHOW:
            PegWindow::Message(Mesg);
            // add your own code here:
            break;

      case USER_DEFINED_MSG1:
            // code for your user message
            break;
      case USER_DEFINED_MSG2:
            // code for another user defined message:
            break;

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

      default:
            // pass all other messages down to the base class:
            return CPWindow::Message(Mesg);
      }
      return 0;
}
```

In the previous section, we created a button with Object ID SAVE_ID. To catch the signal that this button sends, our Message function would look like this:

```
SIGNED AddressWindow:Message(const PegMessage &Mesg)
{
      switch (Mesg.wType)
      {
      case SIGNAL(SAVE_ID, PSF_CLICKED):
            // code for Save button clicked:
            Save();
            break;

      default:
            // pass all other messages down to the base class:
            return CPWindow::Message(Mesg);
      }
      return 0;
}
```

It is recommended that you refer back to the AddressBook example to get more information about how to create an overridden **Message()** function.

Message Flow and Routing

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

Peg Timers

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

PEG timers are maintained by PegMessageQueue. In order for PEG timers to function, your system software must call the PegMessageQueue member function **TimerTick** periodically to indicate to PEG that one tick time has expired. Timers are created and destroyed with the following functions:

```
inline void SetTimer(WORD wId, LONG lCount, LONG lReset)
inline void KillTimer(WORD wId)
```

You start a PegTimer by calling the PegMessageQueue member function **SetTimer()**. The parameters allow you to specify a timer Id value, the first timeout period, and successive timeout periods. The timer Id value can be any number greater than zero. If you have one window or control that creates many timers, you will probably want to assign them unique Id values so that you can recognize each timer expiration message. While you have an active timer running, you will receive a PM_TIMER message in your **Message()** handling function each time the timer expires. When you want to stop a timer, you use the PegMessageQueue member function **KillTimer().** If you pass an Id value of zero to the KillTimer function, all timers owned by the calling object are deleted.

For more information on messages, refer to the example add-in DebugExample. This add-in outputs the names of the messages that get sent to the MessageQueue on the ClassPad. It is a very useful add-in to help you understand when message are sent, and in what order they are sent.

Window and Screen Drawing

The WindowsExample Add-in

This section will provide you with information on how drawing works on the ClassPad 300. To help show how these ideas are applied to an add-in application, we have provided an example add-in that uses most of the concepts that will be discussed.

The add-in is located in **Documents\ClassPad 300 SDK \Examples\WindowsExample\WindowsExample.dev**.

It is recommended that you run this application and see what type of drawing functions it performs. It is also recommended that you try changing the code to see how your changes affect what is drawn to the screen. We will refer back to and discuss the source code of this add-in throughout this section to see how the drawing topics we cover are used.

An Overview of Windows in the WindowsExample

This section serves as a brief explanation of the windows used in WindowsExample. For a more extensive discussion of window classes on the ClassPad, see the *User Interfaces* section.

CPMainFrame

Almost every application on the ClassPad has a CPMainFrame as its base window. From using the ClassPad you should be aware that an application can have two main module windows that can be resized and swapped. Depending on which of these two windows has focus different menus, toolbars and status bar are displayed. It is the CPMainFrame's job to handle these multiple module windows and make sure that the correct UI is displayed.

CPModuleWindow

A CPModuleWindow is the base class for applications or "modules" on the ClassPad. Each CPModuleWindow can have its own set of menus, toolbar items, and status bar messages. CPModuleWindows must be added to a CPMainFrame. The CPMainFrame is then in charge of handling any swapping or resizing of multiple CPModuleWindows.

CPWindow

A CPWindow is a rectangular screen area that supports drawing and scrolling. CPWindows are based upon PegWindows. The only difference between the two is that a CPWindow allows drawing to the window in relative coordinates. Note that CPWindows cannot have their own menus or toolbars.

Windows in WindowsExample

Let's take a look at the WindowsExample add-in and see how the three different windows are used to create the application.

First we will look at the creation of the CPMainFrame in PegAppInitialize. The mainframe is created by passing in a peg rectangle that is the size of the mainframe.

```
void PegAppInitialize(PegPresentationManager *pPresentation)
{
      PegRect Rect;
      Rect.Set(MAINFRAME_LEFT,MAINFRAME_TOP,
               MAINFRAME_RIGHT,MAINFRAME_BOTTOM);

      CPMainFrame *mw = new CPMainFrame(Rect);
```

Next, we want to add an ExampleWindow to the mainframe. Remember that only CPModuleWindows can be added to a CPMainFrame. Therefore, ExampleWindow must be, and in fact is, derived from a CPModuleWindow. Here is the code that creates an instance of ExampleWindow that is the size of a full screen application in the mainframe:

```
      PegRect ChildRect = mw->FullAppRectangle();
      ExampleWindow *ex_win = new ExampleWindow(ChildRect,mw);
```

Now let's jump out of PegAppInitialize for a moment, to see what ExampleWindow's constructor does.

```
ExampleWindow::ExampleWindow(PegRect rect, CPMainFrame
*frame) :CPModuleWindow(rect,0,0,frame)
{
      HasLines = false;
      SetScrollMode(WSM_AUTOSCROLL);
      PegRect r = mClient;
      r -= 20; // make the pan window a bit smaller

      m_panWin = new PanWindow(r);
      Add(m_panWin);
}
```

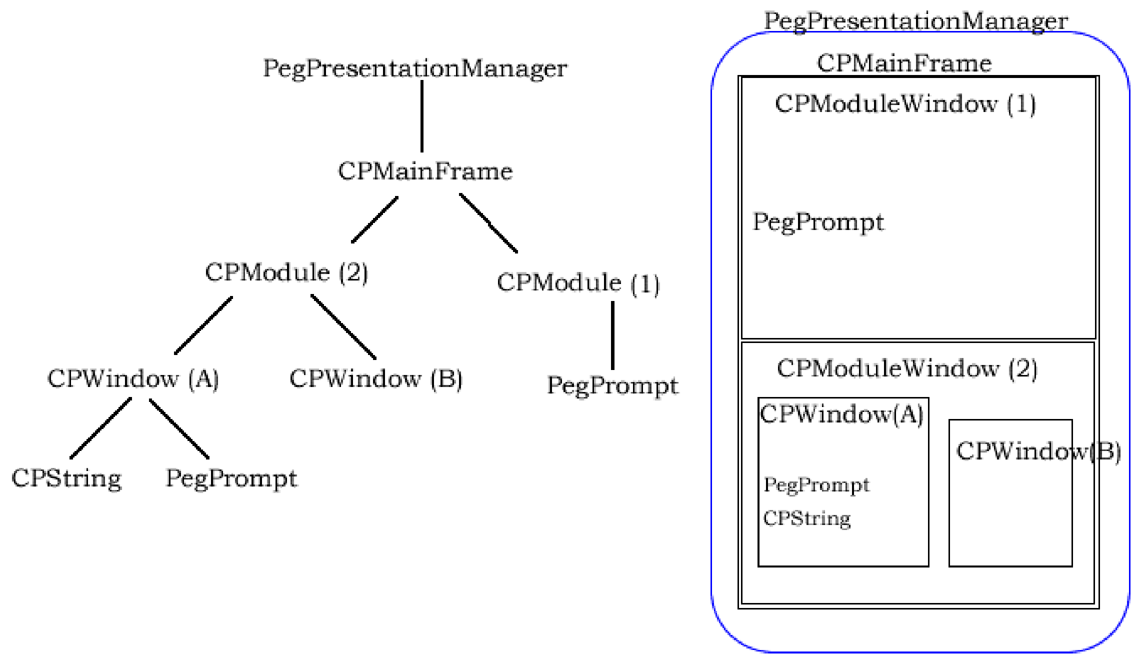
You can see that ExampleWindow has a reference to a PanWindow. PanWindow is based on the CPWindow class. The constructor creates a new PanWindow and adds it to ExampleWindow.

Finally if we jump back to PegAppInitialize we see that the ExampleWindow gets added to the CPMainFrame, and the CPMainFrame gets added to the PegPresentationManager.

```
      mw->SetTopWindow(ex_win);

      mw->SetMainWindow(ex_win);

      pPresentation->Add(mw);
};
```

So to sum that all up: A CPWindow got added to a CPModuleWindow that got added to a CPMainFrame that was added to the PegPresentationManager. All applications created for the ClassPad will follow part of this hierarchy. That is, all CPModuleWindows must be added to a CPMainFrame, and all CPMainFrames must be added to a PegPresentationManager. However, anything that is based on a PegThing can be added to a CPModule Window.

The following graphic illustrates a possible parent-child hierarchy using these windows. On the right is what the hierarchy may look like on the ClassPad. Note that the PegPresentationManager is not actually visible on the screen.



Coordinates on the ClassPad

When designing applications with a graphical user interface on a specific platform it is imperative that you know what type of screen coordinates the platform uses. In the ClassPad, all coordinates sizes are based on pixels. In all PEG base classes coordinates are absolute starting from the top left corner of the screen, which is (0,0). While this may not seem to be a problem at first, when you start adding several windows with toolbars and menu bars using the top left corner as a reference point can become confusing.

For example, let's say that you wanted to add a PegPrompt to (0,0) of your PegWindow inside a CPMainFrame. Using the code:

```
text = new PegPrompt(0,0, (PEGCHAR*)"Prompt at 0,0");
Add(text);
```

you might expect a result like the screenshot on the left:

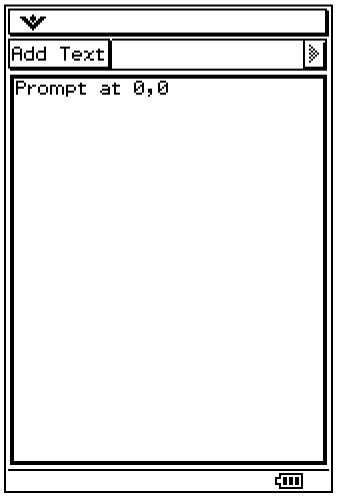


Fig1. AddR a prompt at 0,0

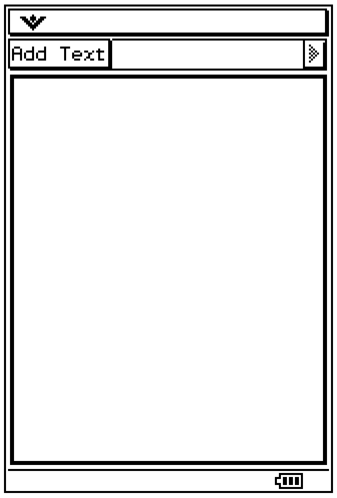


Fig2. Add a prompt at 0,0

However, while the prompt may be at (0,0) in your PegWindow's coordinates, it is not at (0,0) according to the CPMainFrame coordinates. Placing the prompt at absolute (0,0) would create it outside of your window – somewhere underneath the menu bar, and therefore it would not get drawn (Fig2).

To fix this problem the ClassPad 300 SDK includes the CPWindow. CPWindow and all objects that are derived from CPWindow, support a function called **AddR()**. **AddR()** does the same thing as **Add()** – adds a PegThing to "this". However, **AddR()** allows you to add objects to coordinates relative to the window that called **AddR()**. Therefore the following code would produce the screenshot in Fig1:

```
text = new PegPrompt(0,0, (PEGCHAR*)"Prompt at 0,0");
AddR(text);
```

It is not required that you use **AddR()** with a CPWindow, but if you are dealing with windows that are being moved and resized it is easier than trying keep up with absolute coordinates.

Drawing on the ClassPad

Overriding the Draw() Function

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

Usually when you override the **Draw()** function you will allow the base-class **Draw()** function to execute at some point in your routine. A common question is "When do I call the base-class **Draw()** function?". This depends on whether you want your custom drawing to appear on-top or below the default operation. If you want your customizations to appear ‘on-top’ (which is usually the case), you should call the base-class draw function before you do your own drawing. In some cases you may not want to invoke the base-class **Draw()** function at all. This is perfectly OK, as long as you remember a few rules:

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

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

The **Draw()** method in ExampleWindow.cpp does not call its base-class **Draw()**, but instead the previously stated three rules are followed properly.

```
void ExampleWindow::Draw()
{
      BeginDraw();
      DrawFrame();
      DrawLines();
      DrawChildren();
      EndDraw();
}
```

First the function starts with **BeginDraw()**. Next, the custom drawing is done by functions **DrawFrame()** and **DrawLines()**. After that the custom drawing is finished, PanWindow is drawn by calling the **DrawChildren()** function. Finally, we finish the **Draw()** method with a call to **EndDraw()**.

Invalidating and Drawing outside of the Draw() Method

You can also write functions that draw on the screen outside of the **Draw()** function. These functions must be members of a PegThing derived class, or at least have access to a PegThing object, since all of the PegScreen drawing functions require as a parameter a pointer to the PegThing object calling the drawing function. PegScreen requires this pointer to insure that an object is not allowed to draw outside of the area it ‘owns’ on the screen.

PegScreen only allows drawing to occur to areas of the screen that have been invalidated. Areas of the screen are invalidated by calling the **Invalidate()** function. If all of your drawing is done with an overridden **Draw()** function, you don’t need to worry about screen invalidation, since your **Draw()** function is called specifically ***because*** an area of the screen has been invalidated.

If you need to draw on the screen outside of the draw function you need to remember to invalidate the area you are going to draw to before you start drawing. If you want to be allowed to draw anywhere within the client area of your object, you can simply call the **Invalidate()** function with no parameters, which invalidates the area of the screen corresponding to an objects client area. You can also calculate and specify a more limiting rectangle to clip your drawing, and pass that rectangle to the **Invalidate()** function. No matter how large the invalidated rectangle on the screen, you are never allowed to draw outside of an object's borders.

Drawing and Invalidating in WindowsExample

WindowsExample uses invalidation to draw in the **DrawLines()** function of ExampleWindow.cpp. This function is called when a user clicks on the Toggle Lines button.

Before **DrawLines()** can begin drawing to the screen, it must first invalidate the area where it will draw. In this case the entire ExampleWindow will be drawn to, so **Invalidate()** with no clip region is called to invalidate then entire mClient. Let's take a closer look at DrawLines:

```
void ExampleWindow::DrawLines(void)
{
    PegColor Color(BLACK, WHITE, CF_FILL);
    SIGNED yPos = mClient.wTop;
    Invalidate(); // invalidate my client area
    BeginDraw(); // prepare for drawing
    Rectangle(mClient, Color, 0);
    if(HasLines)
    {
        while(yPos <= mClient.wBottom)
        {
            Line(mClient.wLeft, yPos, mClient.wRight, yPos, Color);
            yPos += 4;
        }
    }
    EndDraw();
}
```

As you can see, before drawing the lines the entire ExampleWindow area is invalidated with the call to Invalidate. Commenting out the **Invalidate()** call will give you the result you should expect -- nothing will get drawn to the screen. You should also notice that all drawing functions are placed in between a **BeginDraw()** and **EndDraw()** call.

Looking at ExampleWindow's draw function you may wonder why there is a call to **DrawLines()**. As mentioned before, Draw gets called **because** the screen was invalidated. We want to make sure the lines are redrawn after an invalidation occurs that wasn't because a user clicked on the Toggle Lines button. Consider moving the PanWindow with the pen. Since we are moving the location of PanWindow, a new portion of ExampleWindow will become visible. ExampleWindow must be redrawn to display this portion. This is done by calling the parent's draw function in **OnPointerMove()** in PanWindow.cpp:

```
Parent()->Draw();
```

If we did not call **DrawLines()** in the **Draw()** function then ExampleWindow would be redrawn without the lines. You should try commenting out the call to **DrawLines()** in the Draw function to see this for yourself.

If you look in PanWindow.cpp you will see that we are drawing in the functions **AddText()** and **OnPointerMove()**, but do not call **Invalidate()**. In both of these functions the **Resize()** method is called to either expand or move the PanWindow. The **Resize()** method automatically calls **Invalidate()** before the window is moved or resized and after the move or resize is complete. This eliminates the user from being responsible for calling **Invalidate()** when using **Resize().**

Object Boundaries

mReal, mClient and PegRects

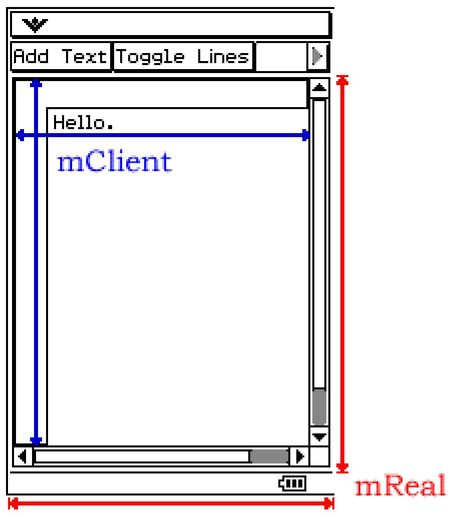
All PegThing derived classes have two rectangles associated with them: mReal and mClient. The rectangle mReal defines the outermost limits of an object. The object and all children of the object are prevented from drawing outside the mReal rectangle.

The mClient rectangle defines the interior boundaries of an object. The mClient rectangle is always a sub-set of the mReal rectangle. All children of an object are clipped to the parent's mClient rectangle, unless the children have PSF_NONCLIENT system status, in which case they are clipped to the parent's mReal rectangle.

For simple objects such as PegButton and PegString, the mClient rectangle is smaller than the mReal rectangle only by the width of the object border. If the object has no border, the mClient and mReal rectangles are identical. For PegWindow and derived classes, the mClient rectangle is further reduced by the size of the non-client decorations such as a title bar, menu bar, status bar, and horizontal and vertical scroll bars. In other words, non-client children are positioned in the region between the mClient rectangle limits and the mReal rectangle limits.

The rectangle you pass to most PEG object constructors defines the outermost limits of the object, hence this rectangle becomes the mReal member rectangle. PEG objects initialize their mClient area by calling the PegThing member function **InitClient()**, which reduces the mClient area by the object border width. PegWindow performs further operations to reduce the mClient area as decorations are added to the window.

For example, here is what ExampleWindow's mReal and mClient look like when there are scroll bars:



Notice that the mClient, the area you can draw to, does not extend over the scrollbars. This means that you cannot draw over the scroll bars and the drawable area in your window shrinks when scroll bars are added.

mClient can shrink in other ways as well. If you look back at the **Draw()** function in ExampleWindow, you will see that there is a call to **DrawFrame()**. This puts a one pixel border around ExampleWindow and makes mClient one pixel smaller than mReal on all sides. The thicker you make this frame, the more mClient will shrink.

Using Object Boundaries in WindowsExample

Bounding rectangles are used in WindowsExample in the functions **DrawText()** and **OnPointerMove()** of PanWindow.cpp. We will first take a look at how they are used in **DrawText()**, then in **OnPointerMove().**

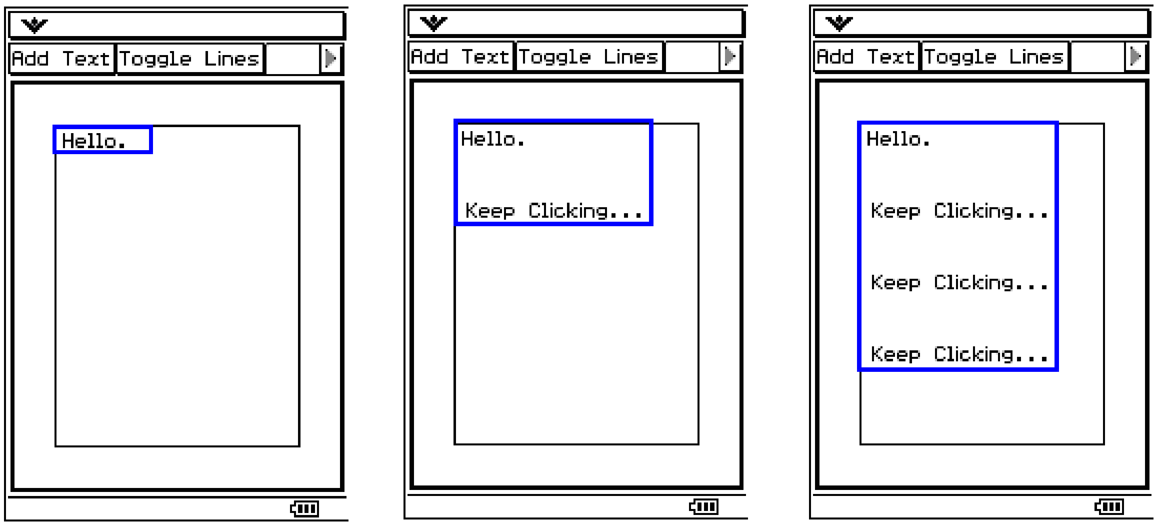
Bounding Rectangles in DrawText()

The **DrawText()** function adds a new PegPrompt to the bottom of PanWindow each time it is called. At first this seems simple enough, but what happens when you run out of room in PanWindow? How do you know when the PegPrompts have grown past the height of PanWindow?

As mentioned in the previous section, all objects derived from PegThing have a mClient and an mReal associated with them. To know when we have run out of room in PanWindow, we need a running rectangle that is the union of the mClients for the PegPrompts that have been added. To accomplish this we create a class member called promptRect that is a PegRect. Each time we add a new PegPrompt, we take the union of the new PegPrompt's mClient and the existing promptRect. The code looks like this:

```
text = new PegPrompt(4, promptRect.Height()+25,
                        (PEGCHAR*)"Window Resized!");
promptRect |= text->mReal;
AddR(text);
```

Notice that we add the PegPrompt 25 pixels lower than the height of promptRect. This spaces the PegPrompts out so they are not drawn on top of each other. The following figure will give you an idea of how promptRect grows as a PegPrompt is added:



A rough idea of what promptRect's boundary, drawn in blue, looks like on startup and after adding a couple of new PegPrompts

In a couple more clicks, the height of promptRect will grow larger than PanWindow's mClient height. When this happens, there will be nowhere to put the next PegPrompt and we will have to resize PanWindow.

To resize PanWindow, we need to create a rectangle that will be the new size of the window and pass it to the **Resize()** function. Here is the code to do this:

```
if(promptRect.Height() + 25 > mReal.Height())
{
      text = new PegPrompt(4, promptRect.Height()+25,
                              (PEGCHAR*)"Window Resized!");
      promptRect |= text->mReal;
      AddR(text);

      BeginDraw();

      PegRect new_rect = mReal;
      new_rect.wBottom += ( 25 + text->mReal.Height());
      Resize(new_rect);

      // Calling Resize with the same size is used as a "trick" to
      // force the parent to check for and add or remove scrollbars
      Parent()->Resize(Parent()->mReal);
      Parent()->Draw();
      EndDraw();
}
```

First, we create a new rectangle and set it equal to PanWindow's mReal. We then add the height of the PegPrompt plus the 25 pixels of spacing to the bottom of it. This rectangle is passed to **Resize()** to set PanWindow's mReal to the new rectangle. Make sure you do not set mReal and mClient explicitly. You can end up in an invalid state where mClient is larger than mReal. By using the function **Resize()** it will make sure that if mClient grows that mReal will also grow if necessary.

Bounding Rectangles in OnPointerMove()

In **OnPointerMove()**, PanWindow's mReal is being moved by the amount that the pen has been dragged. The ideas used here are very similar to what we did in **AddText()**. Again we will pass **Resize()** a rectangle representing where PanWindow's new mReal is located. However, this time the window will not change size, just location. Here is the code that accomplishes this:

```
BeginDraw();
PegRect rect;

// find the difference in x and y from this point p, to the lastPoint
diffx += p.x - m_lastPoint.x;
diffy += p.y - m_lastPoint.y;


// Set rect to mReal and then shift it by the difference in x and y
rect = mReal;
rect.Shift(diffx,diffy);

// Resize invalidates the old rect and new one for us
// It also shifts all children in the window (so we don't
// have to worry about repositioning the PegPrompts)
Resize(rect);

// Calling Resize with the same size is used as a "trick" to
// force the parent to check for and add or remove scrollbars
Parent()->Resize(Parent()->mReal);
Parent()->Draw();
EndDraw();
```

First we find how much the pen has moved in both the x and y direction by subtracting the current point from the previous point. We then take a rectangle equal to mReal and shift it by the x and y deltas. Finally, we pass this rectangle to **Resize()**. This time **Resize()** does not change the size of PanWindow, just its location. Since we are using **Resize()** there is no reason to call invalidate.

Scrollbars

How Scrolling Works

PegWindow provides the capability of adding scroll bars, and using these scroll bars to pan or move the client area of the window. Scroll bars are added by calling the **SetScrollMode()** PegWindow member function.

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 wMin, wMax, wCurrent, wStep, and wVisible 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, and use 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 an overridden version of the GetHScrollInfo function to make the horizontal scroll bar reflect the accumulated time values. In this case, the ScrollInfo 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.

Scrolling in WindowsExample

The WindowsExample add-in is a good example of using the automatic scrolling provided with PEG. We know that once PanWindow is resized or moved, it will need to scroll within ExampleWindow. To support scrolling we need to make sure that the following are all true:

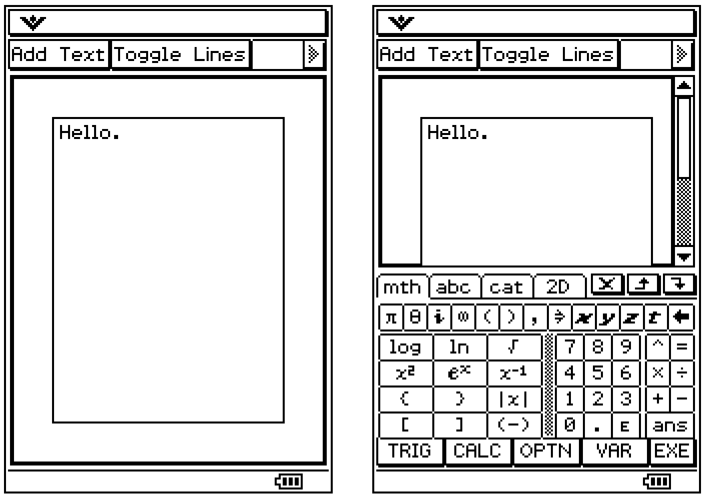
- ExampleWindow is a scrollable window
- ExampleWindow is the parent of the window we want to scroll (PanWindow)
- ExampleWindow sets the correct scrolling mode

These three tasks are easily accomplished, most without any thought about scrolling. ExampleWindow is derived from a CPModuleWindow, which is a scrollable window. When we add PanWindow to ExampleWindow, ExampleWindow becomes PanWindow’s parent. Finally, we need to set ExampleWindow’s scroll mode to be WSM_AUTOSCROLL in its constructor:

```
ExampleWindow::ExampleWindow(PegRect rect, CPMainFrame
*frame) :CPModuleWindow(rect,0,0,frame)
{
      HasLines = false;
      SetScrollMode(WSM_AUTOSCROLL);
      PegRect r = mClient;
      r -= 20; // make the pan window a bit smaller
      m_panWin = new PanWindow(r);
      Add(m_panWin);
}
```

Scrolling is now set up to work in ExampleWindow.

When you run WindowsExample, you'll notice that PanWindow is smaller than ExampleWindow and there are no scrollbars on startup. However, if you bring up the soft keyboard, you will see that scrollbars automatically appear thanks to the automatic scroll mode.



ExamlpeWindow on startup with no scrollbars, and ExampleWindow with automatic scrollbars after bringing up the soft keyboard.

In both **AddText()** and **OnPointerMove()**, PanWindow can change so that scrollbars might be required. In **AddText()** the height of PanWindow can grow larger than the height of ExampleWindow. In **OnPointerMove()**, PanWindow can be moved outside of ExampleWindow's mClient. In both cases we can force ExampleWindow to check its scrollbars by calling **Resize()** and passing in its mReal. The following code is used in both functions:

```
// Calling Resize with the same size is used as a "trick" to force the
// parent to check for and add or remove scrollbars
Parent()->Resize(Parent()->mReal);
Parent()->Draw();
```

As the comments say, this is a bit of a "trick" to cause ExampleWindow to realize that it needs scrollbars to completely hold PanWindow. Try commenting out the **Resize()** and see what happens. If you move PanWindow off the screen, no scrollbars are added. But if you bring up the soft keyboard the appropriate scrollbars will be drawn.

We suggest that you continue to add and comment lines out of the WindowsExample add-in. Once you understand everything in the example, you will have a good understanding of how drawing works on the ClassPad. If you have more questions, refer to the ClassPad 300 SDK Reference Guide.

User Interfaces

We've discussed how to draw on the ClassPad and briefly touched on its windowing architecture. In this section we will discuss in detail what user interfaces are available on the ClassPad as well as visit each type of window that is supported in the SDK.

Windows on the ClassPad

In the Window and Screen Drawing section, we gave an overview of the windows that were used in the WindowsExample add-in application. In this section we will discuss these again as well as all of the other windows available in the SDK.

PegWindow and PegWindow Derived Windows

Class Name	Derived From	Styles	Signals
PegWindow	PegThing	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

A PegWindow is a basic rectangular screen area supporting scrolling and clipping. Many of the Windows in the ClassPad are based on a PegWindow. PegWindow provides the capabilities of being re-sized by the user, having a virtual client area, having one of several frame styles, and controlling non-client-area scroll bars.

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.

PegWindow and PegWindow derived classes are also by default Viewports. 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 also provides improved visual appeal.

The following example will create a PegWindow 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);
    Add(pWin);
}
```

PegPresentationManager

Class Name	Derived From	Styles	Signals
PegPresentationManager	PegWindow	None	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

The 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.

The PegPresentationManager is also responsible for routing keyboard and pen input to the object with the current focus.

PegDecoratedWindow

Class Name	Derived From	Styles	Signals
PegDecoratedWindow	PegWindow	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

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 the 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 types of child objects added. You can even nest PegDecoratedWindow objects with themselves, creating complex and interesting window types.

The following example adds a PegDecoratedWindow that is the same size as your mainframe:

```
PegRect Rect;
Rect.Set(MAINFRAME_LEFT, MAINFRAME_TOP, MAINFRAME_RIGHT,
                                    MAINFRAME_BOTTOM);
pPresentation->Add(new PegDecoratedWindow(Rect));
```

CPMainFrame

Class Name	Derived From	Styles	Signals
CPMainFrame	PegDecoratedWindow	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

CPMainFrame is derived from PegDecoratedWindow. It also supports a menu bar, toolbar and status bar. A CPMainFrame has the ability to handle more than one CPModuleWindow. This includes updating the menus, toolbar and statusbar depending on which CPMoudleWindow is active. If you create an add-in with a CPModuleWindow, you must place it inside of a CPMainFrame.

Here is an example that creates a CPMainFrame that is the size of the mainframe window. This is done in every example that comes with the SDK in the PegAppInitialize function:

```
PegRect Rect;
Rect.Set(MAINFRAME_LEFT, MAINFRAME_TOP, MAINFRAME_RIGHT,
                                    MAINFRAME_BOTTOM);

CPMainFrame *mw = new CPMainFrame(Rect);
```

CPModuleWindow

Class Name	Derived From	Styles	Signals
CPModuleWindow	CPWindow	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

A CPModuleWindow is the base class for windows that represent modules or applications. When you create an add-in, your main module windows will most likely be based on a CPModuleWindow. All of the examples that come with the SDK are built this way.

A CPModuleWindow can have a menu, toolbar and statusbar. CPModuleWindows are added to a CPMainFrame, which controls the resizing and swapping of multiple CPModuleWindows as well as displaying the correct UI.

Here is an example of creating a CPModuleWindow and adding it to the top of a CPMainFrame:

```
CPMainFrame *mw = new CPMainFrame(Rect);
PegRect ChildRect = mw->FullAppRectangle();
CPModuleWindow* win = new CPModuleWindow (ChildRect,mw);
```

```
mw->SetTopWindow(win);
```

CPTabbedWindow

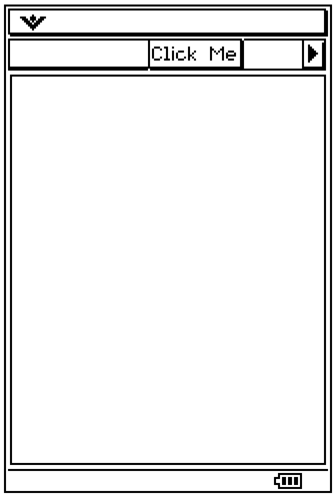
Class Name	Derived From	Styles	Signals
CPTabbedWindow	CPWindow	FF_NONE	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

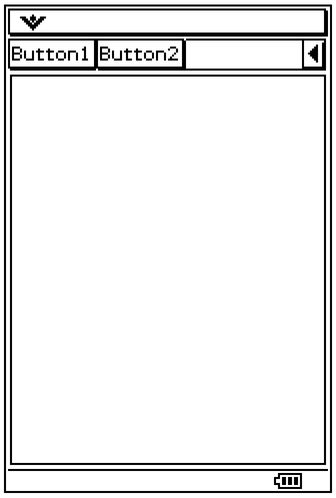
A CPTabbedWindow allows a virtual toolbar that is longer than the physical toolbar. This is helpful when your user interface controls will not all fit on the standard toolbar. The CPTabbedWindow has two panes each of which can contain PEG controls, such as buttons, text entry fields, etc.   It also has an arrow button that lets the user tab between panes.

The following is an example of creating a tabbed window in an object's **AddUI()** function:

```
void YourModule::AddUI()
{
     CPTabbedWindow* tui = (CPTabbedWindow*) m_ui;
     CPWindow* pane0 = tui->GetFirstPane();
     CPWindow* pane1 = tui->GetSecondPane();

     PegRect rr = {1,1,70,15};
     PegEditBox *m_eb = new PegEditBox(rr,0,FF_THIN | EF_EDIT, NULL,30);

     pane0-> AddToolbarButton (m_eb);
     PegTextButton* b = new PegTextButton(71,1, "Click Me",
                                    CLICKME_ID,AF_ENABLED|TT_COPY);
      pane0-> AddToolbarButton (b);

      b = new PegTextButton(1,1, "Button1",
                                    BUTTON1_ID,AF_ENABLED|TT_COPY);
      pane1-> AddToolbarButton (b);

      b = new PegTextButton(40,1, "Button2", BUTTON2_ID,
                                     AF_ENABLED|TT_COPY);
      pane1-> AddToolbarButton (b);
}
```

The result of this code is:





PegNotebook

Class Name	Derived From	Styles	Signals
PegNotebook	PegWindow	FF_RAISED FF_RECESSED NS_TEXTTABS	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_PAGE_SELECT

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 either contain 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 very custom appearance. Regardless of tab type, the tabs can be displayed at the top or bottom of the notebook window.

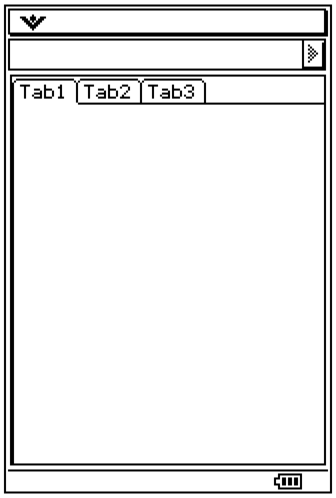
Constructing and displaying PegNotebook requires the following steps:

- 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 sub-objects of the page. This window is then populated with the desired group of child objects, and set as the notebook client object.

The following code adds a PegNotebook with text tabs to a CPModuleWindow:

```
CPMainFrame *mw = new CPMainFrame(Rect);

PegRect ChildRect = mw->FullAppRectangle();
CPModuleWindow* swin = new CPModuleWindow(ChildRect,0,0,mw);
mw->SetTopWindow(swin);

PegNotebook *p = new PegNotebook(ChildRect,
                        NS_TOPTABS|NS_TEXTTABS,3);
p->SetTabString(0, (PEGCHAR*)"Tab1");
p->SetTabString(1, (PEGCHAR*)"Tab2");
p->SetTabString(2, (PEGCHAR*)"Tab3");

swin->AddR(p);

mw->SetMainWindow(swin);
```

The result on the ClassPad is:



PegMessageWindow

Class Name	Derived From	Styles	Signals
PegMessageWindow	PegWindow	FF_THIN MW_OK MW_YES MW_NO MW_ABORT MW_RETRY MW_CANCEL	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

PegMessageWindow is a popup window class for display warning, error, or other status information to the user.

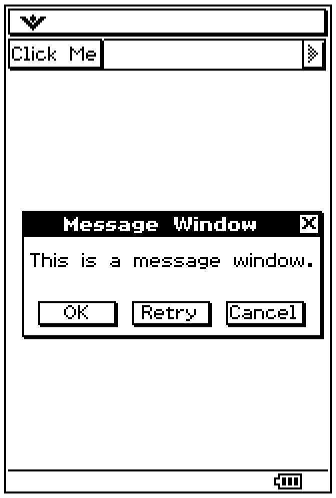
The PegMessageWindow class provides a quick way to display information messages. PegMessageWindow may contain a title bar, message line, and miscellaneous buttons. PegMessageWindow supports both modal and non-modal execution. In addition, the signal generated when the MessageWindow is closed by the user may be directed to any top-level window.

Modal execution is achieved by calling the MessageWindow **Execute()** function. **Execute()** will add the MessageWindow 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 MessageWindow.

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 pre-defined message window buttons.

Here is a simple example of creating a PegMessageWindow:

```
void MyWindow::ModalMessage(void)
{
      PegMessageWindow *pWin = new PegMessageWindow("Message Window",
              "This is a message window.", MW_OK|MW_CANCEL|MW_RETRY);

      Add(pWin);
      pWin->Execute();
}
```

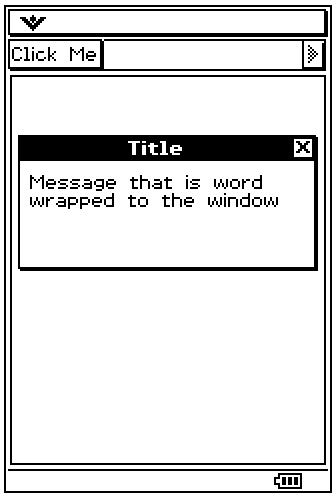


For the most part PegMessageWindow assumes that your error message will fit on one line. If you need line-wrapping you should use the following constructor:

```
PegMessageWindow(const PEGCHAR *Title, const PEGCHAR *Message,
                    const PEGCHAR *Comment, WORD wStyle, WORD wStyle2,
                    PegBitmap *pIcon, WORD dummy1, MessageWindowTypeEnum
                    type=ERROR_WINDOW);
```

The following example will create a PegMessageWindow with wrapping:

```
const PEGCHAR* pTitle = (PEGCHAR*) "Title";
const PEGCHAR* pMessage = "Message that is word wrapped to the window";
PegMessageWindow *win = new PegMessageWindow(pTitle,NULL,pMessage,
                        MW_OK|FF_THIN, 0, NULL, 0, ERROR_WINDOW);
win->Execute();
```



PegProgressWindow

Class Name	Derived From	Styles	Signals
PegProgressWindow	PegMessageWindow	FF_THIN MW_OK MW_YES MW_NO MW_ABORT MW_RETRY MW_CANCEL	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

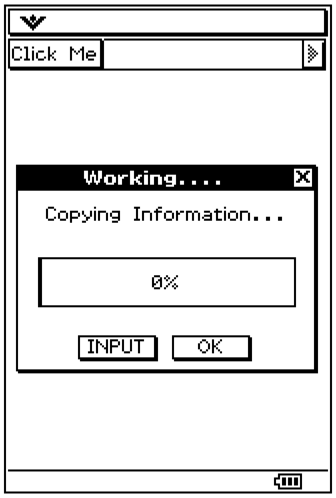
PegProgressWindow is an extension to PegMessageWindow that adds a progress bar to a message window. This makes it very easy to create and display a message and progress bar to the user during a long operation.

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. For more information on the PegProgressBar, see the section *Other User Interface Controls*.

The progress bar added to a PegProgressWindow always has a scale of 0 to 100. It is up to the application software to pre-scale the input value accordingly.

The style of the progress bar displayed in the window client area is passed to the PegProgressWindow constructor.

```
void ExampleWindow::ModalMessage(void)
{
    PegProgressWindow *pWin = new PegProgressWindow("Working....",
                "Copying Information...", MW_OK|FF_RAISED, FF_THIN);

    Center(pWin);
    Add(pWin);
}
```



CPFrameWindow and CPFrameWindow Derived Windows

Class Name	Derived From	Styles	Signals
CPFrameWindow	PegThing	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

A CPFrameWindow is a lightweight window class similar to PegWindow. Because it uses less memory, the CPFrameWindow does not support scrolling. CPFrameWindow serves as the base class for the windows in the SDK that are not based off of PegWindow. Like PegWindow, CPFrameWindow also supports viewports.

To add a CPFrameWindow to another window, use the following code:

```
PegRect Rect = mClient;
CPFrameWindow *f = new CPFrameWindow(Rect);
Add(f);
```

SCWindow

Class Name	Derived From	Styles	Signals
SCWindow	CPFrameWindow	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

SCWindow is an extension of the CPFrameWindow that adds support for events. Instead of handling user input like pen and keyboard events in the **Message()** method, SCWindow extends handling of events with the following event handlers:

- OnLButtonDownEvent(const SCEvent& e)
- OnLButtonUpEvent(const SCEvent& e)
- OnMouseMoveEvent(const SCEvent& e)
- OnKeyEvent(const SCEvent& e)
- OnExtendedKeyEvent(const SCEvent& e)

These methods are virtual. If you derive a sub-class from SCWindow you can create your own event handlers. Of course you can still decide to handle these events directly in the Message method if you'd like. If you handle these events in the Message method then the event handlers listed above will never be called.

The typical usage for SCWindow is to first derive your own CPModuleWindow subclass, and then create another window class like SCWindow or SCWindowWithMode to sit inside your CPModuleWindow class. The CPModuleWindow class can manage scrolling of the SCWindow class while the SCWindow or SCWindowWithMode class can manage events or modes.

Here is an example that creates a SCWindow inside a CPModuleWindow:

```
SCWindow *sc = new SCWindow(Rect);
swin->Add(sc);
```

SCWindowWithMode

Class Name	Derived From	Styles	Signals
SCWindowWithMode	SCWindow	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

SCWindowWithMode further extends a SCWindow by adding modes. Modes encapsulate the most important events for a given task, allowing you to change the behavior of events in you window at runtime.

For example, say that you have two drawing modes: point drawing and line drawing. The pen down in point drawing mode would create a point at that location. A pen down in line drawing mode would either mark the end of a line or the beginning of a line. You can isolate these two pen down events by creating two SCMode classes and using them with an SCWindowWithMode. Before you begin a line draw, switch the window to the line drawing mode. Then all events will be handled appropriately. If you change to point drawing, switch modes to point drawing. To change modes simply pass your SCMode class to your SCWindowWithMode with this function:

```
void SetMode (SCMode *mode);
```

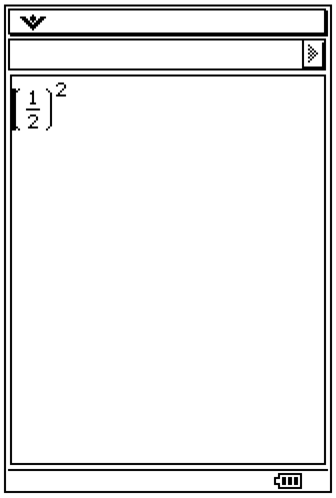
MathWindow

Class Name	Derived From	Styles	Signals
MathWindow	SCWindowWithMode/ CPUndoThing	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

MathWindow is a simple window that allows the editing or displaying of 2D math. This window can be placed anywhere inside a PEG window class, but is normally put inside an AbstractMathWindow object. MathWindow does not support scrolling. Instead, the AbstractMathWindow class provides a frame in which the MathWindow can be posititioned inside.

The following is an example that creates a MathWindow and places some 2D math inside:

```
PegRect rr = {0,1,100,100};
MathWindow *m_math = new MathWindow(rr,1 ,0 ,false ,10 , 10);
m_math->SetMathObject(CPString_to_LinearMathObject("((1/2)^2)"));
AddR(m_math);
```



AbstractMathWindow

Class Name	Derived From	Styles	Signals
AbstractMathWindow	CPFrameWindow	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

AbstractMathWindow is an abstract class that provides scrolling or a frame for a MathWindow. MathWindows are usually placed inside one of the following AbstractMathWindows:

- SlidingMathWindow – Does not allow scrolling, but puts the MathWindow in a simple frame.
- TabArrowMathWindow - Allows scrolling using small arrow buttons placed in the window.
- ScrollableMathWindow - Allows scrolling using PegHScroll, and PegVScroll scrollbar classes

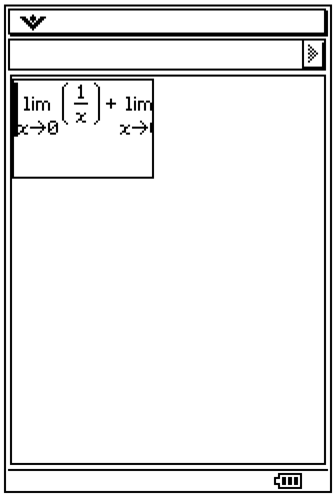
We will discuss each of these classes below.

SlidingMathWindow

Class Name	Derived From	Styles	Signals
SlidingMathWindow	AbstractMathWindow	FF_NONE FF_THIN	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

SlidingMathWindow is a simple frame around 2D math. It does not provide scrolling.

```
PegRect rr = {0,1,70,50};
CPString math = "lim(1/x,x,0) + lim(1/x,x,0)";
CLinearMathObject lmo = CPString_to_LinearMathObject(math);

SlidingMathWindow* math0 = new SlidingMathWindow(rr);
math0->GetMathWindow()->SetMathObject(lmo);
AddR(math0);
```

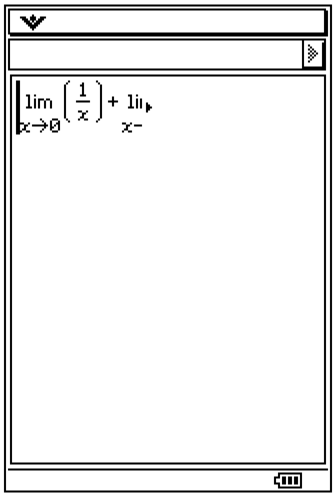


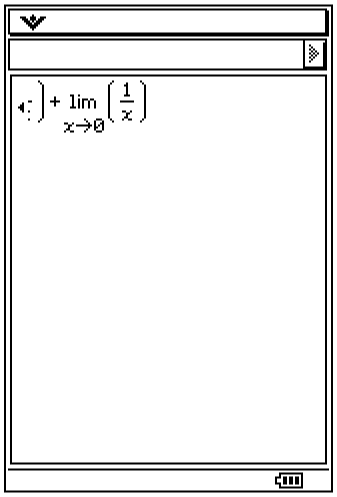
TabArrowMathWindow

Class Name	Derived From	Styles	Signals
TabArrowMathWindow	SlidingMathWindow	FF_NONE	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

A TabArrowMathWindow uses small arrows on the left and right sides of the 2D math to scroll. The creation of a TabArrowWindow is similar to a ScrollableMathWindow:

```
PegRect rr = {0,1,70,50};
CPString math = "lim(1/x,x,0) + lim(1/x,x,0)";
CLinearMathObject lmo = CPString_to_LinearMathObject(math);

TabArrowMathWindow* math0 = new TabArrowMathWindow(rr);
math0->GetMathWindow()->SetMathObject(lmo);
AddR(math0);
```





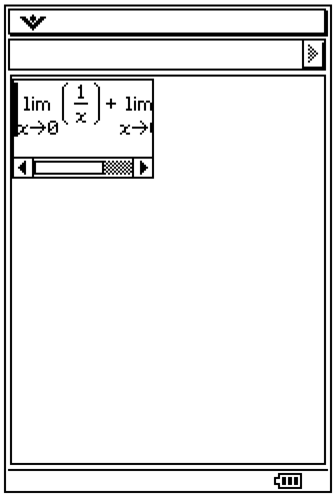
ScrollableMathWindow

Class Name	Derived From	Styles	Signals
ScrollableMathWindow	PegWindow	FF_NONE FF_THIN	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

A ScrollableMathWindow contains a MathWindow and allows scrolling using the Peg horizontal and vertical scrollbars. An example of a MathWindow in a ScrollableMathWindow follows:

```
PegRect rr = {0,1,70,50};
CPString math = "lim(1/x,x,0) + lim(1/x,x,0)";
CLinearMathObject lmo = CPString_to_LinearMathObject(math);

ScrollableMathWindow* math0 = new ScrollableMathWindow(rr);
math0->SetScrollMode(WSM_AUTOSCROLL);
math0->GetMathWindow()->SetMathObject(lmo);
AddR(math0);
```

This produces 2D math in a frame with the PEG scrollbars:

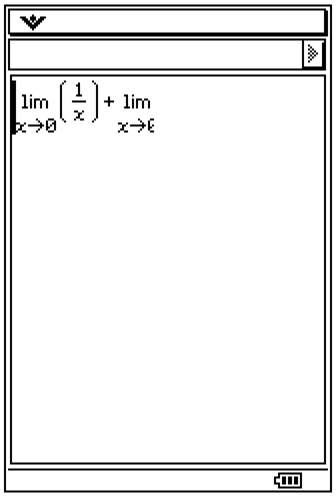


TextMathWindow

Class Name	Derived From	Styles	Signals
TextMathWindow	MathWindow	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

TextMathWindow is a MathWindow derived class that is used to display 2D math. The window does not allow user input or editing.

```
PegRect rr = {0,1,70,50};
CPString math = "lim(1/x,x,0) + lim(1/x,x,0)";
CLinearMathObject lmo = CPString_to_LinearMathObject(math);

TextMathWindow * m = new TextMathWindow (rr);
m->SetMathObject(lmo);

AddR(m);
```



Creating UI in a CPModuleWindow

Most, if not all, of the applications that you build will extend the CPModuleWindow to create your application's main window. This is because the CPModuleWindow makes it easy to create menus, toolbars and a status bar. In this section we will discuss what needs to be done to create these user interface controls in a CPModuleWindow.

Menus

Struct Name	Style	Signals
PegMenuDescriptionML	BF_SEPARATOR BF_CHECKABLE BF_CHECKED BF_DOTABLE BF_DOTTED	N/A

To create a menu we must first define a menu description. This description will identify what items will appear in each drop down menu. The first descriptor we define holds the names of the main menu headers that will appear across the top of the screen:

```
PegMenuDescriptionML MainMenu[] =
{
      {“Menu2”, CMN_NO_ID, 0, AF_ENABLED, SubMenu1 },
      {“Menu1”, CMN_NO_ID, 0, AF_ENABLED, SubMenu2 },
      {“”, CMN_NO_ID, 0, 0, 0}
};
```

You may notice right away that the order of the menu seems backwards – that Menu2 is listed before Menu1. This is the way that PEG is designed. When the menus appear on the ClassPad, Menu1 will be the left most menu. It is also required that the last entry in a PegMenuDescriptorML be a blank menu item.

Now let’s take a closer look at the parameters in each PegMenuDescriptionML. The first two parameters decide the text that will appear as the title of the menu. If the first value is defined, then the second value should be CMN_NO_ID. However, if the first value is NULL, then the second value must be a valid ID into a language database (see the section *Multiple Language Support in the ClassPad*).

In the above example, the values “Menu1” and “Menu2” are hard coded into the menu, so there is no need for an ID into a language array. This means that regardless of the current language of the ClassPad these menus will always have the values “Menu1” and “Menu2”. To make a menu that allows for multiple languages you would define your menus as:

```
static PegMenuDescriptionML MainMenu[] =
{
      { NULL, MENU_2, 0, AF_ENABLED, SubMenu1 },
      { NULL, MENU_1, 0, AF_ENABLED, SubMenu2 },
      { NULL, CMN_NO_ID, 0, 0, NULL }
};
```

In this case MENU_1 and MENU_2 would have to be defined in a language enumeration that corresponds to an entry in a language array.

The third parameter of the descriptor is the object ID of the menu. This ID is what is used to create a signal. Since this is the top-level menu, we can leave these IDs as 0 to prevent a signal from being sent. The fourth ID is a style flag, and the fifth parameter is the name of the sub menu that will be opened by clicking on this menu.

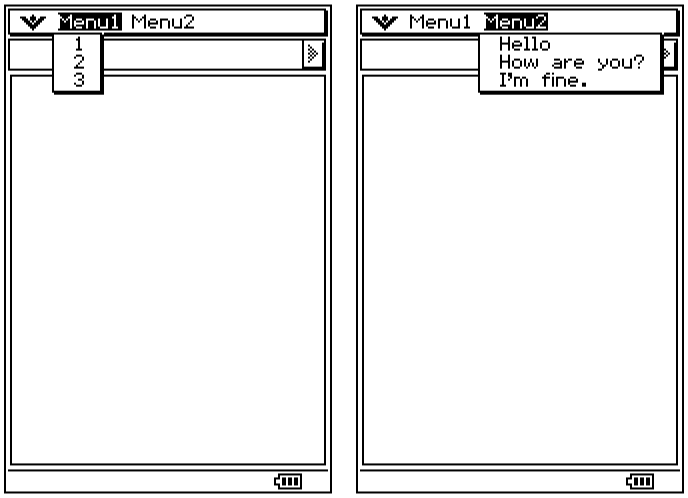
Submenus are created the same way as main menus. When creating a submenu, be sure to give it a signal ID or else you will not be able to respond to a user selection.

```
PegMenuDescriptionML SubMenu1[] = {
      {“I’m fine.”,     CMN_NO_ID, SUB1_3, AF_ENABLED, NULL },
      {“How are you?”,  CMN_NO_ID, SUB1_2, AF_ENABLED, NULL },
      {“Hello”,         CMN_NO_ID, SUB1_1, AF_ENABLED, NULL },
      {“”, CMN_NO_ID, 0, 0, 0}
};

PegMenuDescriptionML SubMenu2[] = {
      {“3”, CMN_NO_ID, SUB2_3, AF_ENABLED, NULL },
      {“2”, CMN_NO_ID, SUB2_2, AF_ENABLED, NULL },
      {“1”, CMN_NO_ID, SUB2_1, AF_ENABLED, NULL },
      {“”, CMN_NO_ID, 0, 0, 0}
};
```

Once you have created all of your menus and submenus, you must override the virtual function **GetMenuDescriptionML()** in your module. This function simply returns a pointer to your main menu descriptor:

```
PegMenuDescriptionML* YourModuleWindow::GetMenuDescriptionML()
{
      return MainMenu;
}
```

Once you have done that, you will have menus! Here is what the menus created above look like on the ClassPad:



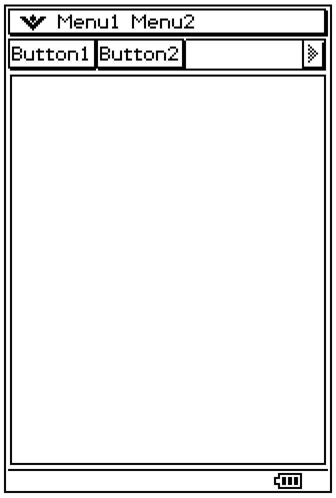
Toolbars

To add buttons to the toolbar you must override CPModuleWindow's virtual function **AddUI().**

Here is a simple toolbar example with two text buttons:

```
void YourWindow::AddUI()
{
      PegTextButton* b = new PegTextButton(1,1, "Button1", BUTTON1_ID,
                                           AF_ENABLED|TT_COPY);
      m_ui->AddToolbarButton(b);

      PegTextButton* b2 = new PegTextButton(35,1, "Button2",
                              BUTTON2_ID, AF_ENABLED|TT_COPY);
      m_ui->AddToolbarButton(b2);
}
```

Which creates the following on the ClassPad:



As shown in the previous section dealing with windows, there is a CPTabbedWindow that you can use to give your application a toolbar that is 2 times as long. This is also done inside of the **AddUI()** function.

Toolbars don't have to be text buttons. They can hold any PegThing derived object, including PegBitmapButton, CPDropDownButton or PegEditBox.

Status Bar

CPModuleWindow has a protected member of type PegStatusBar* that refers to the status bar at the bottom of the screen. You can gain access to this variable by using the **GetStatusBar()** function.

Once you have the status bar, all you have to do to add to text to it is call the **SetTextField()** function:

```
virtual void SetTextField (WORD wId, const PEGCHAR *Text);
```

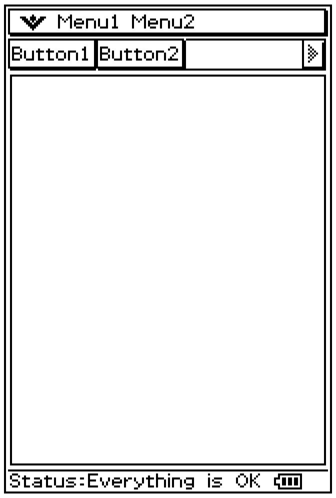
The first parameter will always be 1, and the second is the text string you wish to display.

You can make a function that controls setting the status bar in your module. For example, this function would set the status bar to whatever text is passed as a parameter:

```
void YourModuleWindow::SetStatusBar(PEGCHAR* message)
{
      // Get a pointer to the status bar
      PegStatusBar* bar = GetStatusBar();

      // Set the text
      bar->SetTextField(1, message);
}
```

Usually you will want to change the status bar after some event or message has occurred. If this is the case, in your overridden **Message()** function call the **SetStatusBar()** function after you have received the message that you wish to respond to. For example, the following **Message()** function calls **SetStatusBar()** with the message “Status: Everything is OK” on a PM_SHOW message:

```
SIGNED YourModuleWindow::Message(const PegMessage &Mesg)
{
      switch(Mesg.wType)
      {
            case PM_SHOW:
                  CPModuleWindow::Message(Mesg);
                  SetStatusBar("Status: Everything is OK");
                  break;
:
:
```



Buttons

PEG and the SDK provide several types of buttons that you can add to your application. We will now go through each one providing examples on how to create the button, and a screenshot of what the button looks like on the ClassPad.

Note: In most of these cases there is more than one constructor for each class. Refer to the ClassPad 300 SDK Reference Guide to see details on all available constructors.

PegButton

Class Name	Derived From	Styles	Signals
PegButton	PegThing	FF_NONE FF_THIN FF_THICK BF_REPEAT BF_DOWNACTION	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_CLICKED

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 SIGNALS. You would not normally create an instance of PegButton in your application, however PegButton is very useful as a base class for your own custom button styles.

PegTextButton

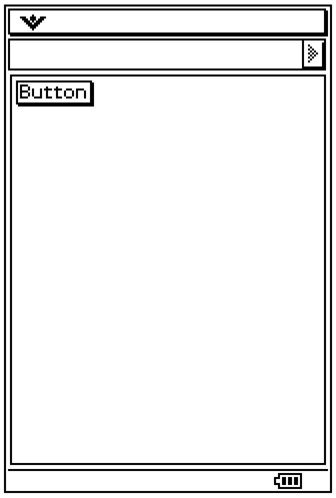
Class Name	Derived From	Styles	Signals
PegTextButton	PegButton	AF_ENABLED FF_NONE FF_THIN BF_REPEAT BF_DOWNACTION TT_COPY TJ_RIGHT TJ_LEFT TJ_CENTER	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_CLICKED

PegTextButton is simply a button with user-defined text. The text string displayed on the button face is vertically centered over button client area, and may be horizontally justified in different ways using the text justification style flags.

A PegButton sends the signal PSF_CLICKED when it is clicked.

The following is an example of creating a PegTextButton. The first two parameters give the left and top justification respectively.

```
PegTextButton *button = new PegTextButton(2, 2, "Button");
AddR(button);
```



PegBitmapButton

Class Name	Derived From	Styles	Signals
PegBitmapButton	PegButton	AF_ENABLED	PSF_SIZED

		BF_REPEAT BF_DOWNACTION	PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_CLICKED

PEG also allows for a button to display an image instead of text. These buttons support the standard border frames, but the text style flags do no apply. A PegBitmapButton also sends the signal PSF_CLICKED.

To create a PegBitmapButton you must first create an image and define a PegBitmap that represents this image. Creating a PegBitmap for a button is simple with the help of the Bitmap Converter tool, accessible under the Tools menu.

First, create a bitmap in your favorite image-editing program. When you save the file, **make sure that the image is monochrome.** Since the ClassPad is monochrome, only monochrome bitmaps will work. In this example, we will use the following bitmap named smile.bmp:

Next, open the BMP Converter tool. Browse to your bitmap image and type in the name of the output C++ file you wish to create. The C++ file will contain the byte data for the bitmap and create a PegBitmap. Press the convert button and add the output file to your add-in project. Open the file and you will see that a PegBitmap called gbsmileBitmap was created.

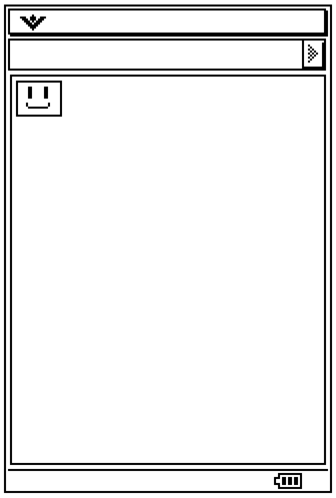
Finally, go back to the window class where you would like to add this button. At the top of the file declare an extern to the name of your bitmap:

```
extern PegBitmap gbsmileBitmap;
```

Then, create your PegBitmapButton like this:

```
PegBitmapButton *button = new PegBitmapButton(2, 2, & gbsmileBitmap);
AddR(button);
```

Here is the result on the ClassPad:

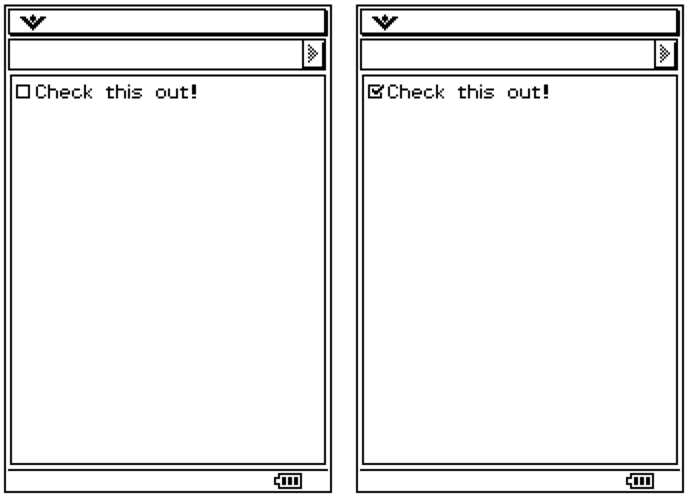


PegCheckBox

Class Name	Derived From	Styles	Signals
PegCheckBox	PegButton	BF_REPEAT BF_DOWNACTION BF_SELECTED AF_ENABLED	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_CHECK_ON PSF_CHECK_OFF

PEG also supports the creation of text-labeled checkboxes. Creating a checkbox is very similar to creating a button:

```
PegCheckBox *box = new PegCheckBox(2, 2, "Check this out!");
AddR(box);
```



A checkbox sends the signals PSF_CHECK_ON when selected, and PSF_CHECK_OFF when de-selected. Checkboxes support the BF_SELECTED and AF_ENABLED styles.

PegRadioButton

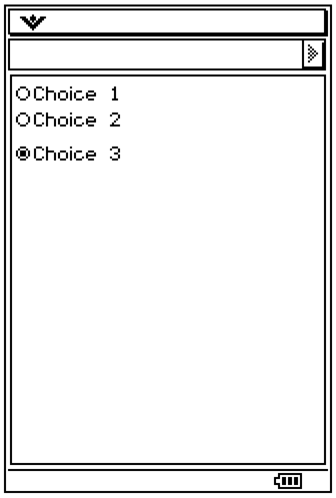
Class Name	Derived From	Styles	Signals
PegRadioButton	PegButton/ PegTextThing	BF_REPEAT BF_DOWNACTION BF_SELECTED AF_ENABLED	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_DOT_ON PSF_DOT_OFF

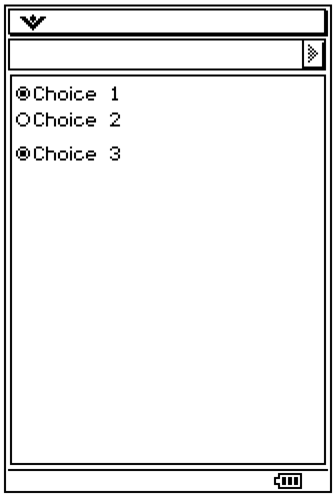
PegRadioButton provides a mutually exclusive selection indicator. When a PegRadioButton is selected by the user, it finds all sibling radio buttons and de-selects them. Therefore, in order to allow more than one radio-button to be selected on a single window or dialog you must group the buttons into separate containers or parents. Placing a radio button in a transparent PegThing is one way to accomplish this.

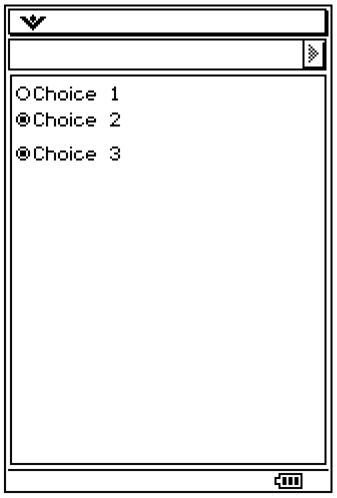
For example, look at the following code:

```
PegRadioButton *b1 = new PegRadioButton(2, 2, "Choice 1");
AddR(b1);

PegRadioButton  *b2 = new PegRadioButton(2, 15, "Choice 2");
AddR(b2);

PegRect r;
r = mClient;
PegThing *container = new PegThing(r);
PegRadioButton *b3 = new PegRadioButton(2, 32, "Choice 3");
container->Add(b3);

AddR(container);
```

b1 and b2 will be mutually exclusive because they are both children of this class. b3, on the other hand, has been placed in a different container, and has that container as its parent. Therefore, its selection will not have any effect on b1 or b2:







PegRadioButtons support the style flags AF_ENABLED and BF_SELECTED. They send the PSF_DOT_ON and PSF_DOT_OFF signals.

CPDropDownButton

Class Name	Derived From	Styles	Signals
CPDropDownButton	PegBitmapButton	BF_REPEAT BF_DOWNACTION BF_SELECTED AF_ENABLED	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_CLICKED

CPDropDownButtons are used by many applications on the ClassPad. They allow a user to select an item from a list of bitmap buttons. For example, the tool selection dropdown in the Geometry toolbar is a CPDropDownButton.

Much like the menus that we looked at before, CPDropDownButtons have a descriptor to define what items it will include. Here is an example of a descriptor:

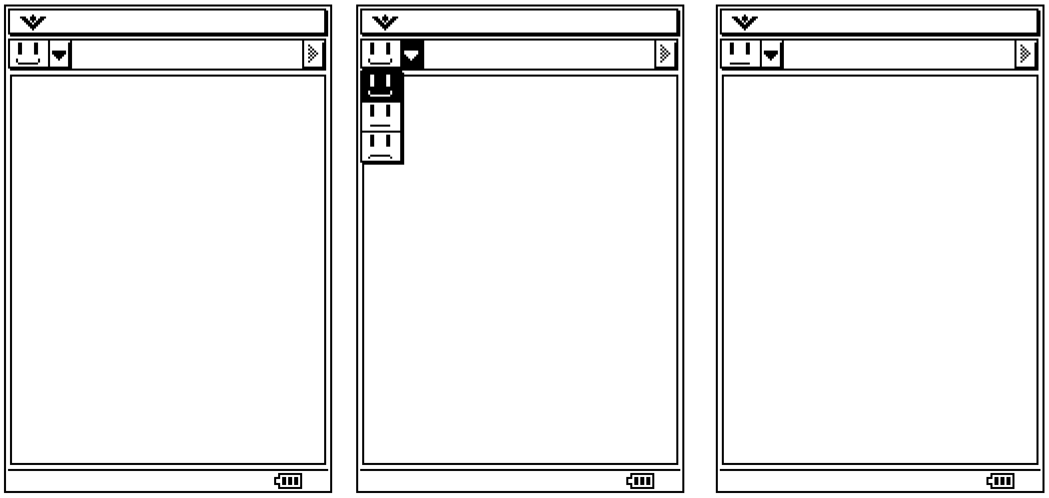
```
struct CPMultiButtonDescription buttons[] =
{
      {&gbsmileBitmap,        SMILE_ID},
      {&gbcontentBitmap,      CONTENT_ID},
      {&gbsadBitmap,          SAD_ID},
      {NULL,                  NULL},
};
```

Each entry in a CPMultiButtonDescriptor defines a button that will be in the drop down list. Each button must define the PegBitmap that it will display and its Object ID. The last entry in the list is a pair of NULLs.

Here is an example of creating a CPDropDownButton in the AddUI function of a module window with the buttons[] descriptor:

```
void YOURWINDOW::AddUI()
{
      PegRect r = GetToolbarButtonRect();
      CPDropDownButton *button = new CPDropDownButton(r, buttons);
      m_ui->AddToolbarButton(button);
}
```

The result is a dropdown button with three bitmap buttons. Selecting one button makes it visible and closes the dropdown.



CPMultiButton

Class Name	Derived From	Styles	Signals
CPMultiButton	PegBitmapButton	BF_REPEAT BF_DOWNACTION BF_SELECTED AF_ENABLED	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_CLICKED

A CPMultiButton is very similar to a CPDropDownButton. With a CPMultiButton, the bitmaps are cycled through instead of being chosen from a drop down. The bold button in eActivity is an example of a CPMultiButton.

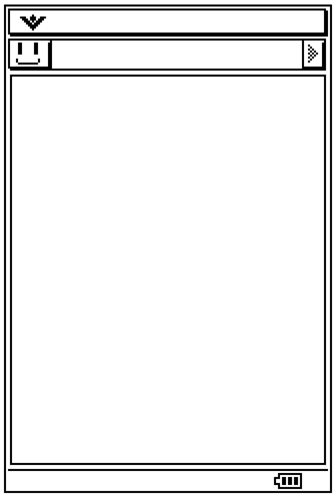
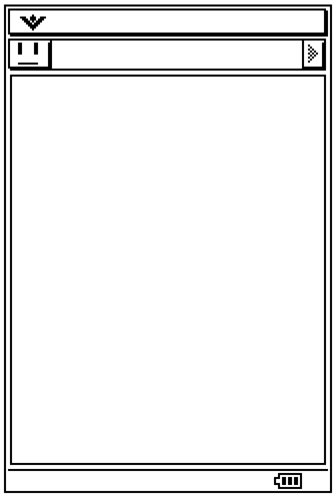
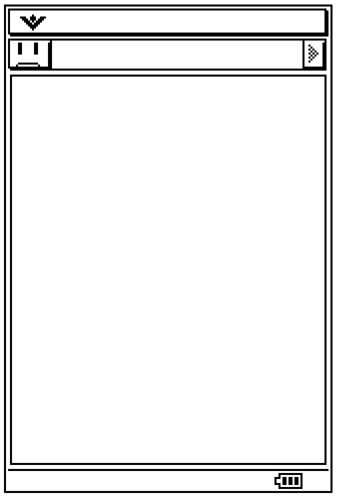
To create a CPMultiButton, you first make a CPMultiButtonDescription:

```
struct CPMultiButtonDescription buttons[] =
{
      {&gbsmileBitmap,   SMILE_ID},
      {&gbcontentBitmap, CONTENT_ID},
      {&gbsadBitmap,     SAD_ID},
      {NULL,             NULL},
};
```

This button is also mostly used in toolbars, so we will add it in the AddUI function:

```
void YOURWINDOW::AddUI()
{
      PegRect r = GetToolbarButtonRect();
      CPMultiButton *button = new CPMultiButton(r, buttons);
      m_ui->AddToolbarButton(button);
}
```

Each of the following screenshots is taken after the button was clicked. Notice that the images are being cycled through on each click:

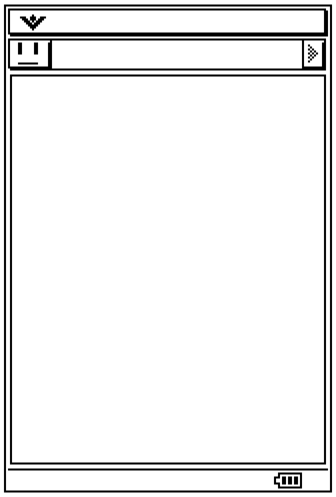
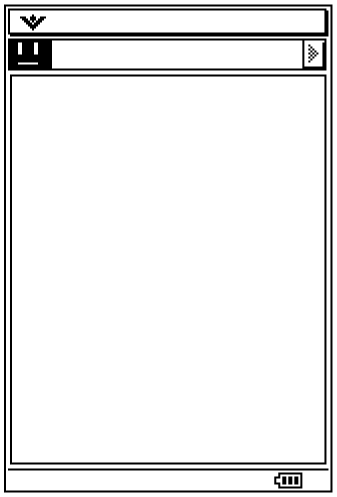
CPToggleButton

Class Name	Derived From	Styles	Signals
CPDropDownButton	PegBitmapButton	BF_REPEAT BF_DOWNACTION BF_SELECTED AF_ENABLED	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_CLICKED

CPToggleButton implements a two state PegBitmapButton that can be selected or unselected. When selected, the PegBitmap is inversed. Here is the code to create a simple example:

```
void MCSWindow::AddUI()
{
      PegRect r = GetToolbarButtonRect();
      CPToggleButton *button = new CPToggleButton(r, &gbcontentBitmap);
      m_ui->AddToolbarButton(button);
}
```

The first image below is when the button has not been clicked. The second is after a click and the image has been inverted.

Text Controls

PEG also provides several options for displaying text to the user as well as retrieving text input from the user. This section will show you how to create each of these text controls.

Note: In most of these cases there is more than one constructor for each object. Refer to the ClassPad 300 SDK Reference Guide to see details on all available constructors.

PegPrompt

Class Name	Derived From	Styles	Signals
PegPrompt	PegThing/ PegTextThing	FF_NONE FF_THIN TJ_RIGHT TJ_LEFT TJ_CENTER TT_COPY AF_TRANSPARENT AF_ENABLED	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_CLICKED

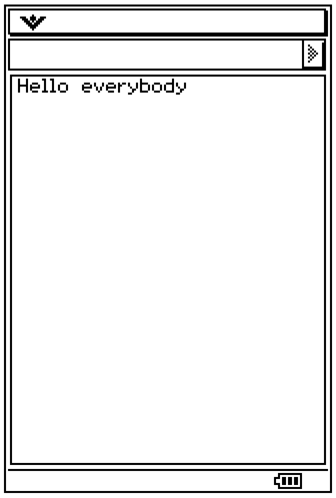
PegPrompt is a text display object. PegPrompt can be drawn with several different border styles, and can be updated dynamically for interactive updates or real-time information display. PegPrompt does not support user editing.

PegPrompt will by default 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.

The following code demonstrates how to create a PegPrompt:

```
      PegPrompt *pp = new PegPrompt(2, 0, "Hello everybody");
      AddR(pp);
```

Which simply adds the given text to your window:


Hello everybody

Be aware that if you do not pass in a static string to a PegPrompt, you should set the style flag TT_COPY. This causes PegPrompt to keep a copy of the string you pass in, so even if the string becomes invalid the PegPrompt will still have its own copy.

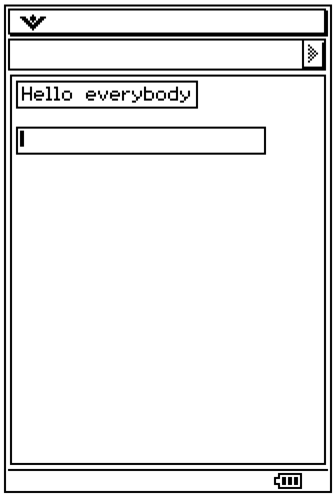
PegString

Class Name	Derived From	Styles	Signals
PegString	PegThing/ PegTextThing	FF_NONE FF_THIN TT_COPY AF_TRANSPARENT AF_ENABLED EF_EDIT	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_TEXT_SELECT PSF_TEXT_EDIT PSF_TEXT_EDITDONE

PegString is a user-editable graphical string object. In addition to the common signals defined by PegThing, PegString also supports the signals listed above.

Here is an example of how to create a PegString:

```
PegString *p = new PegString(2, 2, "Hello everybody");
AddR(p);

p = new PegString(2, 25, 125);
AddR(p);
```

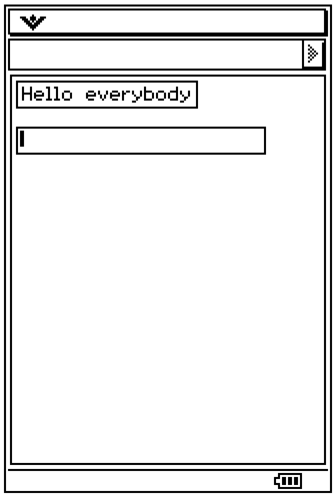


CPPegString

Class Name	Derived From	Styles	Signals
CPPegString	PegString/ CPUndoThing	FF_NONE FF_THIN TT_COPY AF_TRANSPARENT AF_ENABLED EF_EDIT	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_TEXT_SELECT PSF_TEXT_EDIT PSF_TEXT_EDITDONE

A CPPegString is a subclass of PegString. It has advanced features not available in PegString such as drag and drop, cut and paste and undo. The constructor to create a CPPegString can take the exact same parameters as our last example:

```
PegRect r = mReal;
CPPegString *p = new CPPegString(2, 2, "Hello everybody");
AddR(p);

p = new CPPegString(2, 25, 125);
AddR(p);
```



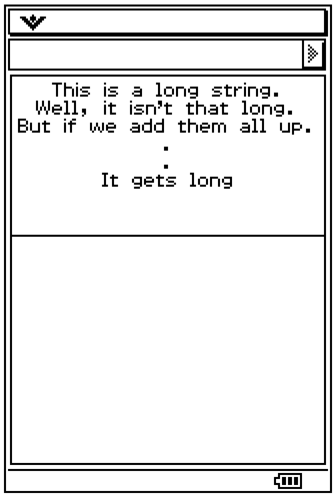
PegTextBox

Class Name	Derived From	Styles	Signals
PegTextBox	PegWindow/ PegTextThing	FF_NONE FF_THIN FF_THICK EF_WRAP TT_COPY TJ_RIGHT TJ_LEFT TJ_CENTER	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

PegTextBox is a multi-line text display control that does not support editing. By default, PegTextBox left-justifies the displayed text. Center-justification is also supported. Lines of text that are too long to fit in the client width of the textbox 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 whitespace, 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 which fits within the client width area.

Here is an example of a centered text box with some default text inside:

```
PegRect r = mReal;
r.wBottom = r.wBottom/2;

PegTextBox *p = new PegTextBox(r, 0, FF_RECESSED|EF_WRAP|TJ_CENTER,
      "This is a long string.\nWell, it isn't that long.\nBut if we add
      them all up.\n.\n.\nIt gets long");

Add(p);
```


This is a long string.
Well, it isn't that long.
But if we add them all up.
.
.
It gets long

If a string is long enough to require scroll bars, they can be added to the textbox by calling:

```
p->SetScrollMode(WSM_AUTOSCROLL);
```

If you would like the textbox to have scroll bars, make sure that you create the textbox with the style flag EF_EDIT. While this will not allow a user to edit the text inside the textbox, but it will allow the user to scroll the text.

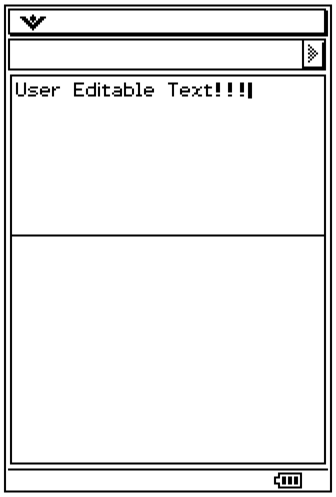
PegEditBox

Class Name	Derived From	Styles	Signals
PegEditBox	PegTextBox	FF_NONE FF_THIN FF_THICK EF_WRAP TT_COPY TJ_RIGHT TJ_LEFT TJ_CENTER	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_TEXT_SELECT PSF_TEXT_EDIT PSF_TEXT_EDITDONE

PegEditBox is a multi-line text display control that allows full user editing via pen and keyboard. A PegEditBox cannot be center justified.

Here is an example of how to create a PegEditBox:

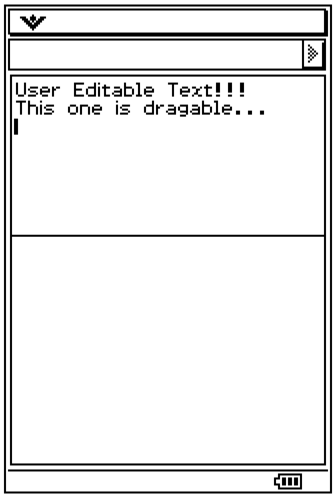
```
      PegRect r = mReal;
      r.wBottom = r.wBottom/2;

      PegEditBox *p = new PegEditBox(r);
      Add(p);
```


User Editable Text!!!

CPEditBox

Class Name	Derived From	Styles	Signals
CPEditBox	PegEditBox/ CPUndoThing	FF_NONE FF_THIN FF_THICK EF_WRAP TT_COPY TJ_RIGHT TJ_LEFT TJ_CENTER	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_TEXT_SELECT PSF_TEXT_EDIT PSF_TEXT_EDITDONE

CPEditBox is a subclass of PegEditBox that supports drag and drop, copy and paste and undo. The constructor takes the same parameters as a PegEditBox.


User Editable Text!!!
This one is dragable...

Other User Interface Controls

PEG has several other user interface controls that can be used to display information to the user or gather information from the user.

PegList

Class Name	Derived From	Styles	Signals
PegList	PegWindow	FF_NONE FF_THIN FF_THICK LS_WRAP_SELECT	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_LIST_SELECT

PegList is a container class that serves as a base class for PegVertList, PegHorzList, and PegComboBox. 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 application, but instead use PegVertList, PegHorzList or PegComboBox.

PegList is a subclass of PegWindow, and enables scrolling in the same way; via the **SetScrollMode()** function.

The three PegList derived controls that we are going to look at all accept PegPrompts as items in the list.

PegVertList / PegHorzList

Class Name	Derived From	Styles	Signals
PegVertList/ PegHorzList	PegList	FF_NONE FF_THIN FF_THICK LS_WRAP_SELECT	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_LIST_SELECT

PegVertLists and PegHorzLists create vertical and horizontal lists respectively. You can use the functions **Add()** or **AddToEnd()** to add PegPrompts to your list. Both PegVertList and PegHorzList manage the position and size of the items in the list, so you do not need to concern yourself about position when creating your PegPrompts.

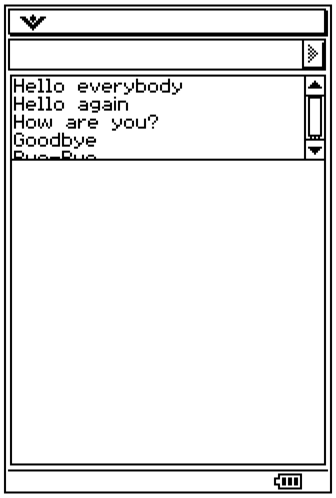
When creating the PegPrompts to insert into a PegList, you should make sure that the AF_ENABLED flag is set. Without it, you will not be able to select the PegPrompt in your list.

Let’s look at how a PegVertList is created:

```
PegRect r = mReal;
r.wBottom = r.wBottom/3;
PegVertList *list = new PegVertList(r);
```

```
// Set Scrolling just like a PegWindow
list->SetScrollMode(WSM_AUTOVSCROLL);

PegPrompt *pp;
pp = new PegPrompt(0, 0, "Hello everybody", 0,
                        FF_NONE|TJ_LEFT|AF_ENABLED|TT_COPY);
list->AddToEnd(pp);

pp = new PegPrompt(0, 0, "Hello again", 0,
                        FF_NONE|TJ_LEFT|AF_ENABLED|TT_COPY);
list->AddToEnd(pp);

pp = new PegPrompt(0, 0, "How are you?", 0,
                        FF_NONE|TJ_LEFT|AF_ENABLED|TT_COPY);
list->AddToEnd(pp);

pp = new PegPrompt(0, 0, "Goodbye", 0,
                        FF_NONE|TJ_LEFT|AF_ENABLED|TT_COPY);
list->AddToEnd(pp);

pp = new PegPrompt(0, 0, "Bye-Bye", 0,
                        FF_NONE|TJ_LEFT|AF_ENABLED|TT_COPY);
list->AddToEnd(pp);
```

Notice that the PegVertList's scroll mode is set in the same way that a PegWindow's is set. We mentioned before the list manages the position of the PegPrompts. In this example even though all PegPrompts are created at the same location, when they are placed in a the list they will be arranged correctly:



On selection, the list sends PSF_LIST_SELECT signals to the parent object.

PegComboBox

Class Name	Derived From	Styles	Signals
PegComboBox	PegVList	FF_NONE FF_THIN FF_THICK	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST

			PSF_KEY_RECEIVED PSF_LIST_SELECT

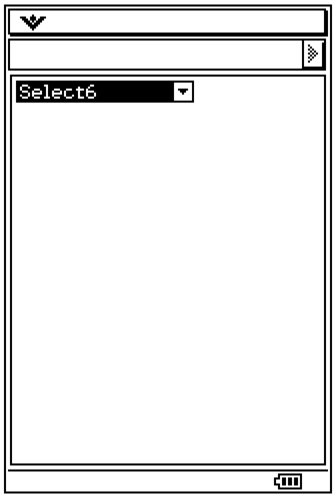
PegComboBox is similar to (and derived from) PegVList. PegComboBox adds the concept of "Opening and Closing", which can conserve space when a large number of items are added to the combo box. A drop-down arrow is provided to open the combo box. The box closes when an item is selected or the combo box loses focus.

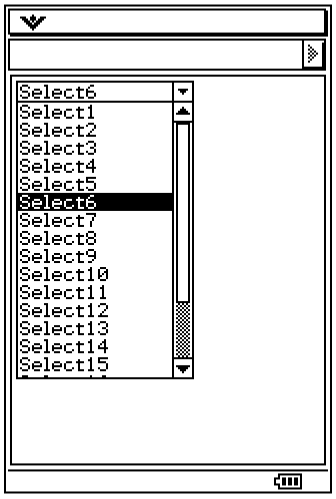
PegComboBox will send the signal PSF_LIST_SELECT to the parent window if the PegComboBox has a non-zero ID value and the selected child also has a non-zero ID value.

The following example creates a PegComboBox and populates it:

```
PegRect ListRect;
ListRect.Set(2, 2, 90, 150);
PegComboBox *pList = new PegComboBox(ListRect);

for (int iLoop = 20; iLoop > 0; iLoop--)
{
    CP_CHAR cTemp[20];
    CP_StringCopy(cTemp, (CP_CHAR*)"Select ");

    CP_CHAR nTemp[10];
    CP_IntToString(iLoop, nTemp);

    CP_StringCat(cTemp, nTemp);
    pList->Add(new PegPrompt(0, 0, (PEGCHAR*)cTemp, iLoop,
        FF_NONE|TJ_LEFT|AF_ENABLED|TT_COPY));
}
pList->SetScrollMode(WSM_VSCROLL);
pList->SetSelected(5);
AddR(pList);
```

The result of this code is:





PegSpinButton

Class Name	Derived From	Styles	Signals
PegSpinButton	PegThing	SB_VERTICAL	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED PSF_SPIN_MORE PSF_SPIN_LESS

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.

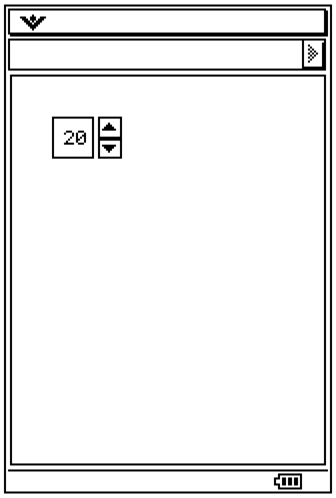
There are two forms of PegSpinButton. The first form is created when the spin button has a 'buddy' object. A buddy object is a PegTextThing derived object that is automatically updated as the spin button is manipulated by the end user. The second form of PegSpinButton has no buddy object, and therefore 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 will first convert the buddy object string to an integer, then increment or decrement the integer value as required, and then convert 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.

The following example creates a PegPrompt that is used as the buddy for the PegSpinButton. The PegSpinButton has a min of 20 and a max of 80 with an increment of 5.

```
PegRect ChildRect;
ChildRect.Set(20, 20, 100, 40);
PegPrompt *pPrompt = new PegPrompt(ChildRect, "20", 0,
       FF_RECESSED|TJ_RIGHT|TT_COPY);
AddR(pPrompt);

// set the spin button position to the right of the prompt:
ChildRect.wLeft = pPrompt->mReal.wRight + 1;
ChildRect.wRight = ChildRect.wLeft + PEG_SCROLL_WIDTH;

//Create the SpinButton with pPrompt as the buddy.  20 as the min, 80
//as the max and 5 as the increment value
PegSpinButton *pSpin = new PegSpinButton(ChildRect,
        pPrompt, 20, 80, 5, SB_VERTICAL);
AddR(pSpin);
```



PegProgressBar

Class Name	Derived From	Styles	Signals
PegProgressButton	PegThing	FF_NONE FF_THIN PS_SHOW_VAL PS_RECESSED PS_LED PS_VERTICAL PS_PERCENT	PSF_SIZED PSF_FOCUS_RECEIVED PSF_FOCUS_LOST PSF_KEY_RECEIVED

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 SIGNED data type, however it is most common to display a value that is a percentage of the completion status.

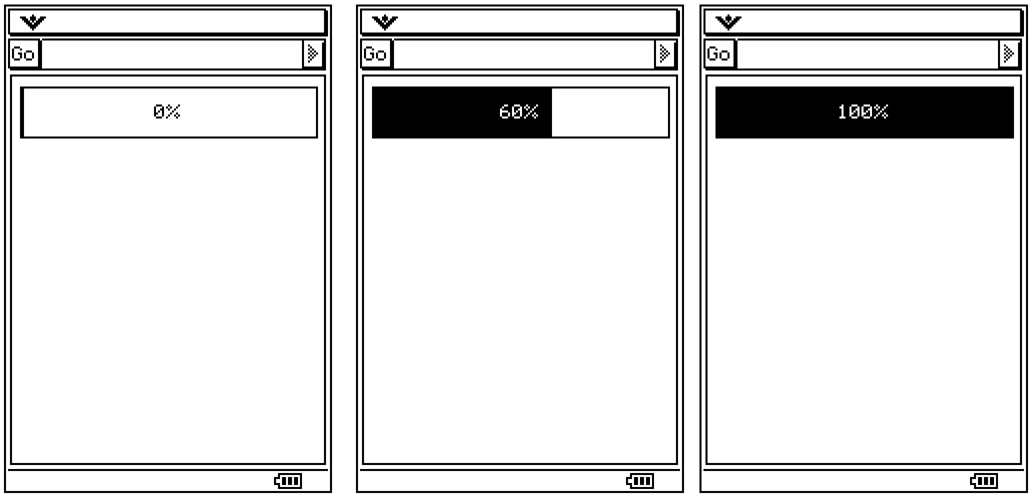
The style, range, and initial value of a PegProgressBar object are passed to the object constructor. As the operation being monitored progresses, the application software calls the **Update()** member function to change the displayed completion value.

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.

The following example creates a PegProgressBar which gets updated during the **Task()** function:

```
void ProgressWindow::CreateProgressBar()
{
    PegRect r = mClient;
    r.wLeft += 2;
    r.wRight -= 5;
    r.wBottom = 30;
    r.wTop = 5;
    bar = new PegProgressBar(r, FF_THIN|PS_SHOW_VAL|PS_PERCENT);
    AddR(bar);
}

void ProgressWindow::Task()
{
    for(int i=1;i<10000001;i++)
    {
        for(int j=0;j<100;j++);

        if(i%100000 == 0)
        {
            bar->Update(i/100000);
        }
    }
}
```

In this example **Task()** could be called from a user event, like clicking on a button. Here is a screenshot where a toolbar button starts **Task():**



For more information, including a complete list of constructors and member functions for each control class, see the ClassPad 300 SDK Reference Guide.

Using Floating-Point Values with the ClassPad

The ClassPad does not support the use of double data types. Instead, the ClassPad has a native representation of doubles called BCD.

There are two types of BCDs: OBCD and CBCD. OBCD is used to represent real numbers, whereas CBCD is used to represent complex numbers. We will look at using OBCD numbers first and then look at CBCD numbers.

OBCD Data Structure

The structure of an OBCD is defined as:

```
typedef struct obcd_ {
      unsigned char mantissa[IM_CAL_INDIGIT];
      unsigned short exponential;
} OBCD_;

typedef union obcd {
      OBCD_ obcd1;
      unsigned long     dummy[3];
} OBCD;
```

The mantissa of a number is stored in obcd1.mantissa. The mantissa is 10 bytes long, with the least significant 2 bytes reserved for system use. The most significant nibble is reserved for a flag. There is also a 2 byte exponent that is stored in a short. The entire structure looks like this:

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	eS	e1	e2	e3
Flag	Mantissa															reserved				Exponent			

OBCD Flag

The most significant nibble of the mantissa is reserved for a flag. When this flag is non-zero it means there is a non-numeric value in the OBCD. The possible values for the flag and the meaning are described in the following table:

flag	Meaning
0	There is a decimal value in mantissa
4	Infinity if the exponent is 0x1000 or negative infinity is the exponent is 0x6000
8	The value is undefined
9	True
a	False
f	Error

Here are some examples of OBCD values that have the flag set to a value other than 0.

Positive Infinity:

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	eS	e1	e2	e3
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Negative Infinity:

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	eS	e1	e2	e3
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0	0	0

True:

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	eS	e1	e2	e3
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Overflow Error:

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	eS	e1	e2	e3
F	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	F

As you can see in the case of infinity or an error, you must inspect the low byte of the exponent for more information about the value. When dealing with infinity these bytes will tell you whether the infinity is positive or negative. We will discuss this in detail in the exponent portion of this section.

If the flag represents an error, the exponent byte tells you which error has occurred. The following is a list of all possible error values and the corresponding error:

Value in Exponent	Error
0x0000	Norm (Normal – no error)
0x0001	Acbreak
0x0002	Syntax ERROR (Syntax error)
0x0003	Undefined
0x0004	Memory ERROR (Memory error)
0x0005	Go ERROR (Jump error)
0x0006	Nesting ERROR (Nesting error)
0x0007	Stack ERROR
0x0008	Argument ERROR (Argument error)
0x0009	Dimension ERROR (Dimension error)
0x000a	Com ERROR (Send and receive error)
0x000b	Transmit ERROR (Transmission error)
0x000c	Receive ERROR (Reception error)
0x000d	Memory Full
0x000e	Undefined
0x000f	Overflow ERROR
0x0010	Domain ERROR (Input range error)
0x0011	Non-Real ERROR
0x0012	No Solution (There is no solution)
0x0013	Mismatch
0x0014	No Variable
0x0015	Not Found

0x0016	Application ERROR
0x0017	System ERROR
0x0018	Already Exists
0x0019	Complex Number In List
0x001a	Complex Number In Mat
0x001b	Can't Solve (There is no solution)
0x001c	Range ERROR
0x001d	Iteration ERROR
0x001e	Condition ERROR
0x001f	NULL

The Mantissa

The remaining space in the mantissa is used to hold the value of your number. Remember that the flag is actually part of the mantissa, and there are 2 bytes of reserved data at the end.

The view of the mantissa:

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
FLAG	Mantissa value															Reserved			

The value stored in the mantissa is the value of the number you want to store without any representation of a decimal point. This means that the numbers 1.75 and 175 both have a mantissa of 175. The distinction is in each number's exponent.

Be aware that the mantissa is left justified – meaning that the first significant digit always follows the flag. In other words your mantissa cannot have leading zeroes.

The Exponent

The exponent portion of an OBCD defines where the decimal point will be. The exponent is stored in 2 bytes as follows:

eS	e1	e2	e3
Sign Bit	Exponent Value		

The sign bit determines the sign of the mantissa and the exponent. Exponents have a range of –999 to 999.

All OBCDs calculate the exponent value as if there were a decimal point right after the most significant digit of the mantissa. That is, the values are stored in a form similar to scientific notation.

How you calculate the value of the OBCD's exponent depends on whether the value you want to store is positive or negative. If you have a positive number, add the value of the exponent to 1000 to get its value. If you have a negative number, add the value of the exponent to 6000 to get the exponent.

For example, let's say you want to store 1750. The value of the mantissa will be 1750. But remember we must calculate the exponent as if 1.750 is actually stored in the mantissa. Therefore the exponent needs to be 10^3 (because 1.75 * 10^3 = 1750). Since 1750 is positive the OBCD's exponent starts at 1000. We then add the exponent value (3) to get:
1000 + 3 = 1003
which means that our exponent has the value of 1003.

The OBCD representation of 1750

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	eS	e1	e2	e3
0	1	7	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	3

Now let's try –0.0065. The mantissa will be 65, so we need the exponent to be –10^-3 (6.5 * -10^-3 = -0.0065). Since our value is negative this time we need to add to 6000. So the value of the exponent is:
6000 + (-3) = 5997

The OBCD representation of –0.0065

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	eS	e1	e2	e3
0	6	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	9	9	7

Finally, let's look at the value 2.25. This will be stored in the mantissa as 225. The value we want for an exponent is 10^0 since 2.25 * 10^0 = 2.25. Again, our number is positive so we add to 1000: 1000 + 0 = 1000. So the representation of 2.25 is:

The OBCD representation of 2.25

eF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	eS	e1	e2	e3
0	2	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Here are some more examples that of exponents and their OBCD representation:

Exponent Value	eS	e1	e2	e3
0	0	0	0	0
10^-999	0	0	0	1
10^-1	0	9	9	9
10^0	1	0	0	0
10^1	1	0	0	1
-10^1	6	0	0	1
-10^-1	5	9	9	9

Setting the Value of an OBCD

To assign a value to an OBCD number there are several functions that you can use. The first three we will look at will look at can be used to set an OBCD to a whole number:

```
word Cal_setn_OBC(word wx, OBCD * x);
word Cal_shortto_OBC(short wx, OBCD * x);
word Cal_longto_OBC(long wx, OBCD * x);
```

These functions place a word (unsigned short), short, and a long into an OBCD respectively. Here is an example using Cal_setn_OBC:

```
OBCD x;
Cal_setn_OBC(5, &x);   //sets x to the value 5
```

Since this function takes an unsigned short we cannot set the value to a negative number. We would have to use the Cal_minus_OBC function as follows:

```
OBCD x;
Cal_setn_OBC(5, &x);   //sets x to the value 5
Cal_minus_OBC(&x);     //sets x to the value -5
```

We could also use the Cal_shortto_OBC or Cal_longto_OBC function to set a negative value:

```
OBCD x;
OBCD y;
Cal_shortto_OBC(-5, &x);   //sets x to the value -5
Cal_longto_OBC(-155, &y);    //sets y to the value -155
```

When dealing with whole numbers, you do not need to worry about exponents as long as you use these functions to set the values. The functions will set the exponent value automatically.

As we mentioned in the *OBCD Data Structure* section, to create a floating-point OBCD we must change the OBCD's exponent value. To do this, we will explicitly set the exponent to the appropriate hex value.

Let's walk through setting the exponent for the value 205.5. First use the function Cal_longto_OBC to set the OBCD to 2055:

```
OBCD x;
Cal_longto_OBC(2055, &x);
```

We want to set the exponent to 10^2 so the number becomes 2.055 * 10^2 = 205.5. Since 205.5 is positive, add the exponent value to 1000: 1000 + 2 = 1002.   So the exponential value is 1002. You can set the exponent with the following code:

```
x.obcd1.exponential=0x1002;
```

x now holds the value 205.5. To make the mantissa negative, we would have set the exponent to 6002 (6000 + 2 = 6002):

```
x.obcd1.exponential=0x6002;  // 2.055 * -10^2 = -205.5
```

Here are some more examples of setting exponent values:

```
OBCD x;
```

```
Cal_longto_OBC(523, &x);

x.obcd1.exponential = 0x1120; // changes the value to 5.23e120

x.obcd1.exponential = 0x6004; // changes the value to -52300

x.obcd1.exponential = 0x5996; // changes the value to -0.000523
```

You can also create an OBCD by explicitly setting all 20 bytes of the mantissa as well as the 2 bytes of the exponent. This is helpful when you need to define a constant OBCD. Remember that even though it looks like the mantissa is starting with a leading zero, this is actually the flag nibble. The first digit of the number is the second nibble and cannot be zero. Also the last 2 bytes of the mantissa should be set to 0x00 0x00.

```
const OBCD PI = {{{0x03, 0x14, 0x15, 0x92, 0x65, 0x35, 0x89, 0x79, 0x00,
0x00}, 0x1000}};

const R_sqrt2by2_INIT {{{0x07, 0x07, 0x10, 0x67, 0x81, 0x18, 0x65, 0x47,
0x00, 0x00}, 0x0999}}; // 0.7071067811865475244

const R_msqrt2by2_INIT {{{0x07, 0x07, 0x10, 0x67, 0x81, 0x18, 0x65,
0x47, 0x00, 0x00}, 0x5999}};  // -0.7071067811865475244
```

While the previous functions require you to work on the low level OBCD structure to set an OBCD value, the function **ExecuteToOBCD()** is much easier to use:

```
int ExecuteToOBCD(const PEGCHAR *str, OBCD &obcd, BOOL bErrCheck=TRUE);
```

This function takes a string representation of a floating-point value and sets the OBCD *obcd* to its value. For example, the following code sets x to 3.21e-5:

```
OBCD x;
ExecuteToOBCD(".0000321", x);
```

If you pass an invalid string representation of a float into the function, then the OBCD will be set to ERROR. For example:

```
OBCD x;
ExecuteToOBCD(".00003.21", x);  // x=OBCD representation of ERROR
```

When using this function you do not have to worry about setting the exponent explicitly; it gets set automatically by the function.

Finally, there are a few functions to set an OBCD to specific values such as infinity, pi or e. Some of these include:

```
void  Cal_setinfp_OBC (OBCD *x);    // Place infinity in x
void  Cal_setinfm_OBC (OBCD *x);    // Place -infinity in x
void  Cal_setundef_OBC (OBCD *x);   // Place undefined in x
void  Cal_settrue_OBC (OBCD *x);    // Place true in x
void  Cal_setfalse_OBC (OBCD *x);   // Place false in x
void  Cal_set05_OBC (OBCD *x);      // Place .5 in x
void  Cal_setwmax_OBC (OBCD *x);    //Place the max possible value in x
void  Cal_setwmin_OBC (OBCD *x);    //Place the min possible value in x
void  Cal_setpi_OBC (OBCD *x);      // Place pi in x
void  Cal_set2pi_OBC (OBCD *x);     // Place 2*pi in x
void  Cal_setpih_OBC (OBCD *x);     // Place pi/2 in x
void  Cal_setpiq_OBC (OBCD *x);     // Place pi/4 in x
void  Cal_sete_OBC (OBCD *x);       // Place e in x
void  Cal_setln10_OBC (OBCD *x);    // Place ln(10) in x
```

Performing Operations on OBCDs

The next step in using OBCDs is performing calculations with them. Here are a few common operations that take two operands:

```
word  Cal_adds_OBC (OBCD *x, OBCD *y);    // Adds x to y (x+y).
word  Cal_subs_OBC (OBCD *x, OBCD *y);    // Subtracts x from y (x-y).
word  Cal_muls_OBC (OBCD *x, OBCD *y);    // Multiplies x and y (x*y).
word  Cal_divs_OBC (OBCD *x, OBCD *y);    // Divides y into x (x/y).
word  Cal_sqrts_OBC (OBCD *x);            //Takes the square root of x.
word  Cal_add_OBC (OBCD *x, OBCD *y);     // Adds x to y (x+y).
word  Cal_sub_OBC (OBCD *x, OBCD *y);     // Subtracts x from y (x-y).
word  Cal_mul_OBC (OBCD *x, OBCD *y);     // Multiplies x and y (x*y).
word  Cal_div_OBC (OBCD *x, OBCD *y);     // Divides y into x (x/y).
word  Cal_pow_OBC (OBCD *x, OBCD *y);     // Raises x to y (x^y).
```

Notice that these functions return an error code, not the result of the operation. The result of the operation is stored in the first parameter, x. For example:

```
word error;
OBCD x, y;
Cal_setn_OBC(10, &x);   // set x = 10
Cal_setn_OBC(15, &y);   // set y = 15

error = Cal_adds_OBC(&x, &y); // set x = x + y
if(error != IM_CAL_NORM)
{
      // An Error Occurred!
}
```

When this code finishes, x has the value of 25. If the return value from the addition function is anything other than IM_CAL_NORM, then an error occurred. You can view all the return values in the ClassPad 300 SDK API Reference Guide under Math Functions->Calculation Error Codes.

There are also several functions that operate on single operands. Some of these include:

```
word  Cal_sqrt_OBC (OBCD *x);  // Takes the square root of x
word  Cal_log_OBC (OBCD *x);   // Takes the log of x (log(x))
word  Cal_log10_OBC (OBCD *x); // Takes the log base 10 of x
                               // (log_10(x))
word  Cal_sin_OBC (OBCD *x, word wdrg);  // Takes the sin of x
word  Cal_cos_OBC (OBCD *x, word wdrg);  // Takes the cos of x
word  Cal_tan_OBC (OBCD *x, word wdrg)  // Takes the tan of x
```

These functions also return an error code and place the result in x.

Converting OBCDs

There are also several functions that convert OBCDs into different data types. An OBCD can be converted to a short, long or string. The functions to convert an OBCD to a short or long are similar to the functions we've already seen:

```
word  Cal_toshort_OBC(OBCD *x, short *wx); //Convert an OBCD to a short
word  Cal_tolong_OBC(OBCD *x, long *wx);   //Convert an OBCD to a long.
```

Once again, be careful not to expect the conversion as the return value:

```
OBCD x;
long y;
word error_code;
Cal_setn_OBC(6, &x);                      // set x = 6
error_code = Cal_tolong_OBC(&x, &y);      // set y = x;
```

There are many different functions that can be used to convert an OBCD to a string. These functions differ by which form of the OBCD is placed in the string – normal mode, scientific notation, etc. We will look at four of these functions. To view all of the available functions, refer to the ClassPad 300 SDK API Reference Guide Strings->Functions to Convert OBCD/CBCD datatypes to strings.

```
//Changes the OBCD value to normal mode and places it in str.
word  CP_Norm_OBC (OBCD *x, CP_CHAR str[], short mode);

//Changes decimal portion of OBCD to fixed size and places it in str
word  CP_Fix_OBC (OBCD *x, CP_CHAR str[], short dig);

// Changes the OBCD value to scientific notation of precision dig and
// places it in str.
word  CP_Sci_OBC (OBCD *x, CP_CHAR str[], short dig);

// Change an OBCD object to a 15 digit string in normal mode.
word  CP_15digit_OBC (OBCD *x, CP_CHAR str[]);
```

Here is an example on how to convert an OBCD to a string in different forms:

```
OBCD x;
CP_CHAR buffer[15];
Cal_setn_OBC(1525, &x);                   // set x = 1525
x.obcd1.exponential = 0x1001;

CP_15digit_OBC(&x, buffer);               // puts 15.25 in buffer
CP_Norm_OBC(&x, buffer, IM_MODE_NORM1);   // puts 15.25 in buffer
CP_Fix_OBC(&x, buffer, 4);                // puts 15.2500 in buffer
CP_Sci_OBC(&x, buffer, 4);                // puts 1.525e1 in buffer
```

C++ Functions

It is important that you understand and can use the OBCD functions that we've covered up to this point. This is how doubles are stored and used natively on the ClassPad. There are, however, some functions available in C++ that are more human readable. You can view all of these functions in the ClassPad 300 SDK API Reference Guide under Math Fuctions->C++ Math Functions. Here is an example of how they work:

```
OBCD x, y;
Cal_longto_OCB(170, &x);

y = sin(x);
if(y == x)
{
      x = x * y;
}
else
{
      y = x + y;
}
```

As you can see, these are much more intuitive than the C functions. Remember that these functions will only work in C++; you cannot use them in C.

CBCD Data Structure

A CBCD is used to hold a complex number. Its structure is defined as:

```
typedef struct cbcd {
      OBCD  repart;
      OBCD  impart;
} CBCD;
```

As you can see from the structure, a CBCD is really just two OBCDs. One OBCD – repart – holds the real part of the CBCD and the second OBCD – impart – holds the imaginary part.

Setting the Value of a CBCD

Since a CBCD is just 2 OBCDs, setting the value of a CDCB is just like setting the value of 2 OBCDs. For example, if you wanted to create the value 2.5 + 3i you just have to create a CBCD with real part 2.5 and imaginary part 3. Here is the code to do that:

```
CBCD x;
Cal_longto_OBC(25, &x.repart);        // set x's real part to 25

// change the real part's exponent to 10^0
x.repart.obcd1.exponential = 0x1000;

Cal_longto_OBC(3, &x.impart);       // set x's imaginary part to 3
```

The result is the value 2.5+3i being stored in x.

Performing Operations on CBCDs

Most of the operations available to OBCDs are also available to CBCDs. A complete list can be found in the ClassPad 300 SDK API Reference Guide under Math Functions->Complex (CBCD) Math Functions. Just like CBCD functions, these functions do not return the result but a return code. Here is a simple example:

```
CBCD x, y;

Cal_longto_OBC(3, &x.repart);
Cal_longto_OBC(5, &x.impart);

Cal_longto_OBC(1, &y.repart);
Cal_longto_OBC(6, &y.impart);

if(Cal_add_CMP(&x, &y) != IM_CAL_NORM)
{
      // an error occurred!
}
```

Converting CBCDs

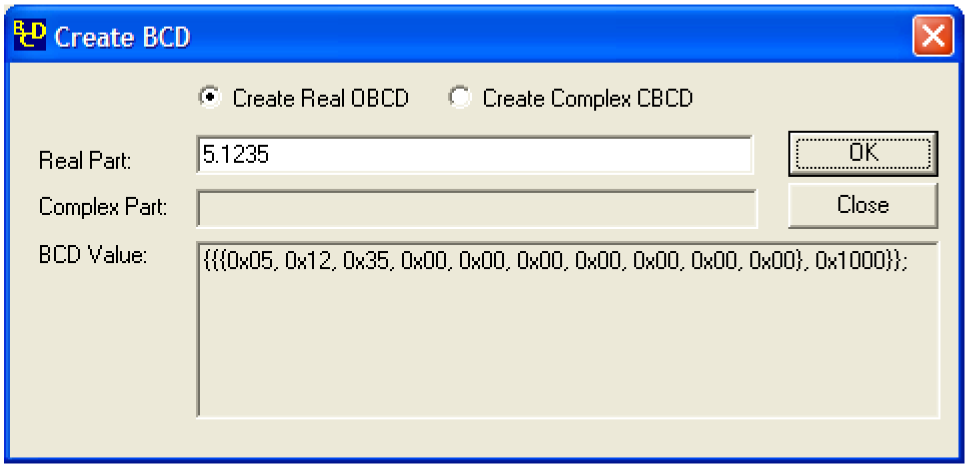
Since CBCDs are complex numbers, they cannot be converted to longs or shorts like OBCDs. You can, however, convert a CBCD to a string. The functions are very similar to the ones used to convert CBCDs. You can view them all in the ClassPad 300 SDK API Reference Guide under Strings->Functions to convert OBCD/CBCD data types to strings.

Here is a quick example of converting a CBCD to a string:

```
CBCD x;
CP_CHAR buffer[30];
Cal_longto_OBC(1525, &x.repart);    // set x = 1525
x.repart.obcd1.exponential = 0x1001;
Cal_longto_OBC(32, &x.impart);

CP_15digit_CMP(&x, buffer);   // puts 15.25+32i in the buffer
```

BCD Converter Tool

The BCD Converter Tool is bundled with the SDK to simplify the creation of BCDs. It is accessible under the Tools Menu of Dev-C++.



[✓ Create Real OBCD] Create Complex CBCD

Real Part: 5.1235

Complex Part:

BCD Value: {{{0x05, 0x12, 0x35, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}, 0x1000}};

To use the tool, first choose whether you are going to create an OBCD or a CBCD. Next type in the decimal value for the real part and/or the complex part of your BCD and click OK.

The hex representation of the value you entered will be displayed in the text box at the bottom of the dialog. You can use this hex string to create a new OBCD as follows:

```
OBCD x = {{{0x05, 0x12, 0x35, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00},
0x1000}};

CBCD y = {{{{0x05, 0x87, 0x64, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00}, 0x1000}}, {{{0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00}, 0x1000}}};
```

More Information

For more information about OBCD/CBCD data structures and the functions they can use, please refer to the ClassPad 300 SDK API Reference Guide.

Strings and String Handling In the ClassPad

In this section we will discuss how strings are represented and used in the ClassPad. We will start with a look at the ClassPad's character set and move on to the CPString class, string conversion and displaying multiple languages.

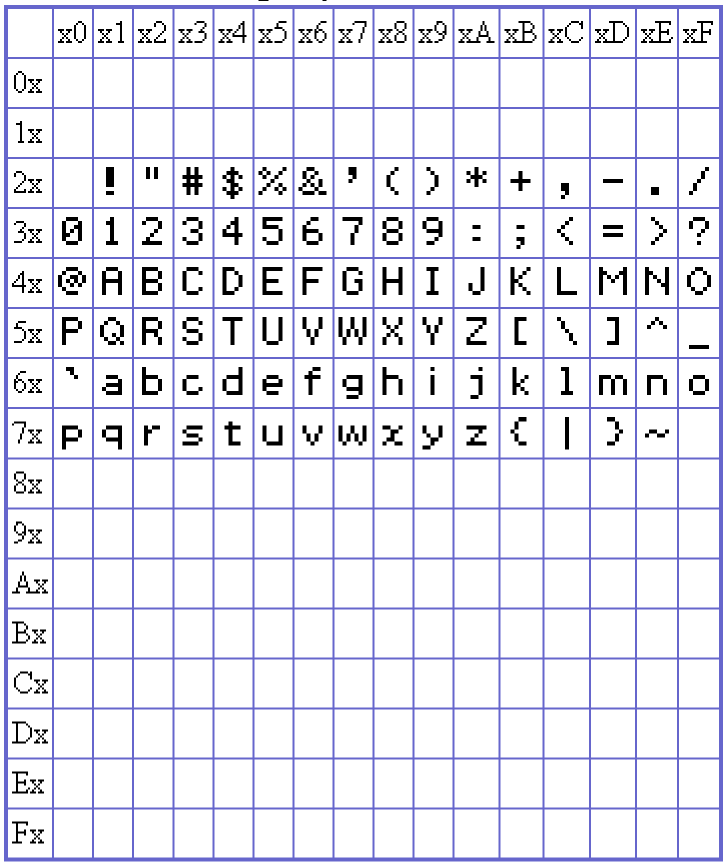
ClassPad Character Set

The ClassPad has a large number of characters it must be able to represent internally. Because of this the ClassPad supports multi-byte character strings.

To differentiate a single byte character from a multi-byte character, a multi-byte character's first byte is always a display code. Display codes are between the range 0xE0 and 0xEF. Currently only three codes are used: 0xEC, 0xED, and 0xEE.

The entire character set on the ClassPad is as follows:

Basic Single-Byte Characters 0xXX



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x																
1x																
2x		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x																
9x																
Ax																
Bx																
Cx																
Dx																
Ex																
Fx																

Multi-Byte Characters 0xEC

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x		À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î
1x	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
2x	Ÿ	Ā	Ă	Ą	Ć	Ĉ	Ċ	Č	Ď	Đ	Ē	Ĕ	Ė	Ę	Ě	Ĝ
3x	Ğ	Ġ	Ģ	Ĥ	Ħ	Ĩ	Ī	Ĭ	Į	İ	Ĳ	Ĵ	Ķ	Ĺ	Ļ	Ľ
4x	Ŀ	Ł	Ń	Ņ	Ň	Ŋ	Ō	Ŏ	Ő	Œ	Ŕ	Ŗ	Ř	Ś	Ŝ	Ş
5x	Š	Ţ	Ť	Ŧ	Ũ	Ū	Ŭ	Ů	Ű	Ų	Ŵ	Ŷ	Ź	Ż	Ž	Ơ
6x	Ư	Ǎ	Ǐ	Ǒ	Ǔ	Ǖ	Ǘ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι
7x	Κ	Λ	Μ	Ν	Ξ	Ο	Π	Ρ	Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ά
8x	Б	В	Г	Д	Е	Ё	Ж	З	И	Й	К	Л	М	Н	О	П
9x	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
Ax	Ɛ	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***	***I***	***J***	***K***	***L***	***M***	***N***	***O***
Bx	***P***	***Q***	***R***	***S***	***T***	***U***	***V***	***W***	***X***	***Y***	***Z***					
Cx																
Dx																
Ex	0	1	2	3	4	5	6	7	8	9	+	-				
Fx	0	1	2	3	4	5	6	7	8	9	+	-	-1	m	n	

Multi-Byte Characters 0xED

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x		à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î
1x	ï	ð	ñ	ò	ó	ô	õ	ö	ø	ù	ú	û	ü	ý	þ	ß
2x	ÿ	ā	ă	ą	ć	ĉ	ċ	č	ď	đ	ē	ĕ	ė	ę	ě	ĝ
3x	ğ	ġ	ģ	ĥ	ħ	ĩ	ī	ĭ	į	ı	ĳ	ĵ	ķ	I	ļ	ľ
4x	ŀ	ł	ń	ņ	ň	ŋ	ō	ŏ	ő	œ	ŕ	ŗ	ř	ś	ŝ	ş
5x	š	ţ	ť	ŧ	ũ	ū	ŭ	ů	ű	ų	ŵ	ŷ	ź	ż	ž	ơ
6x	ư	ǎ	ǐ	ǒ	ǔ	ǖ	ǘ	α	β	γ	δ	ε	ζ	η	θ	ι
7x	κ	λ	μ	ν	ξ	ο	π	ρ	σ	τ	υ	φ	χ	ψ	ω	а
8x	б	в	г	д	е	ё	ж	з	и	й	к	л	м	н	о	п
9x	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
Ax	ɛ	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
Bx	p	q	r	s	t	u	v	w	x	y	z					
Cx																
Dx																
Ex	ʕ	′	″	‴	(	)										
Fx	0	1	2	3	4	5	6	7	8	9	+	-	-1	x	y	

Multi-Byte Characters 0xEE

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x		i	e	E	π	ω	°	ʳ	ᵀ	ʼ	n	∠	x̄	ȳ	p̂	⇛
1x	×	≠	≦	≧	±	÷	≑	≪	≫	∈	∋	⊆	⊇	⊂	⊃	∪
2x	∩	∟	∨	⊻	∧	⊥	≡	⊽	∇	Δ	∵	∴	‘	’	”	“
3x	↵	◢	-	𝒩	𝒵	𝒬	ℛ	𝒞	¨	´	`	^	~	˚	ª	º
4x																
5x	√	Σ	Π	∫	∬	∮	∂	¿	¡	¢	£	¥	Fr	₣	€	†
6x	Å	℃	℉	←	↑	→	↓	↔	↕	↖	↗	↘	↙	¼	½	¾
7x	§	※	¶	¤	‖	¦	[	]	⊿	♪	♫	🔒	🔓	□	■	☑
8x	♠	♣	♥	◇	◆	▲	▶	▼	◀	○	◎	◉	●	🄴	🅁	…
9x	⋯	▫	▪	×	↱	↴	↳	↰	⊠	▲	|	•	↯	⬆	↤	x10
Ax																
Bx																
Cx																
Dx																
Ex																
Fx																

Each character also has a bold representation.

Let’s walk through the creation of the multi-byte character ½. Looking at the character charts, you’ll see that ½ has display code 0xEE and it’s second byte is 0x6E (refer to the chart immediately above). To create this character the code would look like this:

```
CP_CHAR multi[3]={0xEE,0x6E,0};
```

Notice that this array is null terminated.

CP_CHAR and PEGCHAR

While using the ClassPad, you may notice that some string functions expect PEGCHAR whereas others expect CP_CHAR. PEGCHAR is defined as:

```
typedef char PEGCHAR;
```

Whereas CP_CHAR is defined as:

```
#define     CP_CHAR unsigned char
```

In general, these are two things that mean the same thing. If you have a CP_CHAR that needs to be a PEGCHAR (or vice versa), just cast it to the correct type. Because these two types exist, you should avoid using the data type char. Using char will lead to a lot of unnecessary casting.

CP_CHAR Functions

The SDK provides a few string functions that can be used for characters or character arrays of type CP_CHAR.

To change the case of a CP_CHAR*, the following functions are used:

```
void CP_CharacterUpper(CP_CHAR *pS);
void CP_CharacterLower(CP_CHAR *pS);
void CP_FCharacterUpper(CP_CHAR *pS);
void CP_FCharacterLower(CP_CHAR *pS);
```

The first two functions, **CP_CharacterUpper()** and **CP_CharacterLower()**, are not multi-byte safe. When dealing with multi-byte characters be sure to use **CP_FCharacterUpper()** and **CP_FCharacterLower().**

To return the length of a CP_CHAR* array, use:

```
int CP_StringLen(CP_CHAR *pS);
int CP_StringByteLen(CP_CHAR *pS);
```

Notice that **CP_StringLen()** will return the number of characters in CP_CHAR*. This is not the same as the value returned by **CP_StringByteLen(),** which returns the number of bytes in a CP_CHAR. When dealing with multi-byte characters, the number of characters and number of bytes will not be the same.

The following functions are the equivalent of the Standard C strcpy/strncpy and strcat functions. When using multi-byte strings, be sure to use the multi-byte safe version of string copy and concatenate.

```
CP_CHAR *CP_StringCopy(CP_CHAR *dest, CP_CHAR *src);
CP_CHAR *CP_StringnCopy(CP_CHAR *dest, CP_CHAR *src, int maxlen);
CP_CHAR *CP_StringByteCopy(CP_CHAR *dest, CP_CHAR *src, int maxbyte);

CP_CHAR *CP_StringCat(CP_CHAR *dest, CP_CHAR *src);
CP_CHAR *CP_StringnCat(CP_CHAR *dest ,CP_CHAR *src, int maxlen);
CP_CHAR *CP_StringByteCat(CP_CHAR *dest ,CP_CHAR *src, int maxbyte);
```

Finally there are functions to compare two CP_CHAR*. These functions are similar to the standard C function strcmp, and return 0 if *pS1* == *pS2*, <0 if *pS1* > *pS2* or >0 if *pS2* < *pS1*. **CP_StringCmpi()** does a case-insensitive comparison and **CP_StringnCmp()** will compare the strings up to index *n*. When comparing multi-byte arrays, use **CP_StringByteCmp()** to compare on the byte level.

```
int CP_StringCmp(CP_CHAR *pS1, CP_CHAR *pS2);
int CP_StringCmpi(CP_CHAR *pS1, CP_CHAR *pS2);
int CP_StringnCmp(CP_CHAR *pS1, CP_CHAR *pS2, int n);
int CP_StringByteCmp(CP_CHAR *pS1, CP_CHAR *pS2, int maxbyte);
```

CPString

The CPString is a C++ class that encapsulates the memory allocation necessary for string handling and multi-byte string handling, while still providing access to a raw character buffer.

The character buffer is dynamically allocated, and may be reallocated when the string is modified. If the string is modified, destroyed, or goes out of scope, any saved reference to the string (a saved value from **Text()** or an iterator) should be considered invalid. The exception is if an iterator is passed by reference to the operator that modified the string, the iterator will be automatically updated.

CPString **is not** a reference counted string. If you pass a CPString as an argument to a function it will make a complete copy of that string. This is not very efficient and can waste a lot of memory. If you plan on passing a CPString as an argument to a function you should pass it as a constant reference. For example:

```
void MyFunction(const CPString& str)
{
    ... do something ...
}
```

Constructors and Assignment

To **create a new CPString** you can use one of the following constructors:

```
CPString();
CPString(CPMCHAR c);
CPString(const PEGCHAR* s);
CPString(const CPString& y);
CPString(OBCD d, int num_digits);
```

Instead of explicitly creating a string with a constructor, the assignment and concatenation characters are also defined as:

```
CPString& operator=(const CPString& y);
CPString& operator+=(const CPString& y);
CPString& operator+=(const PEGCHAR *y);
```

Here are some examples of creating CPStrings:

```
CPString str1;
str1 = "Hello world.";
str1 += " How are you?";  //str1 now equals “Hello world. How are you?”

CPString str2("I am fine.");
CPString str3(str2);  //str2=str3=”I am fine.”
```

```
OBCD x;
Cal_setpi_OBC(&x);
CPString pi(x, 10); // pi=”3.14159265”
```

CPString Comparison

The **comparison operators** == and != are defined as:

```
int operator==(const CPString& y);
int operator!=(const CPString& y);
```

The Standard C function strcmp is also available by using these functions:

```
int   Compare(const CPString& y) const;
int   Compare(PEGCHAR * lpsz) const;
```

Here is an example that uses these functions:

```
int i;
if(str2 == str3)
{
    if(str1 != str2)
    {
        i = str2.Compare("this will fail");
    }
}
```

Useful String Functions

The CPString class comes with a variety of string functions that will allow you to manipulate a string in many different ways.

First of all, if you need to know the **value of a character at a certain index**, the [] operator is defined so that the following code is valid:

```
CPString str;
str = “test”;
CP_CHAR c = str[1]; // c is ‘e’
```

You can also **retrieve the raw text buffer** with any of the following three functions:

```
const PEGCHAR* Text();
operator LPPEGCHAR();
PEGCHAR *GetBuffer();
```

Be aware that the pointer returned by these functions **is temporary and should never be saved or modified**. This buffer can become invalid when other CPString members are called (like += etc) or when the object goes out of scope

Another common request is for **the length of a string**. The follow functions help get a string’s length:

```
unsigned int ByteLength() const {return m_byteLength;}
unsigned int Length() const {return m_charLength;}
unsigned int ByteLengthTo(int charOffset);
int IsEmpty();
```

Notice that the first two functions are called **ByteLength()** and **Length()**. If you remember, CPStrings are made to hold single-byte or multiple-byte characters. This means that **a string's byte length is not always equal to its character length.** Be careful when allocating a string to use **ByteLength()** and not **Length().**

ByteLengthTo() converts a character offset to a byte offset and **IsEmpty()** returns a 1 if the string is not empty.

For example usage of some of these functions, take a look at the following code:

```
CP_CHAR multi[3]={0xEE,0x6E,0};
CPString str1((PEGCHAR*)multi);  //multi-byte character ½
CPString str2("2");

int j;
j = str1.ByteLength();   // j=2
j = str1.Length());     // j=1

j = str2.ByteLength();  // j=1
j = str2.Length();      // j=1
```

You can **change the case of a string** with the following functions:

```
void ToggleCase();
void UpperCase();
void LowerCase();
```

You can return the **substring of a string** by using the **Mid()** or **Left()** :

```
CPString Mid(int charOffset);
CPString Left(int charOffset);
```

Left() returns a sub-string from the beginning of the string to the character offset charOffset. Whereas **Mid()** extracts a sub-string from the end of the string to the character offset *charOffset*.

To **clear the contents of a CPString** use:

```
void Clear();
```

Finally, you can replace all of the occurrences of a character in a CPString with:

```
void ReplaceCharacter(CPMCHAR c0, CPMCHAR c1);
```

Buffer Ownership

CPStrings control and manage a PEGCHAR* buffer that represents a string. The deletion of these buffers is normally handled by the CPString class. If you have a PEGCHAR* that you have already created, but would like to encapsulate in a CPString class, you can place it in a CPString with:

```
void TakeBufferOwnership(PEGCHAR *buffer);
```

This does not mean that the PEGCHAR buffer is copied into CPString. Instead, the current memory location of PEGCHAR becomes the buffer portion of the CPString. The CPString's current buffer is discarded when it takes ownership of the new buffer.

On the other hand, you can give up the ownership of a buffer and place it back into a PEGCHAR* with:

```
PEGCHAR *GiveBufferOwnership();
```

This function differs from **GetBuffer()** in a very important way: Once a CPString has called **GiveBufferOwnership()** its buffer is gone. The person who called the function is now responsible for keeping track of the returned PEGCHAR* and making sure that it is deleted.

String Conversions

The ClassPad contains several functions to convert strings to and from different data types. Be aware that these functions take CP_CHAR* and not CPStrings. However, once you have the converted value you can easily create a CPString with the appropriate constructor.

Converting Between CPStrings and Supported C native data types

To **convert a string to another data type** the following functions are used:

```
int CP_StringToInt(CP_CHAR *pS);
long CP_StringToLong(CP_CHAR *pS);
short CP_StringToShort(CP_CHAR *pS);
char CP_StringToChar(CP_CHAR *pS);
```

If you want to **convert from an int, long, short or char to a string**, use:

```
CP_CHAR *CP_IntToString(int value, CP_CHAR *pS);
CP_CHAR *CP_LongToString(long value, CP_CHAR *pS);
CP_CHAR *CP_ShortToString(short value, CP_CHAR *pS);
CP_CHAR *CP_CharToString(char value, CP_CHAR *pS);
```

Finally, if you want to **convert from an int, long, short or char to a HEX string**, use these functions:

```
CP_CHAR *CP_IntToStringHex(int value, CP_CHAR *pS);
CP_CHAR *CP_LongToStringHex(long value, CP_CHAR *pS);
CP_CHAR *CP_ShortToStringHex(short value, CP_CHAR *pS);
CP_CHAR *CP_CharToStringHex(char value, CP_CHAR *pS);
```

Here is an example that uses some of these functions:

```
CPString str1("44");
int i = CP_StringToInt((CP_CHAR*)str1.GetBuffer());  // i=44

CP_CHAR c[100];
CP_IntToString(54, c);  // c=”54”

CP_IntToStringHex(15, c);  // c=”0000000F”
```

Converting Between CPStrings and BCDs

The ClassPad does not have native support for doubles. Instead the ClassPad uses its own data type called the BCD to represent floating point numbers. BCDs and their internal representation are discussed in detail in the section titled *Using Floating-Point Values with the ClassPad*.

Because a BCD can have several different visual representations (scientific, normal, fixed, etc) there are several different functions to convert a BCD to a string. There are also usually pairs of functions: one that works with OBCD (real numbers) and another that works with CBCD (complex numbers).

To take the internal representation of a BCD in hex, and place it in a string use:
```
word CP_codech_OBC(CP_CHAR data[], OBCD *x);
```

To convert a BCD to normal mode use:
```
word CP_Norm_OBC(OBCD *x,CP_CHAR str[],short mode);
word CP_Norm_CMP(CBCD *x,CP_CHAR str[],short mode);
```

The parameter *mode* represents which normal mode you would like to use. Valid options are IM_MODE_NORM1 or IM_MODE_NORM2.

To convert a BCD to a fixed size use:
```
word CP_Fix_OBC(OBCD *x,CP_CHAR str[],short dig);
word CP_Fix_CMP(CBCD *x,CP_CHAR str[],short dig);
```

where *dig* is the number of digits to the right of the decimal. For example 5 with a fixed size of 3 would be 5.000.

To create a string representation of a BCD in scientific notation use:
```
word CP_Sci_OBC(OBCD *x,CP_CHAR str[],short dig);
word CP_Sci_CMP(CBCD *x,CP_CHAR str[],short dig);
```

The parameter *dig* represents the number of significant digits. So the BCD 555555 passed with a *dig* of 3 would be 5.55e5.

You can also use the following functions to convert a BCD to format *mode* of length *digit*:

```
word CP_digit_OBC(OBCD *x,CP_CHAR str[],word digit,
                              short mode,short dig);
word CP_digit_CMP(CBCD *x,CP_CHAR str[],word digit,
                              short mode,short dig);
```

The values for *mode* can be:

- IM_MODE_NORM1
- IM_MODE_NORM2
- IM_MODE_FIX
- IM_MODE_SCI

digit can be one of the following:

- IM_DIGIT_9
- IM_DIGIT_6
- IM_DIGIT_4

digit sets the maximum number of characters that can appear in a BCD. For example, pi with IM_DIGIT_4 would be 3.14. Pi with IM_DIGIT_6 would be 3.1415

The parameter *dig* is only used with Sci and Fix modes to determine the number of digits after the decimal point.

Finally, these two functions can be used to convert a BCD to string in degrees, minutes, seconds representation:

```
word CP_dms_OBC(OBCD *x, CP_CHAR str[]);
word CP_dms_CBC(CBCD *x, CP_CHAR str[]);
```

Here is an example that uses some of these functions:

```
OBCD x;
Cal_setpi_OBC(&x);  // set x = 3.141592654…
CP_CHAR c[100];

CP_Norm_OBC(&x, c, IM_MODE_NORM1);  // c=”3.141592654”

CP_Fix_OBC(&x, c, 3); // c=”3.142”

CP_Sci_OBC(&x, c, 2); // c=”3.1e+0”

CP_digit_OBC(&x, c, IM_DIGIT_9, IM_MODE_SCI, 3); // c=”3.14e+0”
```

There is only one function available to convert a string to an OBCD:

```
int ExecuteToOBCD(const PEGCHAR *str, OBCD &obcd, BOOL bErrCheck=TRUE);
```

While the function does not take a CPString as an argument, you can send the buffer of a CPString like this:

```
      CPString str = "123.32";
      ExecuteToOBCD(str.GetBuffer(), x, FALSE);
```

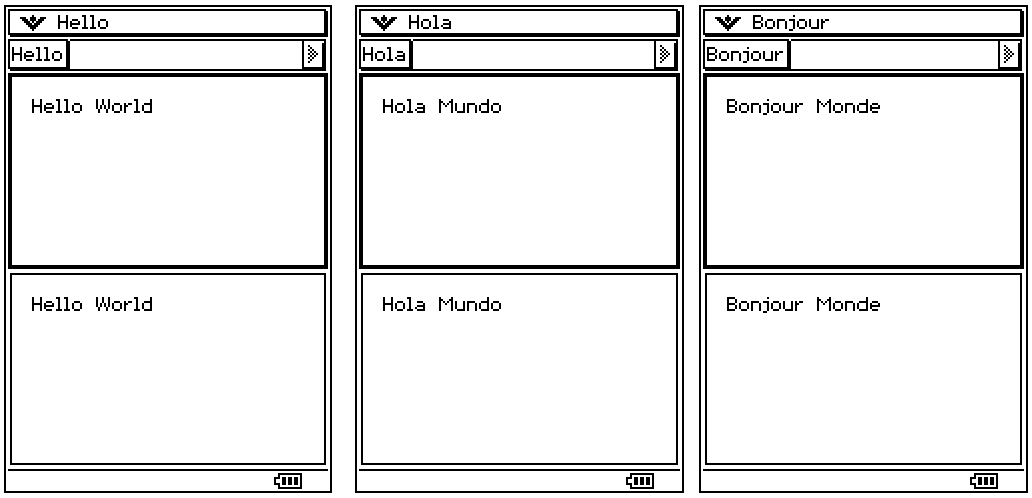
Multiple Language Support in the ClassPad

The ClassPad provides a method to easily allow you to create an add-in application with support for multiple languages. You might have noticed that in every ClassPad add-in that is created, the function

```
char *ExtensionGetLang(ID_MESSAGE MessageNo)
```

must be defined to at least return “”. When properly used, this function will take in a message number of a displayed string and output the correct text for the current language. To work correctly, there is a little setup work that must be done, but once you understand the steps it is very easy to use.

To go step by step through creating an add-in with multiple language support we will refer to the Hello World example add-in that came with the ClassPad 300 SDK. The add-in is located in **Documents\ClassPad 300 SDK \Examples\HelloWorld\**. First, let’s run the program and see how the text on the ClassPad changes depending on what language you are using.



Here you see the same screen in English, Spanish and French. You’ll also notice that there are two different text strings on the screen: “Hello” and “Hello World”.

Message Number Enumeration

The first step that you will need to do when designing your application is keep a running enumeration of IDs for all of the strings that will appear in you application. These IDs are called message numbers. In the Hello World add-in this is done in HelloLangDatabase.h. Here is the code that creates the enumeration:

```
#define HELLO_MESSAGE_START   LOCAL_LANG_START+1

enum HelloWorldMessage {
      HELLO_HELLO= HELLO_MESSAGE_START,
      HELLO_HELLO_WORLD,
      HELLO_MESSAGE_END
};
```

In the enumeration we have the HELLO_HELLO message ID, which refers to the “Hello” message on the button and the menu, and the HELLO_HELLO_WORLD message ID that refers to the “Hello World” string in the Module Windows. The more text messages you have, the more entries in the enumeration you must add. The enumeration should always start from **LOCAL_LANG_START + 1**. This ensures that there is no collision of message IDs. In HelloLangDatabase.h LOCAL_LANG_START + 1 is #defined as HELLO_MESSAGE_START. You should also add an ending entry into your enumeration, like HELLO_MESSAGE_END, so you can check that a message ID is within your enumeration range.

Language Arrays

Each one of the message IDs will become an index into an array. The array that is used depends on what language is currently set. Using this index and the array that corresponds to the current language, the correct text string will be returned. First, let’s take a look at all of the arrays that HelloWorld defines in HelloLangDatabase.cpp:

```
CP_CHAR *HelloMessageData_Eng[]={
      "Hello",
      "Hello World",
};

CP_CHAR *HelloMessageData_Deu[]={
      "Hallo",
      "Hallo Welt",
};

CP_CHAR *HelloMessageData_Esp[]={
      "Hola",
      "Hola Mundo",
};
CP_CHAR *HelloMessageData_Fra[]={
      "Bonjour",
      "Bonjour Monde",
};

CP_CHAR *HelloMessageData_Por[]={
      "Hallo",
      "Hallo Mundo",
};
```

Each supported language in the ClassPad has its own array. You do not have to support every language if you do not want to.

Defining ExtensionGetLang()

Now comes the step of actually defining the function **ExtensionGetLang().** You can view **ExtensionGetLang()** in its entirety by looking in HelloLangDatabase.cpp. We will be looking at the function piece by piece for explanation purposes.

ExtensionGetLang() receives an ID_MESSAGE *MessageNo* as a parameter. This parameter should be one of your message IDs. Just to make sure that *MessageNo* appears in our enumeration, and to normalize it to be an array index we do the following:

```
if (MessageNo<HELLO_MESSAGE_START || MessageNo>HELLO_MESSAGE_END)
            return "";

MessageNo -= HELLO_MESSAGE_START;
```

This simply says if *MessageNo* is not in our enumeration, return “”. Otherwise normalize the message ID to start at 0 so it can function as an index to an array.

The next step that we need to do is determine which array to use to get the language string. To determine the current language we use the function:

```
int GetCurrentLanguageInfo()
```

This function’s return value is used in a switch statement of all of the supported languages:

```
      switch (GetCurrentLanguageInfo())
      {
            case CurrentLanguage_Deu :
                  pStr = (char *)HelloMessageData_Deu[MessageNo];
                  break;
            case CurrentLanguage_Esp :
                  pStr = (char *)HelloMessageData_Esp[MessageNo];
                  break;
            case CurrentLanguage_Fra :
                  pStr = (char *)HelloMessageData_Fra[MessageNo];
                  break;
            case CurrentLanguage_Por :
                  pStr = (char *)HelloMessageData_Por[MessageNo];
                  break;
            case CurrentLanguage_Eng :
            default :
                  pStr = (char *)HelloMessageData_Eng[MessageNo];
                  break;
      }
      return pStr;
```

The current language decides which of our language arrays we will index into. We then return the string at our normalized index to complete the **ExtensionGetLang()** function.

But you may be wondering, how does the menu or button know which message ID it should send? The answer is that you supply this information when you create the object. For example here is the creation of the text button in the toolbar from HelloWorldModule.cpp:

```
PegTextButton* b = new PegTextButton(1,1, GetLang(HELLO_HELLO),
                                    IDB_HELLO,AF_ENABLED|TT_COPY);
```

Instead of creating the button with a text string, we use the function **GetLang()** with the appropriate message ID. The button's text is "Hello" so we use the message ID that represents "Hello", HELLO_HELLO, as the parameter to **GetLang().**

The **GetLang()** function will check this parameter to see if it is a system standard Id. System standard IDs are IDs that already have values for all the languages supported by the ClassPad. These include common menu entries such as "Copy", "Paste", or "Undo". Refer to the SDK Reference Guide for a complete list of all of the system standard messages and their IDs.

If the message ID is not in the system range, then the **ExtensionGetLang()** function that you created will be called to return the correct text string.

You should refer to the HelloWorld example if you have any more questions about creating an add-in that supports multiple languages.

MCS – Memory Control System

MCS, Memory Control System, is used to save data on the ClassPad. This includes saving something as simple as a variable in Main to saving something as complex as an eActivity file with multiple embedded applications. This section will briefly discuss the structure of MCS and then provide information on the BIOS functions to write to MCS as well as the C++ file classes for MCS.

MCS Overview and Structure

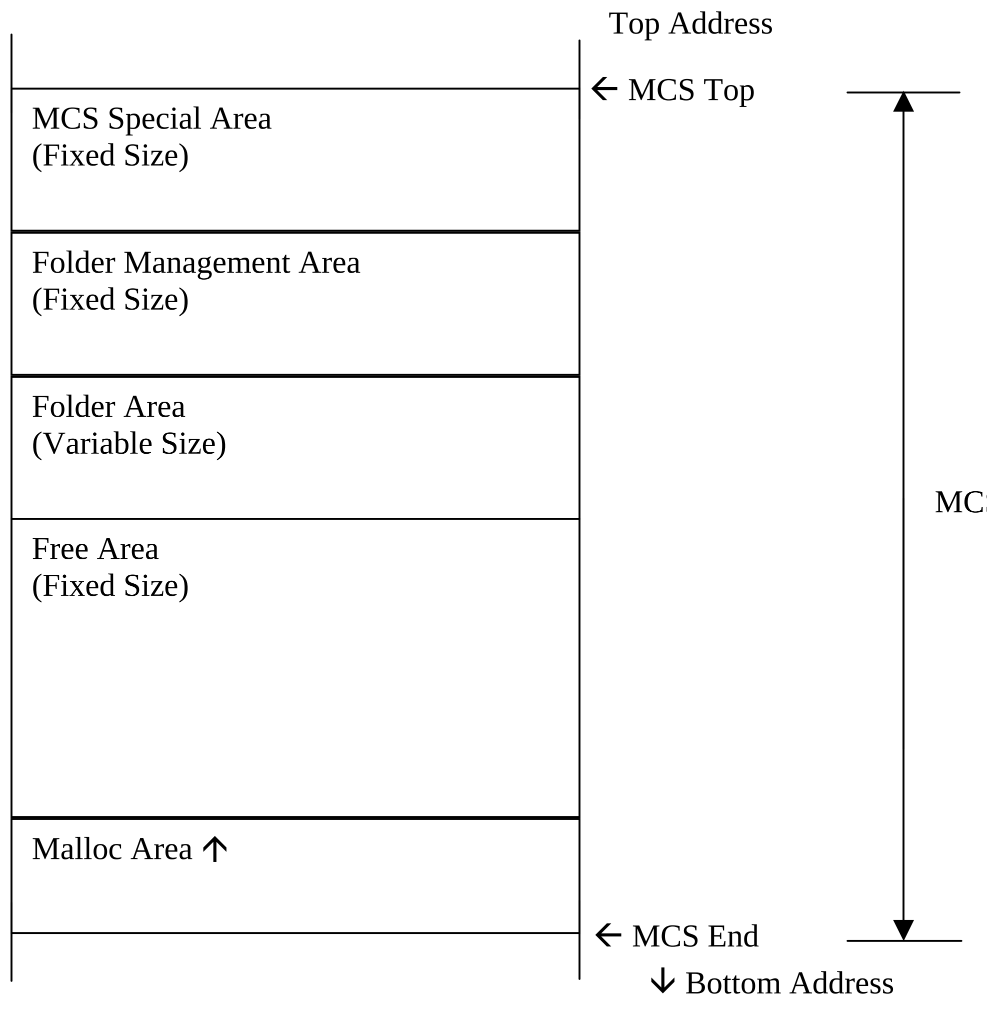
All variables on the ClassPad are saved in MCS. Each variable must have a name and a data type. The following is a list of data types and their size in bytes:

Variable Type	Size
Real number	12
Complex number	24
Integer	4
Float	8
String	n
Expression	n
Program	n
Function	n
File	n
List	n
Vector	n
Matrix	n
GraphPicture	n
mem	n
ProgramExe	n
Gmem	n
3D-Graph	n
Formula process	n

All variable sizes must be divisible by 4.

Variables are stored in folders. A variable's name must be unique to its folder. Folders can only be one level deep.

MCS has the following structure in RAM:



The three sections of MCS each perform their own separate tasks. The MCS Special Area handles the overhead necessary to control the MCS file system. The Folder Management Area manages the folders that are created in MCS. Finally, the Folder Area controls all of the data in MCS and the data attributes.

The Folder Management area uses the following structure to control all of the folders in MCS:

```
typedef struct  _FOLDERMANAGEMENTSTRUCT{
    NAMEBUFFER  naName;     // Folder name
    UCHAR       *pucTopAdr; // Folder Data Area top address
    WORD        wValNumber; // Number of variables in folder
    UCHAR       ucFlag;
    UCHAR       ucReserve;
}FOLDERMANSTRUCT;
```

The NAMEBUFFER holds the name of the folder and is defined as:

```
typedef struct  NAMEBUFFER
{
    CP_CHAR cpcName[8];             // Folder and Variable Name Buffer
}NAMEBUFFER;
```

The variables are stored in structures that are defined as:

```
typedef struct  _VARIABLEMANAGEMENTSTRUCT{
    NAMEBUFFER  naName;         // Variable Name
    DWORD       dwOffsetAdr;    // Variable offest
    DWORD       dwDataSize;     // Variable data size
    UCHAR       ucType;         // Data Type
    UCHAR       ucFlag;
    UCHAR       ucReserve;      // Reserved
    UCHAR       ucFolderNo;     // Folder Management Area
}VALMANSTRUCT;
```

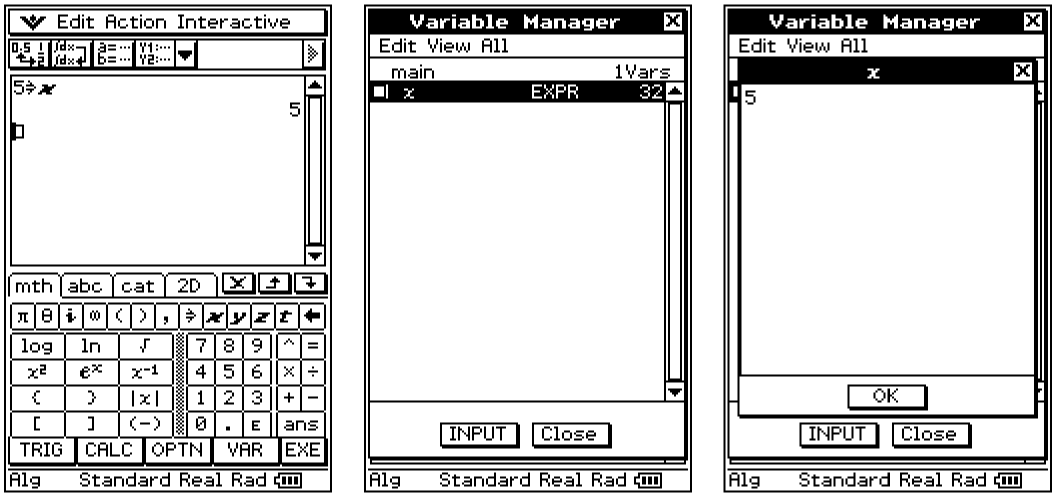
For the most part, you do not need to concern yourself with the internals of MCS or the specifics of these structures. Most of the time you will interact with MCS via the functions that are contained in MCSBiosO.h or the CPFile classes, and never modify a variable or a folder struct directly.

Interacting with MCS via BIOS Functions

The file MCSBiosO.h contains several functions that allow you to interact with MCS in your add-in. In this section we will go through most of the functions in MCSBiosO.h and provide examples on how to use them.

Creating/Deleting Variables and Folders

Before we begin using the BIOS functions to create a variable, let’s see how it is done in Main on the ClassPad. On the ClassPad, start up main and type in 5⇒x. Then click on the ClassPad Menu->Settings->Variable Manager. The variable manager will open and you will see that the main folder has one variable in it. Double click on that folder and you will find your variable x with a value of 5.



To create this same variable in your add-in you would use the function:

```
int BMCSCreateVariable(CP_CHAR *pcpcFolderName, CP_CHAR *pcpcValName,
            UCHAR ucValType, DWORD dwDataLength, UCHAR *pucDataTopAdr)
```

If the function completed successfully, the return value will be IMU_MCS_SUCCESS, #define'd as 0. Otherwise the return value will be one of the error codes defined in MCSLib.h. To create the same value as 5⇒x, we would do the following:

```
OBCD dat;
Cal_setn_OBC(5,&dat);
if(IMU_MCS_SUCCESS != BMCSCreateVariable((CP_CHAR*)"main",
      (CP_CHAR*)"x", IMU_MCS_TypeReal, sizeof(OBCD), (UCHAR*)&dat))
{
      //error handling…
}
```

To delete a variable you would use the function:

```
int BMCSDeleteVariable(CP_CHAR *pcpcFolderName, CP_CHAR *pcpcValName)
```

To delete the variable we created in the previous example, we would do the following:

```
if(IMU_MCS_SUCCESS != BMCSDeleteVariable((CP_CHAR *)”main”,
                                                (CP_CHAR *)”x”))
{
      //error handling…
}
```

The functions to create folders and delete folders are as follows:

```
int BMCSCreateFolder(CP_CHAR *pcpcFolderName, UCHAR *pucFolderNumber)
int BMCSDeleteFolder(CP_CHAR *pcpcFolderName)
```

When creating a folder, you must pass in a reference to a UCHAR to get the folder number back from the function. The following example shows how to create and delete a folder:

```
UCHAR temp;
if(IMU_MCS_SUCCESS != BMCSCreateFolder((CP_CHAR *)"test", &temp))
{
  // error handling…
}

if(IMU_MCS_SUCCESS != BMCSDeleteFolder((CP_CHAR*)"test"))
{
  // error handling…
}
```

Changing a Variable's Name and Attributes

The MCS BIOS provides several functions to control different aspects of a variable. This includes renaming a variable, copying and moving a variable, setting variable attributes and searching for variables.

To rename a variable the following function is used:

```
int BMCSRenameVariable(CP_CHAR *pcpcFolderName,
                  CP_CHAR *pcpcOldValName, CP_CHAR *pcpcNewValName)
```

Variables also support being locked, meaning that their value cannot be altered. To lock or unlock a variable you can use the following functions:

```
int BMCSVariableLockOn(CP_CHAR *pcpcFolderName, CP_CHAR *pcpcValName)
int BMCSVariableLockOff(CP_CHAR *pcpcFolderName, CP_CHAR *pcpcValName)
```

There are also functions to check the attributes of a variable. A variable can have the following possible attributes:

IMU_MCS_FlagLock	Folder/Variable Lock Flag
IMU_MCS_FlagUsing	Folder/Variable In Use Flag
IMU_MCS_FlagUsed	Folder/Variable Used Flag
IMU_MCS_FlagCursor	Cursor on
IMU_MCS_FlagSelect	Select on

To get or set a variable's attributes, use the following functions:

```
int  BMCSSetVariableAttribute (CP_CHAR *pcpcFolderName,
                  CP_CHAR *pcpcValName, UCHAR ucAttributeData)
int  BMCSGetVariableAttribute (CP_CHAR *pcpcFolderName,
                  CP_CHAR *pcpcValName, UCHAR *pucAttributeData)
```

Here is an example that uses some of these functions:

```
UCHAR attr;
OBCD dat;
Cal_setn_OBC(5,&dat);

// Create x=5
BMCSCreateVariable((CP_CHAR*)"main", (CP_CHAR*)"x", IMU_MCS_TypeReal,
                                          sizeof(OBCD), (UCHAR*)&dat);

// Rename x to y
BMCSRenameVariable((CP_CHAR*)"main", (CP_CHAR*)"x", (CP_CHAR*)"y");

// Lock y
BMCSVariableLockOn((CP_CHAR*)"main",(CP_CHAR*)"y");

// Get the attributes of y
BMCSGetVariableAttribute((CP_CHAR*)"main",(CP_CHAR*)"y", &attr);

// Check to see if the lock is set on y
if ((attr & IMU_MCS_FlagLock) != 0)
{
    // The lock is on, so we will turn it off
    BMCSVariableLockOff((CP_CHAR*)"main",(CP_CHAR*)"y");
}
else
{
    // The lock is off, so we will turn it on
    BMCSVariableLockOn((CP_CHAR*)"main",(CP_CHAR*)"y");
}
```

Moving/Copying and Finding a Variable

To copy or move an MCS variable, the following functions are provided:

```
int  BMCSCopyVariable (CP_CHAR *pcpcSourceFolderName,
            CP_CHAR *pcpcSourceValName, CP_CHAR *pcpcDestFolderName,
            CP_CHAR *pcpcDestValName)

int  BMCSMoveVariable (CP_CHAR *pcpcSourceFolderName,
            CP_CHAR *pcpcSourceValName, CP_CHAR *pcpcDestFolderName,
            CP_CHAR *pcpcDestValName)
```

If after moving a variable, you need to find it, there are three functions that allow you to search for a variable:

```
int  BMCSSearchVariable (CP_CHAR *pcpcFolderName, CP_CHAR *pcpcValName,
                        UCHAR *pucValType, UCHAR **ppucManTopAdr,
                        UCHAR **ppucDataTopAdr, DWORD *pdwDataSize)

int  BMCSSearchVal2 (CP_CHAR *pcpcFolderName, CP_CHAR *pcpcValName,
                              CP_CHAR *pcpcValName2, UCHAR ucLength)

int  BMCSSearchVal3 (CP_CHAR *pcpcFolderName, CP_CHAR *pcpcValName,
                              VALMANSTRUCT **ppValMan, UCHAR ucLength)
```

The first function, **BMCSSearchVariable(),** searches for an exact match of the variable name that you pass in. The second and third both perform partial matches. Notice that **BMCSSearchVal3()** takes a VALMANSTRUCT reference as a parameter. In this case, it is important that you know and understand the structure of a variable in MCS before trying to use the function.

The following is an example that uses all three search functions as well as both the copy and move function:

```
UCHAR attr, temp;
OBCD dat;
Cal_setn_OBC(5,&dat);

// Create x=5 in folder main
BMCSCreateVariable((CP_CHAR*)"main", (CP_CHAR*)"x", IMU_MCS_TypeReal,
                                    sizeof(OBCD), (UCHAR*)&dat);

// Create folder test
BMCSCreateFolder((CP_CHAR*)"test", &temp);

// Copy x to test
BMCSCopyVariable((CP_CHAR*)"main", (CP_CHAR*)"x", (CP_CHAR*)"test",
                                          (CP_CHAR*)"copied x");

// Move x to test
BMCSMoveVariable((CP_CHAR*)"main", (CP_CHAR*)"x", (CP_CHAR*)"test",
                                          (CP_CHAR*)"moved x");

// Search using the first search function.  This function returns the
// address of the variable if found.
UCHAR *ucDataTopAddress;
DWORD dwDataSize;
UCHAR *pucManTopAdr;
UCHAR ucValType;

if(BMCSSearchVariable((CP_CHAR*)"test",(CP_CHAR*)"moved x",
&ucValType,&pucManTopAdr, (UCHAR **)&ucDataTopAddress, &dwDataSize)!=0)
{
      // not found!
}
else
{
      // Create a variable using the address from the search function
      BMCSCreateVariable((CP_CHAR*)"test",(CP_CHAR*)"found x1",
                  IMU_MCS_TypeReal,dwDataSize, (UCHAR *)&pucManTopAdr);
}


// The second search function returns the name of the found variable
// in a CP_CHAR buffer
CP_CHAR buffer[100];
if(BMCSSearchVal2((CP_CHAR*)"test",(CP_CHAR*)"moved ", buffer, 5) != 0)
{
    // not found!
}
```

```
else
{
      // If the variable is found, make a copy of it
      BMCSCopyVariable((CP_CHAR*)"test", buffer, (CP_CHAR*)"test",
                                    (CP_CHAR*)"found x2");
}

// The third search function returns the structure of the found
// variable.  This structure contains the variable’s name
VALMANSTRUCT   *pValMan;
if(BMCSSearchVal3((CP_CHAR*)"test",(CP_CHAR*)"copied ",&pValMan, 7)!=0)
{
     // NotFound!!
}
else
{
      // If the variable is found, create a copy of it
      BMCSCopyVariable((CP_CHAR*)"test", pValMan->naName.cpcName,
                        (CP_CHAR*)"test", (CP_CHAR*)"found x3");
}
```

Changing a Folder’s Name/Attributes

Much like variables, you can change the name of a folder, change its locked status and get its attributes with the following functions:

```
int BMCSRenameFolder(CP_CHAR *pcpcOldFolderName,
                              CP_CHAR *pcpcNewFolderName)

int BMCSFolderLockOn(CP_CHAR *pcpcFolderName)
int BMCSFolderLockOff(CP_CHAR *pcpcFolderName)

int BMCSGetFolderAttribute(CP_CHAR *pcpcFolderName,
                              UCHAR *pucAttributeData)
int BMCSSetFolderAttribute(CP_CHAR *pcpcFolderName,
                              UCHAR ucAttributeData)
```

Unlike variables, when dealing with folders you can get and set the current folder by calling these functions:

```
int  BMCSGetCurrentFolder(CP_CHAR *pcpcFolderName,
            FOLDERMANSTRUCT **pFolderMan, UCHAR *pucFolderNumber)
int  BMCSSetCurrentFolder(CP_CHAR *pcpcFolderName,
            UCHAR **ppucManTopAdr, UCHAR *pucFolderNumber)
```

To get the current folder, you pass in a buffer to hold the current folder’s name as well as a FOLDERMANSTRUCT reference to get the current folder’s struct. For example:

```
CP_CHAR folderName[sizeof( NAMEBUFFER)+1];
UCHAR ucFolderNumber;
FOLDERMANSTRUCT *pFolderMan;
if(BMCSGetCurrentFolder(folderName,&pFolderMan,&ucFolderNumber)!=0)
{
   return MEM_ERR;
}
```

To set the current folder, you just need the name of the folder:

```
CP_CHAR    folderName[] = "FOLDER1";
UCHAR         *pucManagementTopAddress;
UCHAR          ucFolderNumber;

if(BMCSSetCurrentFolder(folderName,pucManagementTopAddress,
                                          &ucFolderNumber) != 0)
{
   return MEM_ERR;
}
```

Searching for a Folder

Unlike with variables, there is only one function that is used to search for a folder. It is declared as:

```
int BMCSSearchFolder(CP_CHAR *pcpcFolderName, UCHAR **ppManTopAdr,
                                                UCHAR *pucFolderNumber)
```

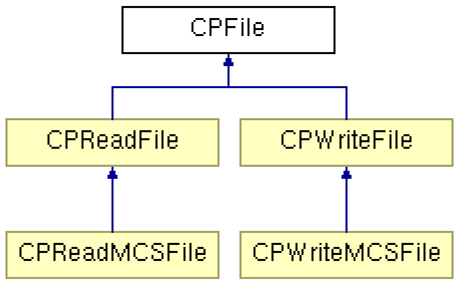
To search, you just need to pass the name of the folder that you are looking for:

```
// To search "FOLDER1".
CP_CHAR     pcpcFolderName[] = "FOLDER1";
UCHAR *pucManagementTopAddress;
UCHAR ucFolderNumber;
if(BMCSSearchFolder(pcpcFolderName, &pucManagementTopAddress,
                                          &ucFolderNumber) != 0)
{
      return MEM_ERR;
}
```

Reading/Writing to MCS using the CPFile Class

Most of the time when you save something in MCS you want to save more than a single variable. For example, in the AddressBook example that comes with the SDK, all of a user's contacts must be saved out to and read in from a single file. To accomplish this, we use the CPFile classes.

The CPFile class hierarchy looks like this:



Since we are reading and writing from MCS we will only create instances of the classes CPReadMCSFile and CPWriteMCSFile to actually read and write. There are some member functions in CPReadFile/CPWriteFile and CPFile that we will use, but you will never instantiate these classes.

All CPFile derived classes of type IMU_MCS_TypeMem should have a header to identify the application that uses the file and what kind of data it contains. In this section we will go through an example of how to use the CPMEMFileHeader class to create a file header as well as how to read and write a file using the CPFile classes.

Reading From MCS

To read from MCS you must create an instance of CPReadMCSFile. You can use one of the following constructors:

```
CPReadMCSFile (UCHAR type)
CPReadMCSFile (const char *name, const char *path=NULL, UCHAR type=0)
```

The first constructor creates CPReadMCSFile without a name or folder. You have to set these attributes before you can use the file.

The CPReadMCSFile does not have any functions to read data directly. It inherits a few functions from CPReadFile to read some data types from MCS. These functions include:

```
int ReadBytes(void *buffer, int nBytes);
virtual char  ReadByte();
WORD ReadWord();
int ReadInt();
OBCD ReadDouble();
int BytesRead();
```

There are also a few important inherited functions from CPFile that should be used before trying to read a file:

```
bool IsNotError();
int FileExists();
```

Here is a simple example of how to open a file and read in an integer:

```
CPReadMCSFile f(FILE_NAME, FOLDER_NAME);
if (f.IsNotError() && f.FileExists())
{
	int i = f.ReadWord();
}
```

While this would work fine if your application only saved integers, how do you read data types that do not have a function in CPReadFile? For the most part, all data types define their own **Read()** function to support being read from MCS.

For example, let's assume a class foo with the following:

```
class Foo
{
	...
	void Read(CPReadMCSFile& f);
}
```

We'll also say that foo's data members that will be read from a file are an int count and a CPString string. Then foo's Read function would look something like this:

```
void Foo::Read(CPReadFile& f)
{
	i = f.ReadInt();

	string.Read(f);
}
```

Just like class foo has its own **Read()** function, so does class CPString. So to read a CPString, we just call its **Read()** function. In the same manner, if we had a class that had a foo as a data member we would just call foo.Read(f) in that class's **Read()** function.

This means that any class that you create and want to read from MCS must have its own read function. Conisder the ContactArray class in the AddressBook example. Remember that the ContactArray is an array of Contact objects. The ContactArray's Read function first reads in an integer that represents the number of contacts saved to the file. It then loops that many times and calls the Contact class's read function to read in each contact:

```
void ContactArray::Read(CPReadFile& f)
{
	//first read in how many contacts are saved
	int count = f.ReadWord();

	// loop that many times
	for(int i=0;i<count;i++)
	{
		//create a contact
		Contact* c = new Contact();

		// Call that contact’s read function
		c->Read(f);
		Add(c);
	}
	//makes sure the array is in order
	sort();
}
```

If we look closely at the Contact class, we see it is just composed of CPStrings that hold all the information for a contact. Therefore, the Contact class's **Read()** function will just be each one of these CPStrings calling its **Read()** function:

```
// Reads a contact from a file.
// Each CPString just calls its read function
void Contact::Read(CPReadFile& f)
{
      firstName.Read(f);
      lastName.Read(f);
      phone1.Read(f);
      phone2.Read(f);
      email.Read(f);
      address.Read(f);
}
```

As you can see, the **Read()** functions that you create will usually just call that class's data members' **Read()** functions.

Writing to MCS

To write to MCS, you must create an instance of the CPWriteMCSFile class. The following constructors are available:

```
CPWriteMCSFile (UCHAR type)
CPWriteMCSFile (const char *name, const char *path=NULL, UCHAR type=0)
```

If you use the first constructor, you must set the name and path of the file before trying to write.

When you create a MCSWriteFile you do not provide a file size in the constructor. Upon creation no memory will be allocated and any write functions you call will not actually write to memory. The write functions will, however, compute the size of the object written. The function **Realize()** can then be used to allocate the memory for the file. A second call to the write functions will then write the file to memory. This method allows you to write a file without determining how much space the file will require in memory prior to creation.

CPWriteMCSFile does not have any functions that write to memory. Like CPReadMCSFile, it inherits a few write functions from its base class CPWriteFile:

```
int WriteBytes(void *buffer, int nBytes);
virtual void WriteByte(char c);
void WriteWord(WORD w);
void WriteInt(int ii);
void WriteDouble(OBCD xx);
```

Here is a simple example of creating a CPMCSWriteFile and writing an int:

```
CPWriteMCSFile f(FILE_NAME, FOLDER_NAME);
f.WriteInt(8);
```

```
f.Realize();
f.WriteInt(8);
```

Without the call to **Realize()** the amount of space needed to write the file would have been computed, but the file would not have been written to memory. Once you call **Realize()** the file is put in write mode, and all subsequent write functions actually write to memory.

Just like when reading files, most of the time you will be interested in writing more than just integers. With write files you will create **Write()** functions for classes that need to write to MCS.

For example, let's look at how the foo class would implement Write:

```
class Foo
{
      ...
      void Write(CPWriteMCSFile& f);
}
```

This time we'll say that foo wants to write data members count and string. Foo's **Write()** function would look something like this:

```
void Foo::Write(CPWriteFile& f)
{
      f.WriteInt(count);

      //just like class CPString had a Read function, it also has
      //a write function
      string.write(f);
}
```

Just like when calling **Read(),** we call CPString's **Write()** function to write a CPString. Keep in mind that foo's **Write()** function will end up getting called twice – once to compute the amount of memory needed to write the file, and a second time to actually write it:

```
Foo foo = new Foo();
CPWriteMCSFile f(FILE_NAME, FOLDER_NAME);
foo.Write(f);
f.Realize();
foo.Write(f);
```

Let's take another look at AddressBook and see how its ContactArray and Contact classes implement their Write functions. First, here is the Write function for ContactArray:

```
void ContactArray::Write(CPWriteFile& f)
{
      //first write out how many contacts are in the array
      f.WriteWord(GetSize());

      //loop through that many times writing each contact
      for(int i=0; i<GetSize();i++)
      {
            Contact *s = (Contact*)GetAt(i);
            s->Write(f);
      }
}
```

All that happens here is that we write out the total number of contacts that will be stored as an integer, then call each contact's **Write()** function to write the contact. Here is the code for the Contact Class's **Write()** function:

```
// Writes a contact to a file.  Since a contact is just a bunch of
// CPStrings, every CPString just writes itself.
void Contact::Write(CPWriteFile& f)
{
      firstName.Write(f);
      lastName.Write(f);
      phone1.Write(f);
      phone2.Write(f);
      email.Write(f);
      address.Write(f);
}
```

Any class that you create that you wish to save to MCS should also have its own **Write()** function.

CPFile Headers

We mentioned earlier that all CPFiles of type IMU_MCS_TypeMem should have a file header. This header identifies what application uses the file and what type of data is contained in the file. So far we haven't dealt with headers when either reading or writing in the AddressBook example. This is because we haven't yet discussed where the ContactArray class's Write or Read methods are called.

Generally, the ClassPad uses a document/view approach to display data on the screen. A document class handles reading/writing the data, and manages the data while the application is running. A view, on the other hand, manages how this data is displayed.

In the AddressBook example you will notice that there is an AddressDocument class that is derived from CPDocument. This is the AddressBook's document class. It contains the array of contacts and the **Read()** and **Write()** methods to save the array.

It is these **Read()** and **Write()** methods that call the ContactArray object's **Read()** and **Write()** functions and handles the creation of the file header.

To create a file header, we use the CPMEMFileHeader class. The constructor simply takes in a PEGCHAR* of your application's name and the name of the data. You can optionally supply it with a major and minor version number:

```
CPMEMFileHeader (const PEGCHAR *AppName, const PEGCHAR *DataName,
                  PEGCHAR MajorVersion=1, PEGCHAR MinorVersion=0);
```

The class only has two member functions, **Read()** and **Write()**.

```
void  Read(CPReadFile &f);
void  Write(CPWriteFile &f);
```

Once you have created a CPMEMFileHeader, you simply call its **Read()** function if you are reading in a file or its **Write()** function if you are writing out a file.   For example, here is the **Read()** function from AddressDocument.cpp:

```
// Application Type
const PEGCHAR* ADDRESSBOOK_MEMTYPE_HEADER = "AddressBook";

//Data Type
const PEGCHAR* ADDRESSBOOK_MEMTYPE_SAVED_STATE = "Data";

void AddressDocument::Read(CPReadFile &f)
{

      if (f.FileExists())
      {
            //Create the file header with the application and data type
            CPMEMFileHeader header(ADDRESSBOOK_MEMTYPE_HEADER,
                                   ADDRESSBOOK_MEMTYPE_SAVED_STATE);

            // Read in the header (basically reads the header and
            // moves the file pointer to the start of the contact data)
            header.Read(f);

            if(f.IsNotError())
            {
                  // Call the ContactArray’s Read Function
                  // (which in turn calls the Contact’s Read Function)
                  contacts.Read(f);
            }
      }
}
```

As you can see it is simply a matter of creating a header and then reading it in before reading in the contacts in the file. Writing a file is done very similarly – before writing the contacts a header is written to the file:

```
void AddressDocument::Write(CPWriteFile &f)
{
      // Create the header
      CPMEMFileHeader header(ADDRESSBOOK_MEMTYPE_HEADER,
                             ADDRESSBOOK_MEMTYPE_SAVED_STATE);

      // Write the Header
      header.Write(f);

      if(f.IsNotError())
      {
            // Call the ContactArray’s Write function
            contacts.Write(f);
      }

}
```

Notice that these Read and Write methods take a CPReadFile *f* and CPWriteFile *f* as a parameter. The methods that call these functions in AddressWindow.cpp are responsible for actually creating this file:

```
void AddressWindow::Save()
{
      CPWriteMCSFile f(FILE_NAME, FOLDER,IMU_MCS_TypeMem);
      m_doc->Write(f);
      f.Realize();
      m_doc->Write(f);
}

/////////////////////////////////////////////////////////////////

void AddressWindow::Open()
{
      CPReadMCSFile f(FILE_NAME, FOLDER,IMU_MCS_TypeMem);
      m_doc->Read(f);
}
```

It is in this **Save()** function that the document's **Write()** is called twice. The first time is to allocate the memory needed for the file, and the second time is to actually write the file.

To summarize the following must be done on a read:
1) Create a CPReadMCSFile of type IMU_MCS_TypeMem
2) Create the appropriate header
3) Read in the header
4) Read in the data in the file

And you must follow these steps on a write:
1) Create a CPWriteMCSFile of type IMU_MCS_TypeMem
2) Create the appropriate header
3) Write the header
4) Write the data for the file
5) Call **Realize()**;
6) Write the header
7) Write the data for the file

For more information about these classes, please refer to the ClassPad 300 SDK Reference Guide or to the AddressBook example add-in.

More Information

For more information on the topics discussed in this document, refer to the SDK Reference Guide. The SDK also contains many example add-ins that were created specifically to help with the explanation of some of the concepts presented in this document. Working through these examples and making changes to discover how your changes affect the program is an excellent way to get a better grasp of how to program on the ClassPad 300.