ClassPad 300 SDK Tutorial

This document will demonstrate the process of developing, compiling and loading a simple add-in application for the ClassPad 300. This document will present the basic classes and an example approach to add-in development, but will not serve as a detailed reference. Please use the included Reference help file as a more detailed reference.

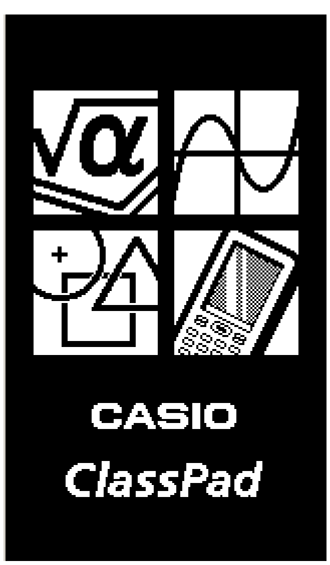


Table of Contents

CLASSPAD 300 SDK ENVIRONMENT 4

DEV-C++ 4
COMPILING AND LOADING AN ADD-IN 5

PROGRAMMING AN EXAMPLE ADD-IN APPLICATION 9

ARCHITECTURE 9

THE MAINFRAME 9
MODULE WINDOWS 10
HELLO WORLD PROGRAM: SCRIBBLE_1 12

USER INTERFACE 13

THE MESSAGE FUNCTION 13
MENUS 14
THE TOOLBAR 15
PEN OR KEYPAD INPUT 16
ADDITION OF UI ELEMENTS: SCRIBBLE_2 17

MULTIPLE WINDOWS 19

ADDING A SECOND WINDOW 19
INVOKING APPLICATIONS 19
SCROLLING 20
MESSAGE BOXES 20
THE STATUS BAR 21
ADDING A NEW WINDOW, DIALOG, AND SCROLLBARS: SCRIBBLE_3 21

INTERACTION BETWEEN WINDOWS 26

DOCUMENTS AND WINDOWS 26
DOCUMENTS AND CHANGING DATA 27
LIVE UPDATING IN THE SCRIBBLE APPLICATION: SCRIBBLE_4 27

SAVING/RESTORING INFORMATION 30

UNDO/REDO 30
SAVING FILES 30
OPENING FILES 32
ADDING SAVE/LOAD AND UNDO TO THE EXAMPLE: SCRIBBLE_5 32

MORE INFORMATION 39

ADVANCED TOPICS 39

UPLOAD ADD-IN TOOL 39
COMPILER AND LINKER 40
USING ASSEMBLY 41
BUILDING FROM THE COMMAND LINE 42

DEBUGGING IN DEV-C++ 44
DEBUGGING ON THE CLASSPAD 45

ClassPad 300 SDK Environment

In this section we will take a brief look at the ClassPad 300 SDK Environment. This includes a description of the SDK's IDE as well as how to compile and load an add-in onto the ClassPad 300. We will assume that you have already installed the SDK into its default directory.

Dev-C++

The ClassPad 300 SDK uses Dev-C++ as its IDE. Dev-C++ is a full-featured Integrated Development Environment (IDE) for the C/C++ programming language. The most recent version available at the time of packaging is included in the SDK installer. However, for bug fixes and new updates be sure to check the developer's web site: http://www.bloodshed.net/. Be aware that the ClassPad DLL has been compiled with MinGW 3.2. Even if you upgrade Dev-C++ it is recommended that you continue to use the MinGW 3.2 compiler.

In general Dev-C++ uses the GNU compiler to create C/C++ programs. A ClassPad add-in, however, is created using the HITACHI SH compiler. In order to seamlessly integrate compilation of an add-in in Dev-C++ a wrapper tool was created to convert GCC syntax into SH syntax. This tool, along with an update to Dev-C++'s ini and cfg file allow you to choose compiler sets that will create an add-in using SHC and/or GCC.

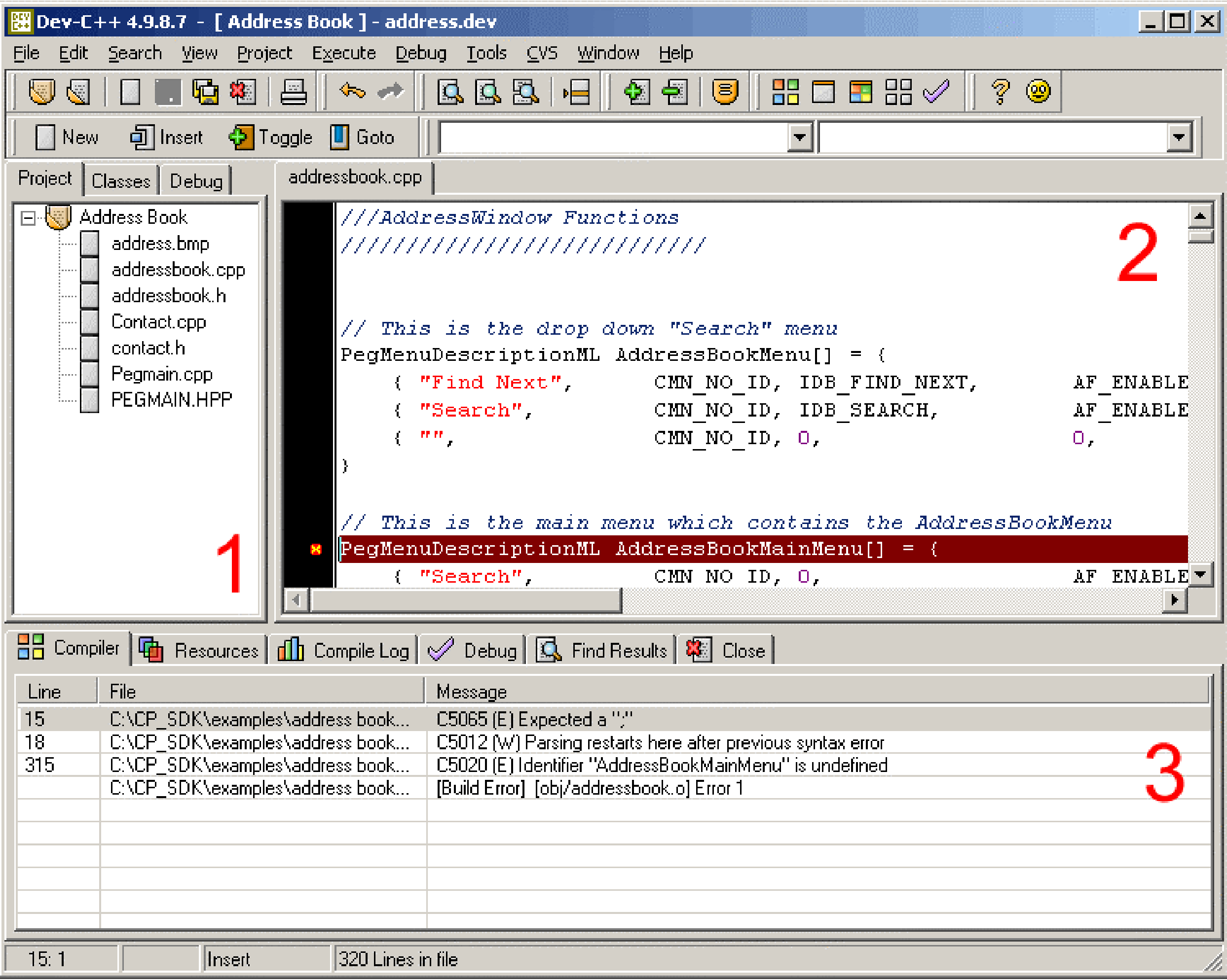
Using Dev-C++

Before concerning ourselves with the details of the wrapper, let's take a moment to go over the basic steps in creating an add-in in Dev-C++.

Dev-C++ will help you organize your projects, and seamlessly compile then with the help of the wrapper tools. Dev-C++ is also used as a front end to the GNU debugger, GDB.

After installing the SDK, you can start Dev-C++ from the Start Menu->Programs-> Dev-Cpp.

To explore the IDE, let's open **address.dev** – the project file for the example Address Book add-in. Click on File->Open Project or File… and browse to **Documents\ClassPad 300 SDK\Examples\AddressBook\address.dev**. We will use this as an example to explain the basics of Dev-C++.



(Fig 1.1) – A typical Dev-C++ screenshot.

Figure 1.1 shows a typical view of Dev-C++. Window 1 lists all of the files in the present project. Double click on any file to open it in the editor (window 2). Window 3 has several tabs that you can click on for different information. Presently it shows errors after compiling this project. Notice that the error selected in window 3 is highlighted in window 2. There is more general information on how to use Dev-C++ on the creator's website at http://www.bloodshed.net/dev/index.html. From here on we will focus on using Dev-C++ for creating an add-in.

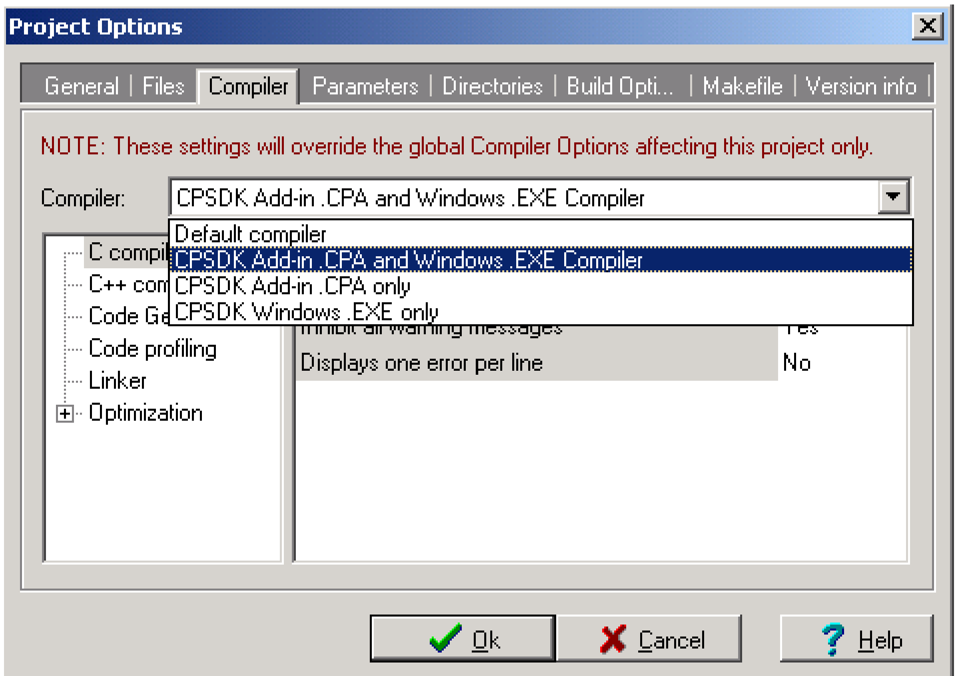
Compiling and loading an add-in

Now that we have loaded the Address Book add-in, let's walk through building and installing it on the ClassPad.

Changing Compiler Sets

The wrapper tools that come with the SDK are capable of doing three different builds: ClassPad Add-in only, Windows .EXE only or both ClassPad Add-in and Windows .EXE at the same time. You can control which compiler is used when by selecting different compiler sets.

For this example we will use the compiler that builds both the Windows .EXE and the ClassPad Add-in – the “CPSDK Add-in .CPA and Windows .EXE Compiler” setting. To ensure that this is the current compiler, click on the Project menu and select Project Options (or press Alt+p). When the Project Options dialog appears, click on the compiler tab.



Here you will see a drop down menu that has the three compiler sets described above plus the Dev-C++ default compiler set (to develop in C/C++). Select “CPSDK Add-in .CPA and Windows .EXE Compiler” from the list to build both a ClassPad add-in and a Windows executable. Remember that the “Default Compiler” is the compiler that comes set up in Dev-C++. You CANNOT use this compiler to build a ClassPad Add-in as it will call GCC (not SH). Also note that this compiler cannot build a Window’s .EXE for the ClassPad because its compiler and linker settings are not set correctly. When building an Add-in you will only use the three compiler sets that the CPSDK installed.

Compiling

Once you have chosen the correct compiler set, building is as simple as clicking on the toolbar. The build button will build any newly modified source files, link them and then build the ClassPad add-in. The build all button will rebuild the entire project. The build and run button will compile the program and then run it automatically. (Note: This only works when using a compiler set that builds a Windows Executable.) **If you switch compiler sets, make sure to do a complete rebuild.** Otherwise GCC may try to use some of SHC’s object files or vice versa.

While the program builds, you can follow the compiler’s output in window 3. The compiler will produce 2 output files in **Documents\ClassPad 300 SDK\Examples\Address Book\outputdir** when it has finished. One of these is a Windows executable that can be run by clicking on the run button in the toolbar. The second is a file called **address.cpa**. This is the compiled add

in file that will be installed on the ClassPad. In general, the .cpa file and .exe file are created in the “outputdir” subdirectory of your project’s directory.

Loading

To load the .cpa file we will use CASIO’s Add-in Installer. From Dev-C++, the add-in installer is accessible from the Tools Menu.

Once you have started the Add-in Installer, plug your ClassPad into your computer using the USB cable. If the ClassPad automatically starts the procedure to transfer via the manager, press the cancel button.

To get the ClassPad ready to receive the add-in, choose communications from the launcher. Next click on the Link Menu -> Install -> Add-in. Finally, you will be asked if you are sending and Add-in App or a new language. Make sure Add-in App is selected and press OK. The ClassPad is now ready to receive the add-in.

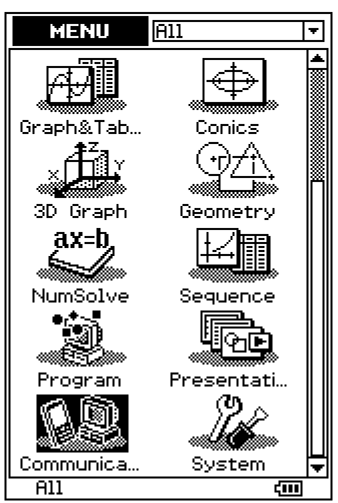


Fig.1.2

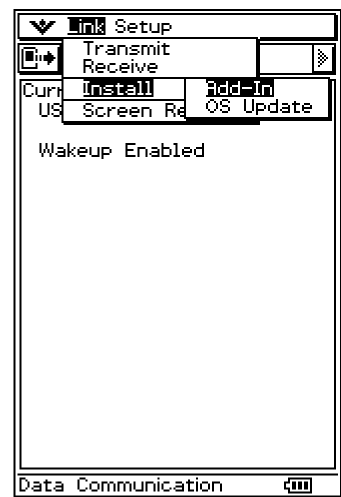
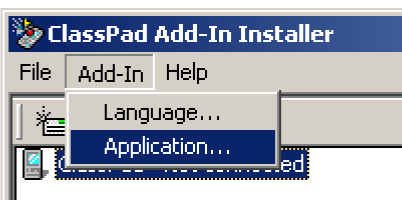


Fig.1.3

Fig. 1.4

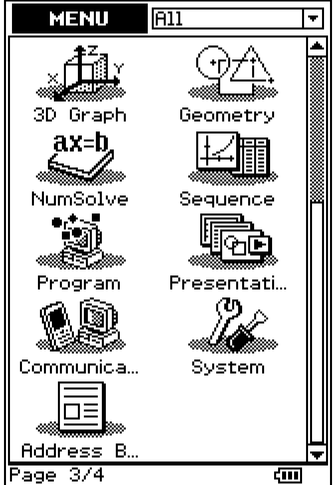
Open Communications from the launcher (Fig1.2). Then click on the Link Menu->Install->Add-in (Fig1.3). Finally, make sure that Add-in App is selected and click OK to begin the transfer (Fig1.4).

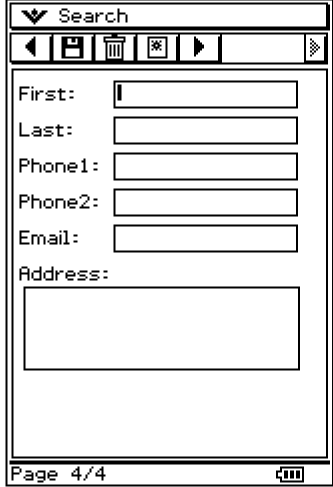


Go back to the Add-in Installer and click on the Add-in -> Application… menu. This will bring up a file Browser. Navigate to **Documents\ClassPad 300 SDK\Examples\AddressBook\outputdir\address.cpa** and press O and the transmission of the add-in from your computer to your ClassPad will begin. When the process has completed you will get a message saying that the connection to the ClassPad is being closed.

You can now disconnect the ClassPad and exit the Add-in Installer.

On the ClassPad, go back to the launcher. At the top of the screen make sure that “All” appears in the drop down next to the Menu title. Now scroll to the bottom of the screen and you will find the Address Book in the launcher. Tap it to start the Address Book add-in.





Two questions might immediately come to mind: Where did the Address Book's icon come from and how did ClassPad know the add-in's name?

Both of these were set up in Dev-C++. Go back to Fig 1.1 and look at the files included in the Address Book project. Notice that one of them is a Bitmap file. When Dev-C++ builds the Add-in, **it will use the bitmap file included in the project as its icon**. If you do not include an icon in the project, a default icon will be used. Note that your icon must be a 45x28 monochrome bitmap.

The name of the Add-in comes from the name of the project. Look at Fig1.1 again and you will see that this project's name is "Address Book", which is what appears on the ClassPad's launcher.

Programming an Example Add-in Application

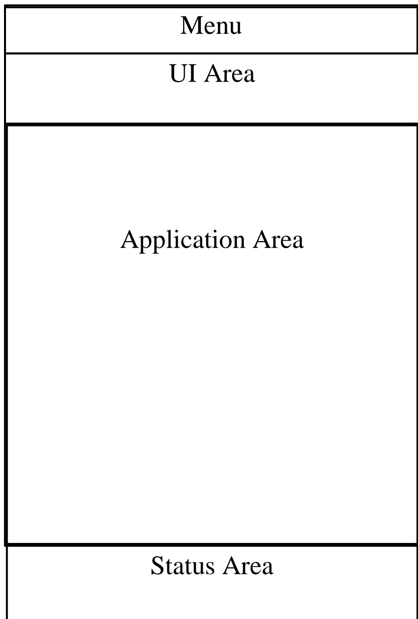
Now that you know how to compile and load projects in the included Development Environment, we will now give some basic programming advice on creating add-in applications. This guide is accompanied by an example add-in: the "Scribble" application, which will eventually let you draw points on the screen, count the number drawn, save and load files, and perform undo/redo operations.

Architecture

In this section we describe the basic structure of an application for the ClassPad, and the necessary components to write a "Hello World" program.

The MainFrame

The base window for every application is called the MainFrame. Applications, menus, toolbars, and the status bar are all loaded into the Mainframe. The basic structure of the Mainframe is as follows:



The application area can be populated by 1 or 2 Application Windows, or by one application window and a virtual keypad.

The Menu area contains the basic Frame menu, merged with the menu for the active application.

A currently active application can put UI elements such as toolbar buttons into the UI area.

The Status area holds a status bar, with text fields that are available for the Mainframe and for the application.

In general, the Mainframe manages which application is active, and also manages interaction between applications. (For example, the mainframe will tell a window to update itself if data that affects that window has been changed. More on this in section 4: Interaction Between Windows.)

Constructing a Mainframe is easy. The constructor is of the form **CPMainFrame(PegRect rect).** The only parameter is a rectangle, which corresponds to the desired size of the mainframe. Since our mainframe should always be the size of the viewable screen, we use the pre-defined rectangle:

```
PegRect rect = {MAINFRAME_LEFT, MAINFRAME_TOP, MAINFRAME_RIGHT,
                MAINFRAME_BOTTOM};
```

(These pre-defined values contain the pixel coordinates for the CPMainFrame rectangle.)

Module Windows

Constructing

The base class for any add-in application is CPModuleWindow. This class contains over-writeable functions needed to customize your UI, and what will take place within your application window. You will need to subclass your application window from CPModuleWindow.

The constructor for this class is as follows:

```
CPModuleWindow( PegRect rect, CPModuleWindow* invoking_window,
                CPDocument* doc, CPMainFrame* frame)
```

Here, **rect** is the rectangle occupied by the window, and **frame** is a pointer to the **CPMainFrame** in which to load the application. We will deal with invoking applications and documents in later sections. For now, setting these parameters to zero will work fine.

Drawing

The **CPModuleWindow** class contains a **Draw()** function which has the instructions on what to display in the window. The **Draw()** function should be overridden in order to add custom features to your new application window. The basic structure of a Draw function is as follows:

```
void YourSubClass::Draw()
{
      BeginDraw();
      DrawFrame();
      //Add other objects to draw here.
      EndDraw();
}
```

All Draw functions must contain **BeginDraw()** and **EndDraw().** The **DrawFrame()** function draws the CPMainFrame into which the window is loaded.

The **Draw()** function is initially called to create your window, and any time your window needs to be re-drawn. This includes if it is ever re-sized, or moved around. Thus, any objects that you want to appear on your window must be specified in the Draw function.

Invalidating

As is typical of GUI programming, when an object needs to be re-drawn, the area of the screen that it occupies must be "invalidated." If the area is not invalidated, a re-draw will not succeed. In most cases, the CPMainFrame will take care of invalidating the proper portion of the screen (e.g. if windows are swapped, moved around, or re-sized.) However this is not always the case; if you find yourself *explicitly* calling the **Draw()** function, or a similar function that draws objects on the screen, you should Invalidate the rectangle which will be re-drawn.

To do so, use the function **Invalidate(PegRect rect)**. A common usage is: **Invalidate(mClient)**. (mClient is a protected member of the base class PegThing, and corresponds to the rectangle occupied by the window. Thus, **Invalidate(mClient)** specifies that the whole window will be re-drawn.

PegAppInitialize

The PegAppInitialize() function is the "main" function for ClassPad applications. Every application must contain this function. In it, you should construct a mainframe, construct any windows, and load these windows into your mainframe. A typical construction is as follows:

```
void PegAppInitialize(PegPresentationManager *pPresentation)
{
      // Create the MainFrame
      PegRect Rect;
      Rect.Set(MAINFRAME_LEFT, MAINFRAME_TOP, MAINFRAME_RIGHT,
               MAINFRAME_BOTTOM);
      CPMainFrame *mw = new CPMainFrame(Rect);

      // Create your window(s)
      PegRect ChildRect = mw->FullAppRectangle();
      CPModuleWindow *swin = new CPModuleWindow(ChildRect,0,0,mw);

      // Load the window into the MainFrame
      mw->SetTopWindow(swin);

      // Set a main window for this module.
      mw->SetMainWindow(swin);

      // Add the MainFrame to the Peg Presentation Manager
      pPresentation->Add(mw);
}
```

The functions **FullAppRectangle()**, **BottomAppRectangle()**, and **TopAppRectangle()** simply return rectangles for the full screen, the bottom half, and top half respectively.

It is a good idea to specify a "main window" for your application, which cannot be closed from the CP Menu. We do this with **SetMainWindow**.

Finally, the last line of the function adds the MainFrame to the Peg Presentation Manager which manages the entire platform.

Hello World Program: Scribble_1

We can now use this basic structure to create a "Hello World" application. This will be the first step in creating a basic add-in application. Our application will be called "Scribble", and will eventually allow you to draw on the screen.

First, we derive a window class for this add-in. It will be called ScribbleWindow. For now, this class needs only a constructor, and a Draw function.

```
class ScribbleWindow : public CPModuleWindow
{
public:

	ScribbleWindow(PegRect rect, CPMainFrame* mf) :
		CPModuleWindow(rect,0,0,mf) { }
	void Draw();
}
```

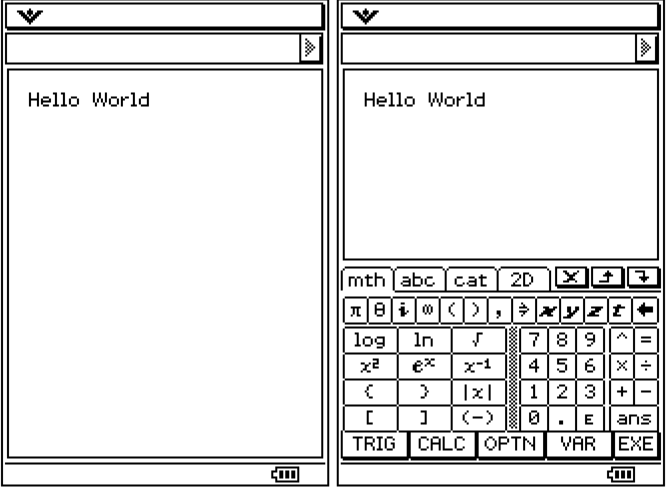
An extra line will be added to the **Draw()** function in order to display "Hello World" in the window.

```
void ScribbleWindow::Draw()
{
	BeginDraw();
	DrawFrame();
	PegPoint pp = {10,10};
	PegColor col = BLACK;
	DrawTextR(pp,"Hello World",col,PegTextThing::GetBasicFont());
	EndDraw();
}
```

The **DrawTextR** function draws the text "Hello World" starting at the location defined by the coordinates of PegPoint pp relative to the top right corner of the window. **DrawText** will do the same, only in absolute coordinates. The third and fourth parameters of this function are the text color, and the font.

Finally, we just need to add a **PegAppInitialize()** function, create a MainFrame, and a ScribbleWindow. We use basically the same code as listed in the template PegAppInitialize, except we replace CPModuleWindow with ScribbleWindow. (In our example code, the **PegAppInitialize()** function is kept in a separate file called PegMain.cpp.)

You can access this example, and build it by compiling and loading the project called **Scribble_1.dev** (inside the Scribble_1 subdirectory). Below is a screenshot of our Hello World window. Notice that there is a system menu in the upper right, but no other UI elements have been added. Also notice that pressing the "Keyboard" button, or selecting Keyboard from the system menu brings up the soft keyboard.



Screenshots from our Hello_World application.

User Interface

The Message Function

The ClassPad receives User input through the **Message** function. Signals are sent to this function when user input is received. These signals are then processed, and the program will then take the appropriate course of action.

The **Message** function is a member of CPModuleWindow, and should be overridden to include any UI signals your application may send. The typical construction is as follows:

```
SIGNED YourModuleWindow::Message(const PegMessage &Mesg)
{
      switch(Mesg.wType)
      {
      case ID1:
            // Put what to do when a message with ID1 is received here
            break;
      case ID2:
            // What to do when a message with ID2 is received.
            break;

      // Other signals are managed by the base classes
      default:
            return CPModuleWindow::Message(Mesg);
      }
      return 0;
}
```

The message ID's are really just numbers, and should be defined somewhere in your project. When you create UI elements of your project, you will assign an ID to each element, it is these ID's that will then be processed by the **Message** function.

Menus

Now that we have a message function to handle User input, we need to define some areas where a user can provide that input.

Menu Description

You first must define a menu description. This will define the drop down menus at the top of the screen. Our menu will have the form:

```
PegMenuDescriptionML ScribbleMainMenu[] = {
      {"Item2",  CMN_NO_ID,    0,   AF_ENABLED,      SubMenu1 },
      {"Item1",  CMN_NO_ID,    0,   AF_ENABLED,      SubMenu2 },
      {"",       CMN_NO_ID, 0,      0,               0}
};
```

As you can see, this declaration contains a list of PegMenuDescriptionML items. Each structure contains a few important parameters. The first parameter is the name of the menu item, and what will be displayed on the screen. *The third parameter is the ID of the signal that will be sent to the message function*. (Since this is only the top menu and the actual menu items will be contained in sub menus, we have left these as zero.) The fifth parameter is the name of the sub-menu that will be opened by clicking on this item. We would then have to define descriptions for these sub-menus as well. For example:

```
PegMenuDescriptionML SubMenu1[] = {
      {"Choice2",   CMN_NO_ID, ID_CHOICE1,          AF_ENABLED,      NULL },
      {"Choice1",   CMN_NO_ID, ID_CHOICE2,          AF_ENABLED,      NULL },
      {"",          CMN_NO_ID,     0,               0,               0}
};
```

The second parameter deals with multi-language support. In this example the value is set to CMN_NO_ID. This means that regardless of the current language of the ClassPad, the menus will always read "Item1" and "Item2". If you are creating menus with common phrases (such as "Copy", "Paste", "Cut", etc) you can use the ClassPad defined language IDs found in CPLangDatabase.h to support multiple languages. For example, this menu would read "Cut" and "Paste" in the current language:

```
PegMenuDescriptionML SubMenu1[] = {
      {NULL,        CMN_MENU_ED_CUT,   ID_CUT,    AF_ENABLED,      NULL },
      {NULL,        CMN_MENU_ED_PASTE, ID_PASTE,  AF_ENABLED,      NULL },
      {"",          CMN_NO_ID,     0,             0,               0}
};
```

We will use some of these common language IDs when creating the menus in Scribble.

The fourth parameter is a style flag. For our purposes, it will work fine as the value given above.

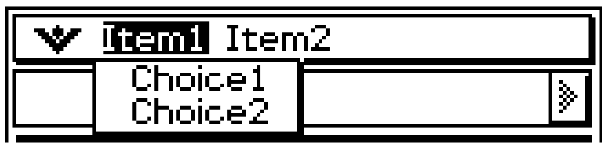
Note: All menus should be terminated by a blank menu item (the third item in the above description.)

The GetMenuDescriptionML Function

Finally, once we have defined our menus, we must add the **GetMenuDescriptionML()** function to our module. This function should return a pointer to the topmost menu. For example:

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

Once we have done this, we have the working menu system shown below. Selecting "Choice1" will send signal ID_CHOICE1 to the Message() function, and "Choice2" will send ID_CHOICE2.



Screenshot of the menu described above

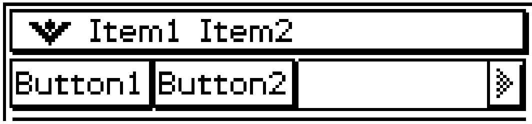
The Toolbar

We can add items to the toolbar in a similar fashion. To do so, we first need to override the **AddUI()** member function of CPModuleWindow. This function is called whenever the UI area of the MainFrame needs to be redrawn. Adding buttons to this area is quite easy; a typical method is shown below:

```
void ScribbleWindow::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);
}
```

In this example, we have chosen to add "text buttons", buttons that contain text. The PegTextButton constructor's first two arguments are the coordinates of the upper left corner of the button (relative to the top left corner of the window.) The third argument is the name that will be displayed on the button. *The fourth argument is the ID that will be sent to the message function when the button is clicked.* Again, the fifth argument is a style flag that we'll leave alone.

Once we have defined our buttons, we add them to the UI window by using **m_ui->AddToolbarButton(b).** m_ui is a protected member of CPModuleWindow, and is a pointer to the UI window. A screenshot of this toolbar is shown below.



Screenshot of the toolbar described above

It is also possible to add bitmap buttons. You can create them using the following method:

```
PegRect rr = GetToolbarButtonRect();
PegBitmapButton *b3 = new PegBitmapButton(rr,&gbPegBitmap,BUTTON_ID);
```

Notice the **PegBitmapButton** constructor takes three parameters: a rectangle that corresponds to the size, a pointer to a **PegBitmap** and the ID that will be sent to the message function. (There is a fourth parameter, which is a style flag, but we'll use the default.)

There is a tool included with the SDK that will convert monochrome bitmaps into **PegBitmaps**. They can then be used to create such buttons. Refer to the section on Scribble_2 for an example of creating a toolbar with **PegBitmapButtons**.

Pen or Keypad Input

In addition to menus and toolbars, we also need to handle user input from the pen and keyboard.

Pen Input

CPWindow has virtual member functions to handle pen input. These are:

```
void OnLButtonDown(const PegPoint & p);    // Called when the pen is first put
                                           // down on the screen.
void OnLButtonUp(const PegPoint & p);      // Called when the pen is picked up
                                           // off the screen.
void OnPointerMove(const PegPoint & p);    // Called when the pen is moved
                                           // around on the screen.
```

The PegPoint p is the location of the pen when the event is received. Simply override these functions in your subclass of CPModuleWindow. Then, whenever a pen event takes place *in that window*, the system will call the appropriate function. These functions are window-specific, so you will have to override these functions for each window in which you wish to handle pen input.

Keyboard Input

The CPWindow function for handling character input is:

```
void OnChar(const PegMessage & Mesg);  // Called when keyboard input is
                                       // received
```

Again, you will need to override this function in order to handle keyboard input. This function will be called when a key is pressed on the ClassPad's hard or soft keyboard.

For ASCII input, **Mesg.iData** will contain the character code for the key pressed.

Addition of UI Elements: Scribble_2

Using the techniques outlined above, we will add the following capabilities to our example Add-in:

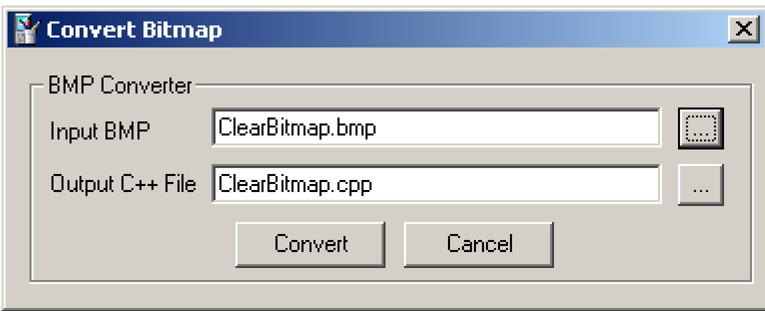
1. Add a **Message** function to handle signals from the menu and toolbar
2. Add a "Draw" menu. This menu will contain a single item "Clear" which will clear the screen
3. Add a button to the toolbar. This will also Clear the screen
4. Add the capability of handling pen input so that a point will be drawn in the window whenever the user touches the pen to the screen, or moves the pen around on the screen.

You can build this stage of the application by compiling and loading the **Scribble_2.dev** project (inside the Scribble_2 subdirectory). Steps 1 and 2 are done almost exactly as shown above.

Adding a Bitmap Button to the Toolbar

In order to add a bitmap button, we must first create a **PegBitmap** object. We first begin with a 19x13 pixel monochrome bitmap. (Ours is called Clear.bmp, and is located in Scribble_2\bitmaps).

To convert **ClearBitmap.bmp** into a **.cpp** file you can use the BMP Converter tool located under the Tools menu of the SDK.



A screenshot of the BMP Converter tool.

Notice that the **.cpp** file defines the **PegBitmap** as **gbClearBitmap**. We can now use this to create a toolbar button in **Scribble.cpp:**

```
extern PegBitmap gbClearBitmap;

void ScribbleWindow::AddUI()
{
	PegRect rr = GetToolbarButtonRect();

	PegBitmapButton *b3 = new PegBitmapButton(rr,&gbClearBitmap,IDB_CLEAR);
	m_ui->AddToolbarButton(b3);

}
```

Notice that we have given the button and the Clear menu item the same ID: IDB_CLEAR. (They both perform the same task.)

Adding a Child Window to the ScribbleWindow

For reasons that will become more obvious in the next section when we discuss scrolling, it is desirable to have all the points drawn within a child window of the ScribbleWindow. We design a separate class called DrawWindow that will keep track of the points drawn with the pointer. The class declaration is shown below:

```
class DrawWindow: public CPWindow
{
protected:

      PegPoint* m_pointlist;
      int m_pointcount;

public:
      DrawWindow(PegRect rect);
      ~DrawWindow();

      // Overwritten function to handle pointer events
      virtual void OnPointerMove(const PegPoint & p);

      // These are functions specific to this add-in,
      // in charge of drawing the points
      void DrawPoint(const PegPoint & p);
      void SavePoint(const PegPoint & p);
      void Draw();
      void ClearPoints();
};
```

As you can see, this class contains a pointer to an array where the coordinates of all the points that have been drawn are stored. The class also keeps track of the total number of points.

Notice that we have only overridden **OnPointerMove** in order to draw the points. This is because **OnPointerMove** is called immediately after an **OnLButtonDown,** so we don't need to override it.

Once we have this class in place, we must add it as a child to the ScribbleWindow class. We first add a pointer to the window as a member of ScribbleWindow, and add the following to the ScribbleWindow constructor:

```
      rect.wBottom -= 1;
      rect.wTop += 1;
      m_win = new DrawWindow(rect);
      Add(m_win);
```

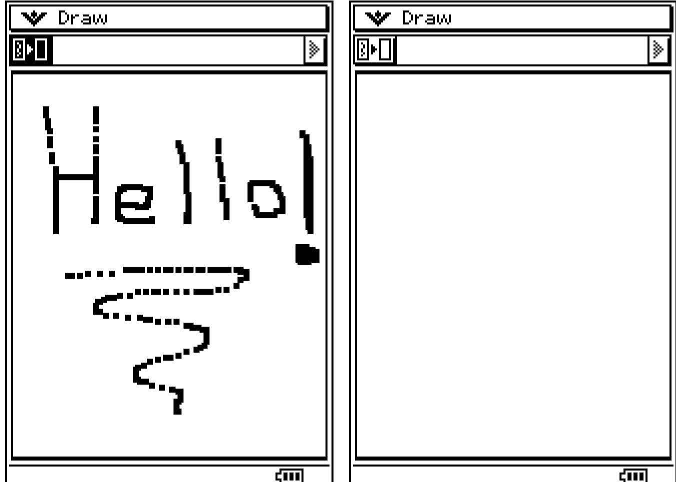
(Here **m_win** is the pointer to the DrawWindow.) We use the command **Add** to add this second window as a child of the first. We make the DrawWindow almost the same size as the full application.

Finally, once we have added a child window to our ScribbleWindow, we must modify the **Draw()** function .

```
void ScribbleWindow::Draw()
{
      BeginDraw();
      DrawFrame();
      DrawChildren();
      EndDraw();
```

```
}
```

The addition of the **DrawChildren()** function ensures that all child windows will be drawn when the parent is re-drawn. Screenshots of our example are shown below.



Writing drawn with "Scribble" is cleared when the "Clear" button is pressed.

Multiple Windows

Adding a Second Window

As stated in the introduction, the ClassPad is capable of displaying two applications windows at the same time. Adding a second window is done almost identically as creating the first window. The window should be a subclass of CPModuleWindow, should be constructed to be the appropriate size, and should then be loaded into the MainFrame. The MainFrame will take care of re-sizing the primary window. The following code is typical for adding a new window to the bottom of the screen.

```
PegRect ChildRect = mw->BottomAppRectangle();
CPModuleWindow *swin = new CPModuleWindow(ChildRect,0,0,mw)
mw->SetBottomWindow(swin);
```

Here, mw is the pointer to the CPMainFrame where the window will be loaded. (The CPModuleWindow function **GetMainFrame()** can be useful when adding a new window from within an existing window. It returns a pointer to the MainFrame in which the current window is loaded.)

Invoking Applications

When you use one window to launch another, it is usually a good choice to make the first window the "Invoking Application" of the second. When a window is closed, its invoking application will replace it (if one exists.) Remember that "main" windows (set with **SetMainWindow**) cannot be closed from the CP menu (at the upper left corner of the screen.)

In order to set an invoking application, we just have to modify the constructor of the new window. Recall that the constructor for CPModuleWindow has the following form:

```
CPModuleWindow( PegRect rect, CPModuleWindow* invoking_window,
                CPDocument* doc, CPMainFrame* frame)
```

Up until now, we have always set the second parameter to zero. If instead we enter a pointer to the invoking window as the second parameter, the invoking window will replace the new window when it is closed. For an example, see the section below where we apply these concepts to the Scribble Application.

Scrolling

Adding scroll bars to your window is very simple, provided your window is structured correctly. In general, the following line needs to be added to the constructor for the window:

```
SetScrollMode(WSM_AUTOSCROLL);
```

The parameter is one of several: WSM_AUTOSCROLL adds automatic horizontal and vertical scrollbars. WSM_AUTOVSCROLL and WSM_AUTOHSCROLL add only vertical and horizontal scrollbars respectively.

Once you have added the above line to your window's constructor, scrollbars will be added to the window *so that all of the window's children can be viewed.* This is why we created a second draw window in the Scribble Application. By making the DrawWindow a child of the ScribbleWindow, *scrollbars will automatically be added to the ScribbleWindow so that the entirety of its child DrawWindow can be viewed*. (By making the DrawWindow the size of a full application, scrollbars will only be necessary when the window is re-sized in order to make room for another window.)

If we had drawn the points within the ScribbleWindow, scrollbars would not be added to the window because the points are not considered children of the ScribbleWindow.

Message Boxes

Adding message boxes to your application is a simple two-step process. Message boxes have their own class: PegMessageWindow. There are three constructors; we will only use one here:

```
PegMessageWindow(const PegRect &Rect, const PEGCHAR *Title, const PEGCHAR
      *Message=NULL, WORD wStyle=MW_OK|FF_RAISED, WORD wStyle2=NULL, PegBitmap
      *pIcon=NULL, PegThing *Owner=NULL)
```

The first two parameters specify the dialog's position (rectangle) and its title. The third parameter is the message you want to display. The forth and fifth parameters are style flags, we'll use the default as usual. The sixth parameter is an icon for the message box, and the seventh specifies whom this message box reports to. We will not use these features.

In general, once you have created a message box, the second step is to call the function **Execute()** which will launch the dialog. The return value of the **Execute()** function will be the

ID of the button clicked. This is useful for determining which option the user has selected. These techniques will be illustrated below.

Important: Note that PegMessageBoxes are self-deleting objects. They delete themselves after they are closed. *Thus, you must make sure that the dialog is created with the **new** operator, but you do not have to worry about **deleting** the object. Simply call the **Execute()** function; everything else will be taken care of.*

The Status Bar

Thus far, we have not addressed the status area located at the bottom of the application screen. This area is often useful for displaying extra information in your application. Adding text to this area is easy to do because of a pointer to the area **CPModuleWindow::m_status_bar**.

This is a protected variable, but you can gain access to it through the function: **GetStatusBar()** which returns a variable of type **PegStatusBar*.**

It is then easy to add text with the function **SetTextField()** which adds text to the already created status bar. (The bar is created when you first create your module window.) To do so, pass in two arguments: the first argument should be "1" denoting the one (and only) text field in the status bar. The second argument should be the text you would like to add.

For example, to add the text "Status: OK" to the status bar, a typical construction is as follows:

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

	// Set the text
	bar->SetTextField(1,"Status: OK");
}
```

Then, of course we would have to call this function from a convenient location to change the status bar's text.

We will use a similar construction in our example. Now that we know how to create multiple windows, we will use the status bar to display which window we are in.

Adding a New Window, Dialog, and Scrollbars: Scribble_3

Using the above techniques, we will add the following functionality to the Scribble Application:

1. The ability to launch a second window that will display the current number of points in the DrawWindow.

2. Make the ScribbleWindow the invoking window for the new window. So the ScribbleWindow will replace it when it is closed.

3. Turn scrolling on in the ScribbleWindow, so scrollbars will be added when the window is resized.

4. Create a status bar for each window which displays which window currently has focus, and the current position (full screen, top window, or bottom window) of the window.

5. Add a dialog box that will pop up when the "Clear" button is pressed. This dialog will ask if the user wants to clear all the points. If they select "OK", the points will be cleared.

You can build this version of Scribble by opening the **Scribble_3.dev** project located inside the Scribble_3 subdirectory. An explanation of important changes is given below.

Creating the Count Window

The second window should again be sub-classed from CPModuleWindow. For now, the only function necessary is **Draw().** Notice that the constructor for the CounterWindow has a parameter for the Invoking window:

```
CounterWindow(PegRect rect, CPModuleWindow* invoking_window,
CPMainFrame* frame) :CPModuleWindow(rect, invoking_window, 0, frame) {}
```

Launching the Count Window

In order to launch the count window, we added a second tool bar button to the Scribble Window, and modified the **Message()** function in order to handle this button. We then added a new function **OnCount()** which actually creates the CounterWindow:

```
void ScribbleWindow::OnCount()
{
	CPMainFrame * mf = GetMainFrame();
	PegRect ChildRect = mf->BottomAppRectangle();
	CounterWindow* cwin = new CounterWindow(ChildRect,this,mf);
	mf->SetBottomWindow(cwin);
}
```

Notice that **this** is set as the invoking window; we have made the ScribbleWindow the invoking application for this window.

Drawing the Count Window

The only function that is necessary for this new window at the moment is **Draw()**. The syntax is below. Notice the use of **GetInvokingWindow()** as a way to get a pointer to the ScribbleWindow. Also note that we have added a new function **CountPoints()** to ScribbleWindow which returns the total number of points on screen.

```
void CounterWindow::Draw()
{
	BeginDraw();
	DrawFrame();

	// Get the number of points from the DrawWindow
	ScribbleWindow* invoker = (ScribbleWindow*) GetInvokingWindow();
	int number_of_points = invoker->CountPoints();

	// Convert number of points to string format
	unsigned char count[5];
	CP_IntToString(number_of_points,count);
```

```
        // Draw a string displaying the number of points.
        CPString str = "Number of Points: ";
        str += (char*) count;
        PegPoint p = {10,10};
        PegColor color = BLACK;
        DrawTextR(p,str.Text(),color, PegTextThing::GetBasicFont());

        EndDraw();
}
```

Also note that the point count will only be current right after launching the window. There is no functionality to update the count as more points are drawn yet. In the above function we have used the utility class CPString. For more information on utility classes, see the SDK Reference Guide

Adding Scrollbars

We have added the **SetScrollMode(WSM_AUTOVSCROLL)** to the constructor of the ScribbleWindow. Notice that when the window is re-sized, you can scroll to see all points that have been drawn.

Adding the Status Bar

We have added the function **SetScribbleStatusBar()** to the scribble window and **SetCounterStatusBar()** to the counter window to set the status bar. These functions are very similar – they both update the status bar to reflect the current screen state. Since each CPModuleWindow derived class has a status bar, both classes need their own function to set their status bar. The code for **SetScribbleStatusBar()** looks like this:

```
void ScribbleWindow::SetScribbleStatusBar()
{
        CPMainFrame *mf = GetMainFrame();
        PegStatusBar *bar = GetStatusBar();
        if (mf && bar) {
        FrameState state = mf->State();
        CPString status = "Scribble: ";
        if(mf->KeypadOn())
                status += "Keypad Open";
        else
        {
                switch (state)
                {
                        case FS_SINGLE_APP:
                                status += "Full Screen";
                                break;

                        case FS_TWO_APPS:
                                if (mReal == mf->TopAppRectangle())
                                        status += "Top Window";
                                else
                                        status += "Bottom Window";
                                break;
                }
        }
        bar->SetTextField(1,status);
        }
}
```

Here we have used the CPMainFrame functions **TopAppActive(), State(),** and **KeypadOn()** to determine the state of the window. Then, we use **GetStatusBar()** and **SetTextField()** to set the text in the status bar. (Again we have used the CPString utility class. For more information, see the SDK Reference Guide.)

To avoid calling these functions explicitly from their classes' **Draw()** functions, we create a user defined message, PM_SCRIBBLE_SIZE_CHANGED, that is pushed to the MessageQueue anytime our windows are resized. The reason for doing this as opposed to just calling our function on the PM_SIZE message is that the window state is not updated until **after** the windows have been resized. Therefore, if we try to call status bar functions when the PM_SIZE message is received, it will be too soon. Here is the code in the message function where we intercept the PM_SIZE message and send our own PM_SCRIBBLE_SIZE_CHANGED message:

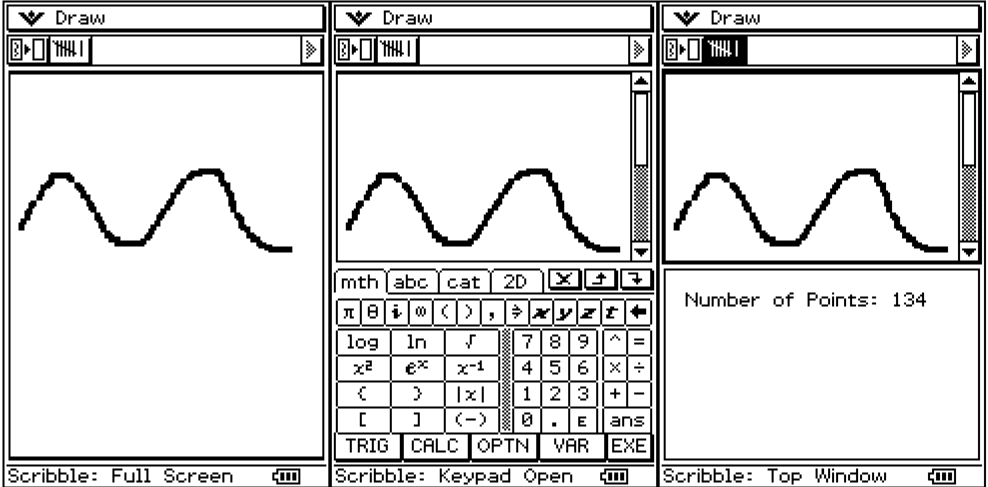
```
SIGNED ScribbleWindow::Message(const PegMessage &Mesg)
{
    switch(Mesg.wType)
    {
      case PM_SIZE:
              CPModuleWindow::Message(Mesg);
              {
                    PegMessage msg(this,PM_SCRIBBLE_SIZE_CHANGED);
                    MessageQueue()->Push(msg);
              }
              break;
:
:
}
```

Message type PM_SCRIBBLE_SIZE_CHANGED is defined in Scribble.h.

Pushing this message will by itself will not accomplish anything. We must then catch the PM_SCRIBBLE_SIZE_CHANGED message in ScribbleWindow's and CoutnerWindow's **Message()** functions, and then call the class function to update the status bar. Here is the portion of ScribbleWindow's **Message()** function that does this:

```
SIGNED ScribbleWindow::Message(const PegMessage &Mesg)
{
      switch(Mesg.wType)
      {
:
:
            case PM_SCRIBBLE_SIZE_CHANGED:
              SetScribbleStatusBar();
              break;
:
:
      }
}
```

Screenshots of the new Scribble Application are shown below. Notice that the status bar at the bottom displays the current window position.



Scrollbars, a status bar, and a new Counter Window added to the Scribble Application

Adding a Message Box

We have added the following function to ScribbleWindow:

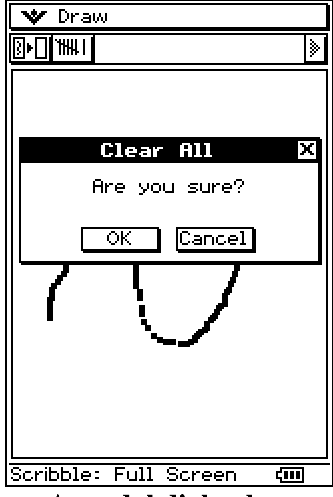
```
WORD ScribbleWindow::ClearAllPopup()
{
      PegMessageWindow *pDlg = new PegMessageWindow(GetLang(CMN_MENU_ED_DEF),
            GetLang(CMN_CLEARALL_SURE), MW_OK|MW_CANCEL|FF_RAISED);
      return pDlg->Execute();
}
```

Notice that we define the title and message using the GetLang() function and an ID from CPLangDatabase.h. This will cause the message box to display the correct message depending on what the current language of the ClassPad is. Also notice that the last line calls the **Execute()** function, which will return the ID of the button clicked when the dialog is closed.

We use this fact in the **ScribbleWindow::OnClear()** function, which is called when the "Clear" button is pressed. We have modified it to include the following:

```
void ScribbleWindow::OnClear()
{
      // Popup a dialog
      // Only clear points it OK is clicked
      if(ClearAllPopup()==IDB_OK)
      {
            m_win->ClearPoints();
            Redraw();
      }
}
```

Notice that the Clear button now first pops up the dialog box. Because the return value of the **ClearAllPopup ()** function is the ID of the button clicked, we proceed to clear the points only if the "OK" button is clicked. A screenshot of the dialog is shown below.



A modal dialog box

Interaction Between Windows

A logical next step is to have the Point Count update as we draw more points, instead of having to launch the Count Window every time. In order to do this, we need to explore the concept of Documents.

Documents and Windows

Until now, we have left the third parameter in our CPModuleWindow's constructor equal to zero. This is the parameter that specifies a document for the window. Typically, a document is an object that contains all the data for a particular window. We have not needed to use one yet, because our application is simple enough that we can keep track of our data within our window. However, in more complicated applications, the use of a document is an excellent way to keep the data (document) separate from the display of the data (window.)

In order to create a document for our window, we need to subclass CPDocument that has a constructor of the form:

```
CPDocument(CPMainFrame * frame)
```

In our class declaration, we also need to override the following pure virtual functions. (They will not affect our program, but we must have them in order to avoid errors.)

```
virtual WORD DocType()  { return 0; }
virtual WORD Version()  { return 1; }
```

Finally, we must link our Module to the newly created document. This is accomplished with the third parameter of our module's constructor:

```
YourModuleWindow(PegRect rect, CPModuleWindow* invoking_window,
                 CPDocument *doc, CPMainFrame * frame)
      : CPModuleWindow(rect, invoking_window, doc,frame) {}
```

We can then access the document using the **GetDocument()** member function of CPModuleWindow.

Documents and Changing Data

Documents are not only useful as a container for your window's data, they also allow for live updating of windows. This interaction comes through two similarly named functions: **OnDataChanged** (a virtual member of CPModuleWindow), and **OnChangedData** (a member of CPDocument.)

Whenever data is changed within a document, one should call the **OnChangedData** function. Once this function is called, the MainFrame will then call the **OnDataChanged** function for the window which points to the changed document. For clarification, it is the developer's responsibility to:

1. Call **OnChangedData** (a member of the document) whenever data is changed in the document, and you would like to update the corresponding windows.

2. Override **OnDataChanged** (a member of the CPModuleWindow) to provide instructions on how to update the window when data has been changed.

The MainFrame will take care of the rest.

Linking Windows Together

The Document-window structure is such that one document can contain data used in many different windows. Further, when you call **OnChangedData(),** the MainFrame will call **OnDataChanged()** for *every* window which is linked to that particular document. This allows changes made in one window to be viewed in a second, and vice versa.

Live Updating in the Scribble Application: Scribble_4

Using the above technique, we will add the following capability to the Scribble application:

1. Create a new document class, and link it to both the ScribbleWindow, and the CounterWindow.

2. Call **OnChangedData()** and override **OnDataChanged()** so the CounterWindow will update the current point count whenever a point is drawn, or the screen is cleared.

Creating the ScribbleDocument class

Because we have neglected documents until now, our project could use a bit of re-structuring in order to accommodate them. First, we need to create our document class. The class declaration is shown below:

```
class ScribbleDocument: public CPDocument
{
protected:
```

```
	int m_counter;
	PegPoint * m_pointlist;

public:

	// Standard constructor for a document, document must
	// be loaded into the mainframe
	ScribbleDocument(CPMainFrame * frame);
	virtual ~ScribbleDocument();

	// Functions used to get a point information
	CPString GetCountAsString();
	PegPoint GetPoint(int i)	{return m_pointlist[i];}

	// Functions to manage the Point list
	void SavePoint(const PegPoint & p);
	void ClearPoints();

	// Functions to manage the counter
	inline int GetCount()		{return m_counter;}

	// These are pure virtual functions that must be overwritten
	virtual WORD DocType();
	virtual WORD Version();
};
```

Notice that the document has completely taken over all management of the points. To do this, we have moved the members m_counter, and m_pointlist here from the DrawWindow, and we have also moved the functions **SavePoint** and **ClearPoints**.

Once we have created the document, we must modify the constructors of the ScribbleWindow, and CounterWindow to accommodate a document. The updated constructors now have one extra parameter—the 3rd parameter now points to the document.

```
	ScribbleWindow(PegRect rect, ScribbleDocument * doc, CPMainFrame* frame)
		:CPModuleWindow(rect,0,doc,frame)

	CounterWindow(PegRect rect, CPModuleWindow* invoking_window,
			ScribbleDocument * doc, CPMainFrame* frame)
		:CPModuleWindow(rect,invoking_window,doc,frame)
```

Finally, in pegmain.cpp, we must create the document, and use it in constructing our windows. (Alternatively, we could use the CPModuleWindow's function **SetDocument()** to link the document to each respective window instead of modifying the constructors.)

We will also want to have access to the document from within the DrawWindow (the document must be updated whenever a point is drawn or cleared.) Because DrawWindow is a CPWindow, not a CPModuleWindow, it can't be linked to the document as above. Instead, we will simply include a pointer to the document as a protected member of DrawWindow.

Restructuring the DrawWindow

Previously, the DrawWindow had been in charge of keeping track of the points. Because we want the document to do this instead, the DrawWindow needs to be changed.

The result of all this shuffling is a cleaner DrawWindow class that is only in charge of drawing the points. Below is the modified DrawWindow class declaration.

```
class DrawWindow: public CPWindow
{
protected:
	// Data abstracted into Document class
	ScribbleDocument * m_doc;

public:
	// Constructor takes a window rectangle and a pointer to the document
	DrawWindow(PegRect rect, ScribbleDocument * doc);

	//Overwritten function to Draw Scribble Data
	virtual void Draw();

	// Overwritten function to handle pointer events
	virtual void OnPointerMove(const PegPoint & p);


	// These are functions specific to this add-in,
	// in charge of drawing the points
	void DrawPoint(const PegPoint & p);
};
```

Allowing for Live Updates

Now, in order to allow for live updates, we need to call **OnChangedData** whenever data in the document changes. We choose to call this function from within the functions **OnPointerMove** (when a point is added) and **OnClear** (when the screen is cleared.) Notice we don't have to override **OnChangedData**, we simply need to call it. It will then call **OnDataChanged** for our windows. The **OnClear** function which now contains **OnChangedData** is shown below:

```
void ScribbleWindow::OnClear()
{
	// Clear Points only if OK is selected from the dialog
	if(ClearAllPopup()==IDB_OK)
	{
		ScribbleDocument* sdoc = (ScribbleDocument*) GetDocument();
		sdoc->ClearPoints();
		Redraw();

		// The document has changed, call OnChangedData.
		// This will update all affected windows
		// by calling OnDataChanged for each one.

		sdoc->OnChangedData(this);
	}
}
```

Finally, we must override **OnDataChanged()** for the CounterWindow, since we need to provide specific instructions about what to do when the document has changed. Since the CounterWindow retrieves the current point count whenever it draws itself, all we need is the function shown below:

```
void CounterWindow::OnDataChanged()
{
	Invalidate(mClient);
```

```
	Draw();
}
```

This simply tells the CounterWindow to re-draw itself whenever the point count changes. This in turn will update the display accordingly.

The Scribble Application is now completely capable of live updating. You can build the example by loading the **Scribble_4.dev** project (located inside the Scribble_4 subdirectory).

Saving/Restoring Information

The last feature we will implement is the ability to save and restore states of the application. This will be useful in implementing Undo/Redo and Save/Load capability.

Undo/Redo

Because some form of the undo mechanism is used in nearly all applications, there is a good deal of functionality already set up to support it.

The relevant class is **CPUndoThing**; it contains most of the necessary functions to implement an undo/redo action. You will have to specify the steps your application will take to actually perform the undo.

The MainFrame carries a pointer to a CPUndoThing. This points to the object that performed the last undoable action. Thus, in order to implement undo functionality, you must make your window (or some other piece of your application) a subclass of **CPUndoThing**.

Secondly, you must call the function **ActivateUndo()** whenever you complete an action that is undoable. **ActivateUndo()** alerts the mainframe that **this** should now be the current undo thing.

Once you have done this, you will need to override the following member functions of CPUndoThing:

1. **Undo()** – This function is called to perform the actual undo. You should include instructions about what steps are needed to perform your undo.
2. **Release() –** This function is called by the Main Frame when the object is no longer the current undo thing. Thus, if possible, you should free up some memory that is used to store the undo state, since the action is no longer undoable.

We will implement undo/redo functionality in the Scribble application at the end of this section

Saving Files

In order to save files into the ClassPad's MCS file system, we make use of the CPWriteMCSFile class.

Creating a CPWriteMCSFile Object

In order to write data, we first need to create a CPWriteMCSFile object. This object has a constructor of the following form:

```
CPWriteMCSFile(const char* name, const char* path=NULL,UCHAR type=0)
```

name and **path** are simply strings that refer to the file's desired name, and folder. The **type** parameter specifies the type of file to be saved. We will always save our variables as type **IMU_MCS_TypeMem**. These variables show up as type "MEM" in the variable manager.

Writing Data to the File

Once you have created the file, simply utilize one of the many "write" members of the base class CPWriteFile to write the data to the file.

WriteInt(int i) –writes an int to the file.
WriteDouble(double xx) – writes a double to the file.
WriteFloat(float xx) – writes a float to the file.
WriteBytes (void* buffer, int nBytes) -- Writes n bytes from the buffer to the file.

Unfortunately there is one extra step before your data is written to the file. This is due to the fact that the **CPWriteMCSFile** is not created with a specific size, thus no memory is allocated for the file when it is created. However, writing the data using the functions above allows the file to keep track of its size. Once everything has been written, call the **Realize()** function to allocate the appropriate memory for the file.

Then, once you have called the **Realize()** function, you must write the data again. This time, since the memory has been allocated, it will actually be written to your file.

The Header for MEM files

Files of type **IMU_MCS_TypeMem** should also include a header that contains their application type, and data type. This header should be the first thing written, and the first thing read out.

To construct and write your header, use the following syntax:

CPMemFileHeader header = CPMemFileHeader("application name","data type");

header.write(f); // f is the CPWriteMCSFile to which you are writing

Example Code

An example of how to write an the integer "count" into a file called "test" is shown below:

```
// Create the CPWriteMCSFile and the header
CPWriteMCSFile f("test","main",IMU_MCS_TypeMem)
CPMemFileHeader header ("test app","test data");

// Write integer the first time to compute the size of the file
header.write(f);
f.WriteInt(count);
```

```
// Call the Realize function to allocate the appropriate memory for the file
f.Realize();

// Write the data for the second time.  This time it is written to memory
header.Write(f);
f.WriteInt(count);
```

This two-step process is always necessary whenever writing data to an MCS file.

Opening Files

In order to load files in from the MCS file system, we make use of the **CPReadMCSFile** class.

Creating the CPReadMCSFile Object

The constructor for a CPReadMCSFile is identical to that for a CPWriteMCSFile:

CPWriteMCSFile(const char* name, const char* path=NULL,UCHAR type=0)

Here name and path are the filename, and folder location of the file, and type is the data type of the file. As before, we will always use a **type** of **IMU_MCS_TypeMem**.

Reading in Data from the File

Once you have created your **CPReadMCSFile** object, simply use one of the following functions to read in the appropriate data type:

int ReadInt()
double ReadDouble()
float ReadFloat()
void ReadBytes (void* buffer, int nBytes) //Read n bytes into the buffer

Reading in the Header for MEM Files

Data should be read in the same order that it was written. For MEM files, the header is the first thing that is written, so we should accordingly read it first.

As before, you need to create an object of type **CPMemFileHeader**, and then use its member function **Read** which takes a **CPReadMCSFile** as its argument**.** See the code of the Scribble Application for an example.

Adding Save/Load and Undo to the Example: Scribble_5

Using the above techniques, we will now add the following capability to the Scribble Application:

1. A simple Undo/Redo function, which will allow the user to undo the last string of points drawn.

2. The ability to save and load files in the application

Implementing the Undo in the Scribble Document

Our undo/redo function will behave as follows: Whenever the user puts the pen down to draw a new string of points, we will save a copy of the point list and point count. These copies will then be restored if the user selects "Undo". We will not allow any other actions (such as clearing the points) to be undoable.

The Scribble Document should be the object that actually performs the undo. Thus, we have to make a few changes to the document. The document will need to carry the current list of points and a count, as well as an undo list of points and a count. Since this is getting a little complicated, it is probably a good time to create a class that abstracts our array of points. This way we will only have to keep up with the current point list and the undo point list in the ScribbleDocument. We call this new class ScribblePointArray and define it as:

```
class ScribblePointArray
{
protected:
      int m_counter;
      PegPoint* m_pointlist;

public:
      // constructor and destructor
      ScribblePointArray();
      ~ScribblePointArray();

      // Size returns the number of points
      int Size()        { return m_counter; }
      // Add a new point
      void Add(const PegPoint& p);
      // clear all points and free up memory
      void Clear();
      // array operator. Get point at index
      PegPoint operator[](int index) const;
      // copy "points"
      ScribblePointArray& operator=(const ScribblePointArray& points);
      // Swap my data and the data from "points"
      void Swap(ScribblePointArray& points);
      // Write data to a file
      void Write(CPWriteFile &f);
      // Read data from a file
      void Read(CPReadFile &f);
};
```

Secondly, we must actually implement the functions that will save the undo state, restore the undo state, and release the undo state. The modified class declaration is shown below.

```
class ScribbleDocument: public CPDocument
{
protected:

      ScribblePointArray m_pointlist;
      // Class for the Undo State
      ScribblePointArray m_undo_pointlist;
public:
:
:
      // Functions to manage the Undo/Redo state
      void SetUndoState();
```

```
	void RestoreUndoState();
	void ReleaseUndoState();
}
```

The **SetUndoState** function will copy the current point count, and point list into the "undo" variables.

The **RestoreUndoState** function will swap the "undo state members" with the current state members.

The **ReleaseUndoState** function will reset the undo state variables to their initial values – freeing up any memory taken up by the undo state. We will call this function whenever the points are cleared from the document. Because we don't want the "Clear" action to be undoable, we should release the undo state whenever this action is performed.

All of these functions are available in the final version of ScribbleDocument.cpp. These tasks are quite straightforward, so we will not list the functions here.

Implementing the Undo in the Scribble Window

Now that we have the desired functionality in the Document, we need to implement the Undo/Redo mechanism from within our application. First, we must subclass our Scribble Window from **CPUndoThing.** We must also override the **Undo()** and **Release()** functions of CPUndoThing which provide instructions on how to actually perform the undo. Finally, we will add a function that saves the current undo state. Portions of the new class declaration are shown below:

```
class ScribbleWindow: public CPModuleWindow, public CPUndoThing
{
protected:

	DrawWindow* m_win;

public:
:
:
	// Undo Functions
	virtual void Undo();
	virtual void Release();
	void SaveUndoState();
}
```

As stated above, all of the work of the Undo will take place within the document. The scribble window is designed simply to handle the user input, pass relevant instructions on to the document, and redraw itself when something changes. This should be evident from the simple implementation of the functions shown below:

```
void ScribbleWindow::Undo()
{
	ScribbleDocument * doc = (ScribbleDocument*) GetDocument();

	// Restore Undo State, and update all dependent windows
	doc->RestoreUndoState();
	doc->OnChangedData(this);
```

```
        // Redraw the window
        Invalidate(mClient);
        Draw();
}

void ScribbleWindow::Release()
{
        ScribbleDocument * doc = (ScribbleDocument*) GetDocument();
        doc->ReleaseUndoState();
}
```

Activating the Undo

We must now select when we want to activate the undo. The only undoable action in our specification is drawing points. Thus, we want to activate the undo whenever the user places the pen down. We've created a function to save the undo state and activate the undo, and called it **SaveUndoState()**. The function is shown below:

```
void ScribbleWindow::SaveUndoState()
{
        // Save the current state in case of Undo.
        ScribbleDocument* doc = (ScribbleDocument*) GetDocument();
        doc->SetUndoState();

        // Notify the Mainframe that this window possesses the current undoable
        // action
        ActivateUndo();
}
```

Notice the call to **ActivateUndo()** at the end of the function. This call must be made whenever you would like an action to be undoable. The function makes **this** the current undo thing. Thus, the **ScribbleWindow::Undo()** will be called when the user performs an Undo.

Finally, we want to set the undo state whenever the user puts the pen down to draw a new string of points. Thus, inside the **DrawWindow** class, we have overridden **OnLButtonDown** and call the **SetUndoState()** function from within it. The code is shown below:

```
void DrawWindow::OnLButtonDown(const PegPoint &p)
{
        // Save the Undo state as the pen is first put down.
        // Undo will then remove the latest scribble
        //(points drawn since the last PenDown)

        ScribbleWindow * parent =
                (ScribbleWindow *) GetMainFrame()->MainWindow();
        parent->SetUndoState();
}
```

Adding the Undo/Redo Menu Item

Finally, we must include a menu item that performs the undo/redo.

```
PegMenuDescriptionML ScribbleEditMenu[] = {
        DECLARE_MENU_ITEM(CMN_MENU_UNDOREDO,        FWM_UNDO)
        { "", CMN_NO_ID,  0, 0, NULL }
};
```

Because the Undo/Redo signal is used frequently, we can simply add a menu item using the code above: **DECLARE_MENU_ITEM(CMN_MENU_UNDOREDO, FWM_UNDO).** This adds a “Undo/Redo” item to the menu, and assigns it the proper ID.

After completing this step, the Undo/Redo mechanism is complete. Screenshots are shown at the end of this section.

Adding the Saving/Loading Functionality to Scribble

Since the **ScribbleDocument** completely describes the state of our application, in order to save the file, we must write the document to a file, and read the document when loading in a file.

For our save/load UI, we will make use of a class called **StorageManager**. This is a dialog box that displays all the files of a particular type in a specified folder. The constructor is shown below:

```
StorageManager::StorageManager(CPString* filename, CPString* pathname,
                               ActionStates action, UCHAR type)
```

filename and **pathname** are strings denoting the name and location of your desired file. You do not need to worry about these values when creating your storage manager, but make sure that you have valid CPString objects that you can pass in.

The third parameter designates whether you will be saving or loading a file. Pass in **STORAGE_ACTION_SAVE** or **STORAGE_ACTION_OPEN** accordingly.

The fourth parameter specifies the type of files to be shown. (Again, we will always use **IMU_MCS_TypeMem.)**

Because the storage manager is a dialog box, we must then call the **Execute()** function to bring it up. Remember that the dialog box will return a value that corresponds to the button pressed when it is closed, and that the dialog will delete itself after calling the **Execute()** function. For the storage manager, we should be expecting the following button ID’s:

IDB_CANCEL, IDB_STORAGE_SAVE, IDB_STORAGE_OPEN.

Writing the Save and Load Functions

We have added functions named **OnSave()** and **OnLoad()** to the ScribbleWindow, and have added message ID’s and menu items as expected. Then, the function implementation uses the storage manager to quickly implement saving and loading. The On**Load** function is listed below:

```
void ScribbleWindow::OnLoad()
{
      CPString folder, name;

      // Create the storage manager window, and call the Execute function to
      // bring it up
      // The filename and foldername are stored in the string variables "name"
      // and "folder" after the dialog
      // is closed.
      StorageManager *manager = new StorageManager(&name,&folder,
            STORAGE_ACTION_OPEN, IMU_MCS_TypeMem);
      int ret = manager->Execute();
```

```
        // Open the file only if the open button is pressed, and both the
        // name and folder strings are not empty.
        if (ret ==IDB_STORAGE_OPEN && name.Length() && folder.Length())
        {
                CPReadMCSFile f(name.Text(),folder.Text(),IMU_MCS_TypeMem);

                // Check that the file exists, and is valid
                if (f.FileExists() && f.IsNotError())
                {
                        ScribbleDocument * doc = (ScribbleDocument *) GetDocument();
                        // Tell the document to read in the data
                        doc->Read(f);

                        // Notify all dependent windows that the document has
                        // changed
                        doc->OnChangedData(this);
                }
                else
                        f.ErrorPopup();
        }

        // Redraw the entire window with the new points
        Invalidate(mClient);
        Draw();
        SaveUndoState();
}
```

Notice that the **name** and **folder** strings are set by actions the user performs while the dialog is open. After the user clicks "Open", the function checks that they have selected a folder and a file, and then creates a **CPReadMCSFile** based on these values. (Additional checking is done to make sure that the file is valid with the functions **f.FileExists(),** and **f.IsNotError**()).

Read and Write Methods for the Document

The above function then calls the **Read** function of the document, which will read in the header and then call our new ScribblePointArray's **Read** function. (This function's definition is shown below.) When creating these functions, make sure that you read and write the data in the same order.

```
void ScribbleDocument::Read(CPReadFile &f)
{
        // Read in the header
        CPMEMFileHeader header(SCRIBBLE_APP_NAME, SCRIBBLE_DATA_NAME);
        header.Read(f);
        m_pointlist.Read(f);
        SetUndoState();
}

void ScribblePointArray::Read(CPReadFile &f)
{
        // Clear out the existing points
        Clear();

        // Read in the point count
        m_counter = f.ReadInt();
        if (m_counter) {
                // Create a new point list of the appropriate size
                m_pointlist = new PegPoint[m_counter];
```

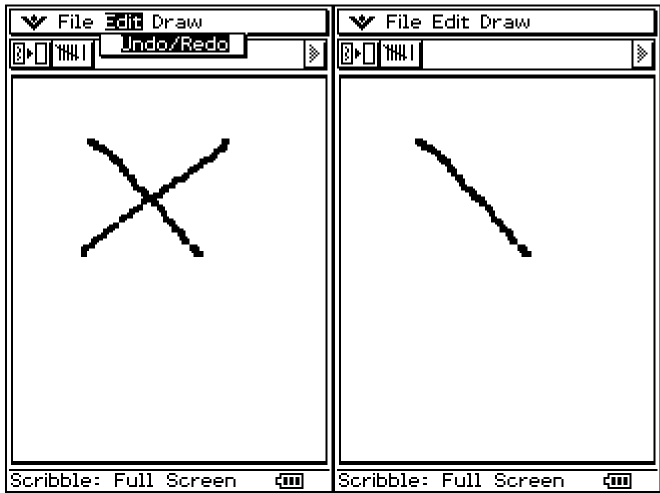
```
            // Read in the points
            PegPoint p;
            for (int ii=0; ii< m_counter && f.IsNotError(); ii++)
            {
                  p.x=f.ReadInt();
                  p.y=f.ReadInt();
                  m_pointlist[ii] = p;
            }
      }
      // If an error happened while reading, then the data is probably bad
      if (f.ErrorFlag()) {
          Clear();
      }
}
```

The procedure for saving the file is quite similar to that shown above, however we must remember to complete the two step write process that was described in the earlier section. Below is **ScribbleDocument::Write** function.

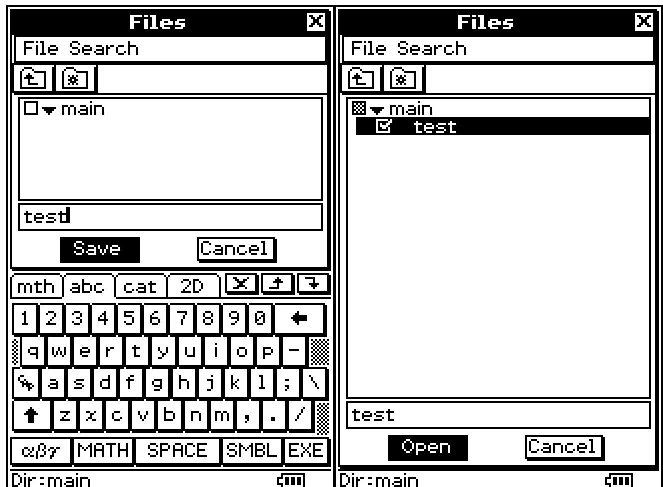
```
void ScribbleDocument::Write(CPWriteFile &f)
{
      // Write once to compute size
      WriteData(f);
      f.Realize();

      // Write a second time
      // this time it is actually written to the allocated memory
      if(f.is_open())
            WriteData(f);

}
```

(The **WriteData** function goes through the details of writing the point count, and point list similar to how the **Read** function reads in this data.) All these functions can be viewed in their entirety inside the **Scribble_5.dev** project (located inside the Scribble_5 subdirectory.

This completes our development of Scribble. You can access all the completed features by loading **Scribble_5.dev**, building the project, and loading it onto your ClassPad. Screenshots of this last phase of the project are shown below.



Performing an Undo



Saving and Loading the file "test"

More Information

The techniques presented in this document are intended to only be a brief introduction to application development for the ClassPad. The ClassPad 300 SDK Programming Guide provides explanations and examples on how to use most of the classes in the SDK. The ClassPad 300 SDK Reference Guide includes full reference on classes and functions available for Add-in development. Please see these documents for more information on programming Add-in applications for the ClassPad 300.

Advanced Topics

Upload Add-in Tool

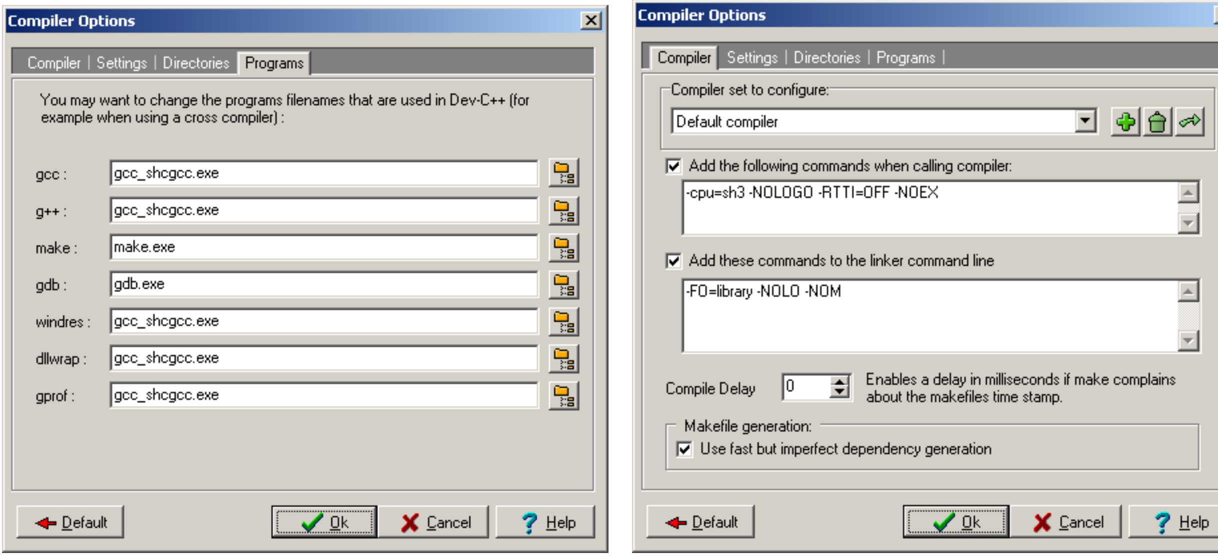
If you grow impatient of browsing to your .cpa and confirming the overwrite each time you upload an Add-in to your ClassPad, then try "Upload Add-in" from the tools menu instead of "Launch Add-in Installer".

Upload Add-in will automatically start the transfer of your project's .cpa file to the ClassPad. Make sure that before starting Load Add-in that your ClassPad is waiting to receive an Add-in Application. When transferring the add-in, any previous add-ins with the same name on the ClassPad will be **overwritten without a warning message**. This tool assumes that you are developing an add-in and will be sending an add-in with the same name to the ClassPad regularly.

Compiler and Linker

Dev-C++ automatically creates a makefile for a project to assist in compilation and linking. Dev-C++ assumes that the user is using GNU's GCC to compile and link a program and creates the makefile using GCC's syntax. Even though the SDK does use GCC to create a windows executable, the compiler that it is uses to compile a ClassPad add-in is **SHC.EXE**. Since **SHC.EXE** does not use the same syntax as GCC, a wrapper is used to convert GCC syntax into SHC syntax.

In Dev-C++, open the **Tools->Compiler Options Menu** and click on the Programs tab. You will see that gcc_shcgcc.exe is listed as the compiler program instead of gcc.exe for the Default Compiler Set. This is the wrapper that will take the commands Dev-C++ sends and convert them into the correct syntax for both GCC.EXE and SHC.EXE.



If you click on the compile tab, you will find a place to send extra arguments to the compiler. The arguments that are listed by default are all in SHC syntax. Any extra options that you wish to send to the SHC compiler must be sent in **SHC.EXE**'s syntax. You can see a list of SHC's options by going to the command line and typing "shc". If you need to include an option that takes a list of arguments, take care not put spaces between the arguments. For example, the syntax for defines are **-DEF=DEFINE1,DEFINE2,DEFINE3** not **–DEF=DEFINE1, DEFINE2, DEFINE3.**

Any options passed to the compiler that are not valid SHC syntax will be sent to the GCC compiler. If there are any options where GCC and SHC share the same syntax, SHC will take precedence.

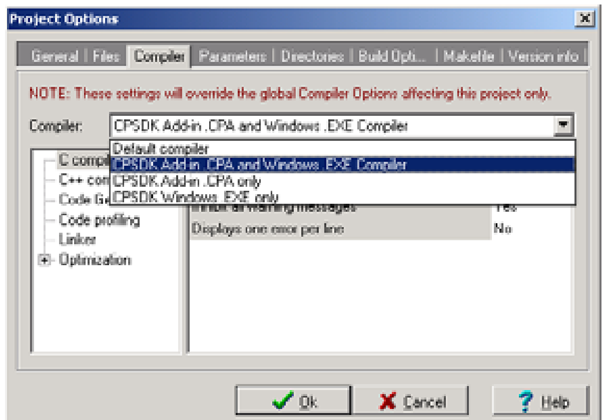
If you wish to send extra arguments to the compiler, be aware that adding extra options to the compiler under Compiler Options will save these options **for all projects**. To add commands to the compiler for the current project, go to Project->Project Options and then click on the parameters tab.

For example, let's say that you want to send a define of GCC to gcc.exe and SHC to shc.exe. To add this to the current project, click on the Project Menu then Project Options. This will bring up

the Project Options Dialog. Then click on the Parameters tab. In the text box under Compiler and C++ compiler type in "-DEF=SHC –DGCC". This will send the –DEF=SHC command to shc.exe and the –DGCC command to gcc.

The wrapper also calls the SHC linker, **OPTLNK.EXE,** and the GCC linker, **LD.EXE**. Options can be sent to the linker in the same fashion that they are sent to the compiler. Once again, make sure that you use OPTLNK.EXE's syntax to send commands to OPTLINK and LD.EXE's syntax to send commands to LD.

After running the linker, the wrapper prepares the add-in that will be installed on the ClassPad. This includes setting the header of the add-in (via **setheader.exe**), adding the name of the add-in (via **putname.exe**) and compressing the add-in (via **compress.exe**). If you'd like to see exactly what the wrapper does, click on the Compile Log tab after building an add-in. All commands that Dev-C++ sends to the wrapper are followed by the actual call to shc.exe then the call to gcc.exe.



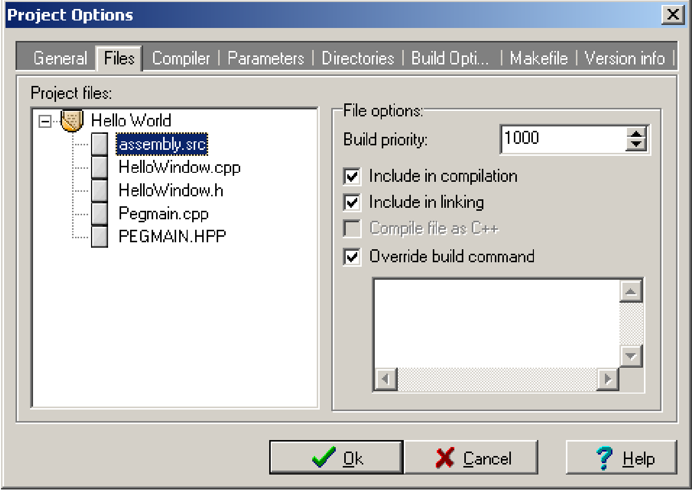
Changing Compiler Sets

If SHC seems to compile correctly while GCC is giving errors, you can choose to only build an add-in and not a Windows executable. To do this, click on the Project menu then Project Options. On the Project Options dialog click on Compiler. At the top you will notice a drop down list labeled "Compiler". Click on this and choose "Add-in Only". This will call a different wrapper that will only compile using shc.exe. If you wish to compile only a Windows executable, that option is also available.

Using Assembly

The wrapper also allows the use of simple assembly files in a project. Not all assembly is supported. If the wrapper is not correctly building an add-in that uses assembly, try building the project from the command line (see the next section).

All assembly files used in a Dev-C++ project must have the extension **.src** and **will only be compiled with SHC**. **Building of assembly files is not supported via the wrapper for GCC.** If you choose to use assembly through the IDE, there is some additional setup that must be done.



Dev-C++ creates a makefile that contains a compile statement for each .c/.cpp file in a project. If a file does not have a .cpp/.c extension then by default there will be no rules to make the file in the makefile. To change this open the Projet->Project Options Menu and click on the Files tab.

Here you will see a list of all files in your project. To include the assembly file in the compilation of your project, select it and check “Include in compilation” and “Include in linking”. The “Override this command” checkbox is automatically selected and “<override this command>” appears in the text box. If you wish to compile the file with the default options that are sent to the compiler, just delete all the text from the textbox. However, if your build requires more options then you can enter the command line call with the arguments needed to the ClassPad assembly compiler, **ashsm.exe**, in this textbox.

Be aware that if you include assembly in your program GCC will not attempt to compile it. You can still build a ClassPad add-in, but cannot build a Windows executable.

Building From the Command Line

If you do not want to use the wrapper, you can build a ClassPad add-in from the command line. Here are the steps that you must follow:

1. **Compile all source files.** The first thing that the wrapper does is to compile all of your source files into object files. For C/C++ code the compiler **SHC.EXE** is used. Here is an example of a command that would compile the file Test.cpp:

```
C:\PROJECT_DIR> shc.exe -OB="outputdir\Test.o" -I="SDK_PATH\cp_include"
-cpu=sh3 -NOLOGO -RTTI=OFF -NOEX  "Test.cpp"
```

If you have any assembly files in your project you must compile them using **ASMSH.EXE.** Here is an example command line call to compile the assembly file Test.src:

```
C:\PROJECT_DIR> ASMSH.EXE  -O="outputdir\test.o" -I="SDK_PATH\cp_include"
"test.src"
```

If you plan on using assembly in your add-in, it is recommended that you build using the command line. While some simple assembly can be made to work with the wrapper, not all assembly files will work.

2. **Link all of your object files.** The next step after compilation is to link all of your object files with the tool **OPTLNK.EXE**. The first thing you need to do is create a command file that lists all of the object files in your project. Each object file should be on its own line and be preceded with “input=”. For example, a project that had the source files Test1.cpp, Test2.cpp and Test3.cpp would have a command file that looked like:

```
input=“outputdir\Test1.o”
input=“outputdir\Test2.o”
input=“outputdir\Test3.o”
```

Name this file objects.sub and pass it to the linker like this:

```
C:\PROJECT_DIR>optlnk.exe -SU="objects.sub" -output=aplmain.lib -FO=library
-NOLO -NOM
```

3. **Link your library file with ClassPad object files and libraries.** Next you need to create an .rld file from your .lib by linking with the ClassPad object and library files. The wrapper

creates and uses a makefile to do this. There is a template for this makefile called **MakeCPA_template.mak** in your SDK's BIN directory. Copy this file to your project location and open it in a text editor.

There are only two lines that you will have to change in this makefile: the location of your SDK and the location of your output directory. When setting these directories, make sure that your paths either do not have spaces or that you **use the short path name**. The place to edit is clearly marked by comments in the makefile:

```
##########################################################################
# EDIT HERE
##########################################################################
##########################################################################

# The ROOT of your SDK installation
#("c:\program files\CASIO\ClassPad 300 SDK" by default)
SDK= C:\PROGRA~1\CASIO\CLASSP~1
# The output directory where you want the .RLD and .MAP files created
OUT_DIR=C:\proj\CPAddins\HELLOW~1\OUTPUT~1

##########################################################################
##########################################################################
```

Once you have successfully edited the makfile, call it from the command line with make:

```
C:\PROJECT_OUTPUT_DIR> make –f MakeCPA.mak
```

This will create the .RLD file and .MAP file in your specified output directory.

Note: If you get any L2310 Warnings, you can safely ignore them.

4. **Set the header and icon for the Add-in.** Next you must set your Add-in's header and icon. This is done using the tool **SETHEADER.EXE** in the SDK\BIN directory. The output is a .CPA file that can be named anything you like. In this example we will name it "Test.CPA":

```
C:\PROJECT_OUTPUT_DIR\> setheader "ADDINAPL.rld" "Test.cpa" -vp0100 -vl1000 -
m2 -pA -o -bI"YourIcon.bmp"
```

5. **Set the Name of the Add-in.** In this step you set the name that will appear on the ClassPad Launcher for your add-in. This time we use the tool **PUTNAME.EXE** in the SDK\BIN directory. We send the executable the .CPA from the previous step and a text string indicating the name of the Add-in:

```
C:\PROJECT_OUTPUT_DIR> putname.exe -p0 -wp "Test.cpa" "My Test"
```

6. **Rename your Add-in and run CPADATAMAKE.EXE.** This step will prepare your add-in for installation on the ClassPad. The command CPADATAMAKE.EXE sets up the file ADDINAPA.BIN, and then appends your add-in to it. Because CPADATAMAKE.EXE expects your file to be named ADDINAPL.BIN, you must rename your .CPA file. The following commands will rename "Test.cpa" and call CPADATAMAKE.EXE:

```
C:\PROJECT_OUTPUT_DIR\>ren Test.cpa ADDINAPL.BIN
```

```
C:\PROJECT_OUTPUT_DIR\>CPADATAMAKE.EXE
```

This will rename ADDINAPL.BIN to ADDINAPA.BIN.

7. **Compress the Add-in.** The final step is to compress the add-in for transfer to the ClassPad. To compress the add-in use COMPRESS.EXE as follows:

```
C:\PROJECT_OUTPUT_DIR\> COMPRESS.EXE -r ADDINAPA.BIN
```

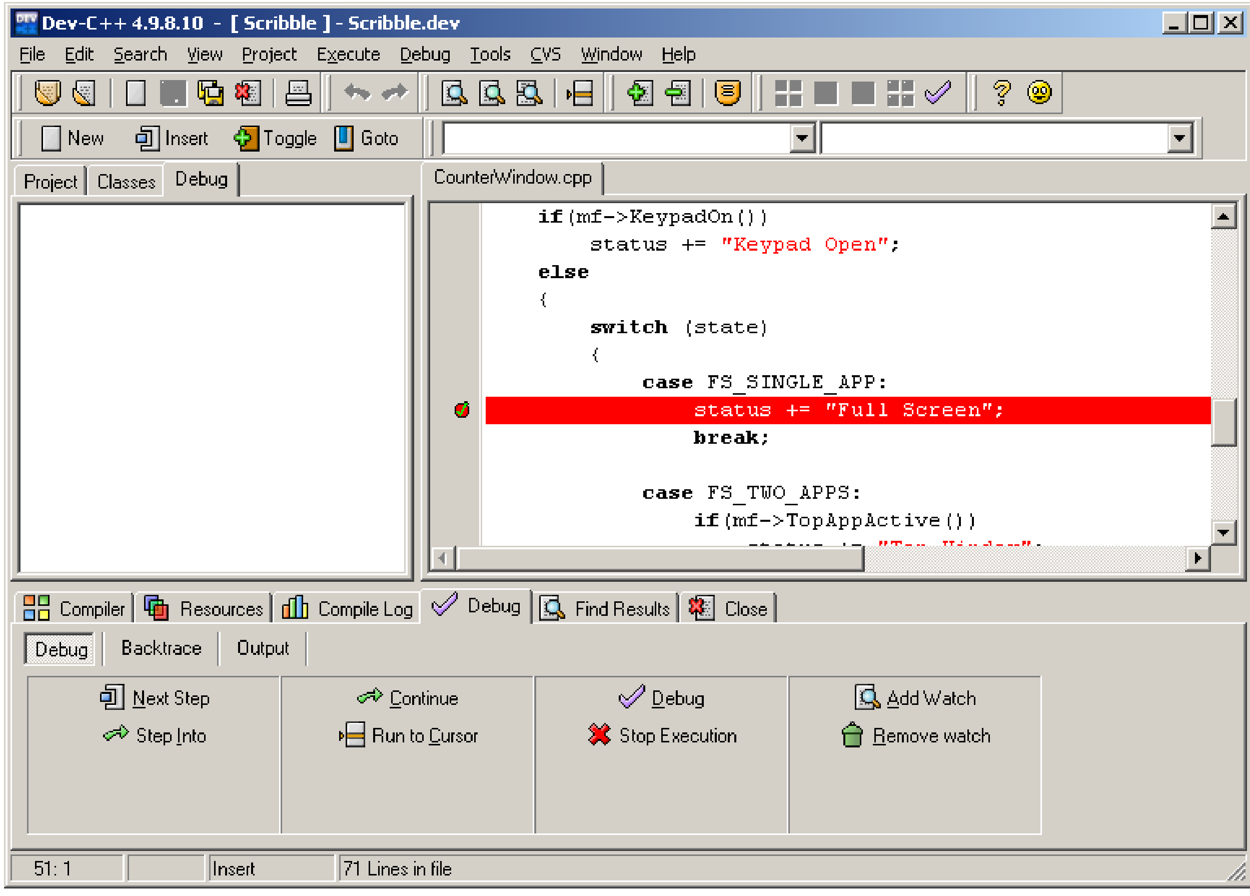
This will create the compressed file ADDINAPA.BI_. Rename this file back to Test.cpa and you're done!

```
C:\PROJECT_OUTPUT_DIR\> ren ADDINAPA.BI_ Test.cpa
```

Debugging in Dev-C++

Dev-C++ GDB Front-end

The Windows executable that the wrapper creates is a debug build by default. You can use Dev-C++ as a front end to the GNU debugger gdb.exe to debug your project.



An active debugging session. Notice the breakpoint set in the editor and the debug window at the bottom of the screen.

To add breakpoints to your project, simply open a file then click in the gutter on the line where you wish to add a breakpoint. Once you have added all of the breakpoints you wish to add, click on the Debug button in the toolbar to start the debugging session.

When the debug session begins, the debug window will appear at the bottom of the screen. Once your program hits the breakpoint you can use this window to step over, step into, continue, run to cursor or stop the execution of the debugger.

Once the debugging session has begun, if you try to add or remove a breakpoint you will get an error message saying that a breakpoint cannot be added while the debugger is running. To pause the debugger without stopping it, bring up the console window that opened with the ClassPad GUI. With the console window having focus press Ctrl-C. This will pause the debugger and allow you to add or remove breakpoints. When you are ready for the debugger to begin again, click the continue button in the debug window.

Printf Debugging

Not only can you use the console window that opens with your ClassPad executable to pause the debugger, but you can also use it to debug by printing to standard out. Since the ClassPad doesn't have printf or cout, you must make sure that any calls to these functions are only compiled by gcc. To do this, surround them with #ifdef WIN32 #endif macros. For example:

```
int main(int argc, char** argv)
{
      int x, z=0;
      x = z+10;
#ifdef WIN32
      printf("%i\n", x);
#endif
      return 0;
}
```

The WIN32 define is sent to gcc when compiling, but is not sent to shc. This allows you to add anything that you want only gcc to compile in an #ifdef.

Debugging on The ClassPad

At the present time there are no advanced debugging tools available to debug on the ClassPad directly. There are, however, a couple of ways to debug using "printf-style" debugging.

Message Boxes

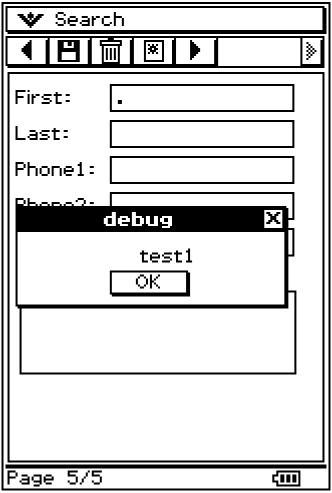
The most straightforward way to debug is with pop-up message boxes. Creating a message box is a very simple process: create a peg rectangle, create a new instance of PegMessageWindow, and execute the dialog.

To make the process easier, you can create a debug function that takes your debug string and displays it in a message box:

```
void DebugPopUp(CPString msg)
{
      PegRect rr = {5,100, 140, 150};
      PegMessageWindow *msg = new PegMessageWindow(rr, "debug", msg);
      msg->Execute();
```

```
}
```

Then to output a debug string simply call DebugPopUp passing in your string.



DebugPopUp("test1");

Status Bar

The status bar can also be used as a debugging tool. The major difference between using pop-ups and the status bar is that the status bar will not pause the program. This can be a disadvantage when you have several messages being replaced before they can be read. If you are echoing several messages and want to read them all, pop-ups are probably the better choice.

To use the status bar create a function like this:

```
void SetStatusBar(CPString str)
{
      PegStatusBar* bar = GetStatusBar();
      bar->SetTextField(1,str);
}
```

When you would like to print debug output to the status bar, just call the function with your debug string.

MCS Variables

A final debugging option is to create MCS variable(s) with different values depending on what code your program executes. Like before, you should create a function to simplify the debugging process:

```
void SetMCSVar(word val)
{
  OBCD dat;
  word size;
  Cal_setn_OBC(val,&dat);
  size = sizeof(OBCD);
  BMCSCreateVariable("main", "debug", IMU_MCS_TypeReal, size, (UCHAR*)&dat);
}
```

At places where you would like to update the variable just call SetMCSVar() with the desired value. After the program has run, you can check the variable manager to see the final value of your variable. This method has the disadvantage of not immediately showing the debug output. But if you do not want to pause your program with pop-ups and cannot use the status bar, this is a good option.