

SUNSTONE CIRCUITS, INC.

PCB123 PRODUCT DIVISION

PCB123 Plugin SDK

PCB123 – A DIVISION OF SUNSTONE. INC.

PCB123 Plugin Software Development Kit

SDK Version 3.0.0.1

© 2006, Sunstone Circuits, Inc.
Freeman Road
Mulino, OR
Phone 503-829-9108 • Fax 503.555.1212

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER(S) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. WE ARE NOT LIABLE FOR ANYTHING HORRIBLE YOU MAY CREATE, NOR WILL WE TAKE CREDIT FOR ANYTHING WONDERFUL YOU MAY CREATE.

Table of Contents

INTRODUCTION	4
PCB123 Plugin in a nutshell.....	5
Intended audience	5
Community.....	6
Sidebar: A little history on restricted access	Error! Bookmark not defined.
A note about the SDK deployment	6
ANATOMY OF A PLUGIN.....	8
Overview	8
The XML Plugin Configuration File.....	8
Event Model	15
Sequence of events.....	15
The Plugin DLL	17
The I/O Parameter Block.....	19
PLUGIN SDK.....	24
The Plugin SDK.....	24
SDK directory structure.....	24
CORELIB	27
CoreDefs.h – Simple types and common definitions	27
Debug.h – Assert and trace macros, CoreTimer	28
Heap.h – Efficient memory management	28
Templ.h – Templated collection classes	30
CoreString.h – General purpose wide string class.....	31
StringTable.h – Global string pool.....	32
Filename.h – Filename parsing, directory scanning, file testing	32
System.h – App settings, PE versioning, Busy Cursor, StatusBar	33
MRU.h – Generic most-recently-used class	34
XML.h – CXmlDomParser abstraction	34
CoreLUT.h – Lookup tables, Color macros, Very random numbers.....	35
ProgressDlg.h –General purpose progress indicator	36
CoreDialog.h – Base class for PCB123 dialogs. Standardizes help	36
LogFile.h –Accumulates messages and formats them into a report.	37
2DLIB	38
2dDefs.h – Declares ANGLE and UNIT type. Angle macros, etc.	39
2dBase.h – Base class for 2D primitives	40
Pnt.h – Declares CPnt class	40
Seg.h – Declares CSeg class.....	44
Arc.h – Declares CArc class.....	47
Rct.h – Declares CRct class.....	51
Gon.h – Declares CGon (Polygon) class	54
Fnt.h – Declares a stroked font	59
PolyTri.h – Triangulates a CGon object.....	59
DRAWLIB	61
Viewport.h – Virtual viewport.....	61
DrawCache.h – Graphics resource manager	62
Cookie.h – Primitive shape drawing functions	62

BASEDBLIB	64
BaseDbLib.h – Basic definitions and master include for BaseDbLib	66
Prop.h – Properties.....	68
TransactionManager.h – Transaction interface.....	69
Base.h – Base class declaration	72
PCBDBLIB	78
PcbBase.h – Base class declaration for PCB objects.....	80
PcbProp.h – Base class for objects with Properties.....	83
PcbError.h – DRC error marker class.....	85
PcbPoly.h – Polygon Class Declaration	87
PcbText.h – Stroked Text Class Declaration	92
PcbPin.h – Pin Class Declaration.....	95
PcbComp.h – Component Class Declaration	101
PcbPackage.h – Footprint Class Declaration	105
PcbTrack.h – Track/Route Class Declaration	107
PcbNet.h – Net Class Declaration	111
A RELAXING TUTORIAL.....	117
Tutorial: Creating a Plugin	117

Introduction

Recently, Sunstone wrote the Gerber import facility in a way that generalized the backend to accept a stream of primitives from anywhere. This of course set off a clamor for a DXF importer and such, all of which would push release dates and new development schedules back.

Besides new development, the Gerber importer opened up another can of worms: of the hundreds of sample Gerber files used during test, a good deal of them exhibited different formatting and several were radically different – even to the point of being non-conforming to the RS-274X specification. This means we pretty much know somebody is going to call us with a problem and we will have to provide a fix and an update.

This is not a new problem but it promises to become more significant as we roll more features and data translators out to our customers. Data translators tend to be moving targets on both ends, so when one format changes or new capabilities are added, we have to schedule a new release.

Our user base has grown to the point where rolling out a new release has is a little bit scary. Any mistake and we have a lot of upset customers. We would rather not have to roll a new release just to fix a translator or two, and you probably agree.

About the only solution to this problem is to decouple the translators from the core system. Before we were willing to remove the translators and re-write them as stand-alone applications we figured we would take a look at the feasibility of writing a private Plugin facility and implement the netlisters and translators as Plugin modules that can be individually updated and distributed with minimal impact to everyone.

After writing the Plugin socket and a couple of test Plugins we became very excited about what had been created. The interface was extremely simple, the protocol concise, and the environment unrestrictive. We liked it so much that we decided to share it with you.

PCB123 Plugin in a nutshell

A PCB123 Plugin is an executable module in the form of a DLL that can interact with a PCB123 database, or even create a new one. The Plugin will also have access to events that occur in the PCB123 host application. Events can be system events such as mouse moves and events can be database events such as a component or route modification.

The Plugin will be handed a top-level database object that represents the board as a whole. It can then navigate through the object hierarchy or iterate through all objects in a direct manor. The Plugin will also be handed an object called the Transaction Manager that it can use to perform modifications to the database. If done through the Transaction Manager, modifications will be recorded for Undo/Redo and will synchronize the display to reflect the changes made.

What the Plugin will not have access to the host application user interface other than specifying menu items to be added under the PCB123 menu. The Plugin can, however, create its own user interface, be it a simple dialog box or a web browser hosting Flash. It is only limited by your imagination.

Here is your chance to create the PCB wiz-bangs you always wanted.

Intended audience

It must be said right up front that if you are a C/C++ developer you will love the Plugin API. However if you are a VB developer or are used to scripting

applications through an automation server then you may find this API a bit of a challenge. Sunstone still reserves the right to move the Plugin API to an automation server in the future.

For the initial release there is an additional restriction on which development environments are supported by the Plugin SDK. Currently only Microsoft Visual Studio 2005 is supported and sample projects for these environments have been included.

In theory, any development environment that can create a regular Win32 DLL (or MFC extension DLL) should be able to produce a valid PCB123 Plugin. Sunstone would be very interested in hearing from you if you either get the SDK working in a different environment or need assistance in doing so.

Community

PCB123 has a new Developer's Exchange on the website. Here, you will have the opportunity to download Plugins developed by Sunstone and others or make your own Plugin available.

If you develop a Plugin, we do not expect you to give it away. That decision is up to you. We can either host the Plugin on our site if it is free, or host a link to your metered site if it is not.

If you do decide to charge money for your Plugin, you will have to provide your own security mechanism. The PCB123 host will always attempt to load a Plugin. It is up to the Plugin whether or not to run.

A note about the SDK deployment

The Plugin SDK is included as part of the standard installation. You do not need to get it from Sunstone.

Whenever PCB123 is compiled, the Plugin SDK directory structure is rebuilt from the same source and libraries that the PCB123 application was

built from. This is to ensure that any Plugin developed on any given machine with PCB123 installed on it will safely share the same DLL's as PCB123. It is for that reason a new version of the SDK will only be distributed with a full installation of PCB123.

The Plugin SDK will reside in a directory named PluginSDK underneath the install directory. All sample Plugin projects will be located underneath the PluginSDK directory. The sample projects all point their output DLL's to the Plugins directory underneath the install directory. That is where PCB123 looks for them when the program is started.

As mentioned before, the PCB123 product group now maintains a Developer's Exchange area on its website. If you wish to share or sell a Plugin on this site then you will be asked to register as a Plugin developer if you have not already done so. Registered Plugin developers will get information on any new version of PCB123 in advance of the general public release. This will allow the Plugin developer time to make any changes that may be required for the new version and to re-submit the Plugin so it can be available to the public when they receive the PCB123 update.

One additional resource Sunstone has created is a custom Application Wizard for Visual Studio. This App Wizard will do most of the leg work in creating a Plugin. The App Wizard will not be part of the standard install. You will have to get it from our website along with instructions on how to install it into Visual Studio.

Anatomy of a Plugin

Chapter

1

Overview

A PCB123 Plugin is a Windows DLL that adheres to a particular protocol to gain full access to a PCB123 board database, footprint database, or application events. Access to a database is accompanied by a transaction manager that can be used to modify the PCB objects in such a way that all graphics are synchronized with the changes and those changes are recorded for Undo/Redo.

The Plugin can specify menu items to be added to the PCB123 application and which functions inside the DLL that should get called when those menu items are clicked. An XML-based Plugin Configuration File is used to configure the Plugin into the system.

Because of how narrow the interface channel is, you will see how easy it is to create tightly focused tools without having to interact with, nor manage an entire application framework.

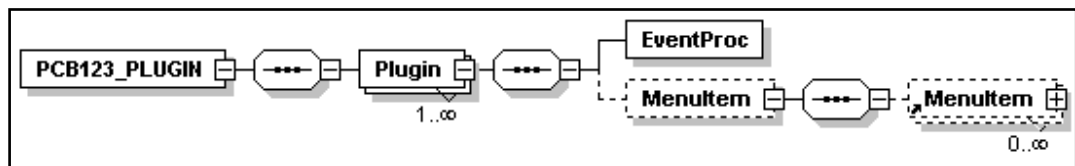
Many code snippets are provided throughout this manual and there is a complete, functional Plugin tutorial at the end.

The XML Plugin Configuration File

The Plugin Configuration File is a small XML file that configures various aspects of a Plugin. This file, along with any Plugin DLL file, must reside in the Plugins subdirectory under the PCB123 installation.

The configuration file can actually configure more than one Plugin. If you have developed several Plugins that you would like to treat as a unit, then you may want to package them all in one configuration file. Each Plugin must be described inside its own "Plugin" element. Do not worry if you are not familiar with XML. The configuration file is quite simple and is documented here.

Like all of Sunstone's XML formats, the Plugin Configuration File has a matching schema named PluginSchema.xsd in the XML subdirectory that you can use to validate against. The schema diagram for the Plugin Configuration File format is shown here.



The schema diagram for the Plugin Configuration File format

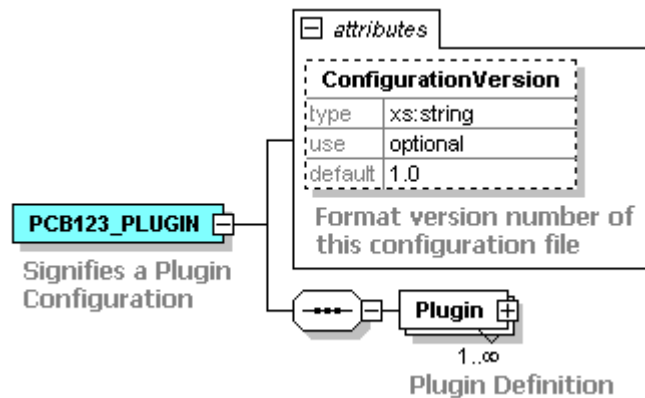
Just so you know what this section is talking about, here is a sample configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<PCB123_PLUGIN ConfigurationVersion="1.0">
  <Plugin PluginVersion="1.0" Name="Density Graph"
    ModuleFileName="Density.dll" Copyright="(C) Copyright 2006,
    Sunstone Circuits, Inc." Author="Keith Ackermann"
    NoDocEnabled="false" FootprintDocEnabled="false"
    PcbDocEnabled="true">
    <EventProc ProcName="DensityEventProc"/>
    <MenuItem ProcName="DensityGraph" ItemString="Density Graph"/>
  </Plugin>
</PCB123_PLUGIN>
```

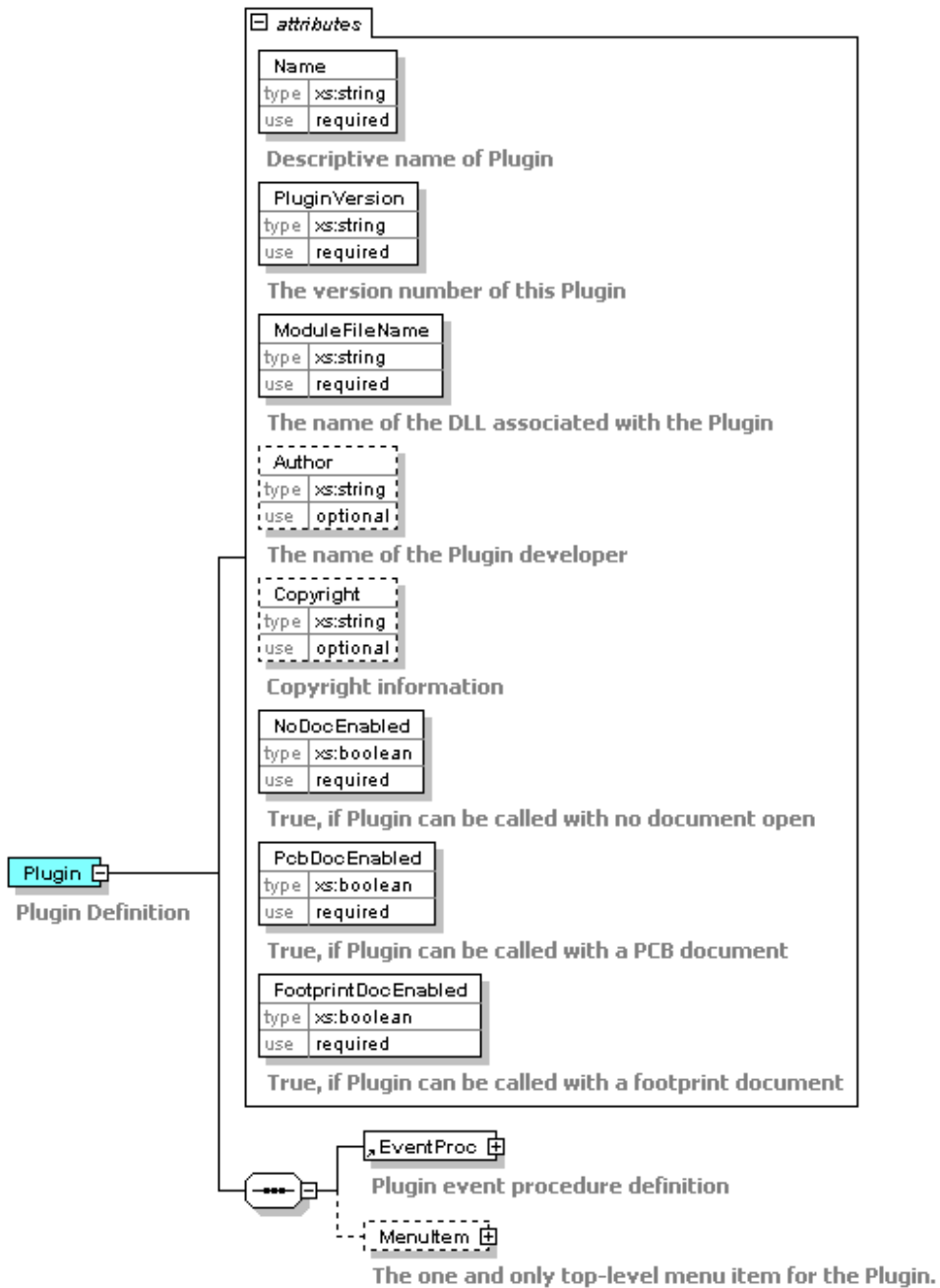
The file begins with a PCB123_PLUGIN element which has one required attribute named ConfigurationVersion. The value of ConfigurationVersion currently must be 1.0. This attribute is not the version of the Plugin but rather the File Format version in case the

format of configuration files is revised in the future. The PCB123_PLUGIN element must have at least one “Plugin” sub-element. The Plugin sub-element is the only kind of child element allowed.

Here is a schematic representation of the PCB123_PLUGIN element:



The next element is the Plugin element. This element begins the actual definition of a Plugin. It declares several attributes which are required and a couple that are optional. Below is a description of each attribute.



Name is just a unique label for the Plugin. You cannot have more than one Plugin element with the same name so try and use something descriptive.

The **Version** attribute is the version number of the Plugin. Currently, no checking is done on this number but eventually the Plugin

Manager may perform version checking. Regardless, it is always a good idea to version your software creations.

The **ModuleFileName** attribute must be present. This is the attribute that points to the name of the DLL that implements this plugin. The name is assumed to be relative to the Plugins directory.

Author and **Copyright** are optional attributes. Nothing is currently done with them. They just establish a minimal form of ownership and protection for you.

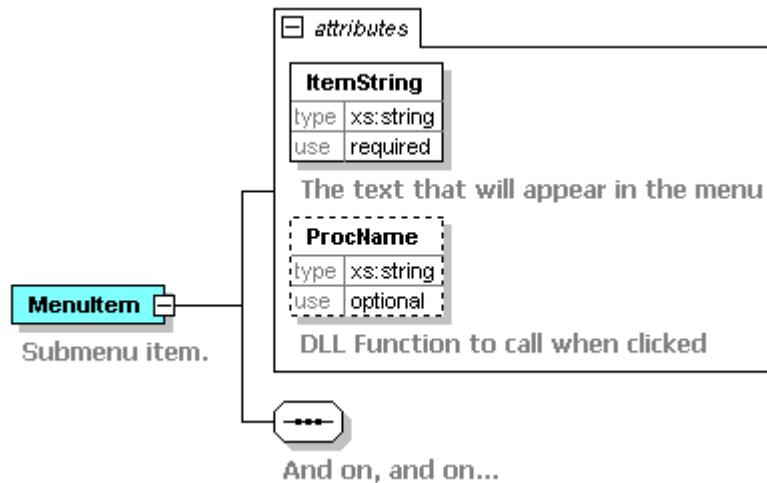
The last three attributes are all related to each other and are significant only if your Plugin adds menu items to the PCB123 host. PCB123 currently has three different menu states: one if there are no documents open, one when a board is the active document, and one when a PCB footprint editor is the active document.

The three attributes **NoDocEnabled**, **BoardDocEnabled**, and **FootprintDocEnabled** control which menu bars will get menus for this Plugin.

At first glance it may not make sense to activate a Plugin if there are no documents open, but it is precisely this state that a Plugin may want to create a new document, such as a translator (more on this later).

The Plugin element may contain two sub-elements. The first is called **EventProc** is used to specify the function name of the main event handler inside the Plugin DLL (described later). The last element is the **MenuItem** element. Each PCB123 Plugin can only specify one menu item but that single menu item can be a popup or sub-menu. In

this way the MenuItem element is a recursive element that allows you to specify any number of menu items and submenus in a cascading fashion.



There is no explicit difference between a menu item and a popup menu. It is simply implied by the presence of child MenuItem elements inside the top-level MenuItem.

The MenuItem element has two attributes. The first is ItemString and is the string that appears in the PCB123 host application. If ItemString == SEPARATOR, then a horizontal bar will be inserted in place of a menu string

The other MenuItem attribute is ProcName. This attribute names the exported function in the DLL to call when the menu is clicked.

Neither SEPARATOR items nor popup menu items can have ProcNames attached to them.

The visual states for the menu items such as checked, grayed, etc. can be controlled by the Plugin itself.

Below is the actual Plugin Configuration File for the density graph Plugin we ship. There are many XML editing tools that will guide you through hand-editing XML files. The one below is XmlSpy from Altova.

XML

version	1.0
encoding	UTF-8

PCB123_PLUGIN

ConfigurationVersion	1.0
Plugin	
PluginVersion	1.0
Name	Density Graph
ModuleFileName	Density.dll
Copyright	(C) Copyright 2006, Sunstone Circuits, Inc.
Author	Keith Ackermann
NoDocEnabled	false
FootprintDocEnabled	false
PcbDocEnabled	true
EventProc	
ProcName	DensityEventProc
MenuItem	
ProcName	DensityGraph
ItemString	Density Graph

And here is what the raw XML data looks like for the same file:

```
<?xml version="1.0" encoding="UTF-8"?>
<PCB123_PLUGIN ConfigurationVersion="1.0">
  <Plugin PluginVersion="1.0" Name="Density Graph"
ModuleFileName="Density.dll" Copyright="(C) Copyright 2006, Sunstone
Circuits, Inc." Author="Keith Ackermann" NoDocEnabled="false"
FootprintDocEnabled="false" PcbDocEnabled="true">
    <EventProc ProcName="DensityEventProc"/>
    <MenuItem ProcName="DensityGraph" ItemString="Density Graph"/>
  </Plugin>
</PCB123_PLUGIN>
```

Event Model

When the PCB123 application starts up, it scans the Plugins subdirectory looking for Plugin modules to load and interface with. The interface model is purely event driven, with events always being initiated by the PCB123 host application.

The Plugin must export a function to catch these events. This function is known as the main event handler. The main event handler is specified in the EventProc element inside the Plugin Configuration File as described above.

The other type of function that a Plugin might export is a menu handler. A menu handler is exactly like the main event handler but is only called when the user invokes it through a Plugin menu item.

When the PCB123 host initiates an event for a Plugin, it fills out a record called an IOB (input/Output Block) that gets passed as the only parameter to the EventProc. This is a synchronous call into the Plugin. In other words, PCB123 execution is blocked until the Plugin returns with a pass/fail return value.

Sequence of events

So what are the events and when does the PCB123 host call the Plugin? It depends on what happens the first time it calls the Plugin. The all-important file named IPlugin.h in the PluginSDK root contains all the definitions used by the Plugin interface. IPlugin.h defines an enumeration called PCB123EventType, which lists five different types of events. They are:

Event Name	Description
InitialEvent	<p>Issued to a Plugin when it is first loaded at system startup. When the Plugin returns from processing this event, it should set the m_ReturnResult member of the parameter block to any combination of the following flags:</p> <p>NO_123_EVENTS – Plugin is not interested in any PCB123 events (other than menu items it installed)</p> <p>DB_EVENTS – Plugin wants to be notified about database modifications every time one occurs.</p> <p>WIN_EVENTS – Plugin wants to listen to all Windows events such as mouse move, etc.</p>
UpdateMenuUIEvent	<p>Issued to the Plugin before a menu item owned by the Plugin is displayed. The Plugin should respond by setting the m_ReturnResult member of the IO parameter block to any combination of the following flags:</p> <p>MENU_DISABLE – Grey the menu item</p> <p>MENU_CHECK – Place a check mark next to menu item.</p>
MessageEvent	<p>If the Plugin requested Windows events (WIN_EVENTS) during InitialEvent, then PCB123 will issue this event to the Plugin before every Windows message is</p>

	<p>processed. The Plugin cannot alter the event or prevent the message from being processed by PCB123.</p> <p>The Plugin should not perform lengthy processing when handling this event type because it will drag down the whole system.</p>
NotifiyEvent	If the Plugin requested these messages during InitialEvent, then PCB123 will issue this event to the Plugin every time a database object is modified, added, or deleted.
FinalEvent	Issued just before PCB123 terminates. It allows the Plugin to perform any cleanup it may need.

So you can see that the Plugin response to the InitialEvent event establishes the frequency and nature of subsequent events that are sent to the Plugin.

If your Plugin is a data translator or a utility that is invoked in response to the user selecting a Plugin menu item, then the Plugin will not listen to Windows events or database notifications and should return NO_123_EVENTS in response the the InitialEvent.

The Plugin DLL

The executable code for the Plugin will reside in a Windows DLL. It can be a standard DLL or an MFC extension DLL. Besides the DllMain entry point that Windows requires (usually automatically generated), the Plugin will export one or more functions that will be

called by the PCB123 host when certain events occur. The exported functions have to be declared as extern and use the C language calling convention. In a Microsoft environment this is usually accomplished by declaring the function prototypes like so:

```
extern "C" {  
    functionPrototype ();  
}
```

Exported functions that the host calls all share a common signature. Namely that of

```
PluginResult PluginProc (PluginIO& iob);
```

Where:

PluginResult – Enumeration of valid Plugin responses. The only two currently defined are PluginFailure and PluginSuccess.

PluginProc – The name of the exported Plugin function. This will either be the Main Event Handler or a Plugin Menu Handler.

iob – A reference to a PluginIO structure. This structure contains the calling context data from the host. It also contains a single member (called m_ReturnValue) where the Plugin can pass results back to the host. This structure is described in detail below.

A couple of points worth mentioning here deal with exception handling and context switching.

The PCB123 system contains a top-level exception handler that eventually catches everything. The handler's job is to save away all open documents, generate a report that gets fired off to Sunstone, and gracefully shut the system down while at the same time firing up

another copy of the application that automatically loads the last active document. Any internally trapped error performs a bunch of duties and ultimately throws an exception of type CSoftCoreException. If you employ exception handling in your Plugin and your Plugin modifies a PCB123 database then be sure to re-throw any exceptions you may trap to ensure proper system behavior.

The other issue is more of a reminder that if you use MFC to develop your Plugin and the Plugin manages resources then make sure you add AFX_MANAGE_STATE(AfxGetStaticModuleState()) to the very beginning of every entry point into your DLL or you will be chasing very strange problems whenever the DLL accesses a resource. See TN058: MFC Module State Implementation in the Microsoft Knowledge Base.

The I/O Block

As mentioned before, exported handlers (both event handlers and menu handlers) in the Plugin share the common signature:

```
PluginResult PluginProc (PluginIO& iob);
```

When we look at the PluginIO structure in the file IPlugin.h, we see the following:

```
struct PluginIO {
    struct { // Common to all Plugin Procs
        CDbBase*      m_pRootObj;           // The root object.
        CTransactionManager* m_pTm;        // The tm to use for changes
        HWND          m_MDIFrame;          // The Frame window
        HWND          m_hCurrentView;      // Current MDI child window
        ULONG         m_ReturnValue;       // Place return values here
    };

    Struct { // Used only in EventProc
        PluginEventType m_EventType;        // Type of event this is
        union { // If m_EventType is...
            PluginProc  m_UpdateMenuEvent;  // UpdateMenuUIEvent
            MSG          m_MsgEvent;        // MessageEvent
            NOTIFY_EVENT m_NotifyEvent;      // NotifyEvent
        };
    };
};
```

```
};
```

You can see that the structure is broken into two sections: a section common to all Plugin procedures and a section only used in the main event handler. The common members are in play for the main event proc and for any menu item handler. The common members are:

m_pRootObj: This is a pointer to the outer-most object of the currently active document and will be in one of three states:

1. It will be NULL, if no document is open. Plugins are still called even when no document is opened because maybe the Plugin will create a document , such as a translator.
2. It will be a Board object. This can be determined by calling `m_pRootObj->GetClass ()`. If it returns `BOARD_CLASS`, then it's a `CPcbBoard` object.
3. It will be a Footprint object. This can be determined by calling `m_pRootObj->GetClass()`. If it return `PACKAGE_CLASS`, then it is a footprint.

m_pTm: This is a pointer to the Transaction Manager for the current document. If there is no document, then it will be set to NULL.

m_MDIFrame: This is a handle to the PCB123 host application window. The intended use for it is to provide a parent window handle for any dialogs or other windows that the Plugin may create. This handle has great potential for abuse and should not be used, but since you are an engineer See Advanced Concepts at the end of this section.

m_hCurrentView: This is a handle to the active MDI child window and may be NULL. Its intent is same as above and carries the same potential for abuse.

m_ReturnValue: This is where the Plugin communicates information back to the host. It is context sensitive, with the host examining this value only under the following conditions:

- Upon returning from a call to the main Plugin event handler with an event type of `InitialEvent`. `m_ReturnValue` will contain flags indicating what further events it wants to receive.
- Upon returning from a call to the main Plugin event handler with an event type of `UpdateMenuUIEvent`. `m_ReturnValue` will contain flags indicating any state changes to be made to the menu item.
- Upon returning from a menu handler. If the host finds any value other than 0 in this member after calling a Plugin menu handler, the value is assumed to be a pointer to a newly created root object that is either a board (`CPcbBoard`) or a footprint (`CPcbPackage`). The host will then create a new document of the appropriate type that is rooted at the newly created object.

The other members of `PluginIO` are only used in the main event procedure and not in menu handlers. There is an event type and a union of members whose use depends on the event type. The event types are:

InitialEvent: This is the one-time initial event. It uses no other data.

UpdateMenuUIEvent: Called whenever the PCB123 host is about to show a menu item. This allows the menu display state to be changed at the last second based on context. This event type uses the `m_UpdateMenuEvent` member to identify the menu item to update. It will be set to the address of the menu handler for that menu item.

MessageEvent: If the Plugin indicated it wants to listen in on everything the host does, then it will be called with this event type quite often. This event type uses the `m_MsgEvent` member for the message parameters. This is a standard Windows structure.

NotifyEvent: If the Plugin indicated it wants these events, then whenever one is received it should examine the `m_NotifyEvent` member for the notification parameters. The `m_NotifyType` member of `m_NotifyEvent` can be one of the following:

- **DocChangedNotice:** The active document has changed.
- **PrefChangedNotice:** A change in the user preferences has occurred.
- **ZoomChangedNotice:** A pan or zoom has occurred.
- **LayerChangedNotice:** The active layer has changed.
- **GridChangedNotice:** The working grid has changed.
- **SelectNotice:** An object has been selected (if `m_IParam == 1`) or deselected (if `m_IParam == 0`)
- **ObjectChangingNotice:** A database object is about to be changed. `m_IParam` points to the object.
- **ObjectChangedNotice:** A database object has been changed. `m_IParam` points to the object.
- **ObjectAddedNotice:** A database object has been added. `m_IParam` points to the object.

- **ObjectDeletedNotice:** A database object has been deleted. `m_IParam` points to the object.
- **IdleTimeNotice:** This notification is sent when the system is in an idle state (doing nothing).

Advanced Concepts

It is possible to violate the synchronous event model with the `m_MDIFrame` and the `m_hCurrentView` members of `iob`. As stated before, these members are primarily provided as parent window handles for dialog boxes.

However, it is possible to drive the host application by calling `SendMessage` or `PostMessage` using these window handles. Because it is inevitable that someone will try it, we provide a file in the Plugin SDK root directory called `HostCommandIds.h`. This file contains the `WM_COMMAND` id's for most of the commands in PCB123. If you elect to use them, we strongly recommend that they are invoked with `PostMessage` and not `SendMessage`. The command id's are not documented but can be figured out by the determined individual.

The Plugin SDK

In this section we will examine the structure of the Plugin SDK and discuss the PCB123 services and structures exposed to you. But first a brief disclaimer.

The current version of the Plugin SDK has been created with the assumption that it will be used in a Microsoft Visual C++ development environment. In fact, the only tested development environment is Microsoft Visual Studio 2005. The project files for the Plugin samples are in this format. With that said, if there is anyone who has managed to get the SDK working in a different environment or needs assistance moving to a different environment then we would love to hear from you.

SDK directory structure

There are two new directories under the PCB123 install path called Plugins and PluginSDK. The Plugins directory is where PCB123 looks to install Plugins during system startup. The PluginSDK directory is where the development of new Plugins takes place. The PluginSDK directory contains IPlugin.h (the main include file) and the following subdirectories:

Directory	Comment
2dLib	Public interface for 2dLib.dll
BaseDbLib	Public interface for BaseDbLib.dll
ContextHelper	Sample Plugin project showcasing event monitoring
CoreLib	Public interface for CoreLib.dll
Density	Sample Plugin project that shows a density graph of a PCB
DrawLib	Public interface for DrawLib.dll
GridCtrlLib	Public interface for GridCtrlLib.dll
Lib	Supplied libraries to link with
PcbDbLib	Public interface for PcbDbLib.dll
RelaxRouting	Sample Plugin that minimizes angles on all routing

The IPlugin.h header file contains all the Plugin-specific definitions and also starts the chain of includes for the PCB123 system interfaces.

The dependency graph for the PCB123 directories is as follows:

--	--

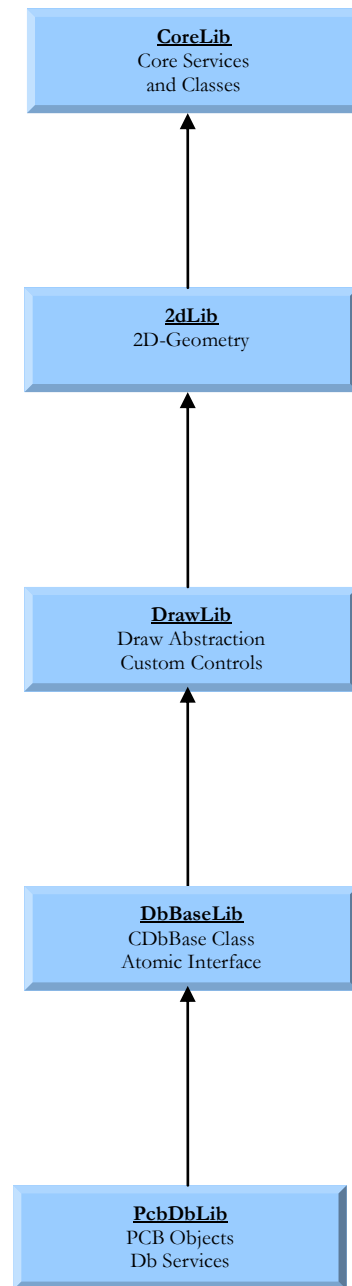
CoreLib.h (included by 2dLib.h)
includes all the header files in the
CoreLib directory.

2dLib.h (included by DrawLib.h)
includes all the header files in the
2dLib directory.

DrawLib.h (included by
DbBaseLib.h) includes all the
header files in the DrawLib
directory.

DbBaseLib.h (included by
PcbDbLib.h) includes all the
header files in the DbBaseLib
directory

PcbDbLib.h (included by
IPlugin.h) includes all the header
files in the PcbDbLib directory



CoreLib

The CoreLib directory contains the public interfaces into CoreLib.dll. The act of implicitly or explicitly linking CoreLib into a process will consume about 2mb of memory due to the Heap Manger and String Table services it offers up. No special initialization or cleanup has to be performed to use or stop using the library.

Below is a brief description of what each file in CoreLib contains.

CoreDefs.h – Simple types and common definitions

CoreDef.h contains the most primitive declarations in the system. It abstracts primitive types with declarations such as INT8, INT16, INT32, FLOAT, DOUBLE, etc. It declares CHAR as a _TCHAR for multi-byte compatibility. STR's are of type CHAR* and CSTR is a const STR. CoreDef.h also defines the usual macros and constants such as SWAP, PI and SQRT2.

Something worth mentioning is the DECLARE_CORE_CLASS(clsName,baseName) and DECLARE_CORE_BASE_CLASS(clsName) macros. If you add one of these macros anywhere (usually at the start) of a class declaration then that class will automatically use the CoreHeap for memory management because these macros overload all variations of operator new and operator delete. See CoreHeap below for details.

Debug.h – Assert and trace macros, CoreTimer

Debug.h defines the following macros for error trapping: ASSUME, which evaluates an expression for truthfulness; RANGE, which tests a value to be within a range; and CRASH, which is an unconditional error assertion. These three error traps declare their first parameter to be an integer which is expected to be unique across the entire system (expected, but not enforced). This presents no problem for development in the main PCB123 system because there is a tool that automatically runs through all the source code and automatically generates unique id's for these macros. It does, however, make them a little awkward for use in Plugins. Even more awkward is the fact that when one of these macros asserts they generate a crash report that is automatically sent to the Sunstone server. That means we, not you will be getting the diagnostics. For these reasons you may find it easier to roll your own error trapping.

Debug.h also defines a macro called STRACE that is just a wrapper for a stream that is directed to the Windows OutputDebugString facility.

Example:

```
ASSUME (9999, pComp != NULL);  
CPnt loc = pComp->GetLoc ();  
STRACE ("Component location is " << loc);
```

Heap.h – Efficient memory management

Heap.h defines a class that is used to manage a dynamic block of memory very efficiently. A CHeap object allocates memory "pages" of a certain granularity (typically 1mb) as it needs them and does out

“slices” of the pages as requested from the `Allocate` member function. If a request is larger than the granularity then a new page of the requested size is allocated.

The `CHeap` object knows about every block of memory it has served up. As allocated memory is returned back to the heap using the `Free` member function, it is first verified that it came from the `CHeap` object and that it has not already been freed. It then places that freed chunk of memory onto a stack of chunks for its particular size. When a new request for memory is made, instead of just slicing off a new chunk, it first attempts to pop a freed chunk off of the stack for the requested size. In this way the `CHeap` object will reuse chunks of memory over and over.

Where this becomes really beneficial is when there is a class or structure that is dynamically created and destroyed many times. If you add `DECLARE_CORE_CLASS` to this class declaration then that class will have its operator `new` and `delete` overridden to use the `CHeap` object. Which `CHeap` object? Well the global one of course.

`Heap.h` also defines several global functions to manage memory – namely `CoreAllocate` and `CoreFree`. These global functions interact with a static `CHeap` object that is created when `CoreLib.dll` is loaded.

The main difference between using the global `CHeap` object and one created dynamically is that all global requests must be freed before the application terminates or memory leaks will be reported. A dynamically created (private) `CHeap` on the other hand may have allocated memory in a very complex way but the application code

can release the entire CHeap object in one swoop without having to bother running through a possibly very complex deallocation process.

Templ.h – Templated collection classes

Templ.h defines the collection class templates used throughout the PCB123 system. These classes are much leaner and meaner than the STL equivalents. The defined classes are:

TSimpleList<T>	This list only burns 4 bytes. Use when item order is not important.
TOrderedList<T>	This list burns 8 bytes. Maintains insertion order.
TQueue<T>	Same as an ordered list but with Queue semantics.
TStack<T>	Same as a simple list but with stack semantics.
TIter<T>	List iterator for all above classes.
TIntMap<K,V>	Map an integer Key to a Value (a pointer works as an integer).
TStrMap<K,V>	Map a string to a value.
TArray<T>	General purpose array class.

When interfacing with a PCB123 database the list iterator class TIter will be the only one of these classes you will use. You will not have access to an object's list members directly. The PCB123 database object base class defines a virtual GetFirstChild() method which returns an iterator pointing to the first item in a list. The base class also defines AddChild, DeleteChild, and GetParent to complete the traversal and update operations.

The TArray class is limited to a 32-bit index. The TArray class is fine for storing a collection of objects that have to be indexed but do not expect to use it and get the same performance as a native C++ array. The payload at two consecutive indexes of a TArray most likely are not at contiguous memory locations, and hence the benefits of the processor cache are lost when doing a sequential access of the array.

Example:

```
// Declare a list that carries CMyClass objects as payload
// Also, declare an iterator for this list type
typedef TOrderedList<CMyClass*>          CMyClassList;
typedef TIter<CMyClass*>                 CMyClassIter;

CMyClassList theList;          // Declare a list

// Create some CMyClass objects and add them to the list
For (int count = 0; count < maxCount; ++count) {
    CMyClass* pMyClass = new CMyClass;
    theList.Add (pMyClass);
}

// Access the objects in the list
// List operator () is overridden to return an iterator
CMyClassIter iter = theList ();
for (; iter(); ++iter) {
    CMyClass* pMyClass = iter.Get ();
    // Do something with the object
}
```

CoreString.h – General purpose wide string class

The CoreString class is yet another general purpose string class. It does pretty much everything a CString object does and then some. It's a little more efficient memory wise too.

StringTable.h – Global string pool

StringTable.h defines the CStringTable class which manages a collection of const strings. Any PCB123 object whose class member is a string declares that member as being of type STRID. You can obtain a STRID from a string by calling the global function StrId(string) and you can convert a STRID to a string by calling the global function StrPtr (STRID). These global functions actually access a global (but private) instance of a CStringTable object. You can declare private CStringTable objects to manage your own collection of strings but usually the global table suffices.

Example:

```
STRID netName = StrId("NET_1");  
CPcbNet* pNet = new CPcbNet (netName);  
CoreString string = StrPtr (pNet->GetName());
```

Filename.h – Filename parsing, directory scanning, file testing

Filename.h declares functions that deal with filenames and collections of files.

GetDriveName, GetPathName, GetFileName, GetExtName, crack a filename.

StripDriveName, StripPathName, StripExtName remove pieces of a filename.

BuildFilename will safely construct a filename from pieces.

NormalizeFilename will make sure all the path separators are the '/' character and optionally change it to upper case.

FileExists and DirectoryExists are tests. MakeRelativePath will attempt to make one path relative to another.

FindFiles and FindSubdirectories will return a list of all files matching the filter criteria. These functions can work recursively.

System.h – App settings, PE versioning, Busy Cursor, StatusBar

System.h defines functions and services that are fairly dependent on the Windows operating system.

GetAppPath and GetAppName return names from the currently running application.

The GetAppValue and SetAppValue functions read and write entries in a private configuration file. If a config file does not exist then one will be created that is the same name as the application with a .ini extension.

The CoreVersion class and associated functions all deal with the PE (Portable Executable) versioning resources. The class also overloads the comparison operators so you can detect if a version is less than, greater than, or equal to another version.

ShowBusyCursor displays or hides the hourglass cursor.

SetStatusMessage displays a string in the status bar.

WaitForKeypress will halt the execution of a program until a key is pressed or the timeout interval has been reached. A message asking for a keypress will flash in the status bar.

CheckEscapeKeyPressed will report if the escape key has been pressed since the last time it has been asked.

ExecProgram will execute a command line.

MRU.h – Generic most-recently-used class

The CMRUManager maintains a list of Most-Recently-Used items based on a given key in the application settings. This is very handy for populating drop lists, etc.

XML.h – CXmlDomParser abstraction

The CXmlDomParser is a very handy abstraction of Microsoft's XML DOM Parser. If you point it at an XML file it will parse the entire document into a tree of elements that can be navigated.

You can iterate over the elements in a tree or search for an element.

In an element, you can iterate through its attributes or search for an attribute.

The whole thing works in reverse. You can build up a tree of elements and attributes and save the document as a well-formed XML document.

Example:

```
CCoreXmlDom xmlDoc;  
if (! xmlDoc.LoadDocument (filename)) {  
    return false;  
}  
CCoreXmlDomElement* pRoot = xmlDoc.FindElement ("PCB123_NLF");  
if (! pRoot) {  
    AfxMessageBox ("Not a valid NLF file.");  
    return false;  
}  
CString desc = pRoot->GetAttributeValue ("Description");  
CCoreXmlDomElementIter iter = pRoot->GetFirstChild ();  
for (; iter(); ++iter) {  
    CCoreXmlDomElement* pElem = iter.Get ();  
}
```

CoreLUT.h – Lookup tables, Color macros, Very random numbers

The most likely reason you will refer to this file is for the color macros. PCB123 encodes an INVISIBLE bit into a Windows COLORREF object. IS_INVIS and MAKE_COLOR(color,invis) are handy macros.

CoreLUT has both look-up and computed prime number functions. It contains a look-up table that covers 32 bits worth of prime numbers distributed roughly logarithmic through the interval. In this way you can ask to return a prime nearest some given number and get an answer back very quickly. Very useful for getting hash keys, etc.

CoreRand returns a very random number that falls within a given range. It uses the Windows Cryptography API to get the values which are supposedly of extremely high quality. It is not recommended that you use this for generating large numbers of random values because the PCB123 implementation performs some trickery on reusing a generated sequence. If you need, you can contact us to get a source code snippet for the general case which is suspected to run slow.

ProgressDlg.h –General purpose progress indicator

This file declares the CProgressDlg class which displays a modeless dialog box with a message area and a progress bar indicator. You can set its range and control the progress explicitly or simply call Tick increment the indicator.

CoreDialog.h – Base class for PCB123 dialogs. Standardizes help

The CCoreDialog class adds context help to dialogs and will automatically supply a Help Topic with the name of the dialog class. To derive a dialog box from CCoreDialog you will have to check Help ID's for all the controls and override the following virtual functions:

```
DWORD* GetContextIdMap ();  
ToolText* GetToolTextMap () const;
```

In the dialog cpp file, add the following:

```
static DWORD helpIds[] = {  
    { IDC_SOME_CONTROL_ID, HIDD_SOME_CONTROL_ID, },  
    { IDC_SOME_CONTROL2_ID, HIDD_SOME_CONTROL2_ID, },  
    { 0, 0, },  
};
```

```

};

static ToolText toolText[] = {
    {IDC_SOME_CONTROL_ID, "Help text for this control" },
    {IDC_SOME_CONTROL2_ID, "Help text for this control" },
    { 0, "" },
};

DWORD* CMyDialog::GetContextIdMap ()
{
    return helpIds;
}

ToolText* CMyDialog::GetToolTextMap () const
{
    return toolText;
}

```

LogFile.h –Accumulates messages and formats them into a report.

This file declares the CErrorLog class which allows you to accumulate error and warning strings that may be generated during some operation. It also provides support for filename, line and column numbers for such things as file parsers. The CErrorLog::ShowLog member creates a formatted HTML file and displays it.

2dLib

The 2dLib directory contains the public interfaces into 2dLib.dll. In a word, the 2D library serves up 2D geometry. It contains a fairly small set of classes but a rich set of operations.

The act of implicitly or explicitly linking 2dLib requires no special consideration. It will automatically link in CoreLib.Dll which it is dependent on. The library does not consume any memory other than creating an instance of a stroked font (about 10k).

All the geometry objects define their data members as type UNIT. A UNIT can be any kind of integer but for PCB123 database objects a unit is defined as being one-ten-millionth-of-an-inch (there are 10,000 units per mil). The reason for this fine grain is that while providing a reasonable world size in 32 bits (+/- 200 inches) it is small enough to make round off errors between metric and imperial coordinates insignificant.

All angle values use an integer data type named ANGLE and is defined as one-ten-thousandth-of-a-degree.

The basic 2D shape classes are:

CPnt – Point class. Defined by public x,y UNIT members.

CSeg – Closed interval line segment. Defined by public p1, p2 CPnt members.

CArc – Arc class. Chords of a perfect circle. Defined by public c, p1, p2 CPnts.

CRct – Rectangle class. Defined by public x1, y1, x2, y2 UNIT members.

CGon – Polygon class. Private data members. Has codes for circle, arc, closed, etc.

All of the above classes overload the following operators

Operator	Operand	Meaning
+, -	CPnt	l-value = r-value offset by CPnt
+=, -=	CPnt	Offset self in the x,y.
*, /	Scalar	l-value = r-value scaled by amount.
*=, /=	Scalar	Scale self.
~	None	Mirror self (about the y-axis)
^	Any shape	Distance from shape to self. Returns a UNIT.
<<,>>	ANGLE	l-value = r-value rotated by ANGLE.
<<=,>=	ANGLE	Rotate self by ANGLE.
==,! =	Any	Test for equality. Incorporates an epsilon of 2 UNIT's.

Below is a brief description of what each file in 2dLib contains.

2dDefs.h – Declares ANGLE and UNIT type. Angle macros, etc.

2dDefs.h defines the UNIT and ANGLE data types. It defines some handy ANGLE macros and some conversions.

It defines direction codes for quick box testing for clipping.

This file also defines a class called a GeoTransform. This class simply stores a 2D translation, rotation and mirror flag. GeoTransforms can be accumulated, transforming the transform. Finally the GeoTransform provides an Apply function for each of the geometric primitive types. Apply transforms that primitive by whatever its translation, rotation and mirror settings are.

It also declares two functions that provide the basis for several important computational geometry operations. The function TriArea returns a signed area of a triangle. Besides calculating area, it can be used to test if a point lies to the 'left' or 'right' of a line by looking at the sign of the area. This requires consistency in its use: the line segment starts in point A, the line segment ends in point B, and the point under test is point C. The other function, PntInLine can be used to test for the only degenerate condition to the area test above.

2dBase.h – Base class for 2D primitives

The C2dBase class exists simply to abstract the distance functions used by the DRC engine. It contains no data members.

Pnt.h – Declares CPnt class

The CPnt class provides a rich set of operations for manipulating 2-D points. It is one of the few classes that allow public access to its data members.

Data members	
UNIT x	X-coordinate
UNIT y	Y-coordinate
Constructors	
CPnt ();	Default constructor
CPnt (const CPnt& src);	Copy constructor
CPnt (const CCir& src);	Construct from circle
CPnt (UNIT ix, UNIT iy);	Explicit initialization
Operators	
CPnt operator~ () const;	Mirror
CPnt operator- () const;	Negate x and y
CPnt operator+ (const CPnt& p) const;	Add
CPnt operator- (const CPnt& p) const;	Subtract
CPnt operator* (DOUBLE m) const;	Scale
CPnt operator/ (DOUBLE d) const;	Scale
CPnt operator<< (ANGLE a) const;	Rotate ccw
CPnt operator>> (ANGLE a) const;	Rotate cw
CPnt operator% (const CPnt& grid) const;	Off-grid by...
UNIT operator^ (const CPnt& p) const;	Distance
UNIT operator^ (const CSeg& s) const;	Distance
UNIT operator^ (const CArc& a) const;	Distance
UNIT operator^ (const CRct& r) const;	Distance
UNIT operator^ (const CGon& g) const;	Distance
CPnt&operator= (const CPnt& p);	Assignment
CPnt&operator+= (const CPnt& p);	Offset
CPnt&operator-= (const CPnt& p);	Offset
CPnt&operator*= (DOUBLE mul);	Scale
CPnt&operator/= (DOUBLE div);	Scale
CPnt&operator<<= (ANGLE a);	Rotate ccw
CPnt&operator>>= (ANGLE a);	Rotate cw

bool operator== (const CPnt& src) const;	Same
bool operator!= (const CPnt& src) const;	Not same
Manipulators	
CPnt& Reset ();	Set to 0,0
CPnt& Add (const CPnt& d);	Vector translation
CPnt& Subtract (const CPnt& d);	Vector translation
CPnt& Rotate (ANGLE a);	Absolute rotation
CPnt& RotateAt (const CPnt& org, ANGLE a);	Rotate about org
CPnt& Scale (DOUBLE mx, DOUBLE my);	Anisotropic scale
CPnt& Mirror (UNIT xOrg = 0);	Reflect the x-axis
CPnt& MirrorY (UNIT yOrg = 0);	Reflect the y-axis
CPnt& Transpose ();	Diagonal reflection
CPnt& Round (UNIT xGrid, UNIT yGrid);	Round to nearest
CPnt& RoundDown (UNIT xGrid, UNIT yGrid);	Round down
CPnt& RoundUp (UNIT xGrid, UNIT yGrid);	Round up
Operations	
OUTCODE BoxOut (const CRct& box) const;	
Void UpdateExtent (CRct& ext) const;	
UNIT Magnitude () const;	
UNIT Dot (const CPnt& p2);	
ANGLE Angle () const;	
ANGLE AngleFrom (const CPnt& org) const;	
UNIT DistVertSeg (const CSeg& s) const;	
UNIT DistHorzSeg (const CSeg& s) const;	
UNIT Distance (const CPnt& p) const;	
UNIT Distance (const CCir& c) const;	
UNIT Distance (const CSeg& s) const;	
UNIT Distance (const CArc& a) const;	
UNIT Distance (const CRct& r) const;	
UNIT Distance (const CGon& g) const;	

UNIT Distance (const CPnt& p, CSeg& connector) const;
UNIT Distance (const CCir& c, CSeg& connector) const;
UNIT Distance (const CSeg& s, CSeg& connector) const;
UNIT Distance (const CArc& a, CSeg& connector) const;
UNIT Distance (const CRct& r, bool filled, CSeg& connector) const;
UNIT Distance (const CGon& g, bool filled, CSeg& connector) const;
UNIT ManhattanLength (const CPnt& p) const;
CPnt ClosestPoint(const CPnt& p) const;
CPnt ClosestPoint(const CSeg& s) const;
CPnt ClosestPoint(const CArc& a) const;
CPnt ClosestPoint(const CRct& r) const;
CPnt ClosestPoint(const CGon& g) const;
CPnt ClosestPointOnSeg(const CSeg& s) const;
CPnt AngleSnap (const CPnt& endPt, ANGLE inc) const;
void SetInvalid ();
Tests
bool IsEqual (const CPnt& p) const;
bool IsCollinear (const CPnt& p1, const CPnt& p2) const;
bool IsCollinear (const CSeg& s) const;
bool IsOn (const CSeg& s) const;
bool IsOn (const CArc& a) const;
bool IsOn (const CGon& g) const;
bool IsIn (const CRct& r) const;
bool IsIn (const CGon& g) const;
bool IsClipped (const CRct& box) const;
bool IsInvalid () const;

Seg.h – Declares CSeg class

The CSeg class provides a rich set of operations for manipulating 2-D line segments. It is one of the few classes that allow public access to its data members.

Data members	
CPnt p1	Line start
CPnt p2	Line end
Constructors	
CSeg ();	Default constructor
CSeg (const CSeg& src);	Copy constructor
CSeg (const CPnt& pt1, const CPnt& pt2);	Point constructor
CSeg (UNIT x1, UNIT y1, UNIT x2, UNIT y2);	Explicit constructor
Operators	
CSeg operator~ () const;	Mirror
CSeg operator- () const;	Negate x and y
CSeg operator+ (const CPnt& p) const;	Add
CSeg operator- (const CPnt& p) const;	Subtract
CSeg operator* (DOUBLE m) const;	Scale
CSeg operator/ (DOUBLE d) const;	Scale
CSeg operator<< (ANGLE a) const;	Rotate ccw
CSeg operator>> (ANGLE a) const;	Rotate cw
CSeg operator% (const CPnt& grid) const;	Off-grid by...
UNIT operator^ (const CPnt& p) const;	Distance
UNIT operator^ (const CSeg& s) const;	Distance
UNIT operator^ (const CArc& a) const;	Distance
UNIT operator^ (const CRct& r) const;	Distance
UNIT operator^ (const CGon& g) const;	Distance
CSeg& operator= (const CSeg& p);	Assignment

CSeg& operator+= (const CPnt& p);	Offset
CSeg& operator-= (const CPnt& p);	Offset
CSeg& operator*= (DOUBLE mul);	Scale
CSeg& operator/= (DOUBLE div);	Scale
CSeg& operator<=<=(ANGLE a);	Rotate ccw
CSeg& operator>>=(ANGLE a);	Rotate cw
bool operator==(const CSeg& src) const;	Same
bool operator!=(const CSeg& src) const;	Not same
Manipulators	
CSeg& Reset ();	Set to 0,0
CSeg& Add (const CPnt& d);	Vector translation
CSeg& Subtract (const CPnt& d);	Vector translation
CSeg& Rotate (ANGLE a);	Absolute rotation
CSeg& Scale (DOUBLE mx, DOUBLE my);	Anisotropic scale
CSeg& Mirror (UNIT xOrg = 0);	Reflect the x-axis
CSeg& Transpose ();	Diagonal reflection
CSeg& Round (UNIT xGrid, UNIT yGrid);	Round to nearest
CSeg& RoundDown(UNIT xGrid, UNIT yGrid);	Round toward 0,0
CSeg& RoundUp (UNIT xGrid, UNIT yGrid);	Round away 0,0
CSeg& Reverse ();	Swap endpoints
Operations	
void UpdateExtent (CRct& ext) const;	
void Sillouette (CGon& gon, UNIT segWidth, UNIT oversize = 0);	
ANGLE AngleFrom (const CPnt& org) const;	
ANGLE AngleFrom2 (const CPnt& org) const;	
ANGLE AngleBetween (const CSeg& seg2) const;	
ANGLE Slope () const;	
ANGLE Slope2 () const;	
ANGLE PerpSlope () const;	

DOUBLE AngularCoef () const;
UNIT Length () const;
UNIT ManhattanLength () const;
UNIT Distance (const CPnt& p) const;
UNIT Distance (const CCir& c) const;
UNIT Distance (const CSeg& s) const;
UNIT Distance (const CArc& a) const;
UNIT Distance (const CRct& r) const;
UNIT Distance (const CGon& g) const;
UNIT Distance (const CPnt& p, CSeg& connector) const;
UNIT Distance (const CCir& c, CSeg& connector) const;
UNIT Distance (const CSeg& s, CSeg& connector) const;
UNIT Distance (const CArc& a, CSeg& connector) const;
UNIT Distance (const CRct& r, bool filled, CSeg& connector) const;
UNIT Distance (const CGon& g, bool filled, CSeg& connector) const;
CPnt ClosestPoint (const CPnt& p) const;
CPnt ClosestPoint (const CSeg& s) const;
CPnt ClosestPoint (const CArc& a) const;
CPnt ClosestPoint (const CRct& r) const;
CPnt ClosestPoint (const CGon& g) const;
Bool ClipIn (const CRct& r, CSeg& res) const;
ISECTIntersect (const CSeg& s, CPnt& res) const;
ISECTProjIntersectVert (const CSeg& s, CPnt& result) const;
ISECTProjIntersect (const CSeg& s, CPnt& result) const;
INT32 Intersect (const CRct& r, CPnt& res1, CPnt& res2) const;
INT32 ClipIn (const CGon& g, CSimpleSegList& res) const;
INT32 ClipOut (const CRct& r, CSimpleSegList& res) const;
INT32 ClipOut (const CGon& g, CSimpleSegList& res) const;
void MakePerp ();
Tests

bool IsVert () const;
bool IsHorz () const;
bool Is45 () const;
bool IsEqual (const CSeg& p) const;
bool IsCollinear (const CPnt& p) const;
bool IsCollinear (const CGon& g) const;
bool IsOn (const CSeg& s) const;
bool IsIn (const CRct& r) const;
bool IsIn (const CGon& g) const;
bool IsClipped (const CRct& box) const;
bool IsJoined(const CSeg& s) const;
bool IsJoined(const CArc& s) const;
bool IsBelow (const CPnt& p) const;
bool IsLeft (const CPnt& p) const;
bool IsParallel (const CSeg& seg2) const;

Arc.h – Declares CArc class

The CArc class provides a rich set of operations for manipulating 2-D arcs. It is one of the few classes that allow public access to its data members.

Data members		
CPnt	c;	Arc center
CPnt	p1;	Start of arc
CPnt	p2;	End of arc
Constructors		
CArc ();		

CArc (const CArc& src);	
CArc (const CPnt& c, const CPnt& p1, const CPnt& p2);	
CArc (const CPnt& center, UNIT rad, ANGLE start, ANGLE end);	
CArc (UNIT xc, UNIT yc, UNIT x1, UNIT y1, UNIT x2, UNIT y2);	
Operators	
CArc operator~ () const;	Mirror
CArc operator- () const;	Negate
CArc operator+ (const CPnt& p) const;	Add
CArc operator- (const CPnt& p) const;	Subtract
CArc operator* (DOUBLE m) const;	Scale
CArc operator/ (DOUBLE d) const;	Scale
CArc operator<< (ANGLE a) const;	Rotate ccw
CArc operator>> (ANGLE a) const;	Rotate cw
UNIT operator^ (const CPnt& p) const;	Distance
UNIT operator^ (const CSeg& s) const;	Distance
UNIT operator^ (const CArc& a) const;	Distance
UNIT operator^ (const CRct& r) const;	Distance
UNIT operator^ (const CGon& g) const;	Distance
CArc& operator= (const CArc& a);	Assignment
CArc& operator+= (const CPnt& p);	Offset
CArc& operator-= (const CPnt& p);	Offset
CArc& operator*= (DOUBLE mul);	Scale
CArc& operator/= (DOUBLE div);	Scale
CArc& operator<<= (ANGLE a);	Rotate ccw
CArc& operator>>= (ANGLE a);	Rotate cw
bool operator== (const CArc& src) const;	Same
bool operator!=(const CArc& src) const;	Not same
Manipulators	
CArc& Reset ();	
CArc& Add (const CPnt& d);	

CArc& Subtract (const CPnt& d);
CArc& Rotate (ANGLE a);
CArc& Scale (DOUBLE mx, DOUBLE my);
CArc& Mirror (UNIT xOrg = 0);
CArc& Transpose ();
CArc& Round (UNIT xGrid, UNIT yGrid);
CArc& RoundDown (UNIT xGrid, UNIT yGrid);
CArc& RoundUp (UNIT xGrid, UNIT yGrid);
CArc& Reverse ();
Operations
void UpdateExtent (CRct& ext) const;
UNIT Radius () const;
ANGLE StartAngle () const;
ANGLE EndAngle () const;
ANGLE Sweep () const;
ANGLE Bisect () const;
CPnt Apex () const;
CRct Box () const;
UNIT Distance (const CPnt& p) const;
UNIT Distance (const CCir& c) const;
UNIT Distance (const CSeg& s) const;
UNIT Distance (const CArc& a) const;
UNIT Distance (const CRct& r) const;
UNIT Distance (const CGon& g) const;
UNIT Distance (const CPnt& p, CSeg& connector) const;
UNIT Distance (const CCir& c, CSeg& connector) const;
UNIT Distance (const CSeg& s, CSeg& connector) const;
UNIT Distance (const CArc& a, CSeg& connector) const;
UNIT Distance (const CRct& r, bool filled, CSeg& connector) const;
UNIT Distance (const CGon& g, bool filled, CSeg& connector)

const;
ISECTIntersect (const CSeg& s, CPnt& r1, CPnt& r2) const;
ISECTIntersect (const CArc& a, CPnt& r1, CPnt& r2) const;
INT32 ClipIn (const CRct& r, CSimpleArcList& res) const;
INT32 ClipIn (const CGon& g, CSimpleArcList& res) const;
INT32 ClipOut (const CRct& r, CSimpleArcList& res) const;
INT32 ClipOut (const CGon& g, CSimpleArcList& res) const;
CPnt ClosestPoint (const CPnt& p) const;
CPnt ClosestPoint (const CSeg& s) const;
CPnt ClosestPoint(const CArc& a) const;
CPnt ClosestPoint(const CRct& r) const;
CPnt ClosestPoint(const CGon& g) const;
Tests
bool IsEqual (const CArc& p) const;
bool IsOn (const CArc& a) const;
bool IsIn (const CRct& r) const;
bool IsIn (const CGon& g) const;
bool IsClipped (const CRct& box) const;
bool CommonEnds (const CSeg& s) const;
bool CommonEnds (const CArc& a) const;
bool PntInChord (const CPnt& p) const;
bool AngleInChord (ANGLE a) const;
bool PntInRadius (const CPnt& p) const;

Rct.h – Declares CRct class

The CRct class provides a rich set of operations for manipulating 2-D rectangles. It is one of the few classes that allow public access to its data members.

Constructors		
CRct ();		
CRct (const CRct& src);		
CRct (const CPnt& ll, const CPnt& ur);		
CRct (UNIT w, UNIT h);		
CRct (const CPnt& org, UNIT w, UNIT h);		
CRct (const CPnt& org, UNIT radius);		
CRct (const CCir& c);		
CRct (const CSeg& s);		
CRct (const CArc& a);		
CRct (const CGon& g);		
CRct (UNIT l, UNIT b, UNIT r, UNIT t);		
Operators		
CRct operator~	() const;	Mirror
CRct operator+	(const CPnt& p) const;	Add
CRct operator-	(const CPnt& p) const;	Subtract
CRct operator*	(DOUBLE m) const;	Scale
CRct operator/	(DOUBLE d) const;	Scale
CRct operator<<	(ANGLE a) const;	Rotate ccw
CRct operator>>	(ANGLE a) const;	Rotate cw
UNIT operator^	(const CPnt& p) const;	Distance
UNIT operator^	(const CSeg& s) const;	Distance
UNIT operator^	(const CArc& a) const;	Distance
UNIT operator^	(const CRct& r) const;	Distance

UNIT operator^	(const CGon& g) const;	Distance
CRct&operator=	(const CRct& r);	Assignment
CRct&operator+=	(const CPnt& p);	Offset
CRct&operator-=	(const CPnt& p);	Offset
CRct&operator*=	(DOUBLE mul);	Scale
CRct&operator/=	(DOUBLE div);	Scale
CRct&operator<<=	(ANGLE a);	Rotate ccw
CRct&operator>>=	(ANGLE a);	Rotate cw
CRct&operator =	(const CPnt& pt);	Union
CRct&operator =	(const CRct& r);	Union
bool operator==	(const CRct& src) const;	Same
bool operator!=	(const CRct& src) const;	Not same
Manipulators		
CRct&Reset	();	Set to 0,0
CRct&Invalidate	();	Inverse world extent
CRct&Sanitize	();	Correct order
CRct&Add	(const CPnt& d);	Vector translation
CRct&Subtract	(const CPnt& d);	Vector translation
CRct&Scale	(DOUBLE mx, DOUBLE my);	Anisotropic scale
CRct&Mirror	(UNIT xOrg = 0);	Reflect the x-axis
CRct&Transpose	();	Diagonal reflection
CRct&Rotate	(ANGLE a);	Absolute rotation
CRct&Round	(UNIT xGrid, UNIT yGrid);	Round to nearest
CRct&RoundDown	(UNIT xGrid, UNIT yGrid);	Round toward 0,0
CRct&RoundUp	(UNIT xGrid, UNIT yGrid);	Round away 0,0
CRct&Expand	(UNIT radius);	Add radius
CRct&Expand	(UNIT rx, UNIT ry);	Add radii

CRct&Contract	(UNIT radius);	Subtract radius
CRct&Contract	(UNIT rx, UNIT ry);	Subtract radii
Operations		
UNIT Width	() const;	
UNIT Height	() const;	
UNIT LargeAxis	() const;	
UNIT SmallAxis	() const;	
DOUBLE Area	() const;	
CPnt Center	() const;	
CPnt Corner	(RctCrnId crn) const;	
void Edge	(RctEdgId edge, CSeg& seg) const;	
CRct Overlap	(const CRct& r) const;	
void UpdateExtent	(CRct& ext) const;	
UNIT Distance	(const CPnt& p) const;	
UNIT Distance	(const CCir& c) const;	
UNIT Distance	(const CSeg& s) const;	
UNIT Distance	(const CArc& a) const;	
UNIT Distance	(const CRct& r) const;	
UNIT Distance	(const CGon& g) const;	
UNIT Distance	(const CPnt& p, CSeg& connector) const;	
UNIT Distance	(const CCir& c, CSeg& connector) const;	
UNIT Distance	(const CSeg& s, CSeg& connector) const;	
UNIT Distance	(const CArc& a, CSeg& connector) const;	
UNIT Distance	(const CRct& r, bool filled, CSeg& connector) const;	
UNIT Distance	(const CGon& g, bool filled, CSeg& connector) const;	
CPnt ClosestPoint	(const CPnt& p) const;	
CPnt ClosestPoint	(const CSeg& s) const;	
CPnt ClosestPoint	(const CArc& a) const;	
CPnt ClosestPoint	(const CRct& r) const;	

CPnt	ClosestPoint	(const CGon& g) const;
Tests		
bool	IsEmpty	() const;
bool	IsInvalidated	() const;
bool	IsEqual	(const CRct& p) const;
bool	IsIn	(const CRct& r) const;
bool	IsIn	(const CGon& g) const;
bool	IsClipped	(const CRct& box) const;
bool	PntInRct	(const CPnt& p) const;

Gon.h – Declares CGon (Polygon) class

The CGon class provides a rich set of operations for manipulating 2-D polygons.

Though unified with the other 2D primitives, the CGon class deserves a bit of extra attention. First, all of its data members are private because the array of vertices needs to be managed. There are operators that hide this private nature such as the array subscript operator []. CGon's have to be declared with a known number of vertices or have to grow with AddVertex/InsertVertex. You cannot arbitrarily access a vertex whose index is greater than that reported by GetVertexCount.

CGon's can take on a definite shape by designating them as Circles or Arcs with SetCircle and SetArc. A circular CGon has only two vertices. Vertex 0 is the circle center and the X member of vertex 1 is the circle radius. Arcs contain 3 vertices: Vertex 0 is the arc center, vertex 1 is the start of the arc, and vertex 2 is the end of the arc. The

arc is always assumed to travel counter-clockwise. There is a convenient function called MakeArc which returns a CArc object from arc polygon. You can then perform all the normal arc operations.

Besides a CPnt specifying the location of a vertex, the vertices of a CGon also contain an extra DWORD data member that is managed but not defined in the CGon itself. For instance, a route object in the PCB123 database uses this extra DWORD to encode per-segment width and layer information.

CGons can also be defined as Filled and not Filled. A filled CGon implies a closed polygon and the SetFilled function will ensure this. All the distance functions take the filled attribute into account. The distance from an object to a filled polygon is zero if the object is inside the polygon.

Constructors			
CGon ();			Default constructor
CGon (const CGon& src);			Copy constructor
CGon (const CSeg& seg);			Construct from seg
CGon (const CArc& arc);			Construct from arc
CGon (const CRct& rct, bool filled = false);			Construct from rect
CGon (INT32 vertices, bool filled = false);			Init corner count
Attributes			
bool	GetFilled	() const	Is it filled?
void	SetFilled	(bool filled)	Set to filled
bool	GetArc	() const	Is it an arc?
void	SetArc	(bool arc)	Set to arc
bool	GetCircle	() const	Is it a circle?
void	SetCircle	(bool circ)	Set to circle

Operators		
CGon operator~	() const;	Mirror
CGon operator-	() const;	Negate x and y
CGon operator+	(const CPnt& p) const;	Add
CGon operator-	(const CPnt& p) const;	Subtract
CGon operator*	(DOUBLE m) const;	Scale
CGon operator/	(DOUBLE d) const;	Scale
CGon operator<<	(ANGLE a) const;	Rotate ccw
CGon operator>>	(ANGLE a) const;	Rotate cw
UNIT operator^	(const CPnt& p) const;	Distance
UNIT operator^	(const CSeg& s) const;	Distance
UNIT operator^	(const CArc& a) const;	Distance
UNIT operator^	(const CRct& r) const;	Distance
UNIT operator^	(const CGon& g) const;	Distance
CGon& operator=	(const CGon& g);	Assignment
CGon& operator+=	(const CPnt& p);	Offset
CGon& operator-=	(const CPnt& p);	Offset
CGon& operator*=	(DOUBLE mul);	Scale
CGon& operator/=	(DOUBLE div);	Scale
CGon& operator<<=	(ANGLE a);	Rotate ccw
CGon& operator>>=	(ANGLE a);	Rotate cw
bool operator==	(const CGon& src) const;	Same
bool operator!=	(const CGon& src) const;	Not same
CPnt operator[]	(INT32 idx);	Subscript operator
Manipulators		
void Reset	(INT32 vertices = 0);	
void Add	(const CPnt& d);	
void Subtract	(const CPnt& d);	
void Rotate	(ANGLE a);	
void Scale	(DOUBLE mx, DOUBLE my);	

void	Mirror	(UNIT xOrg = 0);
void	MirrorY	(UNIT yOrg = 0);
void	Transpose	();
void	AddVertex	(const CPnt& pt, DWORD data = 0);
void	InsertVertex	(INT32 at, const CPnt& pt, DWORD data = 0);
void	AddVertex	(const CCrn& crn);
void	InsertVertex	(INT32 at, const CCrn& crn);
void	DeleteVertex	(INT32 at);
void	GetAt	(INT32 at, CPnt& pt);
void	SetAt	(INT32 at, const CPnt& pt);
void	GetAt	(INT32 at, DWORD& data);
void	SetAt	(INT32 at, DWORD data);
void	GetAt	(INT32 at, CCrn& crn);
void	SetAt	(INT32 at, const CCrn& crn);
Void	GetSegmentAt	(INT32 at, CSeg& s) const;
void	Reverse	();
void	Close	();
void	ApproxCircle	(const CPnt& center, UNIT radius, INT32 verts);
void	ApproxArc	(const CArc& arc, INT32 verts);
Operations		
INT32	GetVertexCount	() const;
CPnt	GetAt	(INT32 at) const;
CCrn	GetCrnAt	(INT32 at) const;
DWORD	GetDataAt	(INT32 at) const;
CSeg	GetSegmentAt	(INT32 at) const;
CRct	GetExtent	() const;
CArc	MakeArc	() const;
void	UpdateExtent	(CRct& ext) const;
UNIT	Distance	(const CPnt& p) const;
UNIT	Distance	(const CCir& c) const;

UNIT	Distance	(const CSeg& s) const;
UNIT	Distance	(const CArc& a) const;
UNIT	Distance	(const CRct& r) const;
UNIT	Distance	(const CGon& g) const;
UNIT	Distance	(const CPnt& p, CSeg& connector) const;
UNIT	Distance	(const CCir& c, CSeg& connector) const;
UNIT	Distance	(const CSeg& s, CSeg& connector) const;
UNIT	Distance	(const CArc& a, CSeg& connector) const;
UNIT	Distance	(const CRct& r, bool filled, CSeg& connector);
UNIT	Distance	(const CGon& g, bool filled, CSeg& connector);
CPnt	ClosestPoint	(const CPnt& p) const;
CPnt	ClosestPoint	(const CSeg& s) const;
CPnt	ClosestPoint	(const CArc& a) const;
CPnt	ClosestPoint	(const CRct& r) const;
CPnt	ClosestPoint	(const CGon& g) const;
bool	Intersect	(const CSeg& seg, TOrderedList<CSeg>& results);
void	Sanitize	();
INT32	Reduce	(INT32 startCrn, INT32 endCrn, INT32 trackCrn);
INT32	Unwind	(INT32 trackCrn);
INT32	Reverse	(INT32 trackingCorner);
void	ClipIn	(const CRct& r, CGon& out) const;
DOUBLE	Area	() const;
ANGLE	TurningAngle	(INT32 idx) const;
INT32	ShiftCorners	(INT32 trackingCrn);
bool	GetTurn	(INT32 crn, CPnt& pa, CPnt& pb, CPnt& pc);
Tests		
bool	Inside	(const CPnt& p) const;
bool	IsEqual	(const CGon& g) const;
bool	IsIn	(const CRct& r) const;

bool	IsIn	(const CGon& g) const;
bool	IsClipped	(const CRct& box) const;
bool	IsClosed	() const;
bool	IsFilled	() const;
bool	IsCCW	() const;
bool	IsLeftTurn	(INT32 crnIdx) const;
bool	PntOnSeg	(const CPnt& pnt, CSeg& seg) const;
bool	PntOnSeg	(const CPnt& pnt, INT32& crnIdx) const;
bool	PntOnCrn	(const CPnt& pnt, INT32& crnIdx) const;
bool	IsCrnOnSimulatedArc	(INT32 idx, INT32& firstArcCrn, INT32& lastArcCrn, CArc& resultArc, UNIT epsilon) const;

Fnt.h – Declares a stroked font

Fnt.h declares a stroked font. A stroked font defines each character using line segments. The font is described using an arbitrary but consistent character cell. Rendering a text string involves calling GetStrokeSegs and specifying the desired font height, string location, and rotation. The string can contain embedded line breaks for multi-line text.

PolyTri.h – Triangulates a CGon object

The CPolygonTriangulation class take a CGon object for input and produces a list of Triangle objects. Detecting if a point is inside a

triangle is an extremely fast operation and is not plagued by round-off errors and special cases. It is therefore advantageous to decompose a polygon into triangles to perform a Point-In-Poly test which is what the CGon's Inside operation does.

DrawLib

The DrawLib directory contains the public interfaces into DrawLib.dll. This library started life primarily as a virtual viewport but has since become a repository for several custom controls. Only the viewport and supporting classes are documented here.

The act of implicitly or explicitly linking to DrawLib requires no special consideration. It will automatically link in 2dLib.dll and CoreLib.Dll which it is dependent on. This library may consume a small amount of memory in the form of cached pens and brushes.

Viewport.h – Virtual viewport

The CDrawViewport class provides a virtual viewport manager that maps 32-bit world coordinates to 16-bit screen coordinates. To use this class you first call SetWindow or, in the case of printing, SetPage and supply the x,y window or page size in pixels. You can then call ZoomTo and supply the view center and view radius in world coordinates. This class provides functions that map various structures between world and screen coordinates.

DrawCache.h – Graphics resource manager

The CDrawCache class is derived from CDrawViewport and adds to it the ability to manage pens and brushes. It declares a new data type called an HDRAW which is a handle to a brush/pen pair. The brush and pen are obtained by a call to GetBrush or GetPen and specifying the color, width, etc. These functions will quickly return the brush or pen, even if it has to create a new one. If one is created, it is remembered for future use. The HDRAW handle is used by the next class, described below.

Cookie.h – Primitive shape drawing functions

The CDrawCookie class, which is derived from the CDrawCache class retains drawing state information and provides functions to draw the various 2D primitives in their world coordinates.

The state information retained is drawing mode such as dragging (XOR) erasing, (draw in background color), and translucency. The other important state data is a Windows HDC (actually an MFC CDC*). This is usually supplied with a call to CDrawCookie::BeginTransaction and is return with a call to CDrawCookie::EndTransaction.

The various Draw functions take a 2D primitive and an HDRAW for arguments. The program will assert if a drawing frame has not been set up with a call to BeginTransaction first.

BaseDbLib

The BaseDbLib directory contains the public interfaces into BaseDbLib.dll. BaseDbLib contains the database classes that are fundamental to PCB database objects and any future databases such as a new schematics program.

The act of implicitly or explicitly linking to BaseDbLib requires no special consideration. It will automatically link in DrawLib.dll, 2dLib.dll and CoreLib.Dll which it is dependent on. This library initially consumes no memory but classes such as the Transaction Manager which is responsible for managing the Undo/Redo lists can consume quite a bit of memory.

This library defines the CDbBase object, which is the base class for all database objects in the PCB123 system. CDbBase mainly defines interfaces for derived classes. In fact, CDbBase contains only one data member and that is a reference to a parent (owner) object. All objects in a PCB123 database are expected to be owned by other objects unless it is the one and only root object. For instance, a pin might be owned by a component which might be owned by a board. The board would be root.

There are two distinctly different methods of interfacing with a PCB123 database object. The first is the traditional way of using the different get/set methods of an object to access and update its data members. The other method is not so traditional and is referred to as the Atomic interface. It is this interface which you will primarily want to use when manipulating an object.

When we write a program much effort is spent organizing the program into logical and functional 'blocks'. We create classes and structures which are intrinsic features of a language especially designed to facilitate the functional and logical partitioning of a program. We do all this only to have it torn away from us by the compiler. At run time, we can only ask of an object, 'where are you in memory' (address-of operator '&') and how big are you (sizeof operator). We cannot ask, for instance, "What are your data members and what type are they". This was the kind of question COM tries to answer. A COM object is basically a pure virtual object whose members have to be "discovered" at run time (gross simplification). PCB123 decided it too wanted answers to this question but it didn't want nor could afford the baggage associated with COM. The solution was the Atomic interface and it goes something like this:

Every class has a unique numerical identifier.

Every class data member has a unique numerical identifier.

The class id and the member id are combined to form a unique atomic id.

The type of every class data member is a type known to the atomic interface.

Given this, I can ask any object "what are your members and what type are they". This has some very interesting implications. For instance, when a PCB123 database is written to disk, a very tight loop simply iterates each field of each object and writes out the atomic id and the value associated with it. It does not know nor cares

what kind of object it is. When reading the database from disk, the reverse holds true.

The one very strict requirement for this to work is that the id's or their types never change. There are several internal tables that coordinate the atomic interface and they are all automatically created by defining the macros DB_CLASS_ENTRY and DB_FIELD_ENTRY in various way and #including the file DbClasses.inl. This file contains the master definitions of all database classes and their members.

Why is all this being mentioned? Because any changes you make to the PCB123 database want to be done through the Transaction Manager and the Transaction Manager works with the Atomic Interface.

For now, lets take a look at the header files in BaseDbLib.

BaseDbLib.h – Basic definitions and master include for BaseDbLib

One of the first things this file does is enumerate the various PCB123 database classes. When you iterate through a database you will get pointers-to-objects. Not until you call GetClass on that object will you know its type. The database classes are:

Class Enumeration	Description
BOARD_CLASS	Top-level container object of type CPcbBoard.
COMP_CLASS	Component. Type CPcbComp.
PIN_CLASS	Pin. Type CPcbPin.

POLY_CLASS	Polygon. Type CPcbPoly.
TEXT_CLASS	Text string. Type CPcbText.
NET_CLASS	Net. Type CPcbNet.
TRACK_CLASS	Track or route. Type CPcbTrack.
USERPREF_CLASS	User preferences. Type CPcbUserPref.
LAYERPREF_CLASS	Preferences by layer. Type CPcbLayerPref.
ERROR_CLASS	DRC marker. Type CPcbError.
DEVICE_CLASS	Device info. Type CPcbDevice.
PACKAGE_CLASS	Package or footprint. Type CPcbPackage.

The next set of definitions are DbUnits. The recognized unit types are:

InchUnits – Imperial inches. Abbreviation of In.

MilUnits – Imperial thousandths of an inch. Abbreviation of Mil.

CmUnits – Metric Centimeter units. Abbreviation of Cm.

MmUnits – Metric Millimeter units. Abbreviation of Mm.

There are two global functions that convert between numerical UNIT values and a string representation. They are:

CoreString UnitToString (UNIT value, DbUnits unitsType = InchUnits)

– Convert from UNIT value to string representation in given units. If no units specifier is given then the default is inches.

UNIT StringToUnit (CSTR strVal, DbUnits unitType) – Convert from a string to a numerical UNIT. If the string has a units suffix such as 0.254cm it will take precedence over the units specifier.

If your Plugin has a dialog box that asks for a value that will be converted to a UNIT you may want to qualify the input string with **ValidateUnit** first because StringToUnit will assert on invalid input.

There are a host of conversion macro supplied that allow you to convert between database UNIT's and physical values. Remember, db units are in ten-millionths of an inch. The conversion macros are:

Macro Name	Description
INCHES_TO_UNITS(i)	Convert from inches to UNITs
MILS_TO_UNITS(m)	Convert from mils to UNITs
CM_TO_UNITS(cm)	Convert from centimeters to UNITs
MM_TO_UNITS(mm)	Convert from millimeters to UNITs
UNITS_TO_INCHES(u)	Convert from UNITs to inches
UNITS_TO_MM(u)	Convert from UNITs to millimeters
UNITS_TO_MILS(u)	Convert from UNITs to mils
UNITS_TO_CM(u)	Convert from UNITs to centimeters

Prop.h – Properties

Most PCB123 database objects allow an arbitrarily long list of properties to be attached to them. A Property is a simple name/value pair.

The names of properties are not case sensitive. There is no restriction on the length of a property name. It is recommended that property names do not begin with a digit and it is recommended that they contain no embedded white-space.

Property values can be any length except zero. Setting a property to empty deletes the property. If a property value is a binary string, then it wants to use base64 encoding for safe transport when expressed as XML data.

There are currently two machine-generated properties that PCB123 uses. The first one is OrigName and is generated when a component is renamed for the first time. The other one is AutoGenMech and is attached to polygon and text objects during the creation of the fab and drill drawings.

TransactionManager.h – Transaction interface

The CTransactionManager class provides a mechanism where PCB123 objects can be modified and those modifications recorded for possible Undo and Redo. In addition, the transaction manager will notify any registered event handlers about the changes. In the case of a Plugin, it is not necessary to register an event handler because the PCB123 document, which is a handler, passes along these events to the Plugin anyway.

A modification done through the Transaction Manager is performed at the Atomic level. This means only that Atomic field is changed and recorded, and as a consequence the undo buffer consumes very little memory. So little memory, in fact, that PCB123 imposes no restrictions on the size of the undo buffer.

Before discussing how to make a change to the database we should talk briefly about what a change is. You can look at a change as

anything that modifies the database but in most cases it is advisable to view a change as a collection of modifications initiated by some user request. Viewed this way, many changes will occur in sets and want to be undone or redone as a set. There is a mechanism in place for that and it is to bracket the changes with calls to StartFrame and EndFrame. If you perform a database modification without doing so, a frame will automatically be created for you every time a modification is done. The Undo/Redo buffer always works on a frame and not bracketing sets of changes will result in the user having to perform an Undo many more times than they expect. The tutorial at the end of this document brackets the entire set of changes in a single frame. Though hundreds of routes may be modified a single undo will roll back the effect of the Plugin.

The Transaction Manager has four main object access functions but only three are typically used in hand-generated code. The three main functions are:

```
void AddChild (CDBBase* pParent, INT32 fieldId, CDBBase* pChild);  
void DeleteChild (CDBBase* pParent, INT32 fieldId, CDBBase*  
pChild);  
void Set (CDBBase* pObj, INT32 fld, someType val);
```

The other function is a complimentary Get function but there is no reason to use the Transaction Manager to get the value of some object's data member. This is because any modification to an object is immediately applied to that object and not shadowed in any way. The only reason to use the Get function is if you are writing code that exploits the Atomic interface to iterate over all fields in an object.

In support of object properties the Transaction Manager provides AddProp and DeleteProp. There is no SetProp because AddProp will perform the same function by simply replacing any property value that already exists under the same name.

In reality, there are several Get and Set functions. One for each data type the Atomic Interface supports. The current list of datatypes allowed by the Atomic Interface is:

Data Type	Description
bool	Boolean value. Pass by value.
INT32	32-bit integer. Pass by value.
STRID	String Id. Pass by value.
CDbBase*	Pointer to PCB123 object. Pass by value.
CPnt	Point object. Pass by reference.
CSeg	Line segment. Pass by reference.
CArc	Arc. Pass by reference.
CRct	Rectangle. Pass by reference.
CGon	Polygon. Pass by reference.

It also supports the CDbBaseList (list of CDbBase* objects) but this type is only manipulated through AddChild/DeleteChild.

Use INT32 for enumerations and typedefs such as UNIT.

NOTE: If you set a field that is of type CDbBase* to literal NULL, you will have to cast it to a CDbBase* first because NULL looks like an INT32. You can use the synonym for CDbBase* which is ObjPtr.

If you do not supply the correct argument type to the Get and Set functions the program will assert.

Base.h – Base class declaration

For all intents, the CDbBase class is the base class of all PCB123 database objects. Examining Base.h you will see this is not technically true. CDbBase is actually derived from a class named CFlagBits. This is a purely transient class. It is not persisted nor is there any Atomic interface for it. It is simply a collection of flags that are never to be trusted beyond some local task. There are cases where it makes sense for the sake of efficiency to be able to set a temporary bit for some calculation. For instance, a function that wants to make sure every pin has been visited once would use a temporary bit on the pin object in place of creating some mapping data structure. If you encounter a situation like this, it is permissible to use the m_bSignaled bit, accessed thorough **GetSignaled()** and **SetSignaled()**. It is considered good form to not only initialize the signaled state before entering your procedure but to also clear the bit when your procedure finishes.

As mentioned before, the file DbClasses.inl drives the creation of several tables and structures used by the system. It also drives the creation some commonly used type definitions declared in Base.h but it does so in a nasty way because it uses token-pasting to build up the identifiers. Even though you will not find them in the file, Base.h forward declares the following classes:

class CDbBase

```

class CPcbBase
class CPcbPropBase
class CPcbLayerPref
class CPcbUserPref
class CPcbPin
class CPcbComp
class CPcbNet
class CPcbTrack
class CPcbPoly
class CPcbText
class CPcbError
class CPcbBoard
class CPcbDevice
class CPcbPackage

```

It also declares lists and list iterators for the above classes (partial list shown):

```

typedef TOrderedList<CDBBase*>      CDBBaseList
typedef TIter<CDBBase*>             CDBBaseIter;

typedef TOrderedList<CPcbBase*>     CPcbBaseList
typedef TIter<CPcbBase*>            CPcbBaseIter;

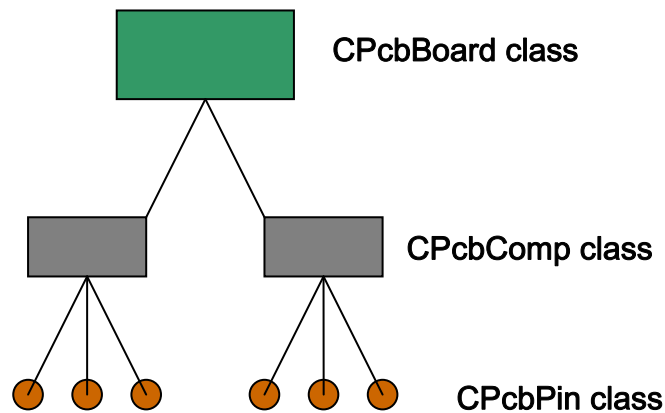
typedef TOrderedList<CPcbPin*>      CPcbPinList
typedef TIter<CPcbPin*>             CPcbPinIter;

typedef TOrderedList<CPcbComp*>     CPcbCompList
typedef TIter<CPcbComp*>            CPcbCompIter;

```

<code>typedef TOrderedList<CPcbNet*></code>	<code>CPcbNetList</code>
<code>typedef TIter<CPcbNet*></code>	<code>CPcbNetIter;</code>
<code>typedef TOrderedList<CPcbTrack*></code>	<code>CPcbTrackList</code>
<code>typedef TIter<CPcbTrack*></code>	<code>CPcbTrackIter;</code>
<code>typedef TOrderedList<CPcbPoly*></code>	<code>CPcbPolyList</code>
<code>typedef TIter<CPcbPoly*></code>	<code>CPcbPolyIter;</code>
<code>typedef TOrderedList<CPcbText*></code>	<code>CPcbTextList</code>
<code>typedef TIter<CPcbText*></code>	<code>CPcbTextIter;</code>

A PCB123 database is composed of a hierarchical collection of objects. There is always one root object that contains the entire collection of objects in the database. An example of this relationship is depicted below:



Because all objects except the root object are contained by another object, a pointer to the parent object is defined and implemented right in the `CDbBase` class. This is the only data member of a `CDbBase` class.

In theory, all PCB123 database objects can be containers for other objects but in practice this is not true. It happens that any renderable object in PCB123 are terminal objects that do not contain children. The table below shows the allowed relationships among the main database classes:

Class	Can be owned by	Can contain
CPcbBoard	Nothing	CPcbPin CPcbComp CPcbPoly CPcbText CPcbNet
CPcbComp	CPcbBoard	CPcbPin CPcbPoly CPcbText
CPcbPin	CPcbBoard CPcbComp	Nothing
CPcbPoly	CPcbBoard CPcbComp	Nothing
CPcbText	CPcbBoard CPcbComp	Nothing
CPcbNet	CPcbBoard	CPcbTrack
CPcbUserPref	CPcbBoard	CPcbLayerPref

The CDbBase class defines a virtual interface for dealing with child lists. Classes that allow children will provide a CDbBaseList data member and override this virtual interface.

The CDbBase class parent/child interface is as follows:

CDbBase*	GetParent	() const
CDbBase*	GetParentRef	() const
Void	SetParent	(CDbBase* pParent)
virtual void	AddChild	(CDbBase* pChild)
virtual void	DeleteChild	(CDbBaselter& iter)
virtual CDbBaselter	GetFirstChild	() const
virtual INT32	GetChildCount	() const

The CDbBase class defines and implements an object name interface. The interface is as follows:

virtual STRID	GetName	() const
virtual void	SetName	(STRID name)
virtual void	GetQualifiedName	(CoreString& name)

The GetQualifiedName function walks up the object hierarchy pre-pending the parent name to the object name separated by a dot ‘.’

There are a couple of pure virtual functions that are always overridden in derived classes. They are:

```
virtual DbClass GetClass () const // Return the class id
virtual CDbBase* Copy () const   // Deep copy this object
```

The last interface CDbBase declares has to do with some fundamental geometric operations. They are:

```
virtual CRct GetExtent () const  
virtual CRct GetRawExtent () const  
virtual void GetTransform (GeoTransform& trans) const
```

The GetTransform function will ascend the object hierarchy accumulating object transforms as it goes. When it returns trans contains the final transformation the object must undergo for rendering.

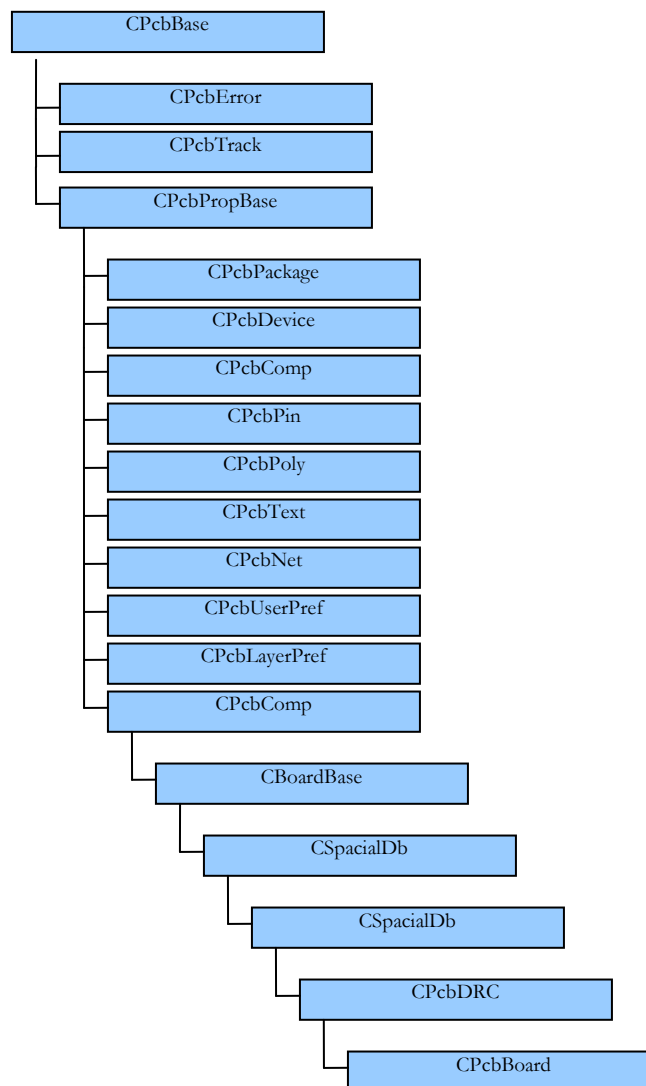
The GetRawExtent returns the relative extent of the object without considering any transforms.

The GetExtent function returns the final bounding rectangle of this object after undergoing all transformations.

PcbDbLib

The PcbDbLib directory contains the public interfaces into PcbDbLib.dll. PcbDbLib contains all the PCB database classes and database services.

Here is what the class graph looks like for PcbDbLib



Notice there is no Layer class even though a layer is a physical entity. A layer is just an attribute of certain objects and there are sets of rules (or preferences) on a per-layer basis.

In PCB123 the internal layer designations are fixed and can be found in the file Layers.inl.

Each of the following database classes will be described in five parts. They are:

Data Graph: This is derived from the actual schema for our XML-based NLF format. It shows the parent-child relationship for the objects and describes the expected or allowed attributes for the object. XML attributes are the mechanism used to store object data members. There is not always a one-to-one mapping between attributes and data members because the values of some data members are inferred from the object relationships.

Data Members: This table lists each data member of an object. The Field column is the name of the data member. You can add “Get” or “Set” to the beginning of any field and it will map to a valid function. The next column is the data type. The Get-functions all return this type, and the Set-functions all accept a single argument of this type. The next column is the AtomicId of the field to use when using the transaction manager to update an object. The last column is a brief description of the data member.

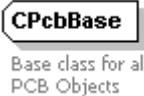
Base Class Overrides: This table lists all the base class interfaces that the class implements. It just lists the function names along with a

brief description. You will have to look at the header for the function arguments.

Operations: This table lists the unique operations that the class provides. Again, it only lists the function name and a description. You will have to look at the header for the function arguments.

Class description: Explains salient features of the class.

PcbBase.h – Base class declaration for PCB objects

diagram						
type	extension of <u>CDbBase</u>					
properties	base CDbBase					
used by	complexTypes <u>CPcbPropBase</u> <u>CPcbTrack</u>					
attributes	Name Nm	Type xs:string	Use optional	Default	Fixed	Annotation
annotation	documentation Base class for all PCB Objects					

DATA MEMBERS			
Field	Type	Atomic Id	Description
Name	STRID	Base_Name	Name of object

BASE CLASS OVERRIDES	
Override	Description
GetQualifiedName	Return name of object lineage

GetName	Returns unqualified name
SetName	Sets object name

OPERATIONS		
Name	Virtual	Description
IsNameLegal	Yes	Check is name would be allowed for object
AddProp	Yes	Add a property to an object
DeleteProp	Yes	Delete a property from an object
GetFirstProp	Yes	Return property iterator for an object
GetPropertyCount	Yes	Returns number of properties object has
GetPropertyValue	Yes	Return value of named property
FindProp	Yes	Attempt to find a named property
GetLayer	Yes	Return layer an object is on
IsOnLayer	Yes	Test if object is on a layer
GetNetRef	Yes	Return a const reference to object's net
GetSillouette	Yes	Return convex outline of object
GetHotSpot	Yes	Returns closest snap-point to given point
GetPrimitives	Yes	Return list of shapes that make up object
GetDistance	Yes	Get distance and closest point to other object

CPcbBase Description

The CPcbBase class, like the CDbBase class, has only one data member but defines several interfaces. The data member is a concrete instantiation of the Name interface declared in CDbBase with one exception: it also declares a virtual function called IsNameLegal which allows each PCB class to implement its own rules with regards to names. Some name rules can be fairly complex. For instance, a board-level pin can have any name at all, whereas a component pin cannot be duplicated within the component with the exception of unnamed pins. Multiple unnamed pins in a component are acceptable.

The abstract interfaces that CPcbBase defines are as follows:

Property interface: The CPcbBase class does not contain a property list but does define the interface for those classes who do. The base class property functions, if not overridden will store nothing and return empty results.

The property functions are:

virtual void	AddProp (const CDbProp& prop)
virtual void	AddProp (CSTR name, CSTR value)
virtual void	AddProp (CSTR name, STRID value)
virtual void	AddProp (CSTR name, INT32 value)
virtual void	AddProp (CSTR name, DOUBLE value)
virtual void	DeleteProp (CDbPropIter iter)
virtual void	DeleteProp (CSTR name)
virtual CDbPropIter	GetFirstProp () const
virtual INT32	GetPropCount () const
virtual STRID	GetPropValue (CSTR name)
virtual bool	FindProp (CSTR name)

Common attributes interface: Many objects reside on a layer, and many objects belong to a net. The base class functions, if called, will return PcbLayerNull for a layer and NULL for a net.

```
virtual PcbLayer      GetLayer () const
virtual bool          IsOnLayer (PcbLayer layer) const
virtual const CPcbNet* GetNetRef () const
```

PcbProp.h – Base class for objects with Properties

diagram	<pre> classDiagram class CPcbPropBase { <<abstract>> } class Property { type CDbProp } CPcbPropBase "1..∞" --> Property </pre> <p>PCB object with properties</p>					
type	extension of <u>CPcbBase</u>					
properties	base CPcbBase					
children	<u>Property</u>					
used by	complexTypes <u>CPcbBoard</u> <u>CPcbComp</u> <u>CPcbDevice</u> <u>CPcbNet</u> <u>CPcbPackage</u> <u>CPcbPin</u> <u>CPcbText</u> <u>PcbShape</u>					
Attributes	Name Nm	Type xs:string	Use optional	Default	Fixed	Annotation
annotation	documentation PCB object with properties					
diagram	<pre> classDiagram class Property { type CDbProp } </pre>					
type	<u>CDbProp</u>					
properties	isRef 0 minOcc 0 maxOcc 1 content complex					
attributes	Name Name Value	Type xs:string xs:string	Use required required	Default	Fixed	Annotation

DATA MEMBERS			
Field	Type	Atomic Id	Description
PropList	CPropList	Prop_List	List of properties

BASE CLASS OVERRIDES	
Override	Description
AddProp	Add a property to the list
DeleteProp	Delete a property from the list
GetFirstProp	Return property list iterator
GetPropCount	Return number of properties in list
GetPropValue	Get value of named property
FindProp	Find a property with the given name

CPcbPropBase Description

The CPcbPropBase class simply adds a list of CDbProp objects to the CPcbBase class. The name of a CDbProp is case insensitive. Only one property with any given name can be stored in a property list. Any attempt to add a second property with the same name will simply replace the value of the original.

There are no built-in properties though there are two system-generated properties that use the following name and value:

OrigName – This property will be added to component objects if they are renamed for the first time. The property value is, as implied, the original component name (reference designator). If this property already exists for a component, it is not updated. The intent here is

that they will only get reset when it is known that the component rename list has been back-annotated to a schematic. (future use).

AutoGenMech – Text and Polygon objects that are automatically generated when creating the Drill drawing and Fab drawing will obtain this property. The value is unimportant, the simple presence of this property signals the drawing generators that the object should be destructed to make way for new ones.

Other than those two, you are free to hang any amount of properties, whose names are virtually anything, and whose values are unrestricted. If you want to hang binary data off an object, then it is highly recommended that you perform a base64 encoding on the data for safe transport in an XML stream.

PcbError.h – DRC error marker class

DATA MEMBERS			
Field	Type	Atomic Id	Description
Reason	Reason	Error_Reason	What went wrong
Layer	PcbLayer	Error_Layer	Layer for DRC marker
Loc	CPnt	Error_Loc	Location of marker
Obj1	CPcbBase*	Error_pObj1	First object in conflict
Obj2	CPcbBase*	Error_pObj2	Second object in conflict
Message	STRID	Error_AdditionalMsg	Extra information

			about error
--	--	--	-------------

BASE CLASS OVERRIDES	
Override	Description
Copy	Makes a deep copy of object
GetExtent	Get object bounds in world coords
GetRawExtent	Get object bounds in local coords
GetTransform	Get transform to world cords
GetLayer	Returns layer object is on

OPERATIONS	
Name	Description
GetErrorString	Returns a computed string with marker details
GetShortErrorString	Returns a simplified marker string
GetSeverity	Returns the severity code of the marker

CPcbError Description

The CPcbErrorMarker class is somewhat of an oddity in that the entire class is not persisted to disk. DRC error markers are highly transient in nature and can be generated or removed dynamically as the board is modified. Due to some setup requirements we have not exposed the full DRC engine directly to the Plugins. The only DRC API you have direct access to is CheckConn, which will DRC all or part of a single CPcbTrack object and return a PASS/FAIL status. It will not actually generate a DRC marker. You can, however, post-back a request for the system to run a DRC as though the user clicked it from the menu.

Errors severity is classified as Informational, Warning, Severe, and FabError. There is another class called ShowStopper but has not been used yet.

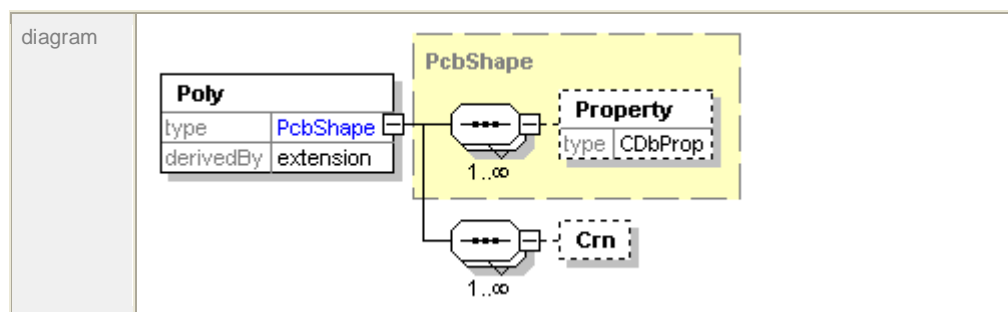
A CPcbError object contains a Reason for existing. It also contains up to two CPcbBase* objects that are in conflict, possibly a Layer that the DRC marker resides on, a location to display the DRC marker at, and optionally a custom message to display when queried.


If you are writing a Plugin that performs a custom DRC-like check and would like to use the DRC marker interface then we have created a special Reason named PluginReason that you should use along with a custom error message describing the marker.

There are a host of methods to interrogate the CPcbError marker with.

You should not alter any DRC marker that you did not create. It can have unintended consequences.

PcbPoly.h – Polygon Class Declaration



type	extension of PcbShape					
properties	isRef 0 content complex					
children	<u>Property Crn</u>					
attributes	Name Nm	Type xs:string	Use optional	Default	Fixed	Annotation
	Typ	derived by: xs:NMTOKEN	optional			
	Lyr	derived by: xs:NMTOKEN	required			
	Wid	derived by: xs:NMTOKEN	required			
	Net	xs:string	optional			
diagram						
properties	isRef 0 minOcc 0 maxOcc unbounded content complex					
attributes	Name Idx	Type xs:nonNegativeInteger	Use required	Default	Fixed	Annotation
	x	<u>UNIT</u>	required			
	y	<u>UNIT</u>	required			

DATA MEMBERS			
Field	Type	Atomic Id	Description
Net	CPcbNet*	Poly_pNet	Net reference
Layer	PcbLayer	Poly_Layer	Layer poly is on
Type	PolyType	Poly_Type	Type of polygon
Arc	Bool	Poly_bArc	Poly is an arc
Width	UNIT	Poly_Width	Width of poly
Pts	CGon	Poly_Pts	The poly corners

BASE CLASS OVERRIDES	
Override	Description

Copy	Makes a deep copy of object
GetExtent	Get object bounds in world coords
GetRawExtent	Get object bounds in local coords
GetTransform	Get transform to world cords
GetQualifiedName	Return name of object lineage
GetLayer	Returns layer object is on
IsOnLayer	Check if object resides on a layer
GetNetRef	Get a const pointer to object's net
GetInfoString	Get info about object
GetHotSpot	Snap point to major feature of object
GetPrimitives	Get list of shapes for whole object
GetDistance	Get distance from another object

OPERATIONS	
Name	Description
MakeArc	If arc poly, returns a CArc object
HitTest	Returns where on poly given point is
Triangulate	Decompose poly into triangles (internal list)
GetFirstTriangle	Returns iterator for triangle list
PntInPolyTri	Test if point is in a poly triangle
GetVertexCount	Returns number of poly corners
AddCrn	Add a corner to poly
DeleteCrn	Delete a corner from poly
GetRawLayer	Returns layer poly was defined on

CPcbPoly Description

A CPcbPoly object is a renderable shape object. It is used to define the board outline, copper pour regions, component silk screen outlines, and other entities.

The CPcbPoly object encapsulates a CGon primitive object and augments it with additional physical characteristics. Besides a general polygonal shape, the CPcbPoly class can represent a circle or an arc. The number of corners (vertices) a CPcbPoly object can have is basically unlimited but circles will always have 2 vertices and arcs 3. You can check if a CPcbPoly object is a circle by calling the GetCircle member function. Likewise, the GetArc member function is used to check for an arc. Corner 0 of a circle contains the x,y coordinates of the circle center, and the x-member of corner 1 contains the circle radius. Corner 0 of an arc is the arc center, corner1 is the beginning of the arc, and corner 2 is the end of the arc. Arcs are always assumed to travel counter-clockwise from start to end. Because circles (and indirectly arcs) store a radius, one with an odd diameter cannot be created. This is usually not a problem because of the extremely fine grain of database units (one 10-millionth of an inch).

A CPcbPoly object is planar and resides on one layer. The only exception to this is a board outline object which is a special case and assumed to be uniformly defined on all layers. The GetLayer member function for CPcbPoly objects will return a “cooked” value that may not reflect the layer the polygon was created on. Calling GetRawLayer will return the original layer. Cooking may change the layer if, for instance, the polygon was created on the top silkscreen layer of a component and that component was placed on the bottom of the board. In that instance the polygon is now effectively on the bottom silkscreen layer.

CPcbPoly objects do not define specific closed or filled attributes but instead are assigned a type whose attributes are implied. This type

(or useage) enumeration is called a PcbPolyType. The current types are:

BoardType – Signals that the polygon is the board outline. This type of CPcbPoly object is assumed to be a closed, hollow polygon or circle. The line width for board outlines is hard-coded to 26 mils to ensure a mandatory 13-mil clearance from the board edge.

FilledType – A solid-filled polygon. It is assumed to be closed and have zero width. Copper pour areas are FilledType CPcbPoly objects that have been assigned to a net. A lot of special processing is done on copper pour polygons whose results are not accessible to a Plugin. Only the border of a copper pour area is stored in a CPcbPoly copper pour. The interior cutouts that you see on the screen are calculated on-the-fly and not cached.

OutlineType – A polyline that may or may not be closed and has definite width. Currently arcs and circles are always set to this type.

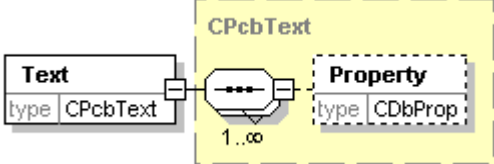
CPcbPoly objects may be net attributed. A pointer to the net can be acquired by using the GetNet() or GetNetRef() base class member functions. GetNetRef() returns a const pointer. If a CPcbPoly is not assigned to a net then its net reference will be set to NULL.

The number of vertices in a polygon is obtained by calling the GetVertexCount member. You can get any point or segment of a CPcbPoly by calling the GetCrn and GetSeg members. You can get a copy of the underlying CGon polygon object by calling the GetGon member.

A CPcbPoly object is either a board-level polygon owned by the CPcbBoard object or it is attached to a component. The coordinates

of the polygon points are in the coordinate system of the parent object. You can call the GetTransform override to transform the polygon to world coordinates.

PcbText.h – Stroked Text Class Declaration

diagram					
type	<u>CPcbText</u>				
properties	isRef 0 content complex				
children	<u>Property</u>				
attributes	Name	Type	Use	Default	Annotation
	Nm	xs:string	optional		
	String	xs:string	required		
	Lyr	derived by: xs:NMTOKEN	required		
	Wid	derived by: xs:NMTOKEN	required		
	Hgt	derived by: xs:NMTOKEN	required		
	X	derived by: xs:NMTOKEN	required		
	Y	derived by: xs:NMTOKEN	required		
	Rot	derived by: xs:NMTOKEN	optional		
	Mir	xs:boolean	optional		

DATA MEMBERS			
Field	Type	Atomic Id	Description
TextString	STRID	Text_TextString	The string to render
Rot	ANGLE	Text_Rot	Text rotation
Layer	PcbLayer	Poly_Layer	Layer poly is on
Loc	CPnt	Text_Loc	Text location

Mirrored	Bool	Text_bMirrored	Text is mirrored
Height	UNIT	Text_Height	Height of character cell
LineWidth	UNIT	Text_LineWidth	Width of stroked segs

BASE CLASS OVERRIDES	
Override	Description
Copy	Makes a deep copy of object
GetExtent	Get object bounds in world coords
GetTransform	Get transform to world cords
GetQualifiedName	Return name of object lineage
GetLayer	Returns layer object is on
IsOnLayer	Check if object resides on a layer
GetInfoString	Get info about object
GetHotSpot	Snap point to major feature of object
GetPrimitives	Get list of shapes for whole object
GetDistance	Get distance from another object
Copy	Makes a deep copy of object
GetExtent	Get object bounds in world coords

OPERATIONS	
Name	Description
GetExpandedString	If text is a macro, returns expanded string
IsRefDes	Returns true if object is a placed reference designator
GetPreciseExtent	Measures every stroke in extent calculation

CPcbText Description

A CPcbText object is a renderable text string. The string is drawn from a built-in font composed of line segments.

The text string to render will accept Unicode characters but will only render the ANSI character set as strokes. Character values greater than 126 behave like spaces in the rendered string. The rendered text string can now contain embedded newlines for multi-line text. Text is always left-justified.

If the text object is a child of a component and its string is set to `$RefDes` then the actual text rendered is the component reference designator. If you need to reproduce this substitution behavior, then use `GetExpandedString` in place of `GetTextString`.

Text objects have a text height value associated with them. This height is in database units and not points. The height value is the character cell height. It can be interpreted as the height of the tallest possible character, but in reality many characters only use about 80% of the cell height.

Text objects also have a line width value that is expressed in database units. The line width is the width of the pen used to stroke the font segments. Care should be taken to ensure the text height to line width ratio stays somewhere between 8:1 and 10:1 or unreadable text may result from the line width overwhelming the stroke lengths causing a loss of information.

A text object resides on a layer. If it is on a routing layer, the text will be etched in copper. If text it is on a plane layer, the strokes will result in an absence of copper. Silkscreen layers result in silkscreen text, etc.

The `GetLayer` member function for `CPcbText` objects will return a “cooked” value that may not reflect the layer the text was created on.

Calling `GetRawLayer` will return the original layer. Cooking may change the layer if, for instance, the text was created on the top silkscreen layer of a component and that component was placed on the bottom of the board. In that instance the text is now effectively on the bottom silkscreen layer.

If text is to appear on the bottom layer or bottom silkscreen then it must be mirrored to read correctly when the board is flipped over. `GetMirrored` and `SetMirrored` get and set this value, but if the text is attached to a component that is placed on the back of the board then it will be implicitly mirrored by the component transformation. As with all rendered objects, the `GetTransform` function should be used for acquire the correct world transform for an object.

Besides mirroring, A `CPcbText` object contains a translation and rotation transformation. The text origin is the lower-left corner of the first character in the string. The text location is relative to the parent object of the text. Again, use `GetTransform` to get the world values.

PcbPin.h – Pin Class Declaration

diagram	<pre> classDiagram class CPcbPin { Pin object for board, comps, and footprints } class CPcbPropBase { +Property :type CDbProp } CPcbPin -- > CPcbPropBase </pre> <p>Pin object for board, comps, and footprints</p>
type	extension of <u>CPcbPropBase</u>
properties	base <code>CPcbPropBase</code>
children	<u>Property</u>
used by	elements <u>CPcbBoard/Pin</u> <u>CPcbPackage/Pin</u> <u>CPcbComp/Pin</u>

attributes	Name	Type	Use	Annotation
	Nm	xs:string	optional	
	SizX	derived by: xs:NMTOKEN	required	
	SizY	derived by: xs:NMTOKEN	required	
	Shp	derived by: xs:NMTOKEN	required	
	Drl	derived by: xs:NMTOKEN	required	
	X	derived by: xs:NMTOKEN	required	
	Y	derived by: xs:NMTOKEN	required	
	Rot	derived by: xs:NMTOKEN	optional	
	NP	xs:boolean	optional	
annotation	documentation Pin object for board, comps, and footprints			

DATA MEMBERS			
Field	Type	Atomic Id	Description
Net	CPcbNet*	Pin_Net	Net reference
Loc	CPnt	Pin_Loc	Pin location
Rot	ANGLE	Pin_Rot	Pin rotation
Drill	UNIT	Pin_Drill	Pin drill size
PadSize	CPnt	Pin_PadSize	x,y size of pad
PadShape	PadShape	Pin_PadShape	Shape of pad
NonPlated	Bool	Pin_bNonPlated	Non-plated flag

BASE CLASS OVERRIDES	
Override	Description
Copy	Makes a deep copy of object
GetExtent	Get object bounds in world coords
GetRawExtent	Get object bounds in local coords
GetTransform	Get transform to world cords
Distance	Get distance from some shape
GetQualifiedName	Return name of object lineage

GetLayer	Returns layer object is on
IsOnLayer	Check if object resides on a layer
GetNetRef	Get a const pointer to object's net
GetSillouette	Get outline of object
GetInfoString	Get info about object
GetHotSpot	Snap point to major feature of object
GetPrimitives	Get list of shapes for whole object
GetDistance	Get distance from another object

OPERATIONS	
Name	Description
GetOblongSeg	Get CSeg that of oblong pad
PadDefined	Check if pad is defined on a given layer
IsPntInPad	Check if point is inside pad on given layer
IsCompPin	Check if pin belongs to a component

Pin Description

A CPcbPin object represents either a component pin or a board-level pin. The only distinction between the two is whether the CPcbPin object is a child object of a CPcbComp or a CPcbBoard. There is a small semantic difference between the two in that a component pin must be uniquely named in the scope of a component (unless the pin has no name at all.) In most cases, the terms 'Pin' and 'Pad' are interchangeable.

Many PCB CAD packages employ a pad-stack structure to define the physical size and shape of pins on different layers of the board. PCB123 opted for a simpler description because in the vast majority of cases, the difference in pad geometry between layers is due to process considerations such as over-sizing for plane layers and

soldermask. By uncoupling these considerations from the design process it allows Sunstone to make the optimal process adjustments for any given board depending on how it is constructed. It also allows for greater flexibility as Sunstone brings new fab capabilities online. It also has the added benefit of simplifying pad specifications when creating new footprints.

Pins can be broadly classified as being surface-mount pins or through-hole pins. The only distinction between the two in PCB123 is the size of the drill assigned to the pin. If the drill is zero then the pin is considered a surface-mount pin. Any other size signifies a through-hole pin. If a pin is surface-mount, then its geometry is only used on the Top layer. When a component is placed on the bottom of the board, then any surface-mount pins it may have are implied to mean Bottom of the board. You can call the `GetDrill()` member function on a `CPcbPin` object to get the drill size. The drill size returned will be in database UNIT's. There is a complimentary `SetDrill(UNIT size)` member function should be used with care. It only checks that the drill size is between 0 and Sunstone's largest drill. In practice it should be set to one of the standard sizes used by Sunstone. A list of the most current drill sizes can be found on the Sunstone website.

A through-hole pin can be plated or non-plated. Drill sizes are always represented as finished hole size but in reality the drill bit used is internally transcoded to account for plating. Specifying a non-plated hole overrides this transcoding and masks the hole from the plating process. Plating can be accessed with the `GetNonPlated()` and `SetNonPlated` members of a `CPcbPin` object.

The shape of a pad is of type `PadShape` and can be one of the following: `RoundShape`, `RectShape`, `OblongShape`, and `NoShape`. Do not use any of the other enumerations. You can use the `GetPadShape()` and `SetPadShape(PadShape shape)` members of a `CPcbPin` to access the shape information. The shape of a pad is the same on all layers.

The size of a pad can be accessed through the `GetPadSize` and `SetPadSize` members of a `CPcbPin` object. A pad size is defined as a `CPnt` object to contain both the x and y sizes. If the pad shape is `RoundShape`, then only the x member is used but for completeness the y is typically set to the same as x. For `RectShape` and `OblongShape` both the x and y values must be set with neither one being zero. Neither the x or y size can be smaller than the drill size.

Pin name rules are enforced by the parent object. If you create a new board-level pin without a name and add that pin to the `CPcbBoard` object, `CPcbBoard` is going to set the pin name to some unique name that begins with the '\$' character. Object names that start with the '\$' character are basically suppressed. They will show up as blank fields in the UI and are considered unnamed objects. If you create a new pin and add it to a component the component will check the name for uniqueness and will generate a soft crash if there is a conflict. You can access the pin name through its base class `GetName` and `SetName` member functions. Object names are of type `STRID`.

Pins may be assigned to a net. A pointer to the net can be acquired by using the `GetNet()` or `GetNetRef()` base class member functions. `GetNetRef()` returns a const pointer. If a pin is not assigned to a net

then its net reference will be set to NULL. Be advised that assigning a pin to a net is not as simple as setting the pin's net reference using the SetNet member. The net reference for a pin is for quick lookup but it is the CPcbNet object that defines the contents of a net as a whole. The following example reassigns pThisPin from pOldNet to pNewNet using the transaction manager so the operation can be undone:

```
// First remove pin from old net
transManager.DeleteChild (pOldNet, Net_CompPinList, pThisPin);
// Set the pin's new net reference
transManager.Set (pThisPin, Pin_Net, pNewNet);
// Add pin to new net
transManager.AddChild (pNewNet, Net_CompPinList, pThisPin);
```

Finally, a pin has location and rotation properties. Pin location and rotation are defined in the local coordinate system of the pin's parent object. For board-level pins, the location and rotation are basically final world coordinates but for component pins the location and rotation will be transformed by the location, rotation, and mirroring of the parent component. If you call the GetLoc() member function on a pin, it will return the location in local coordinates. To get the world coordinates for a pin you can call its GetTransform member to fill in a GeoTransform object. You can then use the GeoTransform object to apply transformations on different kinds of objects. There is a helper member function called GetTransformedPnt() if you only need the center of the pin in world coordinates.

The CPcbPin class overrides the GetSillhouette member function. This function will create a closed CGon polygon object that is the exact size and shape as the pin, and optionally will apply an oversize to the pad. The generated polygon will be in the pin's local coordinate

system I.E. centered around 0,0. You can apply a transformation to this polygon into whatever coordinate system you need.

PcbComp.h – Component Class Declaration

diagram						
type	<u>CPcbComp</u>					
properties	isRef 0 content complex					
children	<u>Property Arc Circle Poly Text Pin</u>					
attributes	Name	Type	Use	Default	Fixed	Annotation
	Nm	xs:string	optional			
	X	derived by: xs:NMTOKEN	required			
	Y	derived by: xs:NMTOKEN	required			
	Rot	derived by: xs:NMTOKEN	optional			
	Prt	xs:string	optional			

Fpr	xs:string	optional
Mir	xs:boolean	optional

DATA MEMBERS			
Field	Type	Atomic Id	Description
Device	STRID	Comp_Device	Part type
Package	STRID	Comp_Package	Footprint derive from
Loc	CPnt	Comp_Loc	Component location
Rot	ANGLE	Comp_Rot	Component rotation
Mirrored	Bool	Comp_bMirrored	True = Mirror comp
PadSize	CPnt	Pin_PadSize	x,y size of pad
PadShape	PadShape	Pin_PadShape	Shape of pad
NonPlated	Bool	Pin_bNonPlated	Non-plated flag

BASE CLASS OVERRIDES	
Override	Description
Copy	Makes a deep copy of object
GetFirstChild	Returns iterator for child list
GetExtent	Get object bounds in world coords
GetRawExtent	Get object bounds in local coords
GetTransform	Get transform to world cords
Distance	Get distance from extents
GetQualifiedName	Return name of object lineage
GetLayer	Only top, or bottom for comps
GetInfoString	Get info about object

OPERATIONS	
Name	Description
GetRefDesObject	Returns a the CPcbText object used to

	present the reference designator. Creates it in needed.
CenterRefDes	Places the ref des in the center of the comp
GetRowsCols	Returns a structure that contains all the rows and columns of pins sorted by location. Useful for calculating pin pitch and classifying package type.
MakePackage	Create a generic footprint object from a specific component.

CPcbComp Description

A CPcbComp object represents a component instance on the board. A component is always a child of the CPcbBoard object. Calling GetFirstComp on the board object will return an iterator that walks the list of components. This does not prevent the use of the more generic GetFirstChild iterator, it is simply more convenient and faster too.

A CPcbComp object acts as a container for a collection of pins, polygons, and text, and is not directly rendered itself. The renderable child objects are accessed with the GetFirstChild member of the CPcbComp object.

There is no restriction on the number of child objects a component can have, but there is a restriction on pin names within a component in that they must be uniquely named or not named at all.

Components have translation, rotation, and mirroring transform information. They establish a local coordinate system for all children.

The mirror transform also implies a change in layer. A mirrored component is on the back of the board and its child objects will reflect that when queried.

A CPcbComp object contains the name of the footprint, and possibly the part type that it was created from. A component is completely encapsulated. It contains no dependencies on external data. The GetFootprint and GetPartType members simply return strings, they do not point to any actual data.

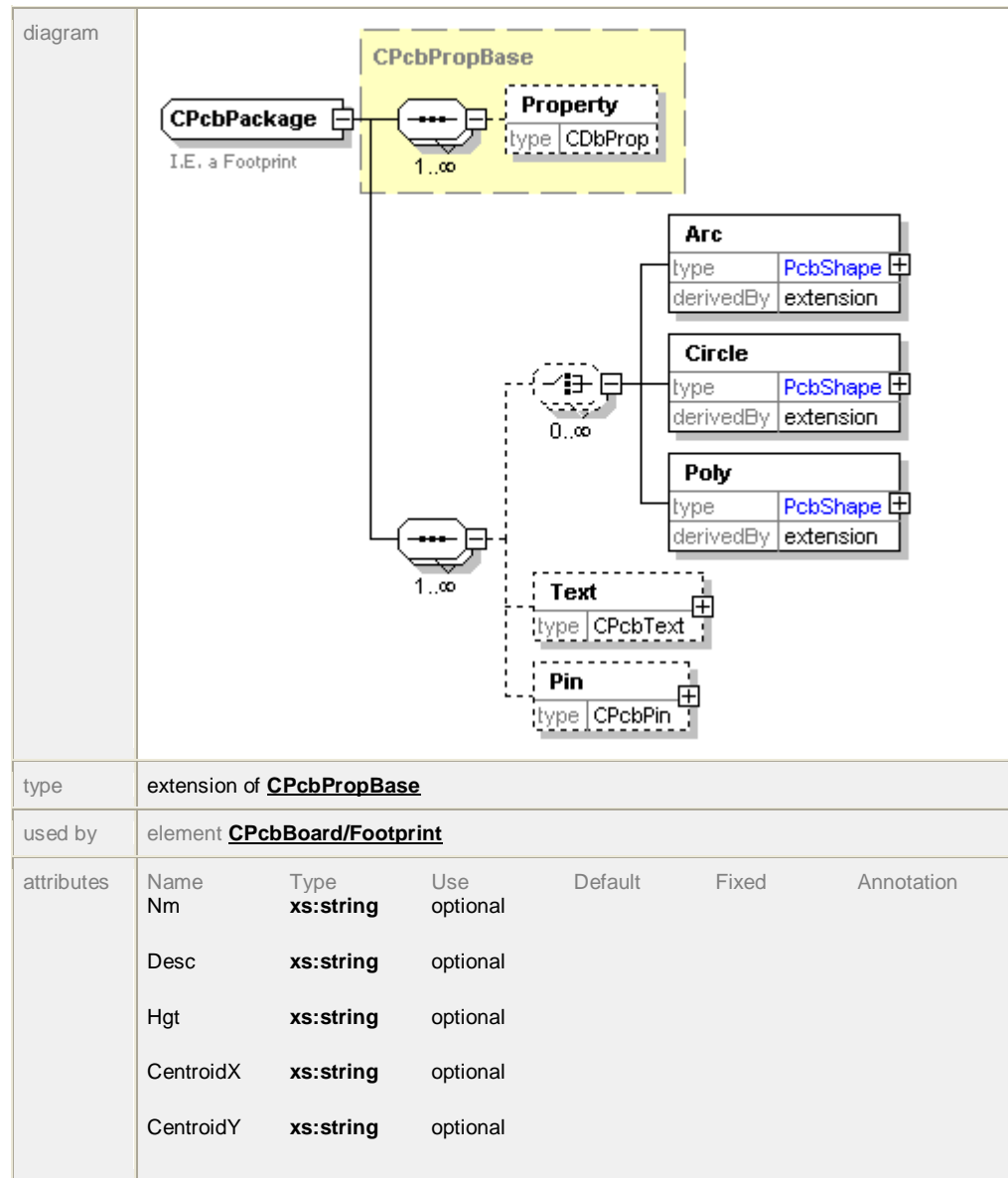
A CPcbComp object's reference designator must be unique among all component instances in the CPcbBoard object. If it is not, then a soft-error will be generated. The reference designator is just the Name field and is accessed through GetName/SetName.

A component should never be moved by simply setting one of the transformation values. Instead, perform the following call sequence:

```
CMoveCompHelper* pHelper = ::BeginMoveComp ();  
MoveComp (transactionManager, pHelper, pComp, loc, rot, mirrored);  
EndMoveComp (transactionManager, pHelper);
```

This sequence ensures that connections and tracks that are attached to the component pins will be modified appropriately to maintain continuity.

PcbPackage.h – Footprint Class Declaration



DATA MEMBERS			
Field	Type	Atomic Id	Description
PackageType	PackageType	Package_PackageType	Type of package
Centroid	CPnt	Package_Centroid	Center of package
Height	UNIT	Package_Height	Height

Desc	STRID	Package_Desc	Description of pack
------	-------	--------------	---------------------

BASE CLASS OVERRIDES	
Override	Description
Copy	Makes a deep copy of object
GetFirstChild	Returns iterator for list of children
GetExtent	Get object bounds in world coords
GetQualifiedName	Return name of object lineage
IsNameLegal	Check if a name is acceptable
GetLayer	Only top, or bottom for comps
GetInfoString	Get info about object

OPERATIONS	
Name	Description
GetNextLogicalPinName	Return next pin name that is valid
CountRowCols	Get the number of rows and cols of pins
GetPinPitch	Get spacing between row and cols of pins
FindPin	Return named pin

CPcbPackage Description

A CPcbPackage object is essentially a component template whose child objects are on library layers that get mapped to physical layers when instantiated as components.

Plugins will have very limited access to CPcbPackages because they are typically flushed from the database. Only if a package has been loaded from a library during the current session will a package be available.

PcbTrack.h – Track/Route Class Declaration

diagram						
type	CPcbTrack					
properties	isRef 0 minOcc 0 maxOcc unbounded content complex					
children	Crn					
attributes	Name	Type	Use	Default	Fixed	Annotation
	Nm	xs:string	optional			
	Crns	xs:positiveInteger	required			
diagram						
properties	isRef 0 content complex					
attributes	Name	Type	Use	Default	Fixed	Annotation
	Idx	xs:integer	required			
	X	derived by: xs:NMTOKEN	required			
	Y	derived by: xs:NMTOKEN	required			
	Wid	derived by: xs:NMTOKEN	optional			
	Lyr	derived by: xs:NMTOKEN	optional			
	Fan	xs:boolean	optional			

DATA MEMBERS			
Field	Type	Atomic Id	Description
Crns	CGon	Track_Crns	Vertices of route
StartObj	CPcbBase*	Track_pStart	Track start object
EndObj	CPcbBase*	Track_pEnd	Track end object

BASE CLASS OVERRIDES	
Override	Description
Copy	Makes a deep copy of object
GetNet	Get the net object this track belongs to
GetExtent	Get object bounds in world coords
GetRawExtent	Get object bounds in local coords
GetTransform	Get transform to world cords
GetQualifiedName	Return name of object lineage
GetLayer	Returns PcbLayerNull.
IsOnLayer	Test if object is on a layer
GetInfoString	Get info about object
GetHotSpot	Snap point to major feature of object
GetPrimitives	Get list of shapes for whole object
GetDistance	Get distance from another object
GetSillouette	Get outline of object

OPERATIONS	
Name	Description
GetRoutedLength	Return length of all segments on routing layers
GetUnroutedLength	Return length of all segs on PcbLayerNull
IsFullyRouted	Tests if copper goes from start to end
GetViaCount	Returns number of vias and fanouts in route
HitTest	Classifies where on the track the given point is
IsUnrouteConnected	Checks if unroute is connected through other copper
-- Corner Operations --	
GetLayer	Returns layer the corner (next segment) is

	on
SetLayer	Set the layer the corner is on
GetWidth	Get width of next segment
SetWidth	Set width of next segment
GetFanout	Check if corner is a fanout via
SetFanout	Set corner to fanout. Corner must be unrouted
IsVia	Checks if corner is an implied via
GetLoc	Get the corner location
SetLoc	Set the corner location
GetCorner	Get a packed corner struct
SetCorner	Set a packed corner struct
GetVia	If corner is a via, returns pad and drill size
GetSeg	Returns the CSeg starting at given corner
GetCornerCount	Get number of corners in route

CPcbTrack Description

A CPcbTrack object is a moderately complex object. There are certain assumptions and implied characteristics that go along with a track, or route, as it is sometimes called.

A CPcbTrack object is always contained by a net. It can be created outside a net but ultimately has to be added to the database as a child of a net object.

ACPcbTrack contains an array of corners in the form of a CGon and two end objects which are pin references (possibly the same pin). The route corners do not necessarily have to coincide with the pin objects as the pin objects primary use is as an aid in the graph connectivity.

ACPcbTrack encodes layer, width, and Fanout Via info into the extra DWORD data member of a CGon vertex. There are member functions that access this data and there are also macros to extract/build the bit fields directly.

When traversing a CPcbTrack from corner 0 to corner n-1, the layer and width fields specify the layer and width for the segment between crn i, and crn i+1.

A CPcbTrack is not automatically assumed to be electrically connected from start to finish. If a corner is assigned to layer PcbLayerNull then that corner begins an unrouted segment (also called a rat). There are situations where a route may contain a rat or even several rat segments but still be electrically complete. This is the case if both ends of the rat are either directly or indirectly tied to a plane layer or touch other electrical objects that make a circuit between the two ends of the rat.

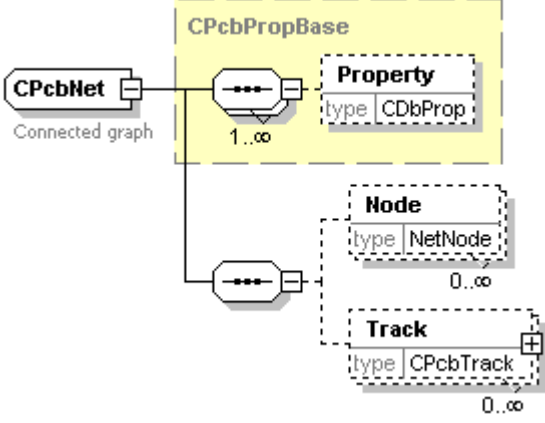
There should never be two rat segments in a row. This condition is usually optimized out but could accidentally be created by a Plugin. The program most likely will assert if detected.

A Change in layer from one corner to the next generally implies a via location. The rules for this are not totally straightforward and that is why a call to IsVia or GetVia on a corner should be used to deal with vias. These functions encapsulate all the via logic.

If a via is implied, its geometry is obtained from the containing net object. There is not enough bits in the extra DWORD of the CCrn to

encode custom via geometry for each via. In fact, the segment width is limited to 23 bits of data (actually clamped to ½ inch).

PcbNet.h – Net Class Declaration

diagram						
type	extension of <u>CPcbPropBase</u>					
properties	base CPcbPropBase					
children	<u>Property</u> <u>Node</u> <u>Track</u>					
used by	element <u>CPcbBoard/Net</u>					
attributes	Name	Type	Use	Default	Fixed	Annotation
	Nm	xs:string	optional			
	Pln	xs:integer	optional			
	Wid	derived by: xs:NMTOKEN	optional			
	Spc	derived by: xs:NMTOKEN	optional			
	Via	derived by: xs:NMTOKEN	optional			
	Drl	derived by: xs:NMTOKEN	optional			
	Pri	xs:integer	optional			
	Reconn	derived by: xs:NMTOKEN	optional			
	Clr	xs:integer	optional			

annotati on	documentation Connected graph
----------------	-------------------------------

DATA MEMBERS			
Field	Type	Atomic Id	Description
Color	INT32	Net_Color	Color of net
Width	UNIT	Net_Width	Default width of net
PlaneLayers	INT32	Net_PlaneLayers	Planes net is on
Priority	INT32	Net_Priority	Autorouter priority
ReconnType	INT32	Net_Reconn	Type of reconnect
Spacing	UNIT	Net_Spacing	Required spacing
ViaSize	UNIT	Net_ViaSize	Size of vias for net
ViaDrill	UNIT	Net_ViaDrill	Drill size for vias
DoNotAutoroute	Bool	Net_DoNotAutoroute	Manual route only

BASE CLASS OVERRIDES	
Override	Description
Copy	Makes a deep copy of object
GetFirstChild	Returns iterator for child list
GetChildCount	Retruns number of child objects
AddChild	Add a child to the net
DeleteChild	Remove a child from the list
GetQualifiedName	Return name of object lineage
GetLayer	Returns PcbLayerAll

OPERATIONS	
Name	Description
FindCompPin	Check if comp.pin name is in pin list
IsPinInNet	Given a pin, check if it is in net
EmptyChildList	Delete all children in net

GetRoutedLength	Returns routed length of all tracks in net
GetUnroutedLength	Returns unrouted length of all tracks in net
GetViaCount	Returns count of all vias in net
MergeNet	Merge given net into this one
FindTracks	Returns list of tracks tied to pin
GetUnrouteList	Returns list of segments that are unrouted

CPcbNet Description

A CPcbNet object is a logical container for the nodes and edges of a graph that represents one net. The net object is a bit of an anomaly with regards to the other database classes in that the only true containment it does is with CPcbTrack objects (edges), and not the CPcbPin (nodes) objects. What it actually stores for nodes is list of *references* to CPcbPin objects. The reason for this is simple: a pin cannot be owned by two objects and they are physically part of a component or board so those are the objects that own pins. Because of this, accessing the pins of a net involve iterating over a different list than the normal child list as is done with GetFirstChild. Instead, pins live in a separate pin list which is accessed using GetFirstCompPin.

CPcbTrack objects are owned by the net and are accessed with GetFirstChild.

Besides object state information, the CPcbNet object also stores various default parameters that get applied to routes as routing progresses. These are:

Width: This is the default width for routes in this net. This can be overridden down to segment granularity on the actual CPcbTrack objects.

ViaSize: The pad size of any vias in the net. Vias are not explicit objects, instead they are implied by a layer change when traversing the corners of a CPcbTrack object.

ViaDrill: The drill size to use for vias in the net.

A distinction must be made when talking about connectivity on the board and connectivity in graph theory. There will always be enough child track objects to connect all the pins in the pin list, so the graph is always connected. There may even be more than enough track objects, resulting in cycles in the graph. A connected graph, even one with cycles does not, however, mean that the net is connected in a physical sense. If any corner of a track object other than the last corner is on layer PcbLayerNull, then the segment starting with that corner is unrouted and does not complete a path by itself. This is a trivial condition to check for but unfortunately is not sufficient for detecting connectivity or lack thereof. It is possible that both ends of that unrouted segment are indirectly connected in the following ways:

Implicitly through a plane layer: if the net is a plane net and both ends of the unrouted segment eventually reach a plated through hole, then the plane layer completes the connection.

Indirectly connected through copper regions or copper polylines that have been assigned to the same net.

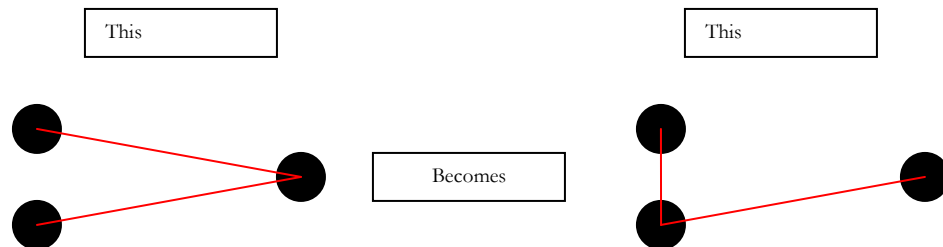
Indirectly by another routed track object, or series of track objects that both ends eventually touch.

Any combination or series of the above three conditions.

To account for this condition, net objects provide a function called `IsUnrouteConnected` that takes the corner index of an unrouted track object and returns a yes/no status for the unrout.

To determine if a net is a plane net, or to set a net to a plane layer, you first have to call `GetPlaneLayers` on the net to get an encoded value that is then passed to the member functions `IsNetAssignedToPlane` and `AssignNetToPlane`. A net can be set to multiple planes this way. There is also a matching member function named `RemoveNetFromPlane`.

While placing the components on the board, while all nets are unrouted, PCB123 will perform length optimizations on the net topology as the components are moved. Length optimization is depicted below:



This behavior can be detected or controlled using the `GetReconnType` and `SetReconnType` member functions.

Lastly, the order in which nets are autorouted are computed by the autorouter using a variety of tests and parameters. One of the

parameters is the Priority of a net, which can be read and updated with GetPriority and SetPriority. Elevating the priority of a net will make the autorouter attempt to route it early when it has a good chance of being completed in a direct manner.

A Relaxing Tutorial

Relax Routing: An iterative routine for minimizing angular deflection of all routes

Chapter

3

Tutorial: Creating a Plugin

In this tutorial we will create a Plugin called Relax Routing that will minimize the angular bending of all the routing on the board.

Create a new MFC DLL project in App Wizard. For a project name choose RelaxRouting and for Location choose the Plugin SDK root.

For the application settings choose Regular DLL Using Shared MFC DLL.

The first thing we'll do is edit the stdafx.h file in the new project to include the main header file and to define the Debug and Release libraries that we will need to link with. At the end of the file add:

```
#include "RelaxRouting.h"

#ifdef _DEBUG
# pragma comment(lib, "../Lib/CoreLibD.lib")
# pragma comment(lib, "../Lib/2dLibD.lib")
# pragma comment(lib, "../Lib/DrawLibD.lib")
# pragma comment(lib, "../Lib/GridCtrlLibD.lib")
# pragma comment(lib, "../Lib/BaseDbLibD.lib")
# pragma comment(lib, "../Lib/PcbDbLibD.lib")
#else
# pragma comment(lib, "../Lib/CoreLib.lib")
# pragma comment(lib, "../Lib/2dLib.lib")
# pragma comment(lib, "../Lib/DrawLib.lib")
# pragma comment(lib, "../Lib/GridCtrlLib.lib")
# pragma comment(lib, "../Lib/BaseDbLib.lib")
# pragma comment(lib, "../Lib/PcbDbLib.lib")
#endif
```

The next step will be to right click on the RelaxRouting project folder and select Properties. The changes we will make are for all configurations so select All Configurations. In the C++/General settings, make sure Detect 64-Bit Portability is off (set to No). In the

Linker/General settings, set the output file to “../Plugins/RelaxRouting.dll”. Select the Ok button.

We will now declare our two exported functions. The main Event Proc and the one menu item handler this Plugin will have. Open the RelaxRouting.def file and at the end of the file add the following:

```
EventProc    @1
Relax        @2
```

We will now declare the prototypes for the two functions. Open the RelaxRouting.h file and add to the end of it the following:

```
extern "C" {
    PluginResponse EventProc (PluginIO& iob);
    PluginResponse Relax (PluginIO& iob);
}
```

Before we close RelaxRouting.h, we want it to add #include “../IPlugin.h” right under the #include “Resource.h”. This will include all the Plugin definitions that we will use.

We will now create the main event procedure for the Plugin. For this Plugin the event procedure is not really interested in anything but setting the state of the menu item to gray if there is no active board document open. Open RelaxRouting.cpp and to the bottom of the file add the following:

```
PluginResponse EventProc (PluginIO& iob)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    iob.m_ReturnValue = 0;

    if (iob.m_EventType == UpdateMenuUIEvent
        && iob.m_UpdateMenuEvent == Relax) {
        if (iob.m_pRootObj == NULL) {
            iob.m_ReturnValue = MENU_DISABLE;
        }
    }

    return PluginSuccess;
}
```

Below this, we will add the menu item handler. When the user selects Relax Routing from the menu, the following code will be executed:

```
PluginResponse Relax (PluginIO& iob)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    if (! iob.m_pRootObj) {
        // No object
        return PluginSuccess;
    }
    if (iob.m_pRootObj->GetClass() != BOARD_CLASS) {
        AfxMessageBox ("Error! Expecting a BOARD object.");
        return PluginFailure;
    }
    CString str = "Relax was called for board ";
    str += StrPtr(iob.m_pRootObj->GetName());

    AfxMessageBox ((CSTR) str);

    return PluginSuccess;
}
```

At this point let's compile the code and test it by running PCB123. Before you load a PCB board you should see a new Plugin entry named Relax Routing that is grayed. Now go ahead and load a board and look at the menu item again. It should be enabled. When you select the Relax Routing menu item you should see a simple message that echo's the name of the currently active PCB board document. Not a very exciting Plugin, yet. Let's change that now. We are going create a new class that is going to perform all the work of route relaxation.

Create a new file called RelaxClass.h and add it to the project's Head Files folder. The contents of RelaxClass.h are as follows:

```
////////////////////////////////////
// RelaxClass.h - Declaration of CRelaxClass.
//              CRelaxClass encapsulates the route relaxation
//              logic. It expects a PcbBoard and a transaction
//              manager to use for making changes to the board.
//
#pragma once
#include "RelaxRouting.h"
#include "../PcbDbLib/PcbDbLib.h"

class CRelaxClass
```



```

{
private:
    CPcbBoard*      m_pBoard;
    CTransactionManager* m_pTm;
    bool            m_bMoveVias;

    bool RelaxOneCorner (CPcbTrack* pTrack, INT32 crnIdx);
    bool RelaxOneTrack (CPcbTrack* pTrack);
    bool RelaxOneNet (CPcbNet* pNet);

public:
    CRelaxClass (CPcbBoard* pBoard, CTransactionManager* pTm);
    virtual ~CRelaxClass ();

    bool RelaxAll ();
};

```

This is a simple class that contains a public constructor and a public member named RelaxAll. The constructor's parameters are directly obtained from the iob parameter of the menu event procedure. Given this header file, let's update the RelaxRouting.cpp file to incorporate this new class.

Directly under the #include "RelaxRouting.h" statement in RelaxRouting.cpp, add #include "RelaxClass.h"

In the Relax menu event procedure we are going to do a very important thing and that is remove the AFX_MANAGE_STATE statement at the very beginning of the procedure. At this point you are probably wondering what gives! "Gee, first you tell me not to forget adding this statement to the beginning of my event procs and now you are telling me to remove it." That's right. The reason for this is because the CRelaxClass is not going to be accessing any resources from this Plugin so there is no need for a module context switch. In fact, the CRelaxClass is going to modify route information through the transaction manager, and one of the jobs of the transaction manager is to synchronously notify anyone who cares about a change to the database and this includes the graphic windows in PCB123! It is for this reason that the module state for the

host application wants to be the active one before making any calls that will modify the database.

Let's finish the Relax menu event procedure by removing the three lines of test code...

```
...
CoreString str = "Relax was called for board ";
str += StrPtr(iob.m_pRootObj->GetName());

AfxMessageBox ((CSTR) str);
...
```

And replacing them with:

```
CRelaxClass rc ((CPcbBoard*) iob.m_pRootObj, iob.m_pTm);
rc.RelaxAll ();
```

Now the entire Relax menu event procedure looks like:

```
PluginResponse Relax (PluginIO& iob)
{
    ASSERT (iob.m_pRootObj != NULL);
    ASSERT (iob.m_pRootObj->GetClass() == BOARD_CLASS);

    CRelaxClass rc ((CPcbBoard*) iob.m_pRootObj, iob.m_pTm);
    rc.RelaxAll ();

    return PluginSuccess;
}
```

We are done with RelaxRouting.cpp leaving only RelaxClass.cpp to be filled in. RelaxClass.cpp is pure application code and is shown below:

```

////////////////////////////////////
// RelaxClass.cpp : Relaxation Class Definition
//

#include "stdafx.h"
#include "RelaxClass.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

////////////////////////////////////
// Constructor - Validate parameters and setup any states.
//
CRelaxClass::CRelaxClass (CPcbBoard* pBoard, CTransactionManager* pTm)
{
    ASSERT (pBoard);
    ASSERT (pTm);

    m_pBoard = pBoard;
    m_pTm = pTm;
    m_bMoveVias = true;      // You can change this (or ask user)
}

////////////////////////////////////
// Destructor - Currently does nothing.
//
CRelaxClass::~CRelaxClass ()
{
}

////////////////////////////////////
//
// RelaxOneCorner - Move corner toward prior corner.
//
bool CRelaxClass::RelaxOneCorner (CPcbTrack* pTrack, INT32 crnIdx)
{
    bool result = false;

    // Get location of corner and previous corner
    CPnt srcPnt = pTrack->GetLoc (crnIdx);
    CPnt destPnt = pTrack->GetLoc (crnIdx-1);
    // Get the distance between the two points
    // (uses 2D distance operator ^)

    UNIT len = srcPnt ^ destPnt;

    if (len <= MILS_TO_UNITS(5)) {
        // Not worth it
        return false;
    }

    // Get the angle from corner to previous corner
    ANGLE a = destPnt.AngleFrom (srcPnt);

    // Start walking toward dest by 5 mils.
    UNIT dist = MILS_TO_UNITS(5);

    for (;;) {
        if (dist > len) {
            // Do not overshoot dest
            dist = len;
        }
        // Create a point at this distance attempt
        CPnt p (dist, 0);
        // Rotate it toward dest
        p <<= a;
        // Add start point
        p += srcPnt;

```

```

// Remember last good location
CPnt lastGood = pTrack->GetLoc (crnIdx);

// Move the track corner
pTrack->SetLoc (crnIdx, p);

// Check if it caused a spacing violation
if (! m_pBoard->CheckConn (pTrack, crnIdx-1, crnIdx+1)) {
    // It did. We've gone as far as we can toward dest
    // Put it back the way we found it
    pTrack->SetLoc (crnIdx, srcPnt);
    // Get the track polygon
    CGon crns = pTrack->GetCorners ();
    // Set the corner to the last valid location
    crns.SetAt (crnIdx, lastGood);
    // Formally update the track and return
    m_pTm->Set (pTrack, Track_Crns, crns);
    return result;
} else {
    // It passed spacing. Try and walk some more
    dist += MILS_TO_UNITS(5);
    if (dist > len) {
        // Reached dest. All done
        // Put corner back the way we found it
        pTrack->SetLoc (crnIdx, srcPnt);
        // Get the track polygon
        CGon crns = pTrack->GetCorners ();
        // Set the corner to the last move
        crns.SetAt (crnIdx, p);
        // Formally update the track and return
        m_pTm->Set (pTrack, Track_Crns, crns);
        return result;
    }
    // If we got here we moved more than 5 mils successfully
    result = true;
}
}

return false;
}

////////////////////////////////////
// RelaxOneTrack - Relax track one corner at a time.
//
bool CRelaxClass::RelaxOneTrack (CPcbTrack* pTrack)
{
    bool result = false;

    // We do not want to move the first or last corner in a track
    for (int idx = 1; idx < pTrack->GetCornerCount()-1; ++idx) {
        PcbLayer prevLayer = pTrack->GetLayer (idx-1);
        PcbLayer nextLayer = pTrack->GetLayer (idx);
        if (prevLayer == PcbLayerNull || nextLayer == PcbLayerNull) {
            // Don't move rat lines
            continue;
        }
        if ((prevLayer != nextLayer) && m_bMoveVias == false) {
            // This corner is a via and we can't move them
            continue;
        }

        if (RelaxOneCorner (pTrack, idx)) {
            result = true;
        }
    }

    return result;
}

```

```

////////////////////////////////////
// RelaxOneNet - Relax each track in the supplied net.
//
bool CRelaxClass::RelaxOneNet (CPcbNet* pNet)
{
    bool result = false;

    CDbBaseIter iter = pNet->GetFirstChild ();
    for (; iter(); ++iter) {
        // Get each child in net
        CPcbBase* pObj = (CPcbBase*) iter.Get ();
        if (pObj->GetClass() == TRACK_CLASS) {
            // Only concerned about tracks
            CPcbTrack* pTrack = (CPcbTrack*) pObj;

            if (RelaxOneTrack (pTrack)) {
                result = true;
            } else {
                // Try the other way
                pTrack->Reverse (0);
                if (RelaxOneTrack (pTrack)) {
                    result = true;
                }
                pTrack->Reverse (0);
            }
        }
    }

    return result;
}

////////////////////////////////////
// RelaxAll - Iterate over the each net in the board and perform relax.
//
bool CRelaxClass::RelaxAll ()
{
    // Start a transaction frame so user can undo entire operation
    m_pTm->StartFrame ();

    bool result = true;

    // Seed to known state
    CheckEscapeKeyPressed ();

    // Indicate we are busy and how to stop this
    SetStatusMessage ("Press Esc to cancel...");
    ShowBusyCursor (true);

    // Loop up to 5 times (as long as a change was made)
    int count = 0;
    while (result == true && count++ <= 5) {
        result = false;
        CPcbNetIter iter = m_pBoard->GetFirstPcbNet ();
        for (; iter(); ++iter) {

            if (CheckEscapeKeyPressed ()) {
                // User pressed the escape key. We can stop here
                m_pTm->EndFrame ();
                return false;
            }

            // Get the net and relax it
            CPcbNet* pNet = iter.Get ();
            if (RelaxOneNet (pNet)) {
                result = true;
            }
        }
    }

    ShowBusyCursor (false);
}

```

```
        // Close the transaction frame
        m_pTm->EndFrame ();

        return result;
    }
```