Create a dogleg channel router using ActiveX technology. You need to follow the following steps.


2. Create a project in Visual C++ 6.0 (5.0 will probably work as well). The project type should be MFC ActiveX ControlWizard. You must give your project a name to proceed. Use your first initial and your last name as the project name. Thus, I would name my project PMaurer. (I will use this as the project name in these notes, wherever a project name is required.) **DO NOT CALL YOUR PROJECT PMaurer!** A project directory will be created for you as part of this process.

3. On the project options, you must choose “Invisible at run time” this is the second checkbox from the top on the second page. Everything else should be left as default.

4. Add two properties to your ActiveX control. Go to the View Menu and choose ClassWizard. This will give you a message box with 5 tabs. Choose the center tab, and click on it. On this tab you will find the Add Property button. Click this to add a property. Add two properties, one named ChannelHandle, and one named Routing Handle. Both must be of type long. Make sure you spell the names correctly!

5. Now add the two files VCG2001.H and VCG2001.CPP to your project. To do this, you go to the file view (note the tabs at the lower left of the window). Copy the two files into your project directory. Right click on the line that says “PMaurer files” and right-click on it. This will allow you to select the two files and add them to your project.

6. In the class view, (tab at lower left) you will have a class named CPMaurerCtrl. This class will have a member function named OnChannelHandleChanged. This is where you put all your code. You can delete whatever lines of code are already in this function, they don’t matter. But don’t delete anything else! To find the OnChannelHandleChanged function, click on the + in front of the class name, then double-click on the function name. If the list of classes isn’t displayed, click on the + in front of “PMaurer Classes” to display the list of classes. There will be a lot of them, most from VCG2001.H.

7. You must define new five variables in the class CPMaurerCtrl. These variables must be of the following types: Channel *, NetList *, SegmentList *, Layout *, and VCG *. They must all be pointers! In the function CPMaurerCtrl (Same as class name) assign NULL to all five of these. Let’s assume you named your Channel * variable Ch, and your Layout * variable Ly. The first line of your OnChannelHandleChanged function must be:
Ch = (Channel *)m_channelHandle;
and the last line must be:
m_routingHandle = (long)Ly;

8. Before proceeding with any other coding, check to see if the NetList *,
SegmentList *, Layout * and VCG * variables contain NULL. If they do not, then
delete them using a statement similar to this: delete Ly;

9. Allocate new variables for the four pointers of type NetList *, SegmentList *,
Layout *, and VCG *, using a statement similar to the following:
Ly = new Layout; DO NOT USE malloc OR free ANYWHERE IN YOUR
CODE!

10. Your channel variable contains two arrays of type long named Top and Bottom. It
also contains a variable named Count, which gives the size of the arrays. They
will always be the same size. Each position in the array represents one pin
position in the channel, and the numbers in the arrays are the net numbers
attached to the top and bottom pins in that position. You must tell the NetList *
variable about the location of the pins. Your NetList * variable has three functions
used to do this, they are:
TopReference(long PinNumber,long NetNumber)
BottomReference(long PinNumber,long NetNumber)
BothReference(long PinNumber,long NetNumber)
Go through the pins one by one, and call the appropriate function for each.
BothReference is used in positions where the top and bottom pins are the same,
otherwise TopReference and BottomReference are used.

11. Now you must break the nets into segments, and give them to the SegmentList *
variable. First tell the SegmentList * variable how wide the channel is using the
SetSize function of that variable. This function has one argument that must have
the same value as the Channel * Count component. Now, create a loop that goes
through the nets one at a time. This is done using the FirstNet, and NextNet
functions of the NetList * variable. Both of these functions return either 1 or 0. 0,
indicates that the end of the list of nets has been reached. Both of these functions
also have one integer argument, which returns the number of the net. (You won’t
need this.) Inside the loop that goes through the nets, create another loop that goes
through the segments of the net. You do this using the FirstSegment and
NextSegment functions of the NetList * variable. Like the net functions, these
functions return a 0 when the end of the list of segments for a net has been
reached, and a 1 otherwise. These two functions have one argument of type
Extent, which is used to return information about the segment. This variable must
be passed intact to the SegmentList * variable, using the AddSegment function of
that variable. The AddSegment function is part of the SegmentList * variable, and
has one argument of type Extent. It does not return a value.
12. Once all segments have been added to the segment list you must begin building
the Vertical constraint graph. Begin by telling the VCG * variable how many
segments there are. This is done by using the SetVertexCount function of the
VCG * variable. The number of segments can be found in the CurrSegID variable
of the SegmentList * variable. The SegmentList * variable has four arrays, Top1,
Top2, Bottom1 and Bottom2. The SegmentList * component “Size” gives the size
of all four arrays. For each pin position i, you must add four edges to the VCG,
one from Top1[i] to Bottom1[i], one from Top2[i] to Bottom1[i], one from
Top1[i] to Bottom2[i], and one from Top2[i] to Bottom2[i]. You do this using the
AddEdge function of the VCG * variable. This function has two arguments, the
from vertex, followed by the to vertex. Go through the pins one at a time and add
these edges.

13. Although a left-edge sort is not strictly necessary, it can’t be done unless you tell
the VCG where each net begins and ends. To do this you have to go through the
segments one by one, and inform the VCG * variable where each segment begins
and ends. To go through each segment, use the FirstSeg and NextSeg functions of
the SegmentList * variable. These functions return 1 or 0, just as the other First
and Next functions mentioned, and in addition, each has a single argument of type
Segment. The Segment argument is used to return information about the segment,
and has the following components:
Left – Pin number of left end.
LeftTop – 0 - Bottom only, 1 - TopOnly, 2 - Both Top and Bottom
Right – Pin number of right end
RightTop – Same as for LeftTop
NetID – Net Number to which segment belongs
SegID – Unique segment number for this segment.
For each segment, call the SetLeft and SetRight functions of the VCG * variable.
These functions both have two arguments, the first is the SegID, and the second is
either Right or Left, depending on which function you are calling.

14. Now for the actual dogleg algorithm. For each row, call the GetFreeList function
of the VCG * variable. This function returns a pointer to an array of longs. Delete
this array (delete [] MyArray) when you don’t need it anymore. This array is
sorted in leftmost order. Go through the array, in order, and pack as many
segments as you can into the current row without overlapping. (Two segments
from the same net cannot overlap.) When you can’t pack in any more, go to the
next row, and call GetFreeList again. When GetFreeList returns NULL, you are
all done.

15. Finally, add each segment and each contact cut to the Layout * variable. This can
be done as you process the segments in step 13. The Layout * variable has three
functions:
AddHorizSeg(long Left,long Right,long RowNumber)
AddVertSeg(long Top,long Bottom,long ColumnNumber)
AddCC(long RowNumber,long ColumnNumber)
When you go through the free list, all you will get is segment numbers. To obtain information about a segment call the GetSegment function of the SegmentList * variable. This function has two arguments, the segment number, followed by a Segment variable. The information will be placed in the Segment variable. Remember to call the DeleteVertex function of the VCG * variable after placing a segment. (One argument, the segment ID.) To route a vertical segment to the top use a Top of 0. To route it to the bottom use a Bottom of –1.

16. As a last step call the SetRC function of the Layout * variable. This function has two arguments, the number of rows followed by the number of columns. The number of columns is the same as the Count variable of the Channel * variable. The number of rows is calculated in step 13.

17. Done correctly this will route all nets, EXCEPT those that have a pin on the top and bottom of a particular column, but no pin anywhere else. These nets have a vertical segment, but no horizontal segment. These are known as VO (Vertical Only) nets. If you want to route them (and you should!), you must go back to the NetList * variable, and use the FirstVO and NextVO functions of this variable to enumerate all VO nets. These functions return one or zero, just as all other First/Next pairs. They also have a single argument of type long. The column containing the VO net will be returned in this variable. Enumerate all VO nets, and add a vertical segment running from top to bottom in each column containing a VO net.

18. Test your program by compiling it and using it in a Visual Basic Program. There is a VB program on my website that has everything you need in it. Add your ActiveX to the program. (Project Menu Components … command, look for the name of your ActiveX in the list and check the box in front of it.) Next add an instance of your ActiveX to the form. (On the right, click the + in front of “Forms” and double click on Form1( …) to display.) Select the icon for your ActiveX, and draw a rectangle some place on the form. Select the instance of your component by clicking on it, and change the name of it to “Router.” (See property list on the right, just type over the old name.) Your VB program is now ready to go. When you go to the VB program download page, two ActiveX controls will be installed on your system. You must allow this to happen, or the program won’t work.