Agilent Technologies

Advanced Design System 2011.01

Feburary 2011
Graphical Cell Compiler

Acknowledgments

Mentor Graphics is a trademark of Mentor Graphics Corporation in the U.S. and other countries. Mentor products and processes are registered trademarks of Mentor Graphics Corporation. * Calibre is a trademark of Mentor Graphics Corporation in the US and other countries. "Microsoft®, Windows®, MS Windows®, Windows NT®, Windows 2000® and Windows Internet Explorer® are U.S. registered trademarks of Microsoft Corporation. Pentium® is a U.S. registered trademark of Intel Corporation. PostScript® and Acrobat® are trademarks of Adobe Systems Incorporated. UNIX® is a registered trademark of the Open Group. Oracle and Java and registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. SystemC® is a registered trademark of Open SystemC Initiative, Inc. in the United States and other countries and is used with permission. MATLAB® is a U.S. registered trademark of The Math Works, Inc.. HiSIM2 source code, and all copyrights, trade secrets or other intellectual property rights in and to the source code in its entirety, is owned by Hiroshima University and STARC. FLEXlm is a trademark of Globetrotter Software, Incorporated. Layout Boolean Engine by Klaas Holwerda, v1.7 http://www.xs4all.nl/~kholwerd/bool.html . FreeType Project, Copyright (c) 1996-1999 by David Turner, Robert Wilhelm, and Werner Lemberg. QuestAgent search engine (c) 2000-2002, JObjects. Motif is a trademark of the Open Software Foundation. Netscape is a trademark of Netscape Communications Corporation. Netscape Portable Runtime (NSPR), Copyright (c) 1998-2003 The Mozilla Organization. A copy of the Mozilla Public License is at http://www.mozilla.org/MPL/ . FFTW, The Fastest Fourier Transform in the West, Copyright (c) 1997-1999 Massachusetts Institute of Technology. All rights reserved.

The following third-party libraries are used by the NlogN Momentum solver:

"This program includes Metis 4.0, Copyright © 1998, Regents of the University of Minnesota", http://www.cs.umn.edu/~metis , METIS was written by George Karypis (karypis@cs.umn.edu).

Intel@ Math Kernel Library, http://www.intel.com/software/products/mkl

SuperLU_MT version 2.0 - Copyright © 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy). All rights reserved. SuperLU Disclaimer: THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS 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 OWNER OR CONTRIBUTORS 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.

7-zip - 7-Zip Copyright: Copyright (C) 1999-2009 Igor Pavlov. Licenses for files are: 7z.dll: GNU LGPL + unRAR restriction, All other files: GNU LGPL. 7-zip License: This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful,but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. unRAR copyright: The decompression engine for RAR archives was developed using source code of unRAR program.All copyrights to original unRAR code are owned by Alexander Roshal. unRAR License: The unRAR sources cannot be used to re-create the RAR compression algorithm, which is proprietary. Distribution of modified unRAR sources in separate form or as a part of other software is permitted, provided that it is clearly stated in the documentation and source comments that the code may not be used to develop a RAR (WinRAR) compatible archiver. 7-zip Availability: http://www.7-zip.org/

AMD Version 2.2 - AMD Notice: The AMD code was modified. Used by permission. AMD copyright: AMD Version 2.2, Copyright © 2007 by Timothy A. Davis, Patrick R. Amestoy, and Iain S. Duff. All Rights Reserved. AMD License: Your use or distribution of AMD or any modified version of AMD implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies.User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. AMD Availability: http://www.cise.ufl.edu/research/sparse/amd

UMFPACK 5.0.2 - UMFPACK Notice: The UMFPACK code was modified. Used by permission. UMFPACK Copyright: UMFPACK Copyright © 1995-2006 by Timothy A. Davis. All Rights Reserved. UMFPACK License: Your use or distribution of UMFPACK or any modified version of UMFPACK implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License

along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies. User documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. UMFPACK Availability: http://www.cise.ufl.edu/research/sparse/umfpack UMFPACK (including versions 2.2.1 and earlier, in FORTRAN) is available at http://www.cise.ufl.edu/research/sparse . MA38 is available in the Harwell Subroutine Library. This version of UMFPACK includes a modified form of COLAMD Version 2.0, originally released on Jan. 31, 2000, also available at http://www.cise.ufl.edu/research/sparse . COLAMD V2.0 is also incorporated as a built-in function in MATLAB version 6.1, by The MathWorks, Inc. http://www.mathworks.com . COLAMD V1.0 appears as a column-preordering in SuperLU (SuperLU is available at http://www.netlib.org). UMFPACK v4.0 is a built-in routine in MATLAB 6.5. UMFPACK v4.3 is a built-in routine in MATLAB 7.1.

Qt Version 4.6.3 - Qt Notice: The Qt code was modified. Used by permission. Qt copyright: Qt Version 4.6.3, Copyright (c) 2010 by Nokia Corporation. All Rights Reserved. Qt License: Your use or distribution of Qt or any modified version of Qt implies that you agree to this License. This library is free software; you can redistribute it and/or modify it under the
terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Permission is hereby granted to use or copy this program under the terms of the GNU LGPL, provided that the Copyright, this License, and the Availability of the original version is retained on all copies.User
documentation of any code that uses this code or any modified version of this code must cite the Copyright, this License, the Availability note, and "Used by permission." Permission to modify the code and to distribute modified code is granted, provided the Copyright, this License, and the Availability note are retained, and a notice that the code was modified is included. Qt Availability: http://www.qtsoftware.com/downloads Patches Applied to Qt can be found in the installation at: $HPEESOF_DIR/prod/licenses/thirdparty/qt/patches. You may also contact Brian Buchanan at Agilent Inc. at brian_buchanan@agilent.com for more information.

The HiSIM_HV source code, and all copyrights, trade secrets or other intellectual property rights in and to the source code, is owned by Hiroshima University and/or STARC.

Errata The ADS product may contain references to "HP" or "HPEESOF" such as in file names and directory names. The business entity formerly known as "HP EEsof" is now part of Agilent Technologies and is known as "Agilent EEsof". To avoid broken functionality and to maintain backward compatibility for our customers, we did not change all the names and labels that contain "HP" or "HPEESOF" references.

Warranty The material contained in this document is provided "as is", and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this documentation and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license. Portions of this product include the SystemC software licensed under Open Source terms, which are available for download at http://systemc.org/ . This software is redistributed by Agilent. The Contributors of the SystemC software provide this software "as is" and offer no warranty of any kind, express or implied, including without limitation warranties or conditions or title and non-infringement, and implied warranties or conditions merchantability and fitness for a particular purpose. Contributors shall not be liable for any damages of any kind including without limitation direct, indirect, special, incidental and consequential damages, such as lost profits. Any provisions that differ from this disclaimer are offered by Agilent only.

Restricted Rights Legend U.S. Government Restricted Rights. Software and technical data rights granted to the federal government include only those rights customarily provided to end user customers. Agilent provides this customary commercial license in Software and technical data pursuant to FAR 12.211 (Technical Data) and 12.212 (Computer Software) and, for the Department of Defense, DFARS 252.227-7015 (Technical Data - Commercial Items) and DFARS 227.7202-3 (Rights in Commercial Computer Software or Computer Software Documentation).

Building Components

The primary purpose of the Graphical Cell Compiler is the creation of new Layout artwork models. But artwork models in themselves do not make a complete component definition. This section walks through a simple example of building a complete component using custom artwork defined using the Graphical Cell Compiler.

The premise of this example is the need to create an alternate MLIN component model which we call MLIN2. MLIN2 simulates as a standard MLIN, but has some additional artwork on non-metal layers for production needs.

Starting with the schematic in a new design named MLIN2, simply place a standard MLIN with a port on each end. The MLIN needs to be parameterized so the physical dimensions are changed from constants to variables. To make the example easier to follow, the new component will have a *Len* and *Wid* parameter instead of the standard *L* and *W*.
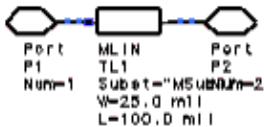
Using the command "*View/Create/Edit Schematic Symbol*" we switch to the symbol view. Now we can create any symbol we want to represent the new MLIN.
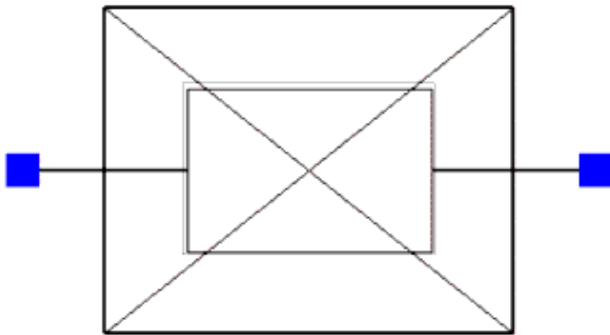
Moving on to the layout the standard MLIN artwork is defined on the *cond* layer. The ports are also added. Then the additional artwork that makes this component different than the standard MLIN is added on non-metallized layers. Finally the construction lines are added for use by the two stretch controls.
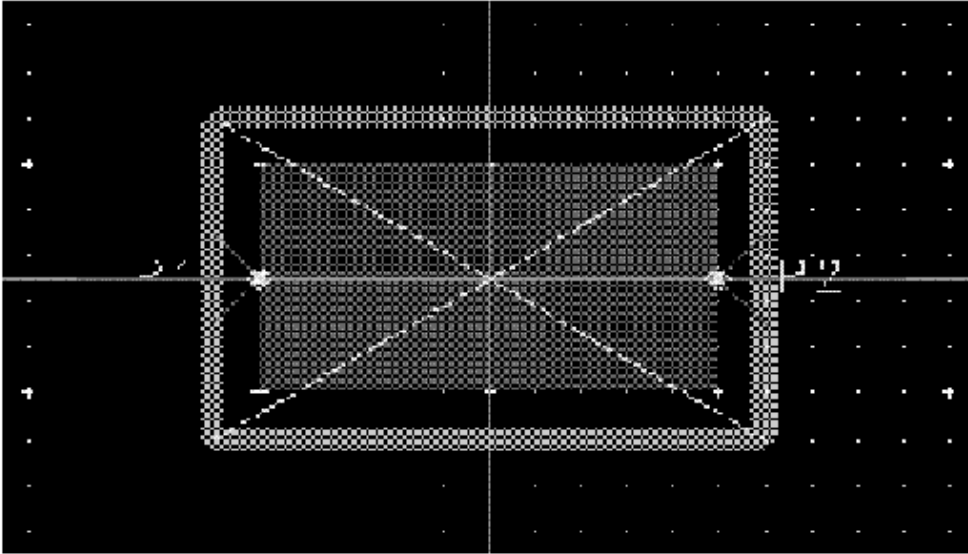
The stretch controls for length and width are defined as:

Control:	Stretch
	Direction: Both
	Length: Wid
	Offset: 50.0 mil

Control:	Stretch
	Direction: Positive
	Length: Len
	Offset: 100.0 mil

After the compile, the last thing to check is the Design/Parameters dialog. In the Parameters section initial values are assigned to the two parameters. The General section is similar to this:

Design/Parameters:2

Name: MLIN2

[General] [Parameters]

Description

MLIN2

Component Instance Name

MLIN2

Symbol Name

MLIN2

Library Name

*

Note: An "*" indicates current project.

◻ Allow only one instance

◻ Include in BOM

◻ Layout Object

◻ Simulate from Layout (SimLay)

Simulation

Model

Subnetwork

Simulate As

◻ Copy Component's Parameters

Artwork

Type

AEL Macro

Name

pam_MLIN2

| Delete Items | Save AEL file | Cancel | Help |

This finishes the new MLIN2 component definition. It's a simple hierarchical design that simulates as a simple MLIN but generates custom artwork. The next step is to use MLIN2 in a design.

Open a new design called *top* . In this design we insert a new MLIN2 component between two standard MLIN components. We add the Termination components, grounds, substrate, and S-Parameter simulation components.
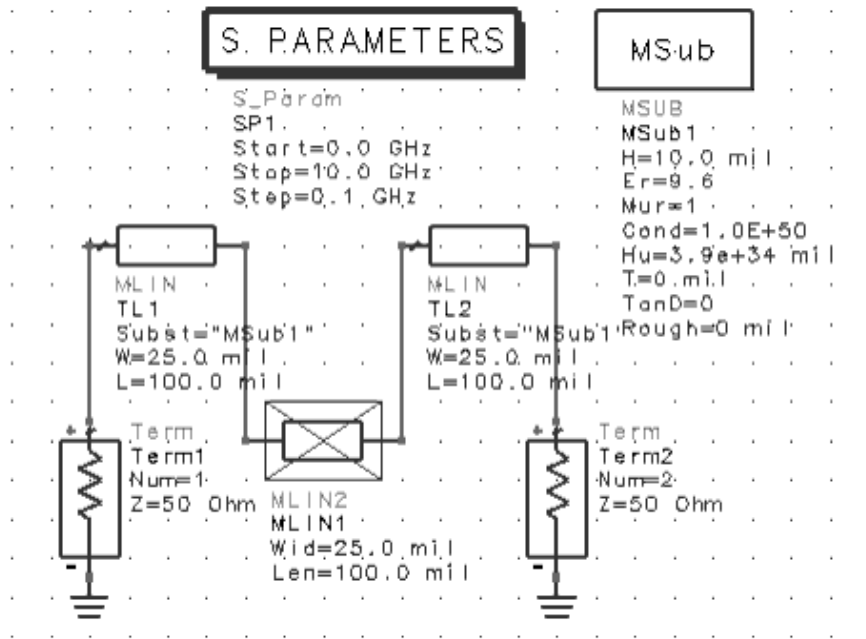
Selecting *Layout/Generate/Update Layout* creates the physical layout using the new artwork.
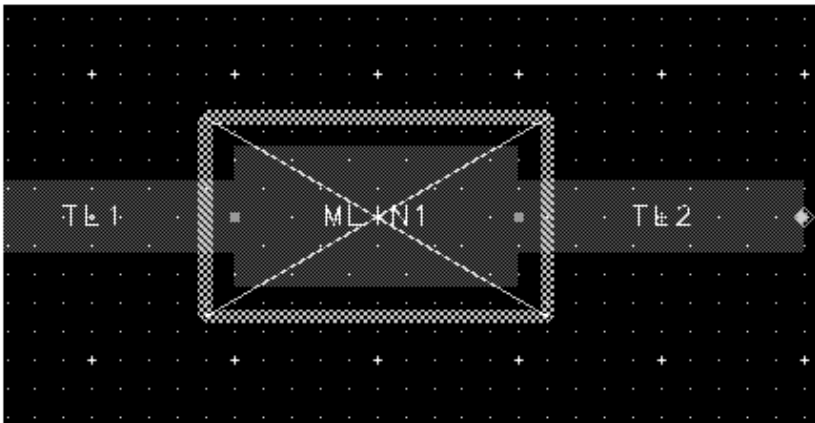
And finally, performing a simulation generates the appropriate S11 response that we would expect for this design.

freq (0.0000 Hz to 10.00GHz)
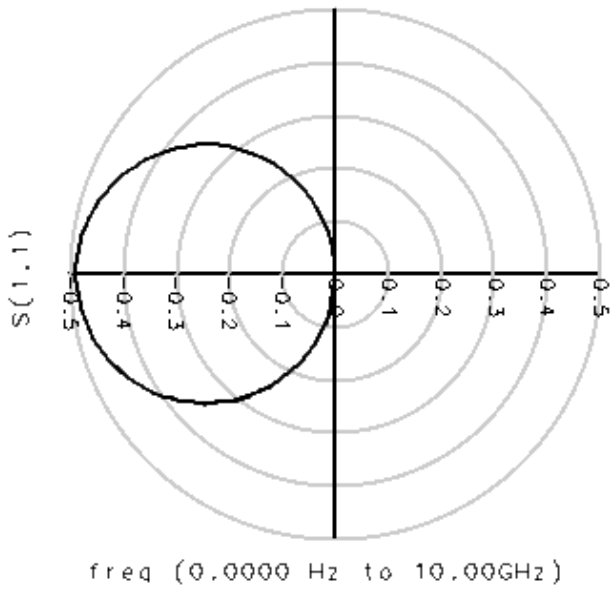
At this point the new component is ready to move to a library for use by designers in creating circuits with the modified artwork. Of course, this process could be applied to any or all of the standard components to make versions with artwork customized to the particular process being use to build the products.

Code Walkthrough

The section describes the code that was generated to produce the model created in *Example: Creating a Multi-Coupled Line.* (gcc)

The source shape definition for the model is a rectangle with two ports and two control lines.
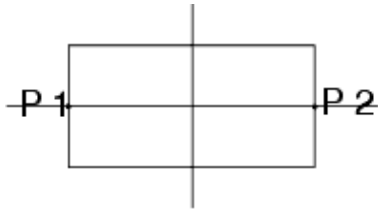
Three controls specify the model:

- A stretch control for length that acts on the rectangle and both pins.
- A stretch control for width that acts only on the rectangle.
- A repeat control for the number of lines to create that acts on the rectangle and both pins.

The actual defined controls, as seen in the Detail list from the Viewer dialog box, look like:

Control:	Stretch
	Direction: Both
	Distance: length
	Offset: 100.0 mil
Control:	Stretch
	Direction: Both
	Distance: width
	Offset: 50.0 mil
Control:	Repeat
	Direction: In X
	X Number: number
	X Distance: width+space

The Generated Code

The model header.

```
/******************************************************************/
/*                                                                */
/*        COPYRIGHT Agilent Technologies 1997, 1998, 1999, 2000   */
/*                                                                */
/* Graphical Cell Compiler generated Parameterized Artwork Macro: */
/*                                                                */
/*        Compiled from:  /a/new/wlv/rivets/d2/stengele/gcc_prj/networks/gcc */
/*        Output to:      /a/new/wlv/rivets/d2/stengele/gcc_prj/networks/gcc_art.ael */
/*        As model:       gcc                                      */
/*        Generated on:   Wed Apr 14 13:35:34 2004                 */
/*        ADS Version:    2.70 (Apr 10 2004)                       */
/*        GCC Revision:   6.0                                      */
/*                                                                */
/******************************************************************/
```

This statement makes sure that the model is loaded into the proper AEL vocabulary. Since AEL code in the ~/networks directory is by default loaded into a different vocabulary, this statement makes sure that existing models and new models being developed all reside in the same vocabulary.

```
#voc(CmdOp)
```

Some comments to help identify what the different variable types are.

```
/* Datatypes:                                  */
/*      l_      primitive list from repeat      */
/*      p_      primitive to modify             */
/*      i_      initial primitive definition    */
/*      r_      primitive rotation              */
/*      w_      primitive width                 */
```

The declarations of the variables for each shape. The comments help indicate what the shape is and what layer it came from.

```
/*              Polygon (from a Rectangle) on mask 1    */
decl l_4127E1E8;
decl p_4127E1E8;
decl i_4127E1E8;
/*              Port    */
decl l_412BBD20;
decl p_412BBD20;
decl i_412BBD20;
decl r_412BBD20;
/*              Port    */
decl l_412BB7E8;
decl p_412BB7E8;
decl i_412BB7E8;
decl r_412BB7E8;
```

This section initializes the above variables with the actual shape data.

```
/******************************************************/
/******************************************************/
defun pam_init_couple()
{
/*              Polygon (from a Rectangle) on mask 1    */
i_4127E1E8 = {
     {-0.00127, -0.000635},
     {0.00127, -0.000635},
```

```
      {0.00127, 0.000635},
      {-0.00127, 0.000635},
      {-0.00127, -0.000635}   };
/*              Port     */
i_412BBD20 = {
      {-0.00127, 0}   };
r_412BBD20 = 90;
/*              Port     */
i_412BB7E8 = {
      {0.00127, 0}    };
r_412BB7E8 = -90;
```

The variable initialization is completed. A test is done to verify that the data is of type real, and the initial data is copied into the primitive structure, which is where actual modifications will take place.

```
l_4127E1E8 = list();
if (array_type(i_4127E1E8, "integer"))
      i_4127E1E8 = convert_array(i_4127E1E8, "real");
p_4127E1E8 = i_4127E1E8;
p_4127E1E8[0,0] = p_4127E1E8[0,0];      /* Force array
copy */
l_412BBD20 = list();
if (array_type(i_412BBD20, "integer"))
      i_412BBD20 = convert_array(i_412BBD20, "real");
p_412BBD20 = i_412BBD20;
p_412BBD20[0,0] = p_412BBD20[0,0];      /* Force array
copy */
l_412BB7E8 = list();
if (array_type(i_412BB7E8, "integer"))
      i_412BB7E8 = convert_array(i_412BB7E8, "real");
p_412BB7E8 = i_412BB7E8;
p_412BB7E8[0,0] = p_412BB7E8[0,0];      /* Force array
copy */
}
```

The actual control operations are performed on the data. For each control, several steps are done:

- Load construction line information. If the control uses a construction line as a reference, the data for that line is sent to the control.
- Evaluate equation. For each control, user-defined equations for each parameter are evaluated to generate a single real value.
- Call control. For each shape acted on, the control is called with that shape's data, and all control parameters.

All data is passed as name/value pairs, so that specific parameter order is not required. This also makes the program very flexible.

```
/*********************************************************/
/*********************************************************/
defun pam_process_couple(length,width,number,space)
{
decl pam_rep;
decl pam_var;
pam_set_cline_info(
```

```
        PAM_LINE_X1,            0,
        PAM_LINE_Y1,            -0.001814322,
        PAM_LINE_X2,            0,
        PAM_LINE_Y2,            0.001814322,
        PAM_LINE_DX,            0,
        PAM_LINE_DY,            1,
        PAM_LINE_SLOPE,         9000000000);
decl p_length_1 = length;
decl p_offset_1 = 100.0 mil;
pam_do_stretch(
        PAM_COMMON_INIT,        i_4127E1E8,
        PAM_COMMON_DATA,        p_4127E1E8,
        PAM_COMMON_PRIM,        PAM_POLYGON_TYPE,
        PAM_STRETCH_DIRECTION,  3,
        PAM_STRETCH_LENGTH,     p_length_1,
        PAM_STRETCH_OFFSET,     p_offset_1);
pam_do_stretch(
        PAM_COMMON_INIT,        i_412BBD20,
        PAM_COMMON_DATA,        p_412BBD20,
        PAM_COMMON_PRIM,        PAM_PORT_TYPE,
        PAM_STRETCH_DIRECTION,  3,
        PAM_STRETCH_LENGTH,     p_length_1,
        PAM_STRETCH_OFFSET,     p_offset_1);
pam_do_stretch(
        PAM_COMMON_INIT,        i_412BB7E8,
        PAM_COMMON_DATA,        p_412BB7E8,
        PAM_COMMON_PRIM,        PAM_PORT_TYPE,
        PAM_STRETCH_DIRECTION,  3,
        PAM_STRETCH_LENGTH,     p_length_1,
        PAM_STRETCH_OFFSET,     p_offset_1);
pam_set_cline_info(
        PAM_LINE_X1,            -0.002561844,
        PAM_LINE_Y1,            0,
        PAM_LINE_X2,            0.005102352,
        PAM_LINE_Y2,            0,
        PAM_LINE_DX,            1,
        PAM_LINE_DY,            0,
        PAM_LINE_SLOPE,         0);
decl p_length_2 = width;
decl p_offset_2 = 50.0 mil;
pam_do_stretch(
        PAM_COMMON_INIT,        i_4127E1E8,
        PAM_COMMON_DATA,        p_4127E1E8,
        PAM_COMMON_PRIM,        PAM_POLYGON_TYPE,
        PAM_STRETCH_DIRECTION,  3,
        PAM_STRETCH_LENGTH,     p_length_2,
        PAM_STRETCH_OFFSET,     p_offset_2);
```

This procedure sends the shape data off so the actual graphics can be created in the Layout window. If the shape was copied by a Repeat or Polar control, the list of shapes is sent. If it is still a single shape, only the one shape is created.

```
/*******************************************************/
/*******************************************************/
defun pam_output_couple()
{
de_set_layer(1);
if (is_list(l_4127E1E8))
    depam_output_polygon_list(l_4127E1E8);
else
```

```
        depam_output_polygon(p_4127E1E8);
de_set_layer(3);
de_set_layer(1);
if (is_list(l_412BBD20))
    depam_output_port_list(l_412BBD20, r_412BBD20);
else
    depam_output_port(p_412BBD20, r_412BBD20);
if (is_list(l_412BB7E8))
    depam_output_port_list(l_412BB7E8, r_412BB7E8);
else
    depam_output_port(p_412BB7E8, r_412BB7E8); }
```

This short procedure is called by the Advanced Design System to get the list of user-defined parameter names for this model.

```
/*******************************************************/
/*******************************************************/
defun pam_couple_parm_info()
{ decl parms;
parms = "length,width,number,space";
return(parms); }
```

This is the start of the main entry-point for the model. The first thing it does is check all the parameters and make sure they are actual values, not strings.

```
/*******************************************************/
/*******************************************************/
defun pam_couple(length,width,number,space)
{
if (is_string(length))
length = evaluate(length);
if (is_string(width))
width = evaluate(width);
if (is_string(number))
number = evaluate(number);
if (is_string(space))
space = evaluate(space);
de_set_global_db_factor();
```

This checks the version of this compiled model with the current version of the Graphical Cell Compiler. This is a check to identify macros that were created with previous versions. If there is any reason why the old macro will not work with the current version a message will be issued telling the user to re-compile the macro.

```
if (\!pam_verify_model(3))
return;
```

The actual work gets done here.

```
pam_init_couple();
pam_process_couple(length,width,number,space);
pam_output_couple();
}
```

A comment ends the file.

```
/\* end of file \*/
```

Compiling a Macro

This section provides details on how compile a PAM, including how to set the defaults for the component parameters that you have defined in the control dialog boxes.

A compiled Parameterized Artwork Macro (AEL script) acts like any other Agilent or user-supplied PAM. At insertion, the macro creates graphics and the artwork is saved as part of the design. As with any other macro, if the original PAM is changed, each design must be revisited and the instances updated.

When a you create a PAM, you give all parameters default values. When you insert the model defined by the macro, you can accepted or modify these values to customize the specific instance.

Compile Dialog Box

The selections in this dialog box control the actual creation of the Artwork Macro (AEL script). After the macro compiles, you define the parameter defaults.
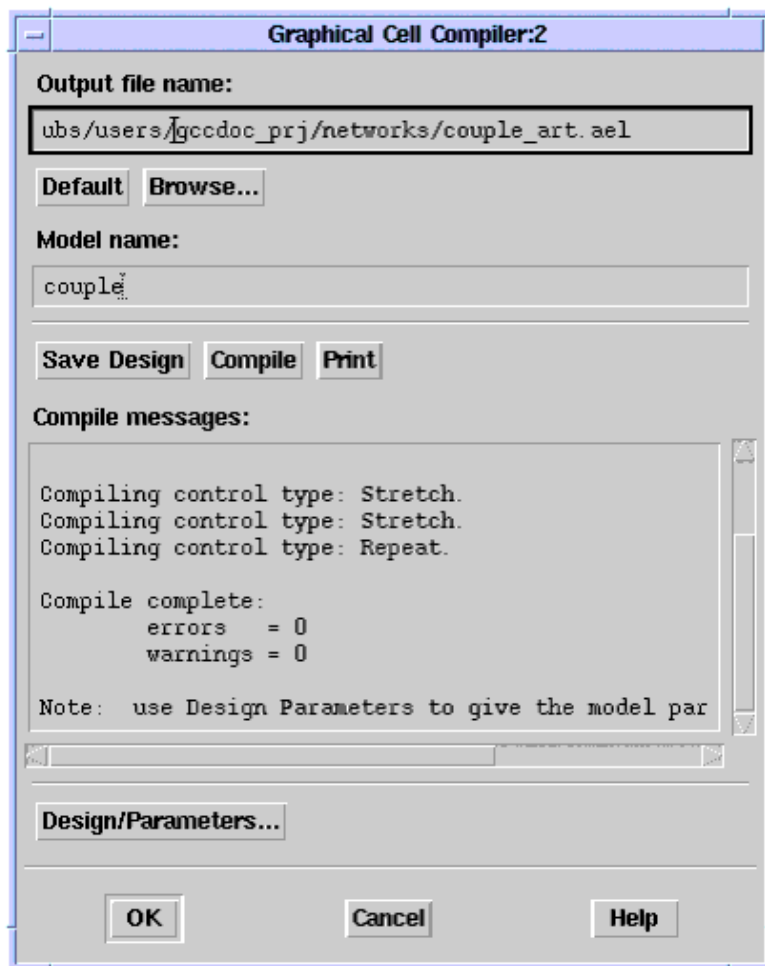
- *Output File Name* is the name of the file that will contain the compiled macro. Default = < *current design name* >_art.ael
- *Model Name* is the name of the component created by the compiled macro. Default =

< current design name >

- *Default* returns both the output file name and the model name to that of the current design.
- *Browse* opens a standard file browser.
- *Save Design* functions the same as the *File > Save* command. If the design has not been previously saved, the Save As dialog box opens so you can enter a name and save the design. If the design has already been save, it is resaved to the same name.

> **ⓘ Note**
> The compile function acts on a design as it is currently displayed, *not* on the design as it was last saved. If you forget to save a design *before* you compile the macro, the compiled macro will include any recent edits but may *not* match the saved design. It is good practice to *always* click *Save Design* before you compile.

Compile starts compiling the macro.
The *Compile Messages* window displays running messages as the macro compiles. Model parameters are listed. Check them to ensure that all of the parameters you expected have been created, and that they are spelled correctly.

Errors are listed. Correct any errors, then recompile the macro. See the section *GCC Error Messages".* (gcc)

Print enables you to print a copy of the contents of the Compile Messages window to use as a reference, should you need to edit the macro.

Design/Parameters displays the Design Definition dialog box. See the section Defining Component Parameter Defaults".

Defining Component Parameter Defaults

Clicking *Design/Parameters* in the PAM Compiler dialog box displays the same dialog box as the menu command *File > Design/Parameters*.

When creating a PAM, you do not need to edit entries in the General panel of this dialog box. By default, Name, and Component Instance Name are the same as the Model Name in the PAM Compiler dialog box; compiled macros are saved in the library called Compiled Artwork Macros.

Design Parameters:2

Name: couple

General | **Parameters**

Description

PAM generated model – couple

Component Instance Name

couple

Symbol Name

SYM_0Port

More Symbols...

Library Name

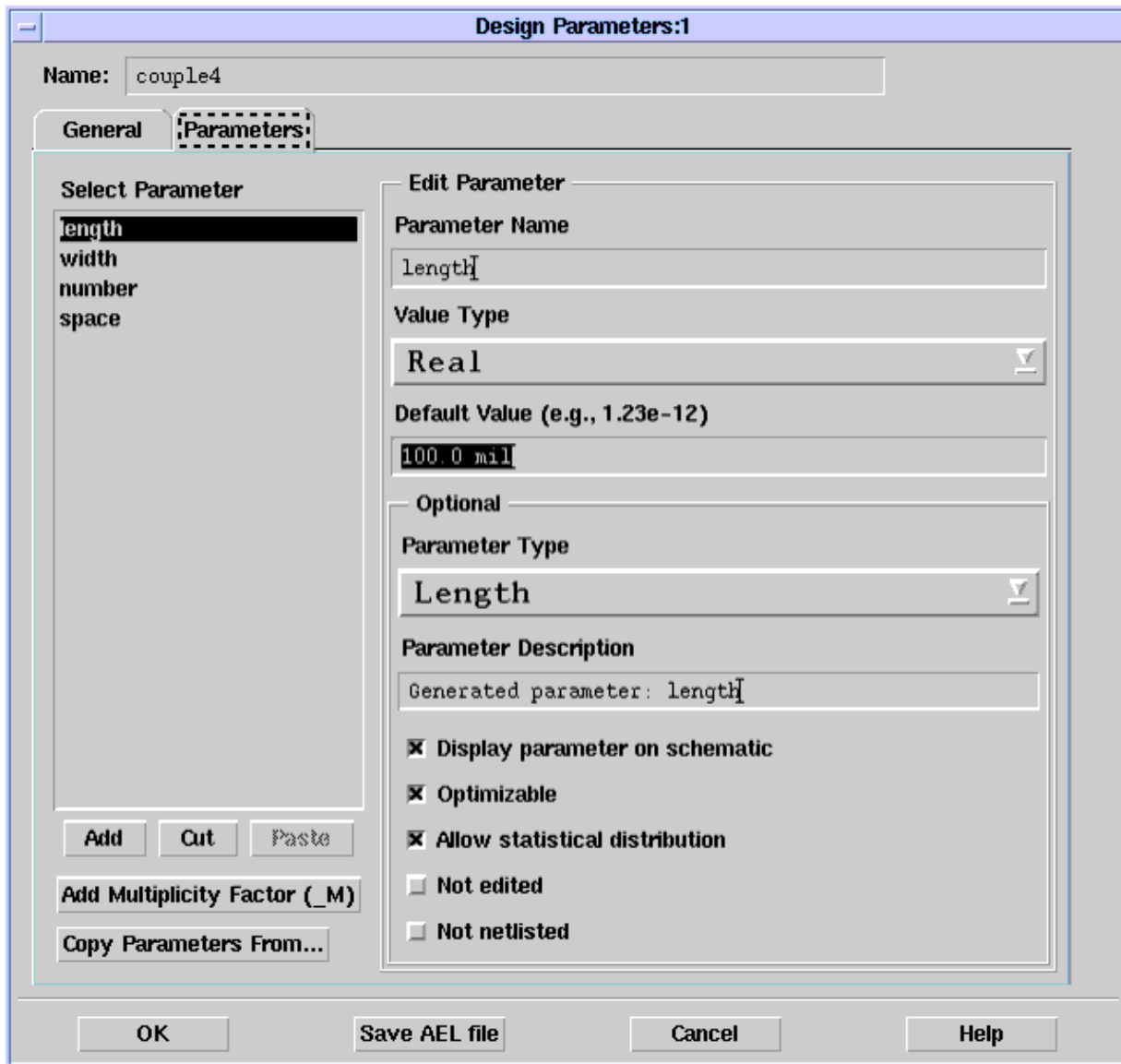Compiled Artwork Macro

Note: An "*" indicates current project.

☐ Allow only one instance

☐ Include in BOM

☐ Layout Object

☐ Simulate from Layout (SimLay)

Simulation

Model

Subnetwork

Simulate As

☐ Copy Component's Parameters

Artwork

Type

AEL Macro

Name

pam_couple

| OK | Save AEL file | Cancel | Help |

Click the Parameters tab.

In the Parameters panel, the Select Parameter field displays the defined component parameters. Make sure these are what you expected to see (number, spelling, and so on).

Ensure that the Default Value for a parameter is realistic. For example, a model with a length of 0.0 mil will not appear when you insert it.

Ensure that the Parameter Type is appropriate.

The Save AEL file button saves the file with the currently defined default parameters. The file is also automatically saved when you exit the PAM Compiler.

> ⚠ **Warning**
> Do not edit PAM parameters, spelling, or order in this dialog box. Edit them only through the View Controls and the control dialog boxes.

Parameter Type

Use the drop-down list in this field of the Design Definitions dialog box to set the type for the selected parameter. The type of parameter selected defines the units used to display a parameter:

Parameter Type	Default Layout Units
String	None
Unitless	None
Frequency	Hertz
Resistance	Ohms
Conductance	Farads
Inductance	Henries
Length	Mils
Time	Seconds
Angle	Degrees
Defined in the Preferences for Layout dialog box (Options > Preferences > Units).	

> ⚠ **Caution**
> If a unitless parameter is defined with a Parameter Type other than Unitless, the program treats the value provided as a measure of the units selected. For example, if a Count parameter is defined with a Parameter Type as Length, when you enter what you believe to be the number of times you want the graphic repeated, the program displays the value entered as length in mils (see *Missing or Incorrect Unit Designators* (gcc)).

Editing Component Parameters

The Select Parameter field in the Design Definitions dialog box lists the component parameters that have been defined for the model. The parameters are listed in the order in which they were compiled, which is the order in which the program looks for them when you use the macro. Because of this, if you want to add/delete parameters, you must do so *only* in the View Controls and the control dialog boxes, and then you must recompile the macro. *Never* perform any editing (Add, Cut, Paste, Copy) in the Design Definition Parameters panel.

You can add additional simulation parameters to the component parameter list. The parameter may be defined in the Edit Parameter side of the dialog. Select the last listed parameter. Fill in all the fields to define a new variable, and click the Add button. The newly defined parameter will be added after the selected one. Make sure any additional simulation parameters are only added *after* the list of compiled parameters. *never* change the order or names of the compiled parameters.

> ℹ **Note**
> See *Hints and Tricks* (gcc) for a technique to help control the parameter order.

If the macro is re-compiled after simulation parameters have been added, they will not be lost. The system marks all the compiled parameters by prepending *Generated parameter* in the Parameter Description field. In this way it knows which parameters are compiled (and my be removed if they are no longer being used) and which ones were added manually and should be retained.

During a compile, the system deals with the parameters in the following manner:

1. Saves all the previously defined parameters (compiled and simulation).
2. Generates the list of compiled parameters for the current set of controls. If the parameter is in the saved list, copy it's Default, Type, and Description fields as saved. Remove it from the saved list.
3. Of those parameters still in the saved list, if the Parameter Description field starts with *Generated parameter* it is a compiled parameter that is no longer needed. Delete it.
4. Copy all remaining simulation parameters from the saved list adding them after the compiled parameters.

It is not an error to edit the Parameter Description field and remove the "Generated parameter" part from the description compiled parameters. All this means is that the system no longer knows the parameter was generated so that if the controls are ever modified to not use it anymore, it will not be removed from the list (it will be treated like a simulation parameter and retained).

Configuration File Options

You can change some PAM defaults by setting configuration options in a local configuration file (*de_sim.cfg*). There is probably a configuration file in your home directory already, but if there is not, create the following:
$HOME/hpeesof/config/de_sim.cfg

The following three options are specific to the Graphical Cell Compiler (default values are shown):

CELL_COMPILER_DIRECTORY=networks

This specifies the default location for the compiled AEL scripts. When placed in the default location (the *networks* directory) they are read at startup, and are available in the next session. If you place the compiled scripts in another location, they are not automatically read in. You will have to read them in manually for subsequent session. For example, you might use the USER_AEL option to do this (see *Model Composer* (modcomp)).

CELL_COMPILER_LIBRARY=Compiled Artwork Macros

This specifies the library name as it appears in the Design/Parameters dialog box. All compiled models will be added to the specified library.

CELL_COMPILER_MAX_COPY=1000

This is a threshold value for the Repeat and Polar controls. It defines the number of copies the controls are allowed to make of each shape. It prevents infinite loops or unreasonable requests due to user-data errors. The default value is arbitrary, and there are many cases where the value should be larger; do not hesitate to change it.

First Spiral Example

This section provides a step-by-step procedure that takes you through using the Graphical Cell Compiler to generate a model for a square (four-sided) spiral. It starts at the very beginning and traces each trial-and-error step as the model is created.

Create the Source Layout

When creating a Parameterized Artwork Macro (PAM) using the Graphical Cell Compiler, we start by placing the required items in a source layout:

1. Create a new design
2. Use the positional coordinate readout at the bottom of the window to insert a *path* with *square* corners between 50,50 and 50,-50.
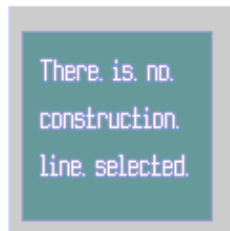
   > ℹ **Note** You must use *square* corners; otherwise, your results will not be accurate.
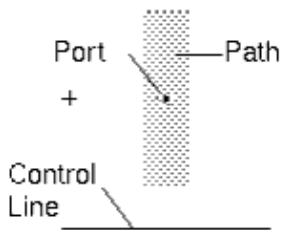
3. Insert a construction line below the path. Do *not* touch the path.
4. Insert a port at coordinates 50,0.
   Now we are ready to create a spiral.

Define a Polar Control

1. Choose the command **Macro** > **Polar** .

   Note the message displayed in the Polar control Definition dialog box. This appears because we opened the dialog box before we selected a control line and the shape that we want the control to apply to. This is not a problem.
2. Select the path and the control line. Do *not* select the port at this time (for now we will just work with the path).
3. We will try using the Polar control default values, so click OK.
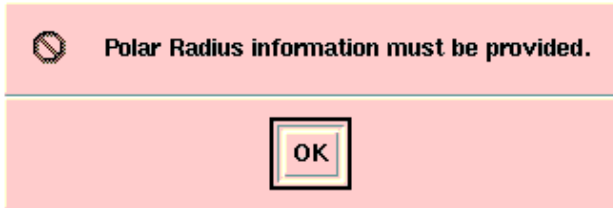
Again the program keeps us from making a mistake. We must define the radius we want to use.

4. In the Radius field, enter **0.0** (no units are needed when the value is 0). Now each field has an entry.
5. Click OK.
6. Choose **Macro * > * View/Edit** . The polar control that we defined is listed here.
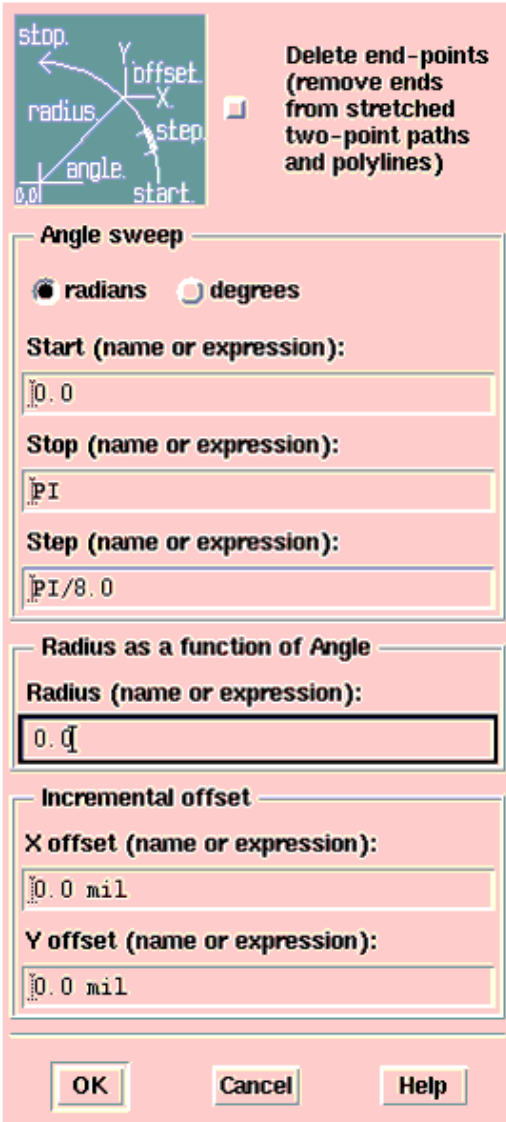
7. Click **Detail** .
   The detail list shows:

Control:	Polar
Delete Ends:	False
Units:	Radians
Start:	0.0
Stop:	PI
Step:	PI/8.0
Radius:	0.0
X Offset:	0.0 mil
Y Offset:	0.0 mil

8. Click OK in both the List and the Viewer dialog boxes.
9. Choose **Macro * > * Compile** .
10. Click **Save Design** .
    Since you saved this design after placing the items, the program simply resaves the design to the same name. Although we have not made any changes to the design (so the save is not necessary) you are reinforcing a good habit by saving before you compile.
11. Click **Compile** .
    The Compile Messages window displays:

Compile Messages:

```
Compile:   /hfs/hpsrmi/d1/users/ednah/MyDesigns/spi

Identified the model parameters:
        <none>.

Compiling control type: Polar.

Compile complete:
        errors   = 0
        warnings = 0
```

- The path and name of the design
- Model parameters (none, in this case)
- The type of control (Polar)
- Any errors or warnings
  If any model parameters had been defined, we would now set their default values, but at this point there are no parameters to set.
12. Click OK.
13. In the Main window, open a new Layout window.
14. Click the Display component library list icon and select the Compiled Artwork Macros library.
15. Double-click the component we just created and insert it in the Layout window:
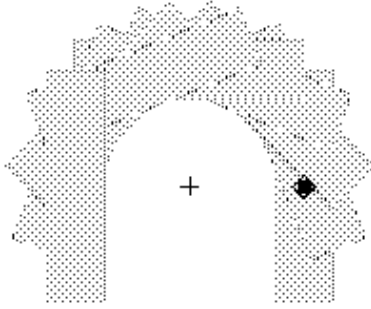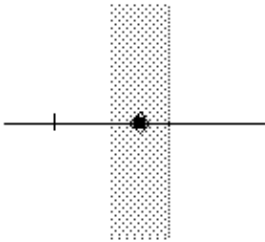
Since this does not look anything like a spiral, it is pretty obvious that we made an (artificial) mistake.

We need to cut the path with the construction line, so that the control *stretches* the path, rather than copying it.

Edit the Source Design and Check the Effects

1. In the source design, move the construction line up so that it crosses the port (and 0,0), as shown.

2. Choose **Macro * > * Compile** .
3. Click **Save Design** to save the change that you just made, then click **Compile** .
4. In the Layout window that contains the model, note that the inserted model did not change. Once you insert a model, it stays as defined by the macro when you inserted it. Either delete the old model, or move it out of the way.
5. Insert the updated model from the Compiled Artwork Macros library.
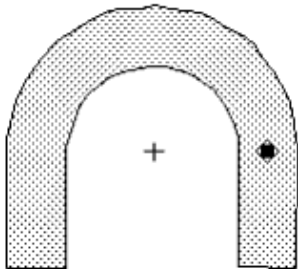
This is better, but we need to remove the straight parts of the path that are caused by the original shape. It is as if the control inserted a curve in the center of the original shape. All we want is the curve.

Remove the Ends of the Path

1. In the source design, choose **Macro** > **View/Edit** .
2. In the Viewer, select the Polar control and click **Edit** .
3. In the Polar Definition dialog box, click the button at the top of the dialog box to turn on the Delete End-Points feature. If you check the Detail List now, you will see that the control is defined as follows:

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	PI
Step:	PI/8.0
Radius:	0.0
X Offset:	0.0 mil
Y Offset:	0.0 mil

4. Save the design and recompile the macro. Then reinsert the model.
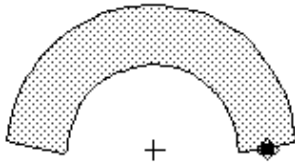
This looks much better.
We know that a spiral needs to go around several times, and that it must spiral out. For this example, we want it to go around twice.

## Set the Spiral for Out, Twice Around

1. In the Polar Definition dialog box, enter **4.0*PI** (twice around) in the *Stop* parameter field.
2. We know the model needs to spiral out, so we will set the radius of the spiral as a function of Angle. Enter **50.0**_angle mil* in the **Radius** parameter field.
   This time you *must* include the units, or you will get something *very* big.
   The Detail List now looks like:

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	4.0*PI
Step:	PI/8.0
Radius:	50.0*_angle mil
X Offset:	0.0 mil
Y Offset:	0.0 mil

Note the use of the Global Variable. For more information, see *Using Variables in the*

Radius & Offset Parameters". (gcc)

3. Save the design, recompile the macro, and reinsert the model.
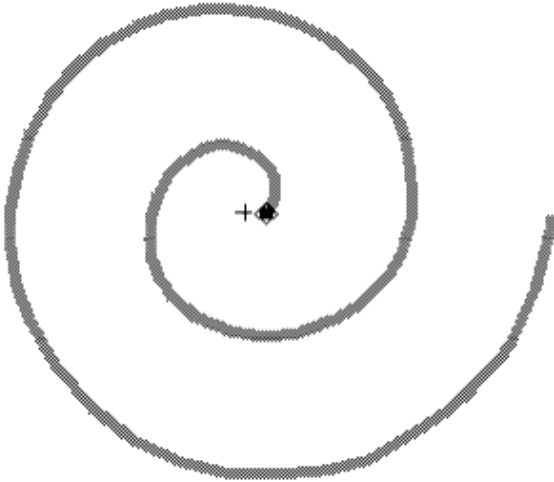
Well, we have a spiral. It is difficult to tell in this figure, but you will notice on your screen that this spiral is very big. This is because the variable *_angle* goes from 0 to 6.28 for the first time around, multiplied by our desired spacing of 50.0 (set in the *Radius* parameter).

Normalize Angle

If we change the value of Radius to normalize *_angle* :

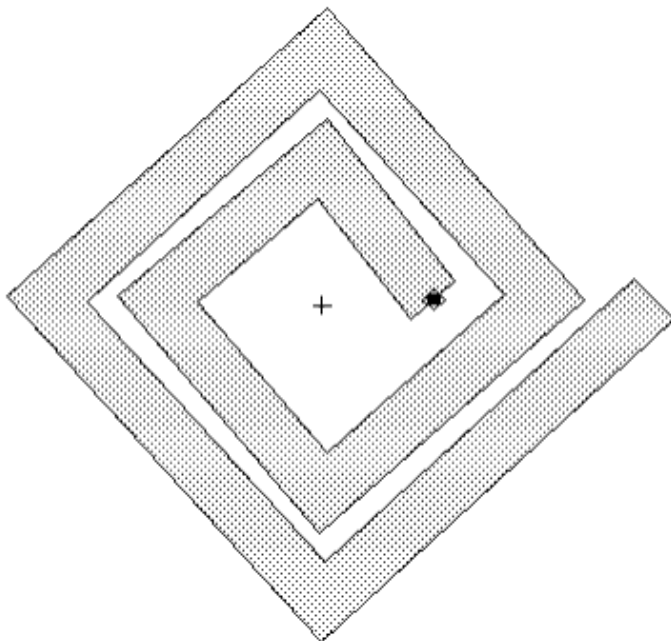| Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	4.0*PI
Step:	PI/8.0
Radius:	_angle/(2.0*PI)*50.0 mil
X Offset:	0.0 mil
Y Offset:	0.0 mil
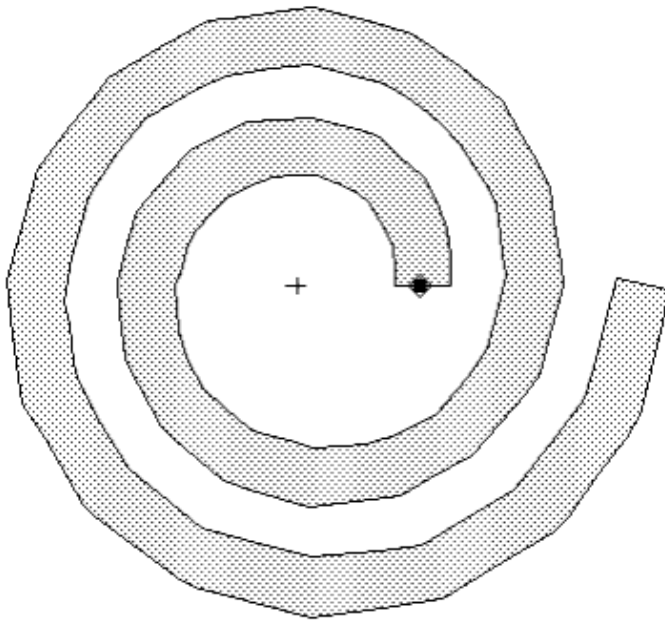
At this point we have a pretty good definition for a rounded spiral. We would want to decrease the step size (by increasing the number of steps) so that it would be smoother, but this is pretty close.
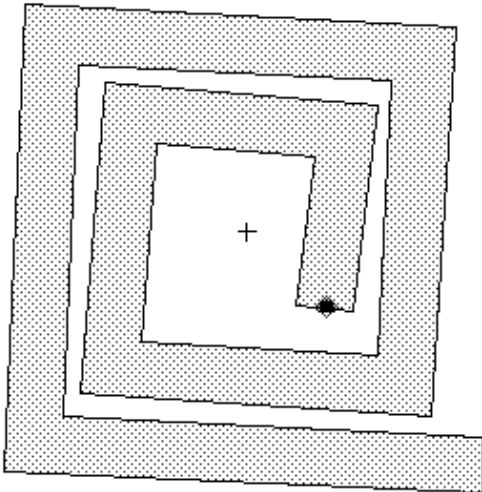
Increase the Step Size

For a square spiral, though, we need to *increase* the step size so we have only four points (steps) per cycle.

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	4.0*PI
Step:	2.0*PI/4.0
Radius:	_angle/(2.0*PI)*50.0 mil
X Offset:	0.0 mil
Y Offset:	0.0 mil

We still have several problems. The first is simply that we would like the spiral to be oriented vertically rather than based on the Angle points. To do this, we add a Rotate/Move/Mirror control.

Add a Rotate/Move/Mirror Control

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	4.0*PI
Step:	2.0*PI/4.0
Radius:	_angle/(2.0*PI)*50.0 mil
X Offset:	0.0 mil
Y Offset:	0.0 mil
Control:	Mirror
Units:	Degrees
Angle:	-45.0
X Offset:	0.0 mil
Y Offset:	0.0 mil
X Mirror:	FALSE
Y Mirror:	FALSE

The next problem, which you may have noticed, is that the lines are not far enough apart. Because we used 50.0 for the space, and the path is 25.0 wide, we would expect the space between lines to be the same as the width of the lines. The problem is in the radius formula , which specifies a distance of 50.0 from one corner to the next corner, but we want a distance of 50.0 from one *side* to the next *side* . The corner-to-corner distance must be larger than 50.0.

A Short Geometry Lesson

We must look into a little geometry to resolve this. We will use an angle greater than 90° (as if we had more than four sides) so we will be forced to come up with a general case.
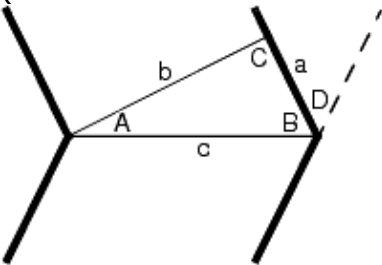
Using our two-turn spiral, the following is what we know about the triangle formed between two vertex points:

1. $\text{Side } b = 50.0 \text{ mil}$
   (line width plus spacing).

2. Since the sides are parallel and b is the distance between sides, then $\text{Angle } C = 90°$.

3. Because $\text{Angle } D = \dfrac{360}{sides}$, the more sides the spiral has the smaller the incremental angle at each vertex.

For this case: $\text{Angle D} = \frac{360}{4}$

.

4. Because $\text{Angle B} = \frac{180 - D}{2} = \frac{180 - 360/4}{2}$

$\text{Angle B} = \frac{180}{2} - \frac{180}{4}$

,

.

5. The Law of sines states that: $\frac{c}{\sin(C)} = \frac{b}{\sin(B)}$

. Since $\sin(90) = 1$

, we get

$c = \frac{b}{\sin(B)}$

.

$$c = \frac{50.0}{\sin\left(\frac{180}{2} - \frac{180}{4}\right)}$$
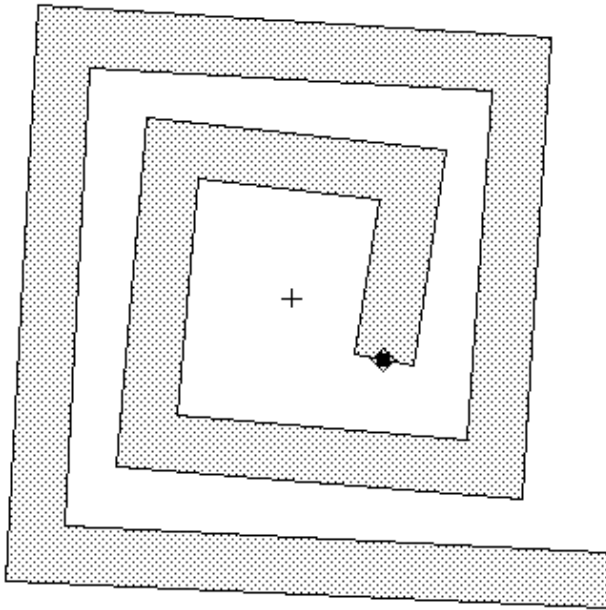
If we substitute the above values:

$$c = \frac{50.0}{\sin\left(\frac{\pi}{2} - \frac{\pi}{4}\right)}$$

Switching to radians:

We could evaluate the constant, but we will leave it like this so it is easier to replace the *4* with a variable for the number-of-sides later.

> **Note** A general purpose spiral would use user-supplied values for the line width, spacing, number of turns, and perhaps number of sides. For now we are using constants for all of these. Once the basic model works, we will go back and add the parameters.

Use the More Accurate Equation for Radius

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	4.0*PI
Step:	2.0*PI/4.0
Radius:	_angle/(2.0*PI)*50.0 mil/sin(PI/2.0-PI/4.0)
X Offset:	0.0 mil
Y Offset:	0.0 mil
Control:	Mirror
Units:	Degrees
Angle:	-45.0
X Offset:	0.0 mil
Y Offset:	0.0 mil
X Mirror:	FALSE
Y Mirror:	FALSE

The last problem (why the spiral is not *square*) is much more serious. For a simple rounded spiral, each point is placed at an ever-increasing radius from the center. As is obvious, that does not work for a square spiral (or any *n* -sided spiral where *n* is small). Basically, each point needs to be *pushed* along the direction of the side by a fraction of the distance between spirals.

Add an Offset

Given that we are at some angle Theta (as we step around the spiral) we need to push the

vertex point for the current side out. The angle B is the same as we derived above. Since each time we move around the spiral we move 50.0 mil out, we move 1/4 (1/sides actually) that amount for each side. Our offset needs to be 1/2 that amount, so we need to move (remembering that most of these constants will become variables later) 50.0 / (4 * 2).
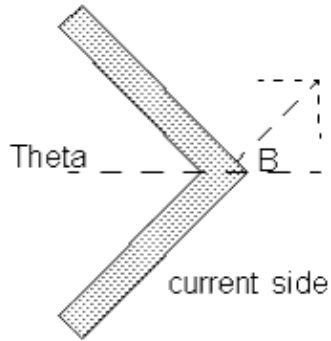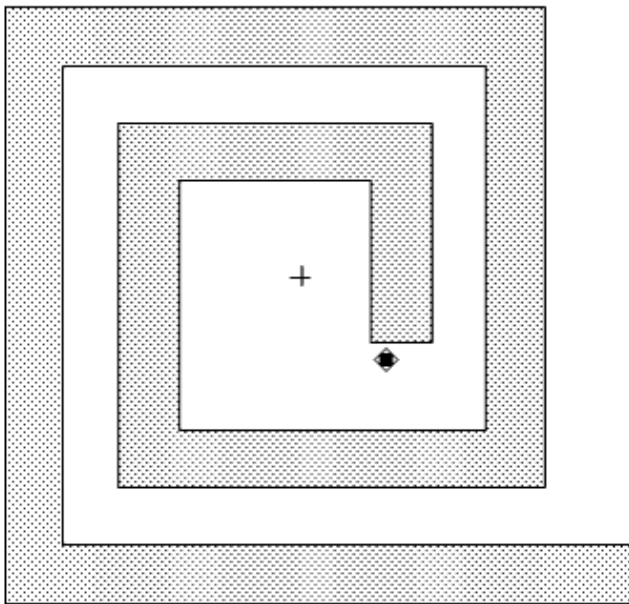

Next, we need to rotate that amount based on the current step angle Theta plus the angle B.

Since rotation is defined as

```
X = r * cos(a) - r * sin(a)
Y = r * cos(a) + r * sin(a)
```

we can substitute everything and we're ready to go. This is where the X and Y offset fields become useful. We take all this and fill it in:
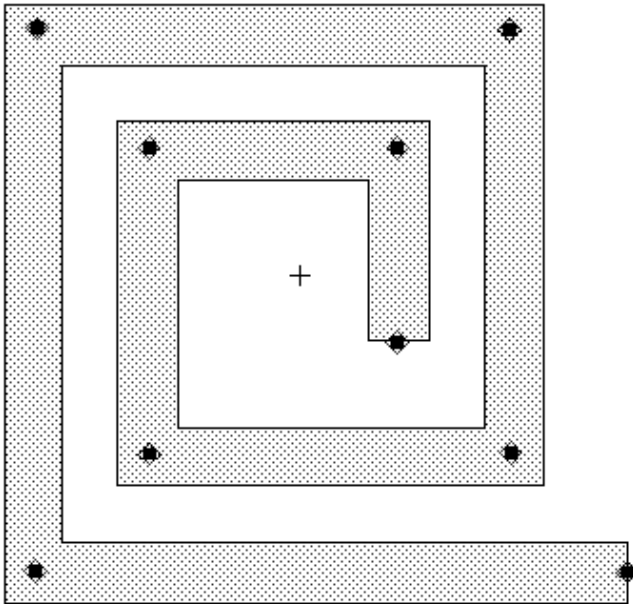
Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	4.0*PI
Step:	2.0*PI/4.0
Radius:	_angle/(2.0*PI)*50.0 mil/sin(PI/2.0-PI/4.0)
X Offset:	50.0 mil/(4*2) *(cos(_angle+PI/2.0-PI/4.0)-sin(_angle+PI/2.0-PI/4.0))
Y Offset:	50.0 mil/(4*2) *(cos(_angle+PI/2.0-PI/4.0)+sin(_angle+PI/2.0-PI/4.0))
Control:	Mirror
Units:	Degrees
Angle:	-45.0
X Offset:	0.0 mil
Y Offset:	0.0 mil
X Mirror:	FALSE
Y Mirror:	FALSE

Now it is time to do something about pins.

Include the Port

We start by simply including the port in both controls. Nothing else changes.

1. In the source design, choose **Macro** > **View/Edit** .
2. In the Viewer, select the Polar control and click **Edit** .
3. In the design, select the path, the control line, *and* the port, then click OK.
4. Save the design, recompile the macro, and reinsert the model.

We need to use the same polar equations for the ports, but we do *not* want all the steps. In fact, we want only one step (the last one).

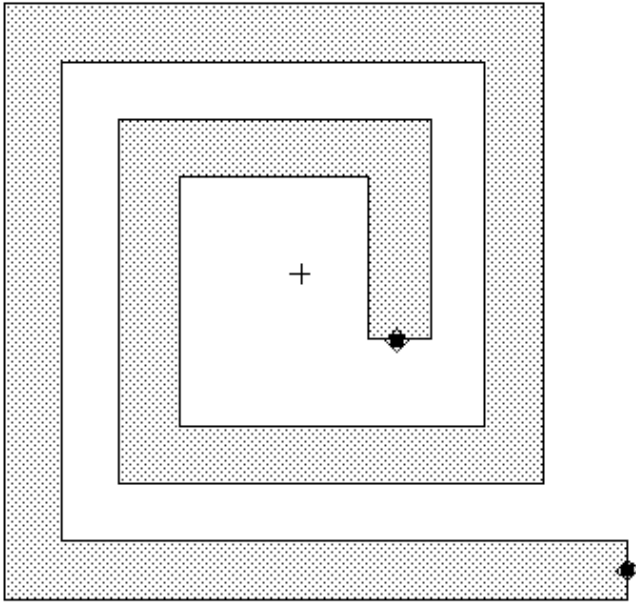# Add a Polar Control for the Port

In order to fix the previous problem of too many ports being created, we need to create a new Polar control that acts just on the port. This means we need to edit the existing Polar control and remove the port from its selected shapes. We then create a new Polar control to operate on just the port. The two Polar controls will be identical except that this second one will have the Step set to the same value as the Stop so that it creates only one copy of the port at the end of the spiral.

It is important that the two Polar controls take place before the Mirror control so that the generated spiral and the ports line up correctly.

Be sure that when all the additions are done, each control is operating on the listed shapes:

- First polar control: construction line and path.
- Second polar control (where Stop = Step): construction line and port.
- Mirror control: path and port.

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	4.0*PI
Step:	2.0*PI/4.0
Radius:	_angle/(2.0*PI)*50.0 mil/sin(PI/2.0-PI/4.0)
X Offset:	50.0 mil/(4*2) *(cos(_angle+PI/2.0-PI/4.0)-sin(_angle+PI/2.0-PI/4.0))
Y Offset:	50.0 mil/(4*2) *(cos(_angle+PI/2.0-PI/4.0)+sin(_angle+PI/2.0-PI/4.0))
Control:	Polar
Delete Ends:	False
Units:	Radians
Start:	0.0
Stop:	4.0*PI
Step:	4.0*PI
Radius:	_angle/(2.0*PI)*50.0 mil/sin(PI/2.0-PI/4.0)
X Offset:	50.0 mil/(4*2) *(cos(_angle+PI/2.0-PI/4.0)-sin(_angle+PI/2.0-PI/4.0))
Y Offset:	50.0 mil/(4*2) *(cos(_angle+PI/2.0-PI/4.0)+sin(_angle+PI/2.0-PI/4.0))
Control:	Mirror
Units:	Degrees
Angle:	-45
X Offset:	0.0 mil
Y Offset:	0.0 mil
X Mirror:	FALSE
Y Mirror:	FALSE

The final step is to replace the constants with parameter names so that the model is variable instead of fixed.

Replace the Constants with Parameters

1. We use the following parameters in the two Polar controls:
    - Turns = the number of complete revolutions
    - Sides = the number of sides
    - Width = the width of the path
    - Space = the space between the revolutions
2. We also add a Width control to the path so that it resizes in response to the width parameter:
    - Control: Width

- Change: Absolute
- Width: width

Up until now, we used constants and had to include the *mil* unit designator. In the following equations we use parameters we will define to be in mils (in step 2), so a designator is no longer needed. If we left it in we would basically be converting to mils twice, and the results of the equations would be too small. Filling everything in and doing a little equation cleanup, we get:

Control:	Width
Change:	Absolute
Width:	width
Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	turns*2*PI
Step:	2.0*PI/sides
Radius:	_angle/(2.0*PI)*(width+space)/sin(PI/2.0-PI/sides)
X Offset:	(width+space)/(sides*2) *(cos(_angle+PI/2.0-PI/sides) -sin(_angle+PI/2.0-PI/sides))
Y Offset:	(width+space)/(sides*2) *(cos(_angle+PI/2.0-PI/sides) +sin(_angle+PI/2.0-PI/sides))
Control:	Polar
Delete Ends:	False
Units:	Radians
Start:	0.0
Stop:	turns*2.0*PI
Step:	turns*2.0*PI
Radius:	_angle/(2.0*PI)*(width+space)/sin(PI/2.0-PI/sides)
X Offset:	(width+space)/(sides*2) *(cos(_angle+PI/2.0-PI/sides) -sin(_angle+PI/2.0-PI/sides))
Y Offset:	(width+space)/(sides*2) *(cos(_angle+PI/2.0-PI/sides) +sin(_angle+PI/2.0-PI/sides))
Control:	Mirror
Units:	Degrees
Angle:	-360/(sides*2)
X Offset:	0.0 mil
Y Offset:	0.0 mil
X Mirror:	FALSE
Y Mirror:	FALSE

3. After compiling the macro, we must set default values for the parameters we have

defined. Note that the parameters are listed in the Compile Messages window.

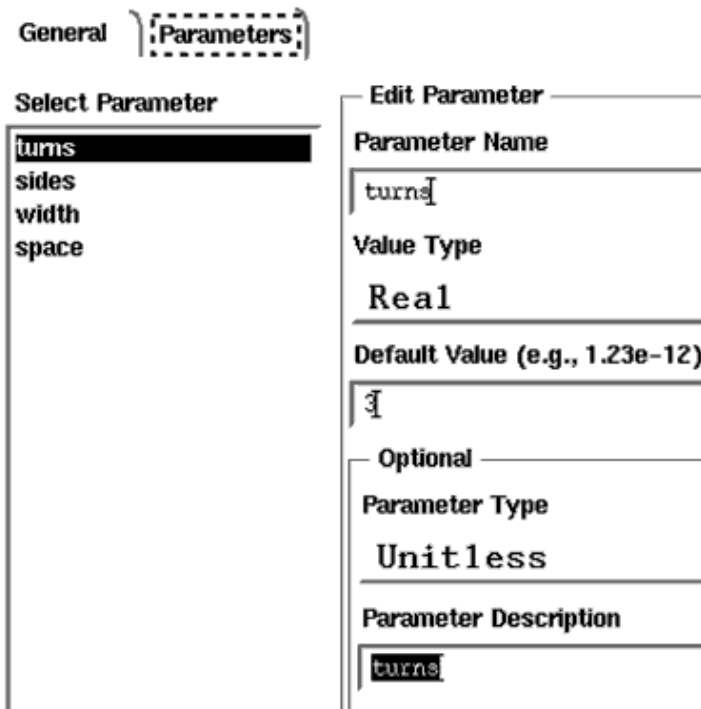**Compile messages:**

```
Identified the model parameters:
        width, turns, sides, space.

Compiling control type: Width.
Compiling control type: Polar.
Compiling control type: Polar.
Compiling control type: Rotate/Move/Mirror.

Compile complete:
        errors   = 0
        warnings = 0
```

ⓘ **Note** Depending on your setup, the messages displayed can differ from those shown.

4. Click **Design/Parameters** .
5. In the dialog box that appears, click the *Parameters* tab.
   The parameters are listed in the Select Parameters window. Set them as follows:

General ⟩ ┆Parameters┆

Select Parameter

turns
sides
width
space

Edit Parameter

Parameter Name

turns

Value Type

Real

Default Value (e.g., 1.23e–12)

3

— **Optional** —

Parameter Type

Unitless

Parameter Description

turns

Turns =	3, Unitless
Sides =	4, Unitless
Width =	20.0 mil, Length
Space =	15.0 mil, Length

6. Click **Save AEL** , then OK.
7. In the PAM Compiler dialog box, click OK.
8. Place the new component.
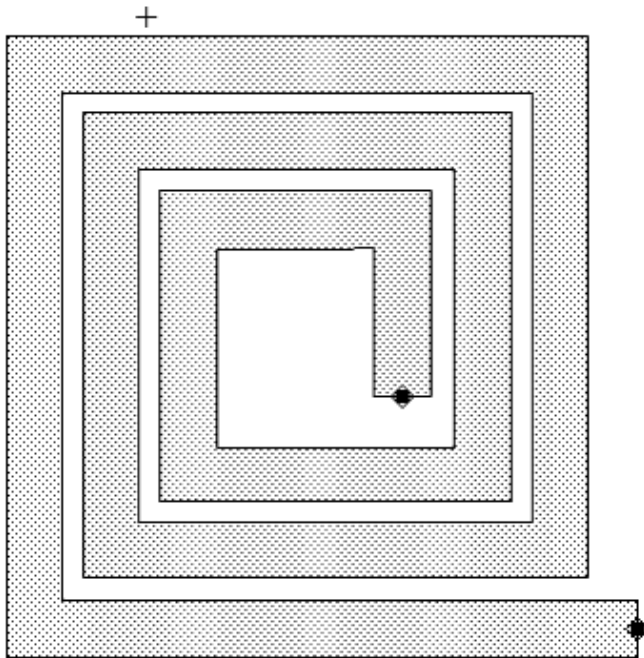
Simplifying Long Equations

In this example, the equations in the Polar controls are pretty long and there is a lot of duplication in them. There is a way to simplify them.

> **ⓘ** **Warning** These types of modifications *must* be done with extreme care. If you reorder the controls or make other changes, it is *easy* to create a situation where constants are referenced *before* they are defined. It is much safer to leave expressions long, with a lot of duplication.

Because the *Start/Stop/Step* equations are evaluated before the *Radius/Offset* equations, and since any local variables are visible during the evaluation of the *Radius/Offset* equations, it is possible to define a local constant to use in them. Using local constants, the controls look like:

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0; decl temp_a = PI/2.0-PI/sides;
Stop:	turns*2*PI
Step:	2.0*PI/sides
Radius:	_angle/(2.0*PI) *(width+space)/sin(temp_a)
X Offset:	(width+space)/(sides*2) *(cos(_angle+temp_a)-sin(_angle+temp_a))
Y Offset:	(width+space)/(sides*2) *(cos(_angle+temp_a)+sin(_angle+temp_a))
Control:	Polar
Delete Ends:	False
Units:	Radians
Start:	0.0
Stop:	turns*2.0*PI
Step:	turns*2.0*PI
Radius:	_angle/(2.0*PI)*(width+space)/sin(temp_a)
X Offset:	(width+space)/(sides*2) *(cos(_angle+temp_a)-sin(_angle+temp_a))
Y Offset:	(width+space)/(sides*2) *(cos(_angle+temp_a)+sin(_angle+temp_a))
Control:	Mirror
Units:	Degrees
Angle:	-360/(sides*2)
X Offset:	0.0 mil
Y Offset:	0.0 mil
X Mirror:	FALSE
Y Mirror:	FALSE

The constant is declared as part of the first Polar control's *Start* parameter as it is evaluated in AEL. It is then available for each evaluation of the *Radius* and *Offset* expressions for the first and second Polar controls.

By editing the AEL file, you can use the spiral model to create spirals with different widths, as shown.
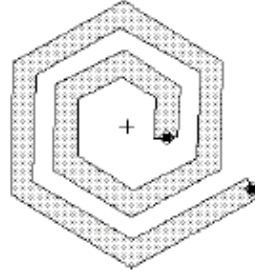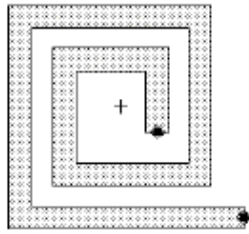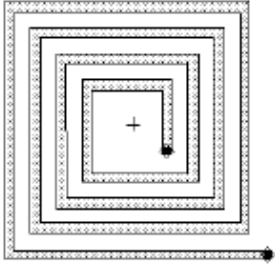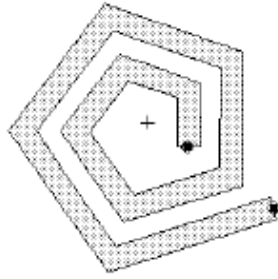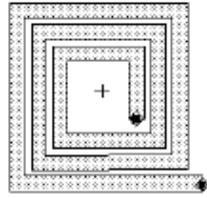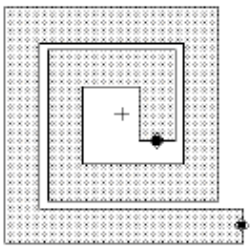
GCC Error Messages

Building a Parameterized Artwork Macro (PAM) is similar to working in a programming language such as Basic, Pascal, or C, in that there is source code that is compiled into an executable program. In the case of a PAM, the source code is the source layout, comprising shapes to manipulate and controls that perform the manipulation. The source layout is compiled into an executable AEL macro (which is itself a programming language used by the Advanced Design System).

As when you use a programming language, you should never delete the source code. Save the source layout as long as the macro remains in use. If you wish to modify the source layout to create a new macro, copy the design into a new layout.

As with programming languages, these types of errors are possible:

- Syntax
- Semantic
- Logic
- Run-time

This section provides details on these types of errors as they pertain to a PAM.

Syntax Errors

In the context of a PAM, syntax errors are typically errors in equations. For example, the equation `length++1` would be a syntax error. Most syntax errors will be trapped in the control dialog when you click OK. At that point, each expression is passed to an AEL parser to check the syntax. Any errors reported by the parser are displayed in a pop-up dialog, and the offending expression is colored red to help you quickly see which one has the error. All syntax errors must be fixed before the control is accepted.
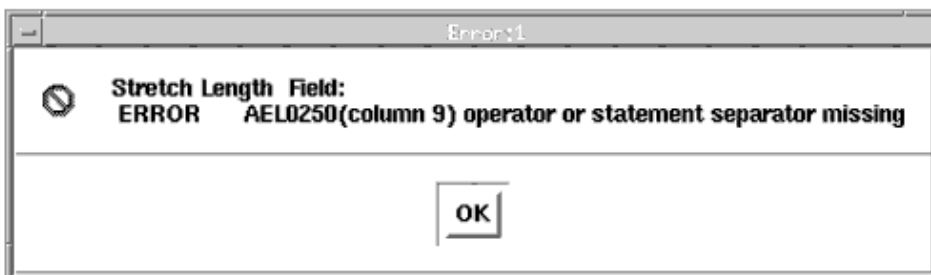
Semantic Errors

Semantic errors are errors in usage. These include trying to do a list index into a variable that is not a list (using the AEL *nth()* function). These types of errors are not identified by the AEL parser (the syntax is correct) and at this point will not be identified until the parser generates a run-time error (see below).

Logic Errors

Logic errors are errors in the flow of the program. Everything seems correct, the macro runs fine, but it generates the wrong results. These errors may be very hard to find and fix in a complicated macro, which is why a macro should be built from a simple starting case and expanded. If you start with something simple that works, it is much easier to add a little bit more. If it fails, the problem is in what was just added, not in anything that was previously working. There is no easy way to find and fix logic errors. You must understand what you want to do, and what the controls are actually doing.

The example in *First Spiral Example* (gcc) illustrates the concept of starting simple and working up to a complicated macro.

Run-Time Errors

In the context of a PAM, run-time errors fall into two classes:

- Math errors
- Untrapped semantic errors

Math Errors

Math errors are typically caused by improper user-input for a parameter. One example is user-input that results in a divide-by-zero in an equation. These errors are trapped by AEL and cause the macro to abort execution.

Another example error would be user-input that causes one of the copy controls (Repeat or Polar) to go into an infinite loop. These errors are trapped by the run-time control code when a loop request exceeds the internal maximum copy limit. Again, if the error occurs the macro execution is aborted. You can set the maximum copy limit using the *CELL_COMPILER_MAX_COPY=1000* (gcc) configuration file option.

Untrapped Semantic Errors

Of all the errors possible, untrapped semantic errors will generate errors that are the most removed from the actual cause. This is best demonstrated by an example:

Defining Repeat Distance as Step

A common mistake is to use the variable name *step* for the Repeat Distance field in the Repeat control. Because *step* is a reserved word in AEL, it does not generate a syntax error, but it is also not identified as a user parameter. One of the reasons it is *important* to check the list of identified parameter names generated by the compile (as well as to check the Design/Parameters dialog box to see what the variables are and what the default values/units are) is that if a parameter is not listed, you may have used an AEL reserved word.

When you try to insert a macro that has been compiled with *step* as the Repeat Distance, the following error message appears:
(couple_art.ael line 164, column 11 in struct2)
operand real value expected
Cannot load component artwork

Because *step* returned a value that was not a number, an error was generated when the program tried to evaluate the expression in which it was used. The best way to deal with an error of this type is to edit the PAM script. Go to the line indicated in the error message. Look there (or a few lines above) for something wrong. In this case the generated code has nothing obviously wrong:

```
decl p_value1X_3 = number;
    decl p_value2X_3 = step;
    decl p_value1Y_3 = 0.0;
    decl p_value2Y_3 = 0.0;
    pam_rep = pam_do_repeat\(
        PAM_COMMON_DATA,        p_416DB340,
        PAM_REPEAT_DIRECTION,   1,
        PAM_REPEAT_MODEX,       3,
        PAM_REPEAT_VALUE1X,     p_value1X_3,
        PAM_REPEAT_VALUE2X,     p_value2X_3,
        PAM_REPEAT_MODEY,       3,
        PAM_REPEAT_VALUE1Y,     p_value1Y_3,
        PAM_REPEAT_VALUE2Y,     p_value2Y_3\);
```

When this type of error occurs, look at each variable/function name and make *sure* it is being identified correctly (*step* is not, in this case) and that, in the case of intrinsic functions, it is being used properly.
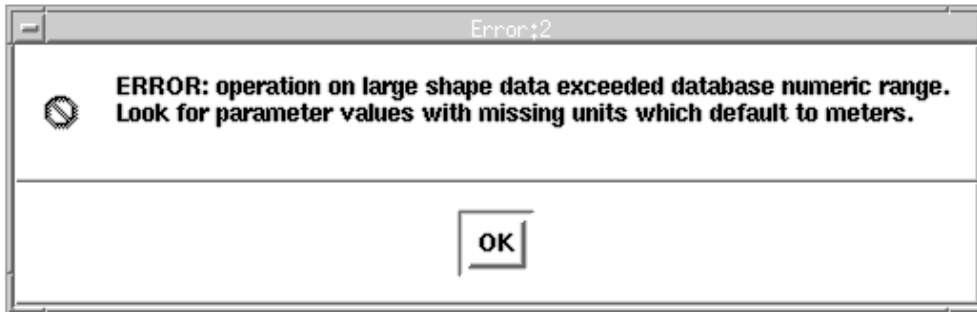
Missing or Incorrect Unit Designators

Another common run-time error is caused by missing or incorrect unit designators in equations. The default MKSA unit for length is meters, but most designers define the Layout window to be in µm or mils. This means that any value without a unit specified is treated as meters.

Example If a Repeat control is defined with a fixed repeat distance of 100 mil, and you forget the mil and just enter 100, the system treats that as a repeat distance of 100 meters. This means that the inserted macro will be *very* big.

The system makes an effort to trap unit problem by checking for shape data values that exceed the database numeric range. A layout will have a specific layout unit (such as mil) and resolution (0.01, for example) as set in the Options/Preferences dialog under the Layout Units tab. These values will define the maximum size of any data value used in the layout, both in terms of absolute size and number of significant digits.

The system monitors the primitive data looking for values that will overflow when converted to database units. If an overflow is identified, the following error message is displayed:

Error:2

⊘ **ERROR: operation on large shape data exceeded database numeric range.
Look for parameter values with missing units which default to meters.**

OK

As the message states, the macro's parameter values as well as any constants used in the control definitions should be checked for missing unit designators. Other things to look for would be various math errors or magnitude errors that results in extremely large numeric values (several orders of magnitude larger than the unit being used).

In addition to the above error checking done by the system, this type of error may have other clues to help you recognize it:

- When you insert the macro, there is no drag image (it is too large to fit in the window).
- When you insert the macro, some or all of the shapes are missing (they are out of view).
- If you choose the command *View > View All* , the view zooms *way* out. Shapes may not be visible (they are too small to see at this zoom distance).
- While inserting the macro you get very unusual error messages, for example: **POLYGON requires at least 3 vertices** . The problem is that due to the large size of the data values significant digits were lost and some of the shape's vertex points have collapsed to a single point. If enough points collapse the polygon no longer has enough *unique* points and an error is generated.

If any of these things happen while you are developing or testing a macro, there is probably a unit mismatch somewhere.

Macro Error Messages

What following section lists the warning and error messages you are likely to encounter when using the Graphical Cell Compiler. The listing provides more detail on each message: what caused the message and what to do to fix it.

The messages are grouped in three sections to parallel the three areas of interaction for macro: definition, compilation, and execution.

Macro Definition Messages

There must be ONE construction line selected as a reference.

Several of the controls (Stretch, Repeat, and Polar) require a construction line to be included as a reference line for the operation. This message indicates that a control definition was completed (the OK button was pressed) but there was more than one construction line selected.

Shapes to be operated on must be selected.

Most of the controls require graphic shapes to be included as the object of the operation (Stretch for example).

This messages indicates that a control definition was completed (the OK button was pressed) but there were no shapes present and selected in the design.

Shapes with width to be operated on must be selected.

The Width control only operates on graphic shapes that support the idea of width. At this time that is a Path. This messages indicates that a control definition was completed (the OK button was pressed) but there were no Paths present and selected in the design.

[field name] Field:

ERROR AELxxxx(column x) [ael error]
This message indicates that there was an AEL syntax error as listed in the specified field for the current dialog. The message will attempt to identify the column number where the error was discovered and list the nature of the error. See the AEL Guide for more information about valid AEL syntax.

[field name] information must be provided.

This message indicates that the named required field must have a value before the control definition is complete.

Macro Compilation Messages

Warning: functions currently undefined:

This warning message indicates that the listed function names were used within the various control expressions, but are not yet defined in the system. This could mean that the function is not a part of AEL or may be mis-spelled. If it's a user-defined function, then it means the file containing the function definition as not yet been loaded into the system.

This warning message has no effect on the compilation of the PAM. If there were no other warning or error messages, the compiled PAM is fine and ready to run. But the listed function must be loaded into the system before the PAM is executed to create the specified artwork. If the function is still not found then a run-time error will be generated.

Identified the model parameters:

This is an informative message only. It lists the undefined variables found in all the control expressions. These variables will be used as the model parameters and will show up in the Design/Parameters dialog.

It's always a good idea to check this list to make sure that all the names you expect show up (you didn't use an AEL reserved word by accident) and that there aren't any extra ones

(perhaps due to a spelling error of an AEL name).

Warning: no primitives found for this control

While processing a control it was discovered that there were no primitive shapes associated with it. This typically happens when a control is defined and then the shapes that were selected for it to operate on were deleted. Perhaps a shape was deleted and re-inserted to fix something but the control was not edited to associate the new shape with the control.

Warning: primitives without width ignored for this control

The Width control can only operate on shapes with width (like a path). If a shape without width is included in the control the above warning is displayed.

Warning: detected primitives with different (x,y) point counts for this control.

In most cases the User-Defined control will be written to work with a given class of shapes, polygons for example. It is much more complicated to write one function that can work with a mixture of shape types, for example polygons (which can have three or more points) and text (which has only one (x,y) point). If while compiling the macro it discovers mixed shape types associated with a User-Defined control, the above warning is displayed.

While this may be in fact intended, one should be very careful about passing different shape types to a give user-defined function.

Warning: ignored unsupported graphical primitive type: [name]

If while processing a control it discovers a primitive shape, component, or connectivity element that the control will not support this message is printed. For example, Wires are not supported by any of the controls.

If this warning is displayed even though the Layout appears to not have any unsupported items present, it is probably because the design was not empty to begin with. Perhaps an old design was used (after deleting everything visible) or perhaps there is or was a Schematic page in the design. It's always best to start with a newly created design to be sure there won't be anything left over from previous work.

ERROR: no construction line found for this control

While processing a control it was discovered that there was no construction line associated with it. This typically happens when a control is defined and then the construction line is deleted and re-inserted to modify it's location or angle. The control will need to be edited and a construction line selected to be associated with it.

Compile complete:

errors =
warnings =
This is simply a summary message which lists the total number of errors and warnings encountered during the compilation of the macro. Any errors mean that the macro is invalid and will not run. Depending on the type of error the compile may have stopped leaving a partially generated file which may not even be valid to AEL. The errors must be corrected before proceeding.

Warning messages do not prevent the macro from being generated, but they do indicate a problem which could result in unexpected results. The warnings should be resolved, especially if the PAM is not working as expected.

Use Design/Parameters to give the model parameters default values.

This is just a reminder message. It is very important that the Design/Parameters dialog be checked for any new PAM as well as any time the number or type of macro parameters change. Since new parameters get a default value of zero it is important to set them to a more reasonable value so that the insertion of a default component (the user didn't modify any of the parameters before insertion) looks correct (no lengths of zero or the like).

Macro Execution Messages

Warning: repeat exceeded limit, operation aborted

Warning: step exceeded limit for polar, operation aborted.

Both the Repeat and Polar controls keep track of how many iterations they perform. If there are to many it is possible that there is something wrong. It could be an infinite loop or just a very large count due to some error in the control specification or parameter value. To prevent any problems the iteration count is checked against a maximum and if it exceeds it the above warnings are displayed.

The default iteration limit is 1,000. This value can be modified using the configuration file option CELL_COMPILER_MAX_COPY. For details, see *Configuration File Options* (gcc).

Warning: infinite step detected for polar, operation aborted

As the Polar control is initializing it checks the angel start, stop, and step values. If <i>start < stop</i> with a negative step value, or if <i>start > stop</i> with a positive step value the control would go into an infinite loop. If these conditions are identified, the above warning is printed.

ERROR: obsolete compiled model version, re-compile model

This message indicates that the PAM was compiled using a previous version of the Graphical Cell Compiler. Due to changes made in the new version the PAM must be re-compiled. The source design does not need to be changed, it just needs to be opened and the Macro/Compile dialog opened to perform a fresh compile.

ERROR: invalid compiled model

The code testing what version of the Graphical Cell Compiler was used to create this model encountered an invalid version number. About the only way this could happen is if the PAM was manually edited and the number changed. In normal usage this message should never occur. If it does, simply re-compile the model and the problem should be corrected.

Control: [name], Shape: [type],

ERROR: operation on large shape data exceeded database numeric range.

Look for parameter values with missing units which default to meters.

As previously discussed, this error is displayed when the calculation for a shape's data points becomes too large. This is typically caused by a missing units designation in a control definition, or an incorrect unit assignment to a macro parameter.

For more information, see Missing or Incorrect Unit Designators.

Control: [name], Shape: [type],

ERROR: initial and current shape data have different number of (x,y) points.
Look for a User-Defined control which added/deleted points.

This error is caused by a User-Defined control which added or removed a number of (x,y) data points from a shape and did a return of type "Shape" which put the data back in the original location. This difference in the array size of the initial shape and current shape structures can cause errors in subsequent controllers.

If the User-Defined control is simply modifying (x,y) points it can return them as type "Shape". But if it is adding or deleting points is must return the resulting shape as type "Replace List" or "Append List" even if it is still a single item.

For more information, see *User-Defined Control.* (gcc)

User-defined shape data for [type]

- not an array.
- is a 1-dimensional array.
- is a >2-dimensional array.
- is not an (x,y) array.
- is not a list.

line [AEL line number] in [AEL file name]
This set of errors are possible results from a series of tests performed on the return data from a User-Defined control. The returned data must be an array of (x,y) values, that is a two-dimensional array.

A well formed array of (x,y) points would look something like:

```
[[1, 2], [3, 4], [5, 6]]
```

The first test is to make sure the return data is an array. If it isn't the "not an array" message is displayed.
Next the data is checked to make sure it is a multi-dimensional array. If it isn't then the "is a 1-dimensional array" message is displayed. A one-dimensional array would look like:

```
[1, 2, 3, 4, 5]
```

It is then checked to make sure it doesn't have more than two dimensions. If it does the "is a >2-dimensional array" message is displayed. A three-dimensional array would look something like:

```
[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
```

The final check is to make sure the second dimension consists of only two elements, that is an (x,y) pair. If it doesn't then the "is not an (x,y) array" message is displayed. A two-dimensional array with two many elements in the second dimension would look something like:

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

An additional test is performed if the User-Defined control is returning a list of primitives to make sure the return value is in fact a list. If it isn't then the "is not a list" message is displayed. If the list is OK, then each element of the list is run through the above tests.

Control: [name], Shape: [type], Parameter: [name]

- ERROR: parameter is NULL
- ERROR: failed to retrieve parameter
- ERROR: array is not of type REAL
- ERROR: array is not a two-dimensional array of (x,y) points (order 2)
- ERROR: array is not a two-dimensional array of (x,y) points (dimension 2)
- ERROR: parameter is not of type LIST
- Warning: extra parameters ignored

As each control starts to execute all the parameters are gathered from AEL. A number of tests are performed to make sure the parameters are of the correct size and type. If any errors are found one of the above messages is printed to help isolate the problem.

In general these messages should never occur. The most likely cause of such an error is through manually editing the PAM and making changes to the Control's parameter list. If such an error were to occur, re-compiling the PAM should resolve it.

Preferred pin with the same number exists

When a PAM has a fixed number of pins, the component pins are assigned the same numbers as assigned to the ports in the source Layout. Since each port must have a unique number the pin numbers will also be unique.

The rules change if a source port is involved in a repeat control. Since all the copied pins must have a unique number, rather than using the port number they are allowed to take on the default numbering (next lowest available integer) as they are inserted. Normally this behavior works fine. A problem develops when an un-copied pin is created after a copied pin.

For example, assume the source design has two ports numbered: 1 and 2. Port 1 is part of a repeat control with a count of 4. So the component is generated with pins 1 through 4. It then generates a pin for port 2, and since it isn't part of a repeat is assigned the port number of 2 which is already in use by the second copy of port 1.

The above error is printed when this new pin (with the default number of 5) is re-assigned a number of 2. There are ways to control the pin creation order to work around the problem of mixing copied and non-copied pins in the same component.

For details, see *Controlling Pin Numbers.* (gcc)

Getting Started with the Graphical Cell Compiler

The Graphical Cell Compiler (GCC) is a tool within Advanced Design System that makes the job of adding parameterized artwork to a layout an easy process. The GCC can benefit Productivity Engineers as well as Circuit Designers:

- For Productivity Engineers, the GCC greatly simplifies developing a library of parts for use by circuit designers. Various parts can be developed and debugged much faster by using the GCC than by developing the AEL macros themselves. Of course, if engineers do know how to program in AEL, they can use AEL to do further customization.
- Circuit Designers can use the GCC to create a special model quickly, without the need to know any AEL.

Parameterized Artwork Macro

A parameterized artwork macro (PAM) is a graphical layout model with specific parameters that define its form. These parameters can control the length or width of some or all of the shapes within the model, the repetition factors for the shapes, and so on can be customized each time you place the model in a layout.

The basic steps for creating and using a PAM are:

- Define the artwork graphically in a Layout window.
- Define parameters to effect the artwork.
- Compile the macro.
- Set the parameter default values and save the macro.
- Insert the new component (macro) in a layout.
- Edit the parameters for that instance of the new component.

For an example that illustrates the basic steps of creating and using a PAM, see *Basic PAM Example* (gcc).

GCC Examples

In addition to the examples provided in this documentation, several Graphical Cell Compiler examples are included in the program's *examples* directory. Examples are improved frequently and new ones are added to the *examples* directory, so the files in your program may differ from what is shown here. However, the basic path is the same.

To view the Examples directory:

1. Select **File > Open > Examples** from the ADS Main window to open the *Choose an Archived Example to open* dialog box.
2. Select **RF Board** > **gcc_examples_wrk.7zap**.

Graphical Cell Compiler Flow Chart

The following figure provides a high-level overview of the way the parts of the Graphical Cell Compiler fit with the Advanced Design System to make a functional tool. The basic use-model is similar to that of a compiled language, such as Basic, Pascal or C. The source code is managed in the context of the Source Layout and is compiled into an executable that is stored in the Component Library. You execute (insert) the component in the context of a Destination Layout.

The two parts to the source for a PAM are graphical shapes and controls that operate on the graphical shapes. Graphical shapes are created in a Layout window using the Graphical Editor to insert, move, stretch, and otherwise manipulate the shapes. The result is a set of graphical data that represents the defined shapes. Then controls are defined from within the Layout window using the Macro menu commands (see Macro Menu). These controls are used to create a set of control data. All the graphical and control data is stored as part of the file when you save the design.

When you are satisfied with the source information, you compile it using the Macro Compiler (also known as the Graphical Cell Compiler). The output is a Parameterized Artwork Macro (an AEL Macro) that is saved on the disk.

Next, you define the parameter unit types, default values, and links (to simulation or schematic data), that complete the component definition. The component can then reside in a component library, ready for use.

When you insert a component created with the Graphical Cell Compiler in a schematic, the component appears the same as any other component in the program. You can set the component parameters using the standard component edit parameters dialog. What is different is that behind the scenes there is an Insertion Engine that processes the various control requests (such as stretch and repeat) to generate the requested graphical shapes. When finished, the generated component appears in the Destination Layout window.

GCC Flow Chart

Macro Menu

The Macro menu is accessed from the menu bar of a Layout window.

Stretch opens a dialog box for defining the controls that adjust how shapes are moved or stretched. See *Stretch Control.* (gcc)

Rotate/Move/Mirror opens a dialog box for defining the controls rotate, move, and mirror shapes. See *Rotate-Move-Mirror Control* (gcc).

Repeat opens a dialog box for defining the controls that set the number of copies of the graphical shapes that are inserted, and where the copies are placed. See *Repeat Control.* (gcc)

Polar opens a dialog box for defining the controls that handle operations in the polar (angle, radius) coordinate system. See *Polar Control.* (gcc)

Width opens a dialog box for defining the controls that handle width for the graphical shapes that support width. See *Width Control.* (gcc)

User-Defined opens a dialog box for specifying a user-created AEL function to be used to modify or create graphical shapes. See *User-Defined Control.* (gcc)

View/Edit opens the controls set for a given PAM, and enables you to reorder, modify, or delete those controls. See the section *Viewing Controls.* (gcc)

Compile opens a dialog box for compiling the PAM and setting defaults and units, as well as other model parameters. See the section *Compiling a Macro.* (gcc)

Defining Artwork

When you use the Graphical Cell Compiler (GCC), you start by creating the artwork by placing simple graphical shapes in a Layout window that the program uses to build an AEL macro. This section provides the basics on creating artwork, by placing components and working with the supported shapes. For details, refer to *Schematic Capture and Layout* (usrguide).

Supported Shapes

The supported shapes are: circle, polyline, path, rectangle (as a polygon), arc (as a polyline), and text. Although these are not shapes, *Connectors* are supported, so you can create components that have pins.

The shapes not supported are: construction line (except as a control line), trace,

component, and wire. While construction lines are needed to define reference points for various controls, these are *not* included in the compiled output and are not displayed when the macro is used. Traces are *not* supported, since a trace is a path that has been turned into a component.

ℹ **Note**
It is *not* possible to include a component within a Parameterized Artwork Macro.

Placing a Port or Shape

The menu and button locations for placing a port or shape are:
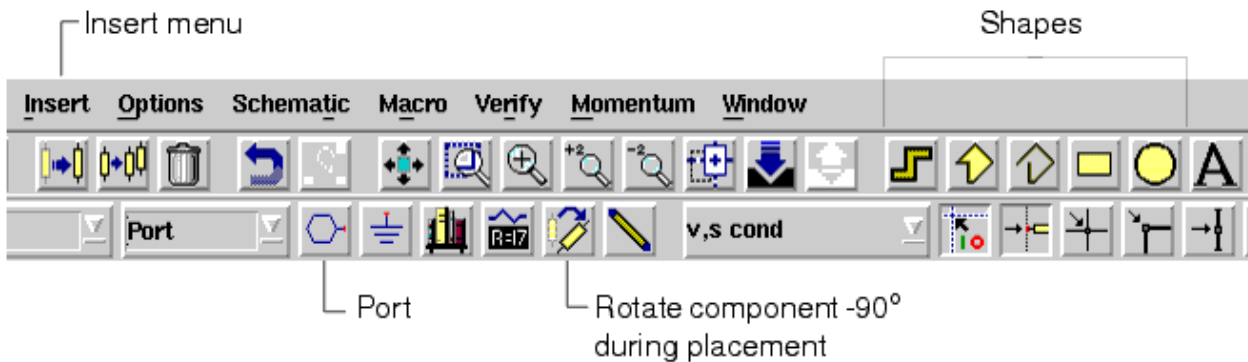
To place a port:

- Select **Port** from the Insert menu or click the Port button on the toolbar.
- Click in the Layout window to place the component.
- Define edge and area ports. See *Schematic Capture and Layout* (usrguide) for more information.

✅ **Hint**
You can save time and mouse-clicks by rotating ports during placement, so that each is properly oriented. If you find that a connector is not properly oriented, click Rotate before you place the port. The connector rotates −45° each time you click the button. When the port is oriented properly, drag it into position in the drawing area and click to place it.

To place a shape:

- Select the shape name from the Insert menu or click the shape on the toolbar.
- Following the instructions that appear in the status bar at the bottom of the window, place the shape in the Layout window.

** Stretching a circle modifies its radius, making it larger or smaller. If you want to make an oval, convert the circle to a polygon *before* running the compiler (**Edit** > **Modify** > **Convert to Polygon**).

- Paths are supported only to the degree that they are like a polyline. Other path attributes (corner type, and so on) are not under parameterized control and are

the same as that of the source shape.

- If you rotate a rectangle, or move any of its corners so that it is no longer a rectangle, the program converts it to a polygon. Therefore, the resulting AEL scripts treat rectangles as four-point polygons.
- Similarly, arcs are converted into a polyline.

Defining the Size of a Shape

When you place a shape, you can use the X,Y coordinates displayed in the status bar at the bottom of the Layout window to help you draw the shape to the size you want (View > Coordinate Readout). The two types of coordinates are: positional and differential.

-420.0000, -150.0000   -150.0000, -230.0000

Positional   Differential
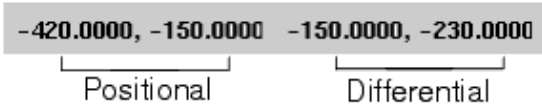
- Positional coordinates display the X,Y coordinates of the cursor position in relation to the total window. By default, the large + in the center of the drawing area is 0,0.
- Differential coordinates display the distance in X,Y that the cursor has traveled since the last click. Set the starting point to 0,0 by clicking anywhere in the drawing area.

For example:

1. Select the rectangle on the toolbar.
2. Click in the Layout window to define the first point on the rectangle. The Differential X,Y coordinate display reads *0.00, 0.00*.
3. Move the cursor until the Differential X,Y coordinate display reads *200.0, 100.0*.
4. Click. A rectangle 200 by 100 mil is placed in the window.

Placing a Construction Line

To place a construction line in a Layout window, choose **Insert** > **Construction Line**, then click twice in the drawing area to provide two points along the line.

Editing Components

You can edit components in a Layout window (either before or after selecting the component you want to edit) by using an Edit menu command or by clicking the appropriate button on the toolbar.

Experiment with these commands and buttons until you feel comfortable with them.

Experimenting with Parameter Values

To better understand how you can use your new component, try these experiments.

1. Try to select one of the rectangles in the component you have just inserted. See that you used one rectangle (in the original artwork) to create a component that contains two rectangles.
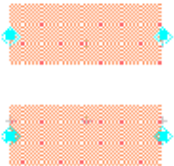
2. Open the Component Parameters dialog box (**Edit** > **Component** > **Edit Component Parameters**) and reset the parameters to these new values:
   length = 100 mil
   width = 25 mil
   number = 3
   space = 25 mil
   Click Apply to see how the component changes.
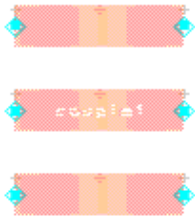
3. Reset the parameters to these new values:
   length = 100 mil
   width = 15 mil
   number = 5
   space = 10 mil
   Click Apply to see how the component changes.
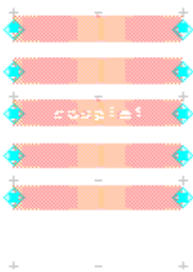
4. Try other settings to see how different settings change the component.

Glossary for Graphical Cell Compiler

Term	Description
Artwork Macro	An Application Extension Language (AEL) script that creates graphical shapes in Layout.
Artwork Instance	A specific occurrence of an artwork macro placed in a layout.
Control	A specific attribute used to modify artwork as it is placed in a layout. A given control (for example: Stretch) may have a number of parameters that define its behavior (for example: orientation, direction, length).
Fixed Artwork	Layout graphics that are fixed in size and do not change from instance to instance.
Graphical Cell	*See* Parameterized Artwork Macro (PAM).
Graphical Cell Compiler	This product. It takes a layout with defined shapes plus controls that operate on those shapes and generates a Parameterized Artwork Macro that can be used to insert Parameterized Artwork instances in a design.
Parameter	A specific aspect of a control that defines its behavior. For example, orientation, direction, and length are all parameters of the stretch control.
Parameterized Artwork	Layout graphics that may change in size or number depending on the parameters provided at the time of insertion of the instance into a layout.
Parameterized Artwork Macro (PAM)	Parameterized Artwork implemented as an AEL macro.
Scalable Artwork	*See* Parameterized Artwork.
Shape	A basic layout primitive such as a circle, rectangle, polygon, and so on.

Hints and Tricks

This section provides a number of suggestions that can help when using the Graphical Cell Compiler to create PAMs. As with any new tool, as you become familiar with it you discover ways to do things that make life easier. The purpose of this section is to save you time by sharing a few of these tricks before you spend the time to discover them on your own.

Selection and Control Definition

Since most all the controls require shapes to be selected as part of their definition and many also require a control line construction line) being able to select the proper shapes for a given control is important. There are times when this can be difficult. Take for example the case of defining a multi-layer stack of shapes that all start the same size but must be stretched in different ways depending on the layer and user-supplied parameter values. You could find yourself with the Stretch dialog open and unable to accurately select the shape or shapes from the stack that must go with that stretch.

Traditionally, one would use the Option/Layers dialog to turn off and on various layers to make it possible to select the required shapes. But this process is rather tedious and slow. In this case there is an easier way.

At this point it is important to remember just when various actions take place in the define, compile, execute sequence. For a given control, the phases perform the following functions:

- Define phase - shapes to be operated on are picked by selection and associated with a given control.
- Compile phase - shape data (the actual (x,y) points) are collected.
- Execution phase - shape data is tested (for example to see if it is intersected by a control line) and modified by the control.

Given the above we can see that the location of the shapes does not matter during the definition of the control.
This means that the stack of shapes can be spread around the screen so they don't lie on top of one another. This makes the selection of particular shapes easy to do. Once all the controls for that stack of shapes are define, the shapes can be moved or undo can be used to put them back in their original location.

This technique can be used to solve a number of different selection problems during the definition of controls. As long as everything is in it's proper location before the compile step, moving shapes around will have no negative effects.

Controlling Pin Numbers

In general, the component pins inserted by a PAM will be created in the same order that the ports were found in the source layout. The search moves through the layers from 1 to

N, and for each layer picks up the ports in order which will be from the first ports inserted up through the last ports inserted or modified/edited. That is, for a given layer, if the first port inserted is edited, it will be re-created at the end of the list and will now be the last port found on that layer.

Most of the time this order doesn't matter since the port numbers are saved in the PAM. This means that as the pins are inserted, the default pin numbering (next lowest available integer) will be modified with the saved port number. The one time where this may matter involves cases where a port is part of a repeat control. See *Preferred pin with the same number exists* (gcc) in *GCC Error Messages* (gcc).

As discussed, if an un-copied pin is created after a copied pin there can be a pin number conflict. The easiest way to resolve this conflict is to change the port order in the source layout so that all the fixed (un-copied) pins will be inserted and numbered before any of the copied pins.

In order to control the pin creation order in the final component, you must insert the ports in the source layout in their desired order. If for any reason a port is edited, it will then be out of order. The easiest way to correct the order is to use the Edit Component Parameters (ports are treated as components) and select each port and then Apply. Going through the ports in the desired order will position them in the database in that order so the resulting PAM will insert them in the desired order.

By controlling this order you can make sure that all the fixed pins are inserted and numbered (with port numbers) before the copied pins. The copied pins will then be given the next available default number skipping over any numbers already is use by an un-copied pin.

Testing for Minimum and/or Maximum Values

It is often the case that a given physical model is only valid for a certain range of parameter values. While these constraints are often checked by the simulation model it is sometimes desirable to have the physical model do some checking also. This may be desired if an out-of-range parameter value (a zero for example) can cause an error in the macro or cause the macro to generate invalid graphics.

While there is no direct support for parameter range checking in ADS, there is a technique that can be used to at least constrain parameters to within certain ranges. This makes use of the ?: syntax supported by AEL.

Let's say for example you have a Repeat control that is using the parameter count for the repeat count field. So the dialog field would simply contain:
```
count
```
And let's say that count should never be less than one. If the user enters zero or a negative number, use one. This could be forced with:
```
count < 1 ? 1 : count
```
Which is sort of equivalent to a small function that does:

```
if (count < 1)
```

```
        return(1)
else
        return(count);
```

If you further wanted to constrain count to be less than ten, you would add a second
nested test:

```
count < 1 ? 1 : ( count > 9 ? 9 : count )
```

Which would be somewhat like:

```
if (count < 1)
        return(1)
else
        if (count > 9)
                        return(9)
        else
                    return(count);
```

Now this is clearly not a general-purpose solution. For example, if the parameter count is
used three different times then the range testing will need to be in place for each
occurrence. But in cases where some range testing is very important (to prevent divide by
zero errors for example) this method will work well.

Advanced Value Testing and Error Reporting

The above example is for simple bounds checking. It does not allow for more complex
testing of parameter values nor does it provide for any sort of error/warning reporting to
the user. By using the User-Defined control it is possible to do more extensive testing of
the component parameters as well as provide messages to the user.

We start with the basic coupled-line model used elsewhere in this documentation. To the
existing two Stretch and one Repeat controls is has, we will add a User-Defined control. It
will be set to Replace List (and end with a return (NULL) statement so that it's a no-op)
and be passed all four of the component parameters: length, width, number and space.
The function called will be named: *check_couple_values()* . Then using the View/Edit
dialog and Cut/Paste the new control is moved to be the first control executed by the PAM.
This last step is very important because if it's left at the end of the list of controls, the
NULL return will remove the copies made by the repeat control.

There are two ways to pass the component parameters to the test function
check_couple_values() . The first it by-value which is the way parameters are typically
passed. If passed by-value the function can use the parameter values, but it can not
modify them back in the calling code. The other way (used here) is by-reference which
means that a pointer to the value is passed to the function rather than the value itself.
The function can still access the value, but if it modifies the value it will be changed back
in the calling procedure also. By passing the parameters by-reference, the testing function
can check AND correct the parameter values before the rest of the PAM executes.

To pass a parameter by-reference, you use the C language syntax of:

```
&name
```

To access the value of a parameter passed by-reference, you use the C language syntax
of:

```
*name
```

So the final control list for the coupled line model will be (the User-Defined control being the only new control):

Control:	User-Defined
	Returns: List Replace
	Function: check_couple_values(&length, &width, &number, &space);
Control:	Stretch
	Direction: Both
	Length: length
	Offset: 100.0 mil
Control:	Stretch
	Direction: Both
	Length: width
	Offset: 50.0 mil
Control:	Repeat
	Direction: Parallel
	Parallel
	Number: number
	Distance: width+space

And the function called by the User-Defined control might look something like:

```
defun check_couple_values(lengthP, widthP, numberP, spaceP)
{
if (*lengthP < 100 mil) {
            de_error_dialog("length must be >= 100 mil, reset",
            NULL, TRUE); *lengthP = 100 mil;
}
if (*lengthP > 500 mil) {
            de_error_dialog("length must be <= 500 mil, reset",
            NULL, TRUE); *lengthP = 500 mil;
            }
// Same type of test for width, and space
if (*numberP > 5) {
            de_error_dialog("number must be <= 5, reset", NULL, TRUE);
            *numberP = 5;
            }
if (*lengthP < *widthP) {
            de_warning_dialog("Warning, length < width", NULL, TRUE);
            }
return(NULL);
}
```

You'll notice that all the parameter names have a *P* appended to their name. This acts as a reminder in the code that these have been passed by-reference and must use the *name* syntax to access the value. A test for a minimum and maximum value for the length parameter has been included. In each case, if the value is out-of-range, a message is displayed to the user and the value is reset to be in range. Notice the use of units (mil) in the expressions. Similar tests could be added for the width and space parameters. A test has been added for number to show what a unitless value would look like.

Finally, a test has been added that goes beyond simple range checking. In this case it's

checking to make sure that the length of the line is always greater than it's width. If the test fails, a warning (not an error) message is printed and the values are not changed. While this may not be a realistic constraint, it serves to demonstrate the types of error checking that can be performed in a User-Defined control.

When using this method for range checking, be aware of the limitations. First, it only works on components built with the GCC. Second, the testing is only performed during Layout component creation, which means that no testing would be done on the Schematic equivalent for this component. Finally, great care must be taken when creating the tests to make sure they are testing the right value (by-value, by-reference) in the right way (with or without units) to get the desired results.

Printing Shape Data

Sometimes when trying to debug a macro there is just no substitute for printing out the intermediate values for shape data. While it is possible to edit the PAM and add print statements in the code, it must be done manually and if you update and re-compile the macro all the added print statements will be lost.

But there is a creative way to use the User-Defined control to print shape data without actually making any changes to the data.

As discussed in the section on the User-Defined control the local variable _shape_data contains the current (x,y) data for the shape being operated on. If the Function returns field is set to Shape, and the x,y points field is simply set to _shape_data it has the effect of assigning the current shape data to itself, that is, a null-operation. No change is made to the data.

This gives us a hook into the AEL code where we can attach an additional AEL statement to print out some values. We will use the *identify_value()* function as defined in the AEL Guide. By using this with the *fputs()* function we can cause the values to be printed out to the terminal window where all the ADS startup messages are printed.

If we want to print out the current (x,y) shape data, then set the User-Defined control as:

```
_shape_data; fputs(stderr, identify_value(_shape_data));
```

You could even give it a label by doing:

```
_shape_data; fputs(stderr, strcat("after stretch: ",
            identify_value(_shape_data)));
```

If multiple shapes are selected for this control, then each will be operated on and therefore each will have it's current data printed.

And while you would continue to use _shape_data as the function value, you could print the contents of any of the local variables defined in the User-Defined control, for example: _cline_data or _shape_list.

In some cases you may want to print out a number of values, but you don't want to

clutter the output with labels.

An example might be to print out some of the variables for each step of a polar control. Similar to the example above, you'd have an expression for the radius length, followed by a semicolon, followed by something like this:

```
fputs (stderr, fmt_tokens(list(_angle_i, _angle, _radius)))
```

This would result in one line being printed per step, each line containing the step number, the computed angle, and the computed radius. This would be repeated for each shape operated on by this control.

Define Parameter Order

As previously noted, component parameters are collected from the expression strings in the various controls as they are processed. You have little or no control over the final order as they will appear in the component insertion dialog. But what if it's really important for component usability or understanding for the parameters to be in a specific order?

As we saw in the previous hint (Printing Shape Data) it's possible to use a User-Defined Control that performs no function as a hook to do other work. So if we make a User-Defined control the first control in the list, and specify all the parameters in that control, it will be the first control processed and we will have essentially defined the parameter order. All we need is a way to use all the parameters in an expression in such a way that it doesn't change any values or have any other side-effects.

One was to do this is to use the AEL list() function to make a list of the parameters that we then never use for anything else. Looking at the Multi-Coupled Line example used elsewhere in this documentation, we might define the first control as a User-Defined control with the following function:

```
_shape_data; list(length, width, number, space)
```

By returning _shape_data for whatever shape we happened to select for this control, it performs no modifications. The list() function contains all the parameters in the desired order to be found during the compile step. But since we don't assign the returned list to anything it is simply discarded. Of course any parameters not listed in this way will be found as they appear by the compiler and added to the list.

The results of this technique is effective control over the component parameter order.

Center of Rotation

All shapes rotate about the origin (0,0). If the shape is setting ON the origin it will sort of spin in place. If the shape is some distance away from the origin the rotation will move it about the origin (it's location AND orientation will change).

So let's say there are a number of shapes (Ports for example) which must all be rotated to some angle. If they are in their final locations when the rotation is applied they will rotate about the origin and be moved away from their desired location.

One trick to deal with this problem is to place all the shapes needing to be rotated on the

origin in the source design. Then you can define Rotate/Move/Mirror controls to act on each shape. Since the actions are performed in the order they appear in the dialog, the rotation is done first (about the origin) followed by the move which can place them in their final location.

Of course if the shapes are also part of a stretch or repeat then having them all start at the origin may make it impossible to place them in the current location with respect to a control's construction line. For this case one might use a more traditional translate, rotate, restore method.

You start with the shape at some known location in the source layout. You then use a Rotate/Move/Mirror control to translate it to the origin. You can then use a second Rotate/Move/Mirror to rotate it and restore it back to it's original location. It is then properly rotated and in position for any subsequent controls to operate on.

The key point to remember is that in developing a complex PAM you are free to start the shapes in any location and/or move them in any that positions them to allow you to perform the operations you need. Don't be limited by methods that would place a shape in it's final location and not move it during the execution of the PAM.

 Deletion of shapes. There may be times when the inclusion of a particular shape should be optional. In other words, under specific condition you may wish to delete a shape or shapes so that they do not appear in the final layout. It's possible to do this using the User-Defined control.

As discussed in *User-Defined Control* (gcc), it is valid to have a User-Defined function return NULL. This has the effect of deleting that particular shape. This means you could define a conditional expression that would either delete the shape or have no effect on it, for example:

```
(delete == TRUE) ? NULL : _shape_data
```

This same method could be used to delete the list of generated shapes (from a Repeat) by setting the return type to *Replace List* and doing:

```
(delete == TRUE) ? NULL : _shape_list
```

Be aware that if you delete the list, then the macro will act as if it was never there and will generate a shape from the current shape data array. So you may need to clear them both, or, you may want to delete the current shape data before the repeat so that it doesn't make the copies in the first place.

Disable Controls

There are times when one is debugging a PAM that it would be real nice to be able to turn off a control. You don't want to actually delete it (it's to much trouble to add it in again), you just want to disable it so you can eliminate its effects.

While there is no built-in way to disable a control, there is a method you can use to modify a control's action to cause it to do nothing. Most controls have a key parameter

that if set to zero will cause the control to have no effect on its shapes. For example setting the stretch distance to zero will result in no stretch taking place.

But you don't want to simply change the control values to zero because you will then have to re-enter the correct equation to turn them on again. What you can do is preface the expression with a conditional expression that will result in the same thing. For example, if the Stretch control has a Distance equation of:

```
length
```

Then inserting the following in front of the expression will have the effect of turning it off:
```
TRUE ? 0.0 : length
```

Which basically says: If TRUE then return(0.0). Since TRUE is always true, then it will always use the value 0.0 for the Stretch Distance. The control has been disabled until you remove the conditional again (or change the TRUE to FALSE to enable it and leave it easy to disable in the future).

Here are some examples of how to disable some of the controls:

- Stretch: set Stretch Distance to: TRUE ? 0.0 : [expression]
- Repeat: set Number of items to: TRUE ? 0 : [expression]
- Width: the same thing can be done with the Width control IF the Width Change is Relative. A 0.0 would result in no change.
- Rotate/Move/Mirror: the same method will work here also although you will need to disable each of the three functions that is being used. If a given control is setup to do both a rotate and a move and you only disable the rotate it will still do the move.
- Polar: this one is a little more complicated. You need to set the Angle Start and Stop to the same value so that it doesn't make any iterations. For example, if Start was 0.0 and Stop was 2*PI, you might set the Stop to:
  ```
  TRUE ? 0.0 : 2*PI
  ```
- User-Defined: as previously discussed (in the section about deletion of shapes), having the function return the shape data unmodified will have the effect of doing nothing. So to disable a User-Defined control one could set the function as:
  ```
  TRUE ? _shape_data : [ function call ]
  ```

Multi-Model Drivers

> **ℹ Note**
> This section is for very advanced library builders who are familiar with AEL and the procedures for building libraries of models for use by designers.

While is should be possible to create a PAM that will support any physical model you can think of, sometimes the effort to design it is just to great and instead you create two or more simpler models. A multi-fingered fet might be an example of this. Due to the physical properties it may be just too hard to make one model that can handle the one finger case and the 2-n finger case. So, you build two models: the one finger model and the 2-n finger model.

The problem is that you would like to have just a single fet model in the library, not the two special-case models. What you would like is a single parameterized point-of-control

that would call the correct special-case model as needed.

Again, this assumes you already understand how to create library elements for use in ADS. This section is not going to provide the details on the use of the *create_item()* , *create_parm()* , and function calls that are used to do this. We will assume you have defined the fet model for the library, and you have specified that the library element will call an artwork procedure named *fet_artwork()* with two parameters: number of fingers, and length (of the fingers).

You now need to create the fet_artwork) procedure such that it can take the provided parameters and call the correct special-case model. Such a procedure might look something like this:

```
defun MWTD_FET_ARTWORK(finger, length)
{
if (is_string(finger))
            finger = evaluate(finger));
if (!is_integer(finger))
            finger = int(finger);
if ((finger < 1) || finger > MAX_FINGER) {
            /* print an out-of-range error message */
            return;
            }
if (finger == 1)
            pam_finger1(length);
else
            pam_fingerN(finger, length);
```

Parameterized Artwork Macro (PAM) Controls

Contents

- *Defining Controls* (gcc)
- *Stretch Control* (gcc)
- *Rotate-Move-Mirror Control* (gcc)
- *Repeat Control* (gcc)
- *Polar Control* (gcc)
- *Width Control* (gcc)
- *User-Defined Control* (gcc)
- *Viewing Controls* (gcc)
- *Temporary Control Variables* (gcc)

Defining Controls

After defining the artwork, you use the Graphical Cell Compiler to define the controls that are available on the finished Parameterized Artwork Macro (PAM).

The basic steps required to define a control are:

- Place and select a control (construction) line to act as the reference for the control.
- Select the shape(s) that the control applies to.
- Select the control (stretch, and so on).
- Define the control parameters.

This section provides details for control precedence, theory of operation, control lines, and the difference between control and component parameters.

Control Precedence

The order in which you define controls is important.

- You *must* define Stretch controls first, since Stretch controls operate on the *original* shape, *not* on copies of the shape.
- You can perform multiple operations on a given shape (such as changing both length and width), and then after the shape is correct, use Repeat and Polar controls to make copies of it.
- Rotate/Move/Mirror controls can operate on the original shape, *and* on copies; you can rotate/move/mirror the original shape, and you can rotate/move/mirror the results of copy operations.

Not all controls are required, but the suggested order of implementation is shown in the following figure.

After you place the shape(s) and control line(s), use Width, Stretch, and Rotate/Move/Mirror controls to manipulate the original graphic(s).

Use the User-Defined control to perform any custom shape manipulations on the original graphic(s).

When the original shape looks exactly as you want, use Repeat and Polar controls to make the desired copies.

Use a Rotate/Move/Mirror control to position the entire group.

Order of Control Precedence

Theory of Operation

To understand *why* controls act the way they do you must understand the different data variables allocated to each shape in a generated macro. Each shape has (at least) these variables assigned to it:

`i_` < *number* >Initial Variable = shape's original data
`p_` < *number* >Primitive Variable = shape's modified data
`l_` < *number* >List Variable = list of generated shapes' data
`r_` < *number* >Rotation Variable = shape's rotation value
`w_` < *number* >Width Variable = shape's width or height
where:
< *number* > is a value assigned at compile time to create a unique name.

When the macro executes, the Initial and Primitive variables are assigned the shape's initial location data. For details, see the section *Code Walkthrough.* (gcc)

- **Initial Variable**   The Initial variable is never modified. It provides a reference so that controls can determine whether the original shape was cut by a control line and if so, where. This means that a shape's response to a control acting on it does not change when a previous control moves or rotates it on/off a control line. For example, if you define a Stretch control to modify the length of a shape, you can be sure that it stretches as defined even if a previous Mirror control moves it away from the stretch control line.
- **Primitive Variable**   The Primitive variable is where all modifications to the shape

take place. When a shape is resized, rotated, mirrored, moved, and so on, it is the points in the Primitive variable that get modified.

- **List Variable** The List variable contains the results of any copy operations as defined by a Repeat or Polar control. In other words, if a control makes copies of the shape defined in the Primitive variable, the original *and* the copies are placed in the List variable.
- **Rotation Variable** This variable contains the shape's rotation value for shapes like ports and text.
- **Width Variable** This variable contains the shape's width (for paths) or height (for text) and is only modified by the Width control.

What is Output

When it comes time to output the results of the macro, a test is performed on the List variable. If the List variable contains a list of shapes, the list is output to generate the shapes in the destination layout. If the List variable is empty, only the Primitive variable is output. The macro *never* outputs the contents of both variables. This explains why a Stretch performed on a shape after a Repeat does not appear to have any effect. The Stretch operates on the Primitive variable, but at output time, the program uses the List variable, which contains the copies created by the Repeat *and* a copy of the original shape. Any modifications made on the Primitive variable by the Stretch *are lost*.

Multiple Repeats

When you perform more than one repeat on the same shape, the Repeat controls use the Primitive variable as the source for the shape data, placing the results in the List variable. If the initial shape does not move between Repeats, *multiple* copies of the shape are made at the starting location. That is, if you perform a Repeat Parallel followed by a Repeat Perpendicular, *two* copies of the shape are made at the starting location: one generated from the Repeat Perpendicular and one generated from the Repeat Parallel.

Modifying Data Between Repeats A shape's data (in the Primitive variable) can be modified between Repeats. Repeats use the same source to copy, so if that source is changed (by a Stretch or Mirror control, for example) after one repeat, the next repeat makes copies of a different-looking shape.

Modifying the Contents of Primitive Variable & List Variable Data

The Rotate/Move/Mirror control operates on *both* the Primitive variable *and* the List variable. This means that you can modify a shape either before or after you copy it.

Rotating a shape and then repeating the shape:
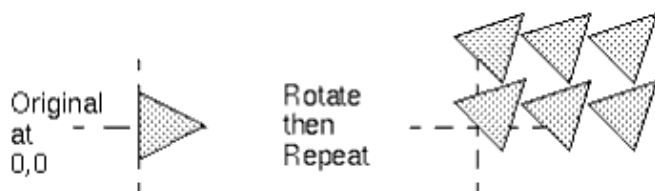
Repeat a shape and then rotate the results of the repetition:
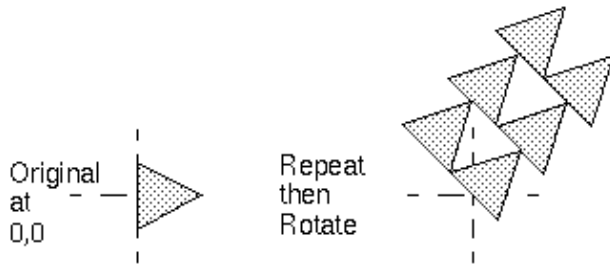
As the example shows, if you modify (rotate) the shape before you copy it, the Repeat control makes copies of the rotated shape. If you copy the shape before you rotate it, the Rotate control operates on the list of shapes as a unit. The Rotate control also modifies the Primitive variable, but because the List variable contains a list, the modification to the Primitive variable is not visible unless you do another Repeat on the same shape. In that case, you do see the modification in the results of the second Repeat.

Using Control Lines

Control lines (construction lines) provide the reference for Stretch, Repeat, and Polar controls. When you define one of these controls, you must select both the shape that you want to manipulate/copy and the control line that you want to use as a reference. If you forget to select a control line, the program prompts you to do so.
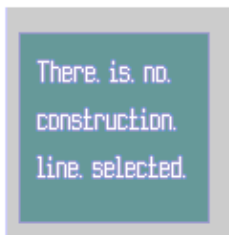
With a Stretch control, you choose whether you want the stretch to be in the positive or negative direction with respect to the reference line.

With a Repeat control, you choose whether you want the repeat to be parallel or perpendicular to the reference line.

With a Polar control, a control line defines where a polyline or path is cut for a stretch-like operation. It does not have any effect on the rotation angle (0° is still the x-axis) or the rotation origin (which is 0,0).

A control line does not have to intersect a shape.

For example, if a shape is to one side of the control line *and* is included in a stretch, the entire shape moves by the stretch amount, rather than being stretched (see *Stretch Control* (gcc)).

 A control line can be at *any* angle; a control line is not limited to 90° angles.

Positive and Negative Stretch Directions

When you select a control line, you use the Stretch Control Definition dialog box to indicate the direction (positive or negative) in which you expect the stretch to occur *relative to the selected line*.

- For a horizontal control line, positive = up
- For a vertical control line, positive = to the right

The following figure illustrates stretch direction with respect to the horizontal or vertical control lines.
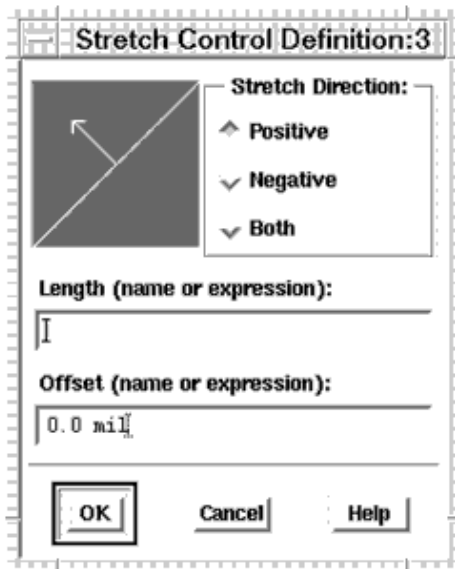
Control line at 45°
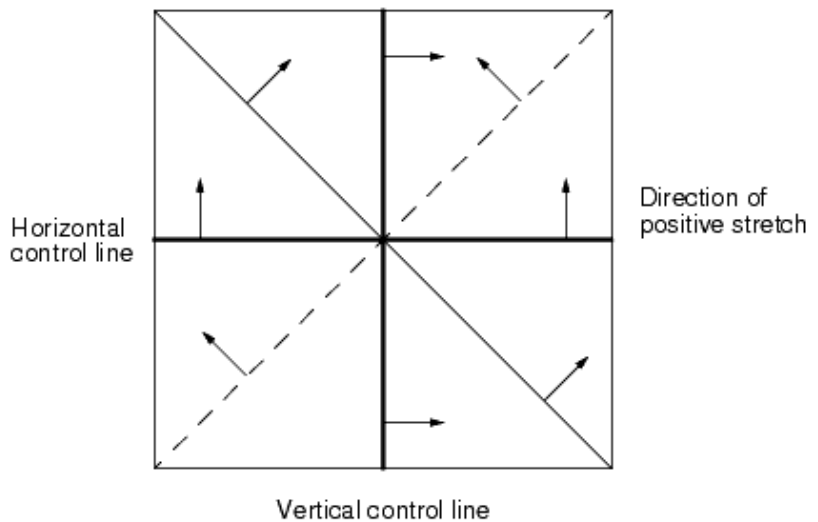
Determining Stretch Direction

The point at which the program sees a line as moving from horizontal (positive = up) to vertical (positive = right) is when the angle of the line is *greater* than 45°.

The stretch direction for a line between 45° and 90° can appear to be backward, as shown below. Just remember that the line is at an angle greater than 45°, even though it may be hard to see the slight difference.
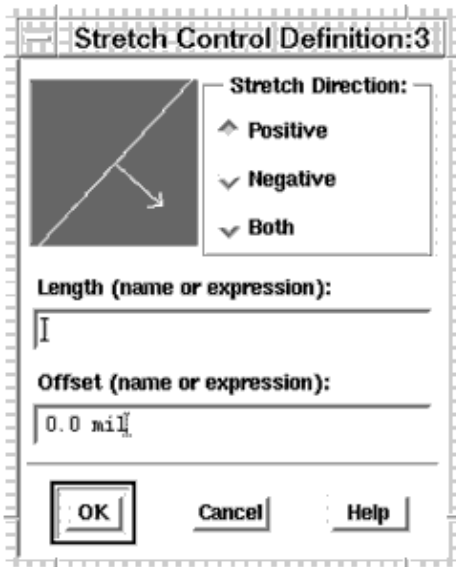
Control line slightly greater than 45°

Parallel and Perpendicular Repeat Directions

When you select a control line, you use the Repeat Control Definition dialog box to indicate the direction (parallel or perpendicular) in which you can expect the copy to occur *relative to the selected line*.

- Parallel = parallel to the control line
- Perpendicular = perpendicular to the control line
- Both = an array-like grid

As with a Stretch control (see Positive and Negative Stretch Directions):

- For a horizontal control line, positive = up
- For a vertical control line, positive = to the right
- And vertical is > 45°

Entering a positive or negative number determines the direction in which the copy is placed along the selected axis.
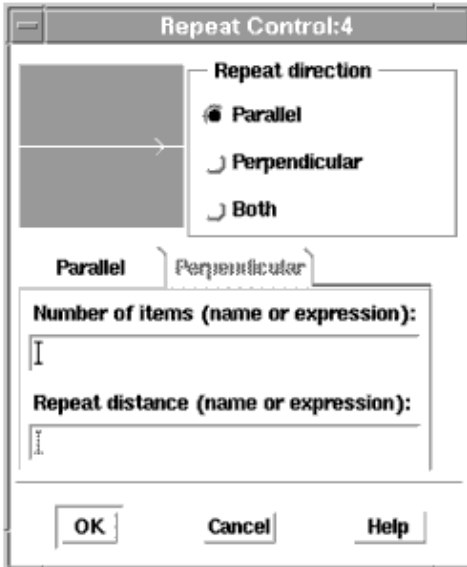
If the control line is horizontal: Parallel goes left/right; Perpendicular goes up/down.
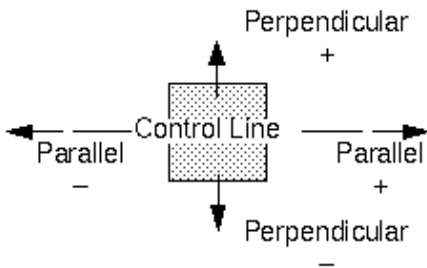
If the control line is vertical: Parallel goes up/down; Perpendicular goes right/left.
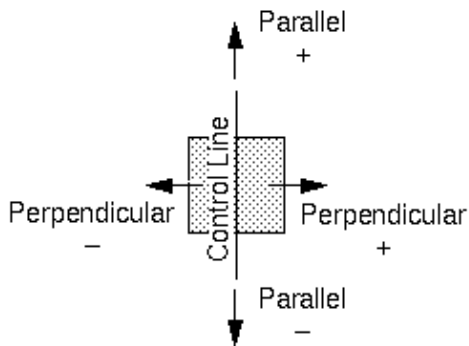
When the control line is at an angle: Parallel is still parallel to the control line and Perpendicular is still perpendicular to it.
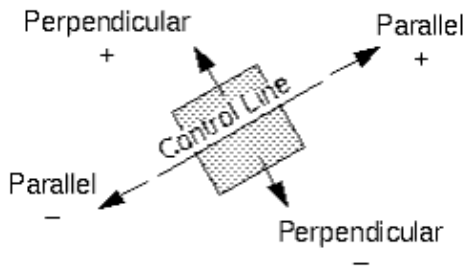
Repeating Both combines a parallel and a perpendicular copy, even at an angle.
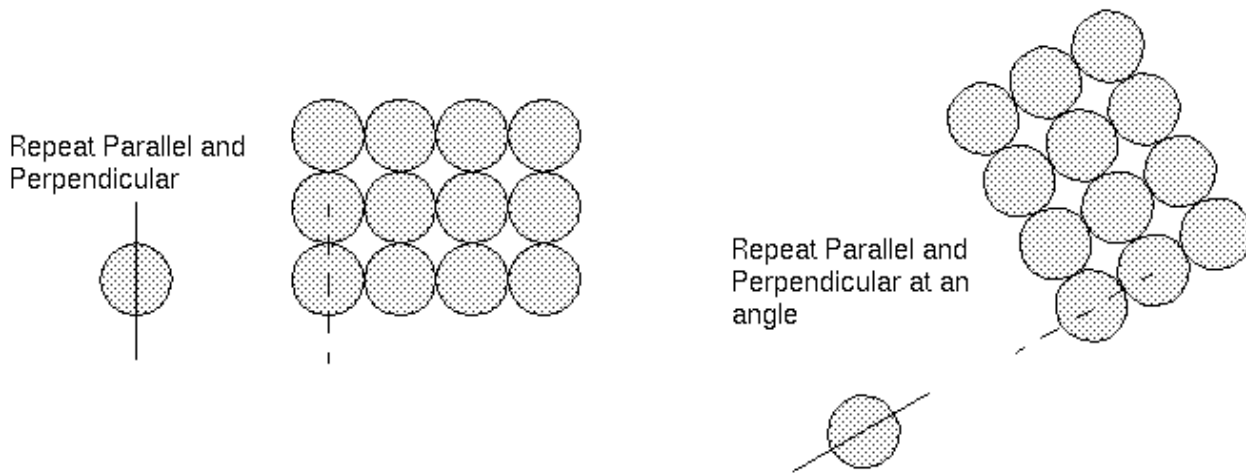
Defining Parameters

When using the Graphical Cell Compiler, you work with Control Parameters and Component (model) Parameters.

Control Parameters

Control parameters are the fields within a control dialog box. For example, in the Stretch Control Definition dialog box, the control parameters are Distance and Offset.
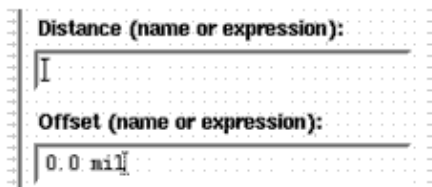
You can use any one of the parameters listed in a control parameter field:

- Constant
  Example: 100
  Comments: This value can not be edited when you use the macro.
- Variable

Example: length
Comments: A variable appears as a component parameter; its actual value is supplied by the user.

- Expression
  Example: size*100
  Comments: Any undefined values appear as component parameters (size, in this example). The actual value is supplied by the user. Expressions enable you to associate component parameters.
  Example: sin(angle)
  Can include AEL function calls.

- AEL code fragment using the ?: syntax from C (also available in AEL)
  Example: size < 100 ? 100 : size * 100
  which is approximately equivalent to:

```
if \(size < 100\) then
    length = 100;
else
    length = size \* 100;
```

but because it does not require a temporary variable *length* it is an expression that can be used as a control parameter.
Comments: Any undefined values appear as component parameters (size, in this example). The actual value is supplied by the user. Enables complicated calculations. See *AEL* (ael) for syntax.

User-Defined Function
Example: build_shape(length, width);
Comments: A user-written AEL function to support calculations more complicated than can fit in a field of a control dialog. See *User-Defined Control* (gcc) for more information about including a user-defined function.

Component Parameters

Component (or model) parameters are the parameters you edit to customize an instance of a given component or model. These parameters appear in these locations.

- In the Component Parameters dialog box (*Edit > Component > Edit Component Parameters*).
- In the Compiler dialog message area.
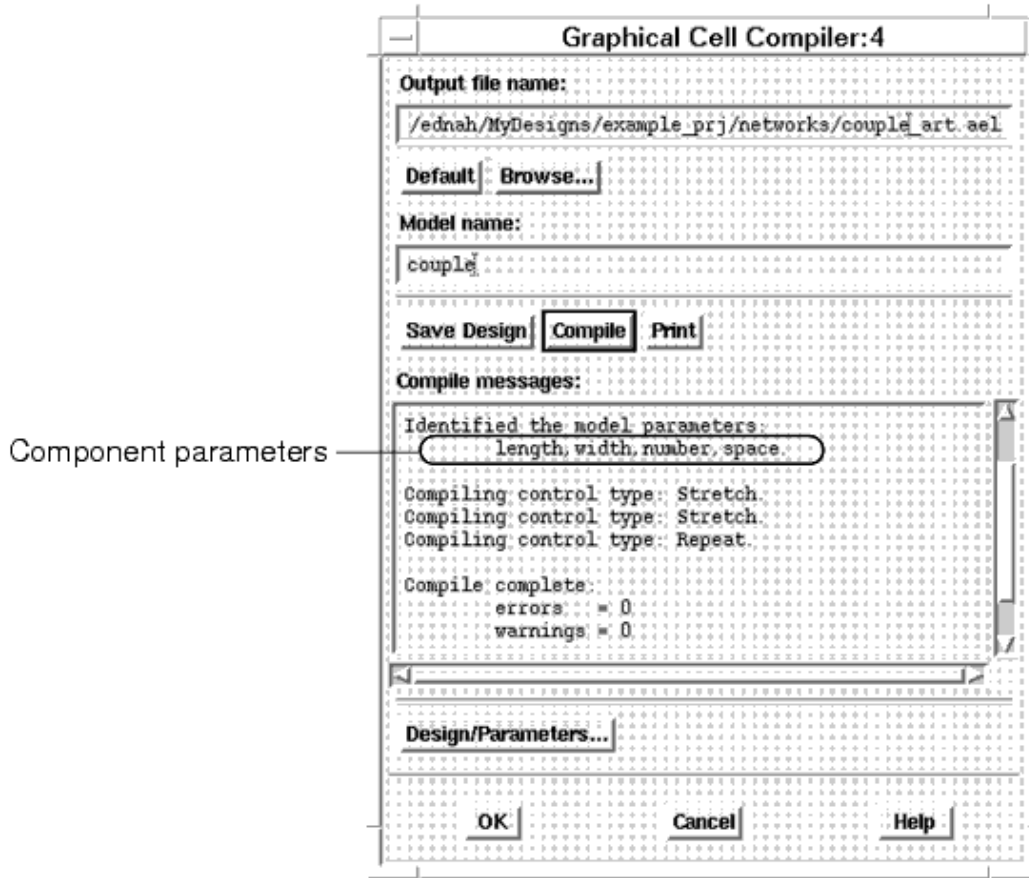
- In the Design Definition dialog box Select Parameters list (in the Parameters panel, where you set default values and units when creating a PAM).
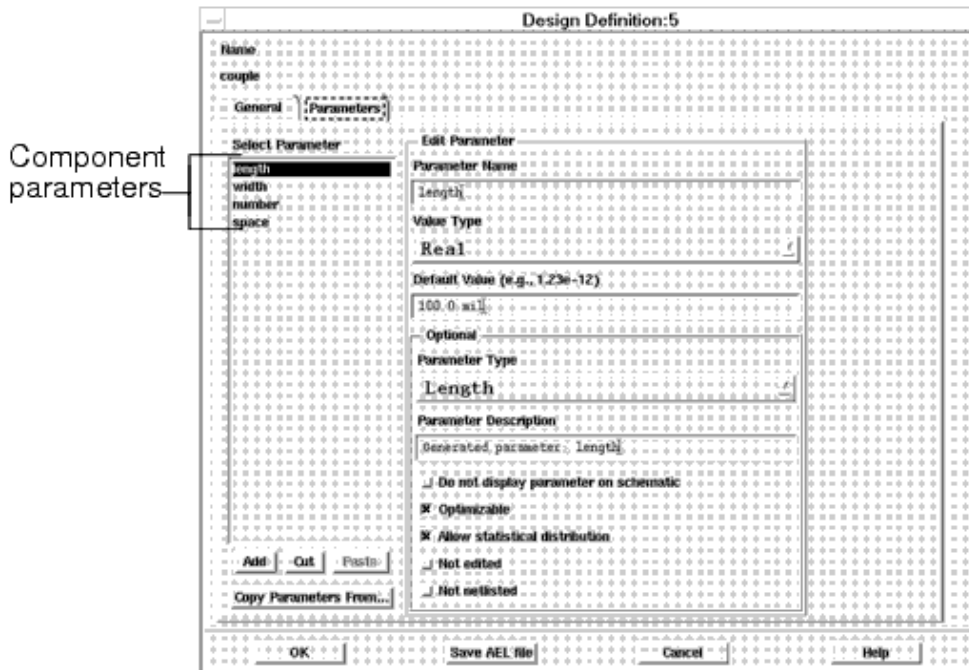
Controlling Multiple Shapes on Different Layers

If you select a shape to be included in a control, it does not matter what layer the shape is on. A control operates on any number of shapes, on any combination of layers.

In the original design, the three shapes selected for a Stretch control are on three different layers. The control is defined with a length of 100 mil and an offset of 100 mil.
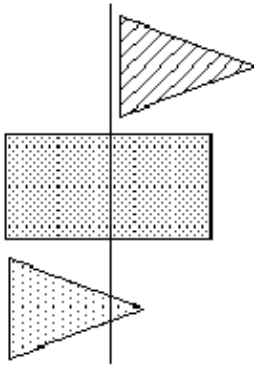
When the model is used and the length reduced to 50 mil, the top shape is moved (because the control line did not touch the original shape), and the other two shapes are stretched back.
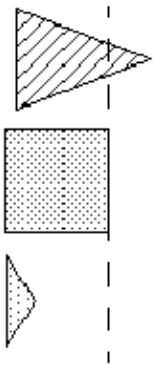
When the length is increased to 150 mil, the top shape is moved again, and the other two shapes are stretched.
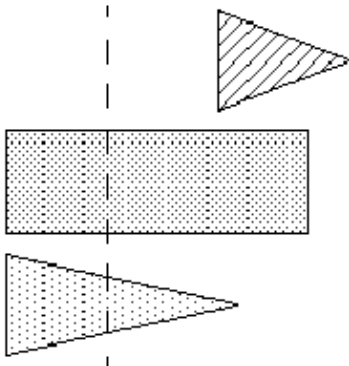
For details on using a Stretch control, refer to the section *Stretch Control.* (gcc)

Stretch Control

You can use the Stretch control to change the size of the graphics in a Parameterized Artwork Macro (PAM). Stretching takes place relative to a control line that may be at any angle. You can stretch that shape in more than one direction by placing more than one stretch control on the same shape.

As with many controls, you must identify the shape(s) that you want to effect and add a control line (construction line) before you define the stretch parameters.

This section describes how you can use the placement of control lines and the selection of stretch parameters to manipulate shapes within a PAM.
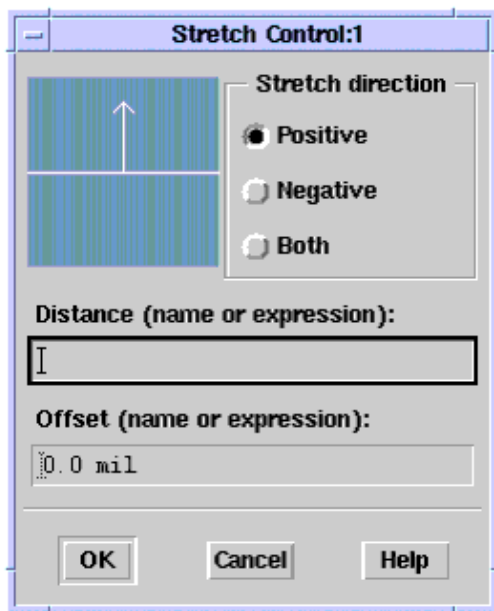
Stretch Direction

You can define stretching as *Positive*, *Negative*, or *Both*. When you define stretching as *Both*, half of the requested distance is stretched to each side of the control line.
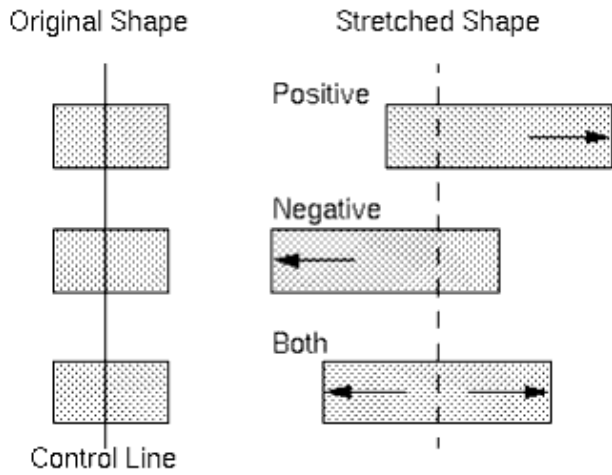
Stretch Orientation

Stretch orientation is defined relative to the stretch control line (construction line). Stretching takes place perpendicular to the control line.

If the control line is horizontal, stretching goes up/down.
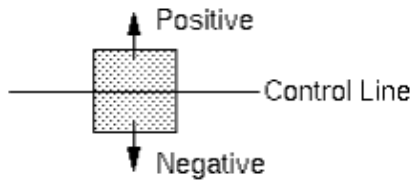
If the control line is vertical, stretching goes left/right.
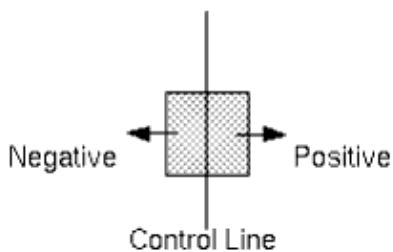
Even when the control line is at an angle, stretching is still perpendicular to it.
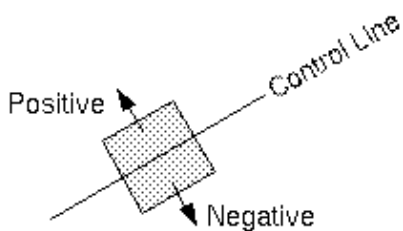
For details on positive and negative directions with regard to control lines, see *Using Control Lines.* (gcc)

Moving a Shape with a Stretch

A control line does not have to intersect a shape. If a shape is to one side of the control line *and* is included in a stretch, the entire shape moves by the stretch amount, rather than being stretched.

All of original shape is to one side of the control line. After a positive stretch, the entire shapes has moved in the positive direction.
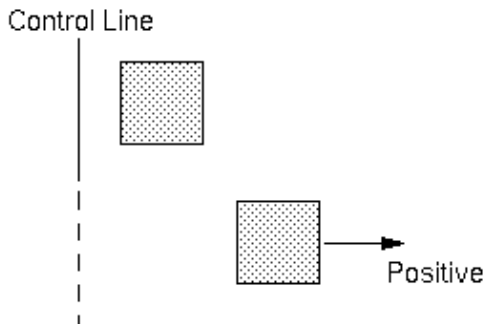
The Stretch control moves vertex points. When a control line intersects a shape, the vertices on the designated (positive or negative) side of the control line move, stretching the shape.
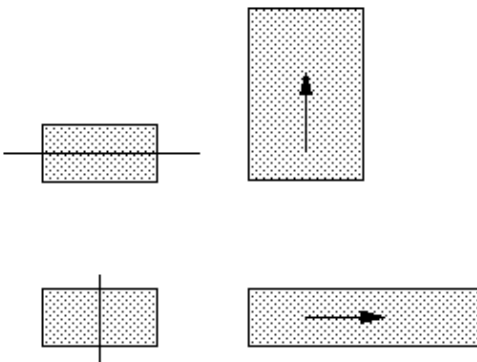
When the entire shape is to one side of the control line, *all* of the vertex points on the shape move in the same direction, by the same amount. This moves rather than stretches the shape. A stretch control is not limited to 90° angles, but can be at *any* angle.
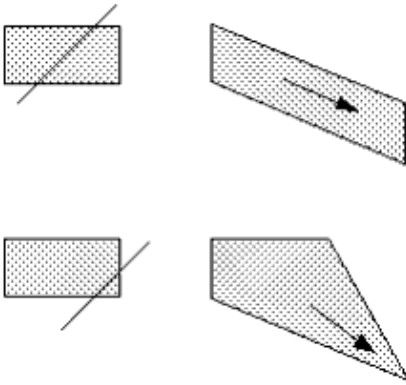
Distance

Distance is the amount to stretch the shape(s). It can be a simple variable name (such as *length*), or an expression that refers to other values. The amount of stretch can be either positive (vertex points move *away* from the stretch control line) or negative (vertex points move *toward* the stretch control line).
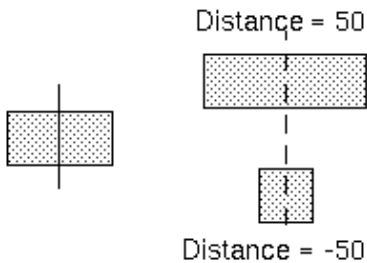
Offset

Offset is the amount to subtract from the entered length to compensate for the original size of the shape. This makes it possible for you to create a PAM using a shape with some size (so it is easy to work with), and use the PAM without needing to know its initial size.
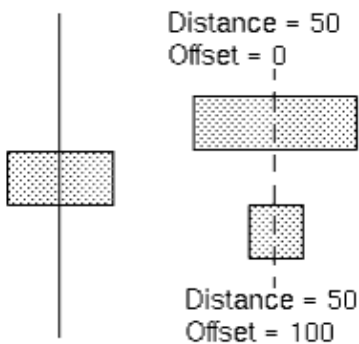
For example, you create the PAM using a rectangle of length 100 and an offset of 100. When you use the PAM and enter a length of 50, you get a rectangle with a total length of 50 rather than 150. That is,

        distance = length
        offset = 100

is the same as:

        distance = length - 100
        offset = 0

Shape Response

Different shapes respond differently to a stretch. Most shapes are defined by vertex points that can be moved by the Stretch control. Polylines and paths are different, in that these are defined by a set of control points that the line or path follows. Polylines have no width to modify; a path's width is defined with respect to the control points. There are no vertex points to move, so a Stretch control cannot modify the width of a path.

Stretching a circle in any direction simply increases its diameter. But if you convert the circle to a polygon (*Edit > Modify > Convert to polygon*) first, the polygon stretches just like any other polygon.

Different Shapes Responding to a Stretch shows how different shapes respond to stretch using both a vertical (Length stretch) and horizontal (Width stretch) control line.
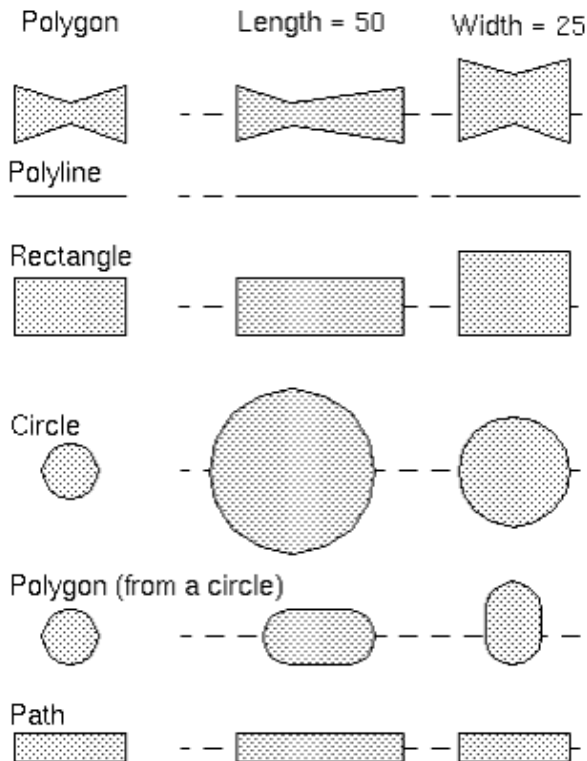
Different Shapes Responding to a Stretch

Repeat Control

This section provides details on how the placement of control lines and the selection of repetition parameters can be used to duplicate shapes within a PAM.
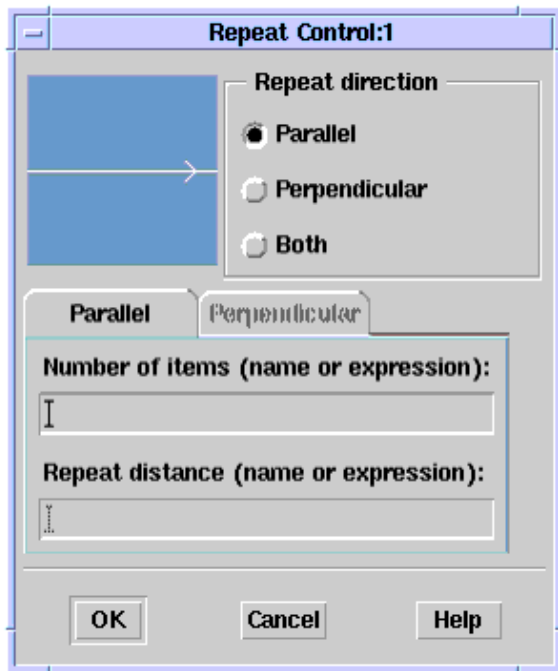

You can use the Repeat control to define the number of copies of one or more shapes in a Parameterized Artwork Macro (PAM) that are made when you place the PAM in a layout. Repetition takes place relative to a control line, which may be at any angle.

As with many controls, before you define the repetition parameters, you must first identify the shape(s) that you want to effect and add a control line (construction line).

Repeat Direction

Repeat direction is defined relative to the repeat control line (construction line). Repetition can take place parallel to a control line, perpendicular to a control line, or in both directions.
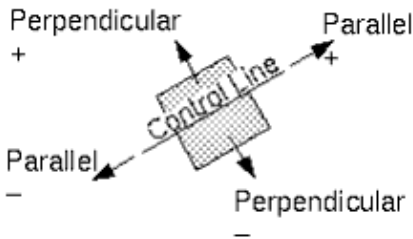
If the control line is horizontal, *Parallel* goes right/left, *Perpendicular* goes up/down.
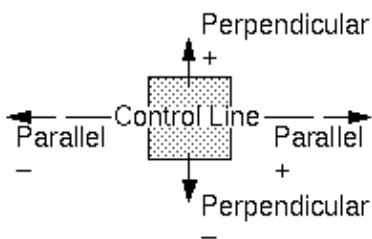
If the control line is vertical, *Parallel* goes up/down, *Perpendicular* goes right/left.
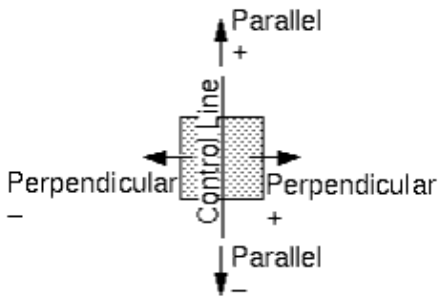
Even when the control line is at an angle, *Parallel* is still parallel to the construction line and *Perpendicular* is still perpendicular to it.
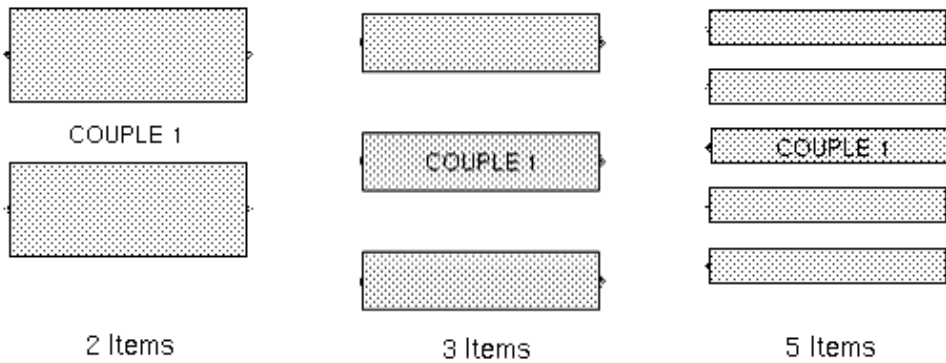
Repeating in *Both* directions generates an array-like grid of the repeated shapes.
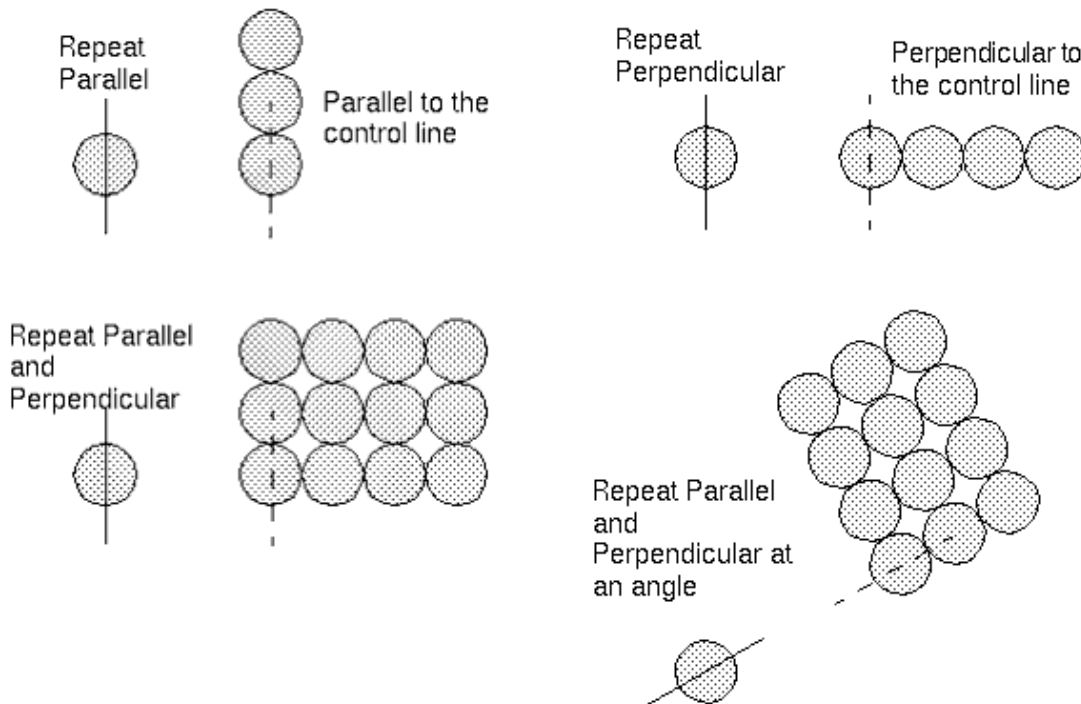
For details on positive and negative directions with regard to control lines, see *Using Control Lines.* (gcc)

Number of Items

This parameter enables you to define how many times the shape or group of shapes are placed when you use the PAM in a layout.


Repeat Distance

Repeat Distance is the distance from the beginning of one occurrence to the beginning of the next. This distance is similar to a wavelength in that it includes both the length of the shape and the space between the end of the first shape and the beginning of the second shape.
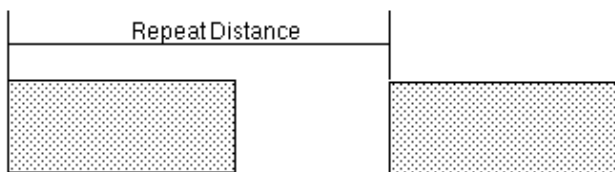
When step distance is negative, the shapes repeat in the opposite direction:
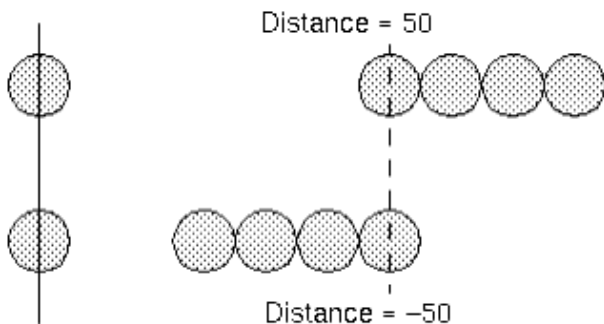
Rotate-Move-Mirror Control

This section provides details for using the Rotate/Move/Mirror parameters to manipulate shapes within a PAM.

You can use the Rotate/Move/Mirror control to rotate, move, and mirror (flip) graphics in a Parameterized Artwork Macro (PAM). Operations are performed in the following order:

- Rotation about a point (default is 0,0)
- X and/or Y offset (move)
- Mirror about the X-axis
- Mirror about the Y-axis

Rotation is performed relative to a point defined in the source layout. The default is the origin (0,0) in the source layout. An arbitrary center for rotation can be defined by the intersection of two selected construction lines.

Mirror about the X-axis and mirror about the Y-axis are performed relative to the X,Y axis that goes through the defined point in the source layout (default is 0,0).
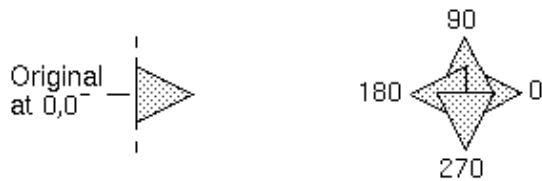
Rotation Angle

You can set the angle of rotation to be in either *radians* or *degrees*. The default is degrees.

The parameter you enter in this field enables you to set the selected shape at an angle when you place the shape in a Layout window. Exactly how the shape behaves when you insert it depends on where the shape was positioned relative to the defined center (default is 0,0) when the control was created.

For example (using the default of 0,0):

- If you place the shape at 0,0 when you create the PAM, the Angle offset parameter appears to spin the shape.

- If the shape is offset from 0,0 when you create the PAM, the Angle offset parameter produces a different effect.
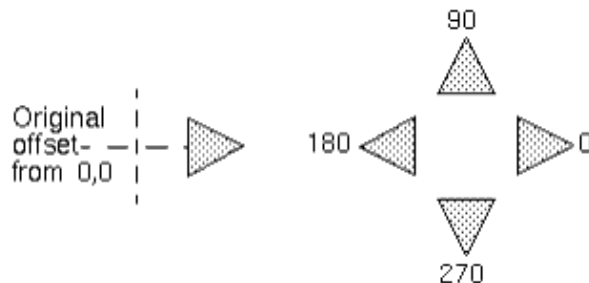
If you are designing a more complicated PAM with several shapes that you want to rotate, and those shapes are in different locations, you could have a problem. It would be impossible to center all of the shapes on 0,0 in their respective locations. There are a number of easy ways to deal with this:

- You can define an alternate center for rotation. This is done by inserting two construction lines in the source layout that intersect. Select these construction lines along with any shapes to be operated on. Their intersection is used as the center (instead of the default of 0,0) for all actions.
- A shape does not have to start in its final position. You can place it at 0,0, apply rotation, then apply a move to relocate the shape to its desired position.
- You can move a shape to 0,0, apply rotation, then move the shape back to its original position.

In any case, after you move a shape into its proper position relative to the other shapes, you can apply any number of stretch, repeat or other operations to it.
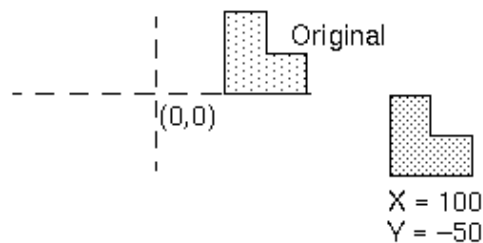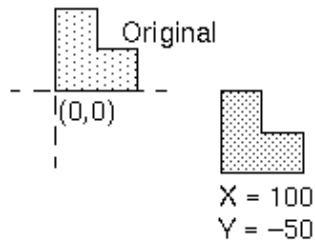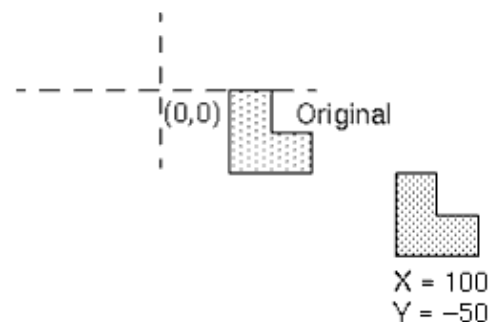
Move

The parameters you enter in these fields enable you to move the shape when you place the shape in a Layout window. These parameters move a shape relative to the o *riginal shape*. Exactly how the shape behaves when you insert it depends on where the shape was positioned relative to the defined center (default is 0,0) when the control was created.

For example (using the default of 0,0):

- If you place the shape at 0,0 when you create the PAM, the X and Y offset parameters move the shape as shown here.
- If the shape is offset from 0,0 when you create the PAM, the X and Y offset parameters produce a different effect.
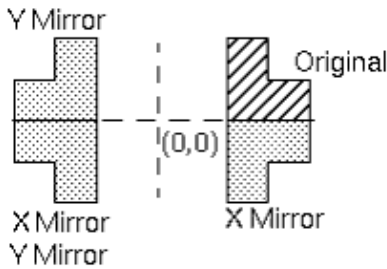
The relative movement is the same, but the absolute position of the shape is determined by the starting position (relative to 0,0).

Mirror

The parameter you enter in these fields enable you to mirror the shape when you insert it in a Layout window.

As with the other Rotate/Move/Mirror parameters, exactly how the PAM behaves when you insert it depends on where the shape was positioned relative to the defined center (default is 0,0) when the control was created.
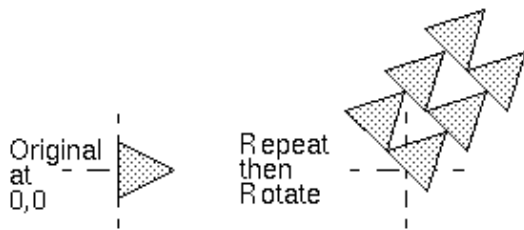
Control Order

You can use a Rotate/Move/Mirror control on shapes either before or after you use other controls. This control can operate on the original shape, or on the results of another control.

For example, you can rotate a shape and then repeat the shape.
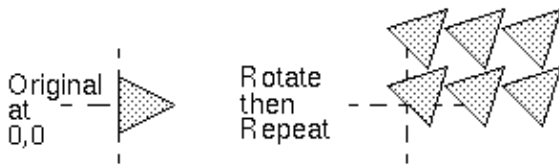
or you can repeat a shape and then rotate the results of the repetition.

Polar Control

This section provides details on how the placement of control lines and the selection of polar parameters can be used to manipulate shapes within a PAM.
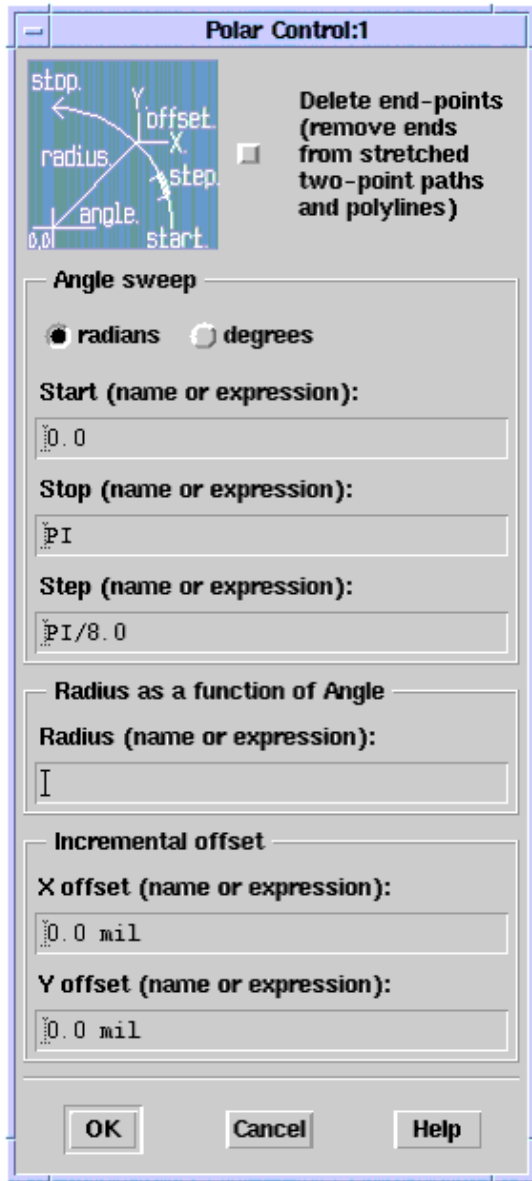
You can use the Polar control to define operations in the polar (angle, radius) coordinate system.

As with many controls, before you define the polar parameters, you must first identify the shape(s) that you want to effect and add a control line (construction line).

This control functions similarly to the Stretch and Repeat controls, but the way the polar control operates on shapes is dependent on the type of shape and the placement of the Polar control line (construction line).

Angle Sweep

You can set the angle of sweep to be in either *radians* or *degrees* . Because all AEL trig functions work in radians, the default is radians. If you want to use degrees, you must use the AEL function *rad()* each time you reference an angle in a trig function, to convert it to radians.

Start

The equation or value you enter in this field defines the starting value for the Angle sweep.
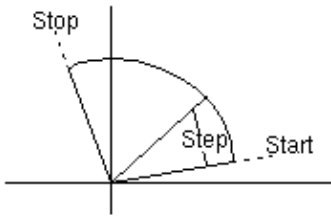
Stop

The equation or value you enter in this field defines the ending value for the Angle sweep. You can define a sweep as greater than 360° from the start. In that case, the sweep goes around more than once.

Step

The equation or value you enter in this field defines the resolution of Angle sweep. A very small step produces a smooth surface for a curved shape; a large step creates line segments.
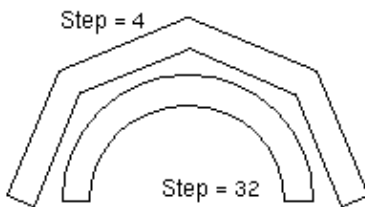
> **Note**
> The current value of Angle sweep is saved in the variable *_angle* , that can be used in expressions for the *Radius* , *X offset* , and *Y offset* parameters. For a list of the variables that can be used in these parameters, see *Using Variables in the Radius & Offset Parameters.* For more details on using expressions, see *Defining Parameters.* (gcc)

Radius and Incremental Offset

The Radius and Incremental Offset parameters are unique in that these parameters are evaluated by the program more than once. Most parameters are evaluated once and then used, but these parameters are evaluated at each Step, and then used appropriately.
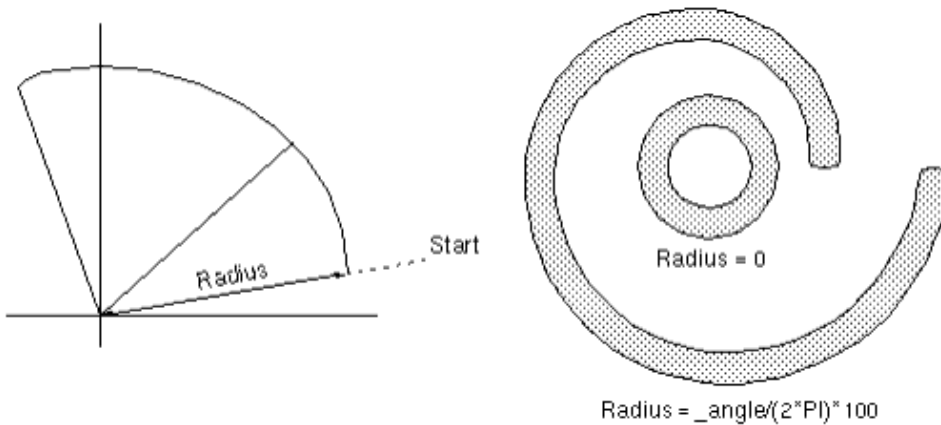
Radius as a Function of Angle

The equation or value you enter in the *Radius* field defines the change in radius from 0.0 to the initial position of the shape. To produce a spiral-like shape, define the radius as a function of Angle sweep.

Radius as a function of angle allows for some unique shapes to be defined. For example, these settings produce the illustrated shape:
Delete Ends: True
Units: Radians
Start: 0.0
Stop: PI*2
Step: PI/32
Radius: sin(2*_angle)*200 mil
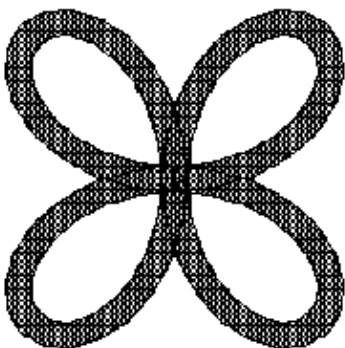
Incremental Offset as a Function of Angle

The parameters you enter in the *X offset* and *Y offset* fields enable you to modify the location of each point that results from the angle and *Radius* parameters, specifically, the point at which each *Step* ends. In the square spiral in the following example, each corner is one of these points.
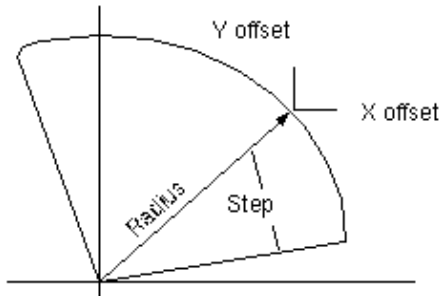
The first spiral is a radial spiral with a step size of PI/8.0. A smooth radial spiral requires a smaller step size; a square spiral, such as in this example, requires that you change the step size so that there are only four points per cycle: 2.0*PI/4.0 (2p = 360°).
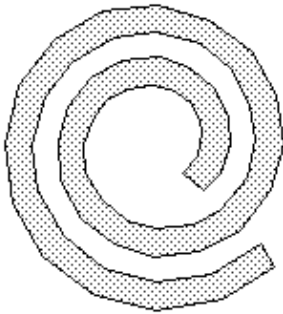
This gives a four-sided spiral. But the spiral is not square because what works for a rounded spiral, where each point is placed at an ever-increasing radius from the center, does not work for a square spiral.
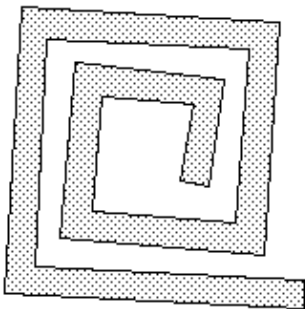
To square things up, you must use to X and Y offsets to apply a small delta (a function of the spiral parameters and the current angle) to each point (corner) to push each point along the direction of the side.
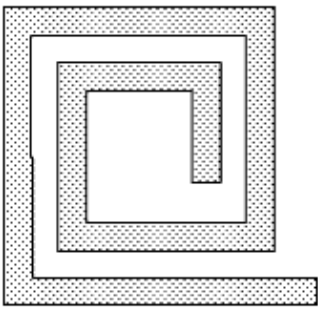
For more details on using offsets, see the section *First Spiral Example.* (gcc)

Shape Response

The polar control acts like both the Stretch and Repeat controls, depending on various factors. The only two shapes that can be stretched are paths and polylines (shapes defined by a set of control points), and these shapes stretch only if they are touched or cut by a Polar control line (construction line).
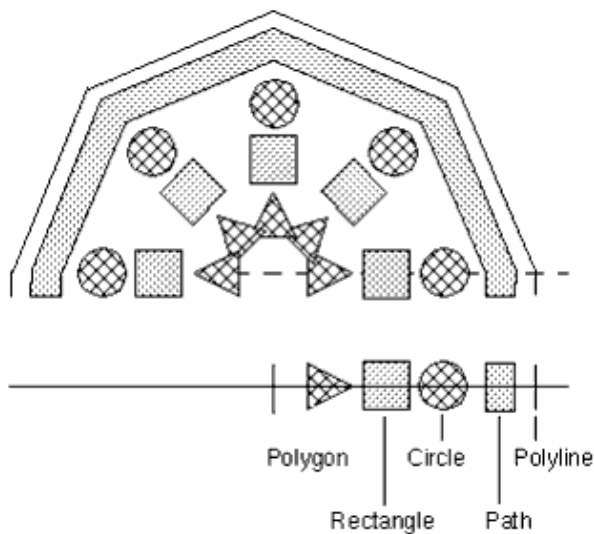
All other shapes (those whose outline or size is defined by a set of vertex points), and paths/polylines not cut by a control line, are copied by the polar control.

Using Paths

Paths support three corner types: mitered, square, and curved. Mitered and curved corner types require a certain amount of distance between the control points to be drawn correctly. If the space is too small, the results are not ideal. Because of this, when working with paths, use the square corner for a Polar stretch with a small step.

Corner = square

Corner = mitered
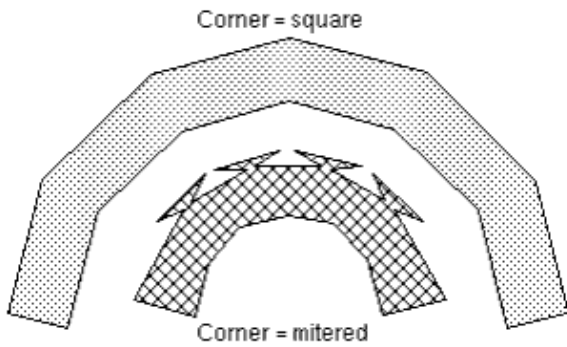
Delete End-Points

In the context of a polar operation, Delete End-Points (toggled at the top of the Polar Control Definition dialog box) means to delete the end-points of a stretched, two-point path or polyline.

In general, a polar stretch on a path or polyline preserves the parts of the shape on either side of the polar control line. This is consistent with the way a normal stretch operates on shapes, but there may be times when you do not want this to happen. All initial shapes must have some size, but you may want to generate a shape that does not have any remnant of the initial starting shape. This is similar to using *Offset* in a Stretch control to compensate for the initial size of a shape (see *Offset* (gcc)). In this case, if:

- the shape is a path or polyline,
  and
- the initial shape has only two points,
  and
- the operation is a stretch (the control line touches or cuts the shape),
  then the Delete End-Points option removes the initial points from the resulting shape. This feature is very useful when you create spiral-like shapes where you do not want the straight remnants from the initial path hanging on to the ends of the generated shape. Remember, though, that Delete End-Points works only on *two-point* paths and polylines.

Delete Ends = On

Delete Ends = Off

Delete Ends = On (more than 2 points)
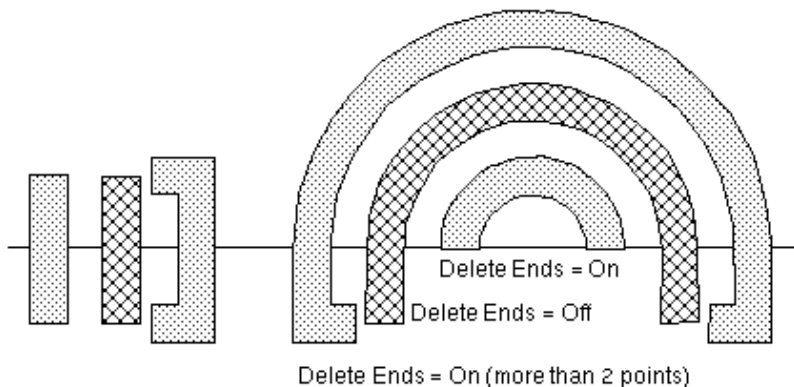
Using Variables in the Radius & Offset Parameters

A number of temporary variables have been defined to hold commonly-needed values for use in the Radius and Offset parameters. These variables act as a shorthand so that you can use the variable in an expression rather than repeating the possibly lengthy expression that defines it.

- _angle (as calculated by the Angle expression for the current step)
- _radius (For use in the Offset field only) (as calculated by the Radius expression for the current step)
- _angle_start   (as calculated by the Start expression)
- _angle_stop   (as calculated by the Stop expression)
- _angle_step   (as calculated by the Step expression)
- _angle_i (step number that increments from 0 to $\dfrac{\_angle\_stop - \_angle\_start}{\_angle\_step}$

The convention of a leading underscore (_) for global variables is the same as is used by the ADS Analog RF Simulator (ADSsim).

For more details on using expressions, see *Defining Parameters.* (gcc)

User-Defined Control

You can use the User-Defined control to perform operations that you can't do with the controls and shapes defined by the Graphical Cell Compiler.
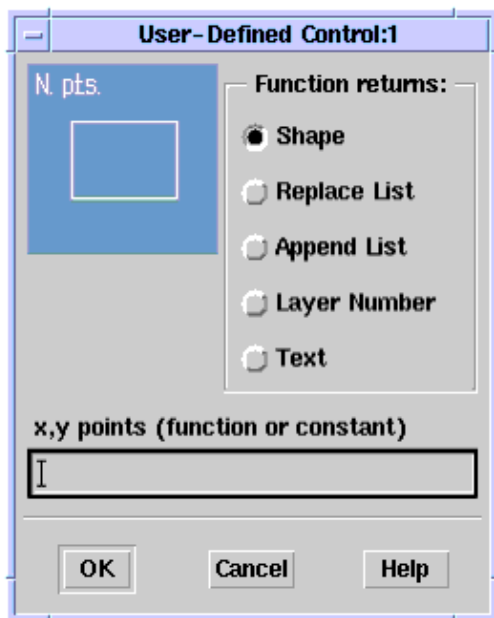
The User-Defined control actually fills the gap for two types of operations:

- Creation of shapes that are not supported by ADS or that can't be formed with existing shapes modified by existing controls (such as, an ellipse).
- Creation of a shape-modifying control of your own design to perform some operation beyond those supported by the base system (such as, using splines to smooth the outline of a polygon).

Using the User-Defined control requires a knowledge of programing in AEL. You supply an AEL function that accepts whatever parameters you want to supply the shape with and returns the (x,y) point data for the created/modified shapes. You can create almost anything. If the AEL code can be written to do it, a PAM can be created to insert it.

The general flow for the setup of a User-Defined control is the same as that for other controls:

- Select one or more shapes to be operated on. This defines the type of shape, what layer it is on, and (if used) the initial data-points of the shape.
- (Optional) Select a construction line as a reference for the control's operation.
- Define the control which includes: what parameters are to be passed to the user-written function, the name of the function to be called, and what type of data is to be returned by the function.

The function field of the User-Defined control is basically just like any other expression field in any of the other controls except that there are a few more restrictions on it. In the other controls the expressions must evaluate to a simple numeric (integer or real) or boolean type. In this case, the results of the expression generally must be (x,y) point data

for a shape, or possibly the data for a list of shapes.

Parameters

The parameters to the user-written function can be anything allowed by AEL syntax: constants, variables, expressions, etc. As with the other controls, any unrecognized variable is taken as a component parameter to the macro.

In addition, a number of pre-defined variables (like the ones available in the Polar control) can be passed to the user-written function:

_cline_data

This is 2-element array of (x,y) values (as in [[x1, y1], [x2, y2]]) which define the construction line (if included, that is selected, when the control was defined) for this control. The position and slope of the construction line can then be used to control how the function operates on the shape, much in the same way that the construction line specifies the direction of movement in the Stretch control or the direction of copy in the Repeat control.

For example, using _cline_data to compute the slope of the construction line could look something like:

```
decl x1, y1, x2, y2;
decl slope;
x1 = _cline_data[0,0];
y1 = _cline_data[0,1];
x2 = _cline_data[1,0];
y2 = _cline_data[1,1];
if ((x2 - x1) == 0.0)
     slope = 999999.      \* vertical line, infinite slope */
else
     slope = (y2 - y1) / (x2 - x1);
```

_shape_type

The type of shape being operated on. One of the following pre-defined AEL constants:

- PAM_PATH_TYPE
- PAM_POLYGON_TYPE
- PAM_POLYLINE_TYPE
- PAM_PORT_TYPE
- PAM_TEXT_TYPE

This is probably most useful as an error check to make sure the selected shape is appropriate for the user-written function. For example, it may well be an error case to pass a text element (which only has one (x,y) point) to a function which operates on polygons (which must have at least three (x,y) points).

An error check might look something like:

```
if (_shape_type != PAM_POLYGON_TYPE) {
    de_error_dialog("Shape is not a polygon");
    return(NULL);
    }
```

_shape_data

The current (x,y) data points for the shape being operated on. This data can be modified from the original shape if it has been operated on by previous controls (such as, Stretch).

You would pass the _shape_data parameter to the user-written function if your function was designed to modify a graphic shape. For example, let's say the function was written to rotate a shape about a user-defined point rather than about the origin (0,0) as the Rotate/Move/Mirror control does. Such a function would take the current _shape_data, perform the rotation operation on it controlled by some number of additional parameters, and return the modified data points.

On the other hand, if the function was written to create a new graphic shape based on the other parameters passed to it (an ellipse for example) then it would not need the _shape_data parameter at all. Or, it might only use the _shape_data parameter to get an initial (x,y) location for the creation of the new shape but ignore the rest of the data points.

If the _shape_data parameter is passed and how the data points contained in it are used is totally up to the requirements of the user-written function.

The *shape_data parameter is a two-dimensional array of values. The first dimension is the data point number (from 0 to the number of points - 1) and the second dimension indexes the x and y value (0 for x, 1 for y). The _array_upperBound()* function is used to access the number of data points in the array. So, a simple iterative loop to add a fixed (x,y) offset to each point of a passed in shape might look something like this:

```
decl i;
for (i=0; i<=array_upperBound(_shape_data, 1); i++) {
    _shape_data[i,0] += x_offset;
    _shape_data[i,1] += y_offset;
    }
return(_shape_data);                \* return the modified data
```

_shape_init

The initial (x,y) data points for the shape as it originally existed in the source layout. This structure is never modified by the PAM so it is always possible to get the original points if needed.

These points are important because the other controls that use a construction line as a reference (Stretch, Repeat, Polar) compare the ORIGINAL shape data with the construction line to determine direction, orientation, intersection, etc. That way the specified action is not changed because a shape was moved on or off a construction line by other controls.

If you are creating a control that you want to work in the same way as the built-in

controls then you will want to use the _shape_init point data for any tests with respect to the reference construction line.

On the other hand, you may specifically want a control which will act on a shape based on previous actions performed on it. In that case you would want to use the _shape_data (described above) for all point data tests with respect to the reference construction line.

_shape_list

The current list of sets of (x,y) data points (the result of a Repeat or Polar control) for the shape being operated on.

Passing the _shape_list parameter to the user-written function would happen when the purpose of the function is to modify a list of shapes that have been created by another control. For example, one may want to take the shapes generated by a Repeat control and rotate each one by some amount dependent on which copy it is.

The *shape_list parameter is a list of two-dimensional arrays (like _shape_data). The _listlen() function will provide the length of the list (number of shapes) and the nth() function will access the individual shapes (zero relative). So, a simple loop to access all the shapes passed to a function in the _shape_list parameter might look something like:*

```
decl n;
    decl i;
    decl shape;
    decl shapes;                \* list of modified shapes to return */
    shapes = NULL;
n = listlen(_shape_list);
for(i=0; i<n; i++) {
    shape = nth(i, _shape_list);
    \* operate on array of (x,y) points in shape */
    shapes = append(shapes, list(shape));
}
    return(shapes);
```

_shape_layer

The ID number for the layer the shape is on.

Knowing the Layer ID for a shape allows the AEL function to perform layer-specific operation on the shapes. It may also be useful if a layer mapping function is to be performed, taking the initial layer, mapping it to a different layer, and returning that value to be use when the shape is inserted (see Return Value).

There is one important point to be aware of in using the above shape variables. While AEL syntax does allow for any function parameter to be passed by-reference (using the "&name" syntax) doing so for any of the above variables will NOT allow you to modify the actual shape's values. That is because the above variables are already copies of the shape's data and even if they were modified the contents are NOT copied back to the original shape's values.

Function Call

109

The function call as defined in the User-Defined dialog is really just any AEL expression that generates an array of (x,y) points (for the case of a simple shape) or a list of shapes (for the list case). In most cases this will be a function call since it will require multiple lines of AEL code to perform the desired operation. But there are cases where the expression entered could be simpler:

NULL

This would have the effect of deleting the shape or list of shapes. This could be used as part of a conditional expression (using the ?: syntax) to optionally delete a shape, for example:

```
(delete == TRUE) ? NULL : _shape_data
Which is functionally equivalent to a function that did:
    if (delete == TRUE)
            return(NULL);
else
    return(_shape_data);
```

This would test for the parameter "delete" being set to TRUE. If it is then a NULL would be returned essentially deleting the shape. If "delete" is FALSE then it would return the un-modified shape data, basically doing no operation to the data.

Constant

Using a constant would have the same effect as defining the shape in the source layout and not performing any operation on it. This would define it's location and shape programatically rather than graphically. So defining a fixed-sized circle this way would have something like the following in the dialog field:

```
[ [0.0,0.0], [100.0, 0.0] ]
```

Which would define a circle of radius 100 centered at the origin.

Variables

This would be similar to the use of a constant, but would allow the values to be specified by the user as component parameters. So a user-defined circle might look like:

```
[ [center_x, center_y], [center_x + radius, center_y] ]
```

where the user would supply the values for center (x,y) and radius at component insertion time.

Function

The most general purpose usage is to call a function. A simple example might look like:

```
modify_shape(_shape_data, length, width)
```

Where modify_shape is the user-written function. It receives three arguments: the current (x,y) array of data points for the shape, and a length and width value which are component parameters specified by the user at component insertion time.

Return Value

The type of value returned by the function is specified in the User-Defined dialog by the "Function returns:" radio buttons. This informs the compiler what is being returned and what to do with it. The four choices are:

- *Shape* is the value returned is simple shape data (and array of (x,y) points) and should replace the current shape data for the shape being operated on.
- *Replace List* is the value returned is a list of shapes (each being an array of (x,y) points) and should replace the current list for the shape being operated on. This case is used when the user-written function performed modifications of the existing list of shapes but did not generate any new shapes.
- *Append List* is the value returned is a list of shapes (each being an array of (x,y) points) and should be appended to the current list for the shape being operated on. This case is used when the user-written function is generating additional copies of the shape being operated on and does not want to loose any existing list (perhaps generated by a previous Repeat control).
- *Layer ID* is the value returned is the new layer ID (integer value) to use when the shape is created and inserted into the Layout.

There is one more very important issue to be aware of when deciding what the return type is. If the user-defined function is making simple modifications to the (x,y) data points (as in a stretch or similar action) then the resulting shape would be returned as type "Shape". But if the function is adding or deleting (x,y) data points (as the Polar controller adds points to a path) then the resulting shape must be returned as a "Replace List" or "Append List".

The reason for this is that the Initial and Current shape data arrays must always be of the same size (same number of (x,y) points). Several of the controls rely on this fact in processing the shape. For example, in the Stretch control, it is comparing the Initial array with the construction line to decide which points will be modified. If it decides that the shape's fourth point is to be modified then there had better be a fourth point in the Current structure to apply the modification to.

For those controls using both the Initial and Current arrays their size is checked during initialization and an error is displayed if they aren't equal.

This also helps demonstrate why certain controls (like Stretch) should appear before other controls (like Repeat or Polar). For more information see *Defining Controls* (gcc).
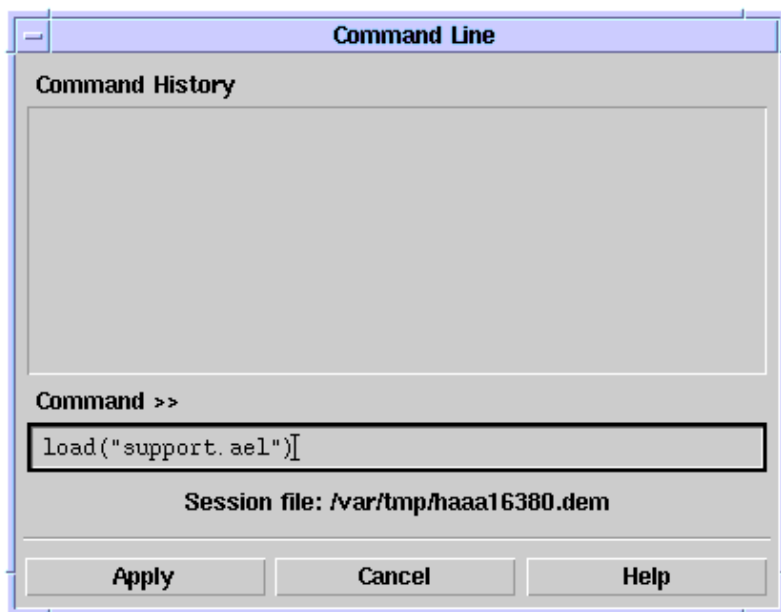
So if the user-defined function is modifying the (x,y) data points of a shape you should set the return type to "Shape". But if the function is adding or deleting (x,y) data points then you should set the return type to either "Replace List" or "Append List".

Function Implementation

As already stated the user must supply the actual AEL function called by the User-Defined control. The function my use any parameters the user wants including any of the pre-defined ones listed above. the output must be either an (x,y) array of shape data points, a list of shapes as described above, or a layer ID. What's in between is totally up to you.

If the function does not exist in the system at compile time then a warning message will be issued by the compiler. This is fine as long as the function exists by the time the PAM is executed. If it doesn't exist then executing the macro will generate an AEL error.

During development and testing of the function the best way to load it into the system is by executing a *load()* command from the Command Line dialog (accessed from the main window with the "Options/Command Line" menu pick). This allows you to load the AEL file to test the function, then make changes to it and re-load it without the need to exit ADS. See *AEL* (ael) for details on the load() command.

Command Line
Command History
Command >>
load("support.ael")
Session file: /var/tmp/haaa16380.dem
Apply   Cancel   Help

If you want the AEL function to be visible to the compiler to eliminate the warning messages, it must be loaded into the system before the compile is done. The function does not need to be complete at this time. In fact it can be a NULL function, that is, a function definition with no actual code. Simply defining the function's name will be enough to prevent the PAM compile warning. Of course the function must be fully implemented when the PAM is inserted in order to get the desired results.

After the AEL function is completed and tested, you should make sure it's loaded each time ADS is started. You can use the "USER_AEL" directive in the de_sim.cfg file (see the AEL Guide for details). Basically, the procedure is:

First, edit your $HOME/hpeesof/config/de_sim.cfg file and add a line like this:

```
USER_AEL=$HOME/user.ael
```

which will load the file user.ael from your home directory. That file should be a list of load() commands for all the AEL function you have written that you wish to load as part of

the system. It might look something like:

```
load("/users/brett/my_models/spiral.ael");
load("/users/brett/my_models/couple.ael");
load("/users/brett/my_models/square.ael");
fputs(stderr, "End Loading user.ael");
```

Obviously the path and file names can be anything you want such that the organization of your PAM support code makes sense.

The *fputs()* line is simply a way to output a message to the ADS startup window as a sort of verification that the file did run and load all the support functions. It can easily be commented out once satisfied that everything is working the way you want.

Now each time ADS is started, the user.ael file will be executed which will load all the other AEL files which define the various user-written functions needed.

A Simple Example

This section provides a simple example of how to use a User-Defined control to perform an action that would be very hard to do with the default controls. While the example may not be very realistic, it does demonstrate how easy it is to extend the Graphical Cell Compiler with user-written AEL code.

For this example we assume that you want to make an NxM grid of rectangles. But it's not just a simple grid, you want it to look like a checker-board (every other shape is missing).

Starting with the source layout, add a simple rectangle with two construction lines for use by the various controls:
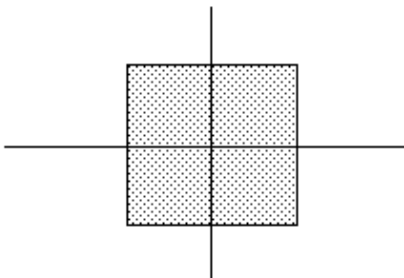
Next add two Stretch controls to set the length and width of the rectangle:

Control:	Stretch
	Direction: Both
	Length: length
	Offset: 50.0 mil

Control:	Stretch
	Direction: Positive
	Length: width
	Offset: 50.0 mil

Then add a Repeat control (selecting the horizontal construction line) to build the grid:

Control:	Repeat
	Direction: Both
	Parallel Number: x_num Distance: length
	Perpendicular Number: y_num Distance: width

If this model is compiled and inserted you'll get an x_num by y_num grid of rectangles (size length by width) with no space between the individual shapes. If the fill-type is set to filled for that layer it will look like one big rectangle.

Now add a User-Defined control to call an AEL function which will do the work of removing the shapes needed to create the final structure.

Control:	User-Defined
	Returns: List Replace
	Function: remove_some(x_num, y_num, _shape_list)

The purpose of the user-written function *remove_some()* is to take the list of shapes generated by the Repeat control, iterate over it removing the ones no longer needed, and return the modified list. We pass the function three parameters: the X and Y dimension using the same names as in the Repeat control, and the pre-defined variable _shape_list which is the generates list of shapes for the rectangle.

An example of the code for *remove_some()* is:

```
defun remove_some(x, y, shapes)
{
decl i, j;
decl odd;
decl keep;
keep = list();        /* initialize the return list     */
odd = 1;        /* removing odd/even elements                   */
for (i=0; i<x; i++) {       /* iterate over X axis      */
     for (j=0; j<y; j++)      /* iterate over Y axis      */
          if ((j%2) != odd)
                   keep = append(keep, list(nth(j*x+i, shapes)));
     odd = odd^1;       /* toggle for the next column      */
     }
return(keep);       /* return reduced list      */
}
```

The Repeat controller iterates over X (Parallel) and then Y (Perpendicular) which is a Column Major Order. Therefore, we need to process the generated list the same way in order to know which (x,y) rectangle is being considered for removal (that is, NOT to be

114

copied to the output list).

The final artwork created by the PAM as defined above would look something like:
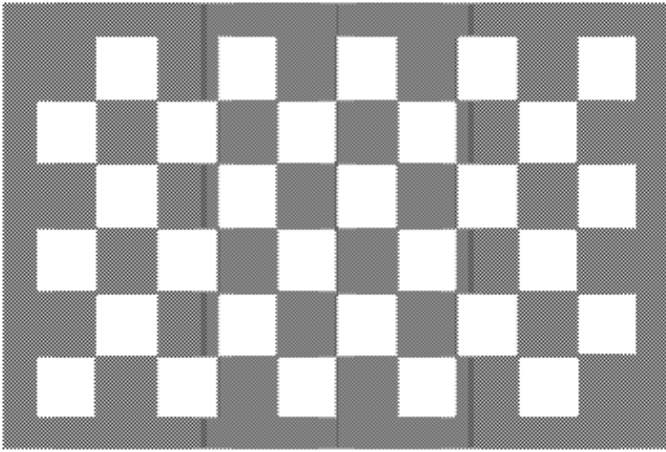
Of course the parameters defined give full control over the size of the rectangles and the number in each axis.

Width Control

This section provides details on how to use the Width control to control the width attribute on paths and height attribute on text. You can use the Width control to specify the width of primitive shapes that have the width attribute.
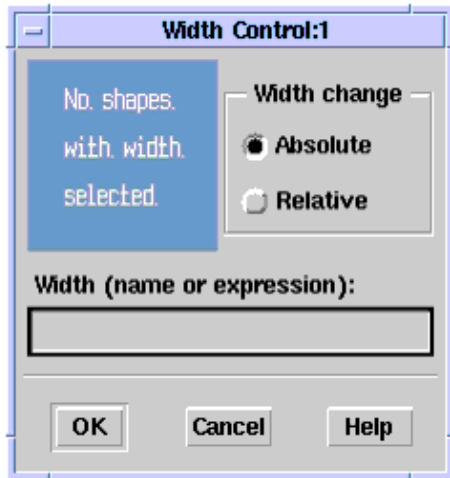
The width attribute defines the *thickness* of certain shapes. Paths are the only shape with width supported by the Graphical Cell Compiler at this time. Since the width attribute is not affected by other controls (like Stretch) the Width control provides the only access to change it.

While text does not have width (in the sense that paths do), the text height attribute is basically the same concept and is also supported by the Width control. This means that one can use the Width control to make absolute or relative changes to text height.

Width Change

The Width Change parameter defines how you wish to apply the specified width to the selected shape. If Absolute is selected, then all shapes associated with this control are set to the specified value, ignoring the value they were created with.

If Relative is selected, then the specified value is added to the currently-defined shape's attribute.
Again, this applies to width for paths and height for text. The illustration shows the differences between Absolute and Relative width changes.

Absolute width change = 15
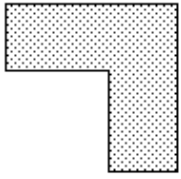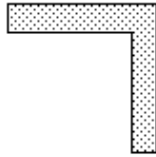
| Before: | After: | Before: | After: |
| Width = 25 | Width = 10 | Width = 15 | Width = 15 |

Relative width change = 15
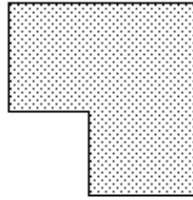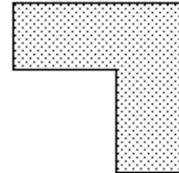
| Before: | After: | Before: | After: |
| Width = 25 | Width = 10 | Width = 40 | Width = 25 |

Width

This parameter specifies the actual value the shapes are to be set to Absolute or modified by Relative.

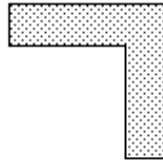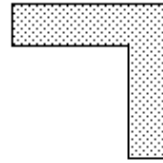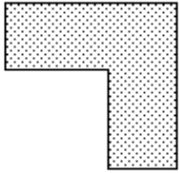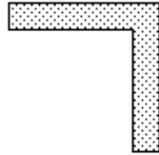The results of the Width equation can be negative. This could make sense if the initial shape had a specific value which needed to be reduced as part of the execution of the PAM. But when all the width controls are done, the value being assigned to any specific shape must be positive. A negative value is not allowed. The compiled macro forces this condition by taking the computed width and operating on it with the *max2()* AEL function. For paths the function is:

```
de_set_path_width(max2(computed_width, 0.0));
```

This way if the computed width happens to be negative, the value of 0.0 is used instead.

This further implies that if one is having trouble with shapes being generated with zero width, make sure the width calculations are not going negative.

Since the width/height parameter is associated with, but still separate from, a given shape (or shapes) it's value as modified by the Width control is independent of any other actions performed on the shape (such as, stretch, repeat, etc.). This also implies that if a shape is copied (by a Repeat or Polar control) that all the resulting shapes have the same width/height. And since the value applies to all copies of a given shape it doesn't matter whether the Width control is executed before or after a copy control.

Viewing Controls

View Controls Dialog Box

This dialog box (*Macro > Edit/View*) displays a scrolling list of all currently defined controls in the order of execution. Use this dialog box to delete, modify, or change the order of controls. Selecting a control highlights the shapes and control lines used in that control.

Edit opens the control dialog box for the selected control so you can change parameter values, change the shapes operated on by the control, or change the control line that the control uses as a reference (see Editing a Control).

Cut removes the selected control from the list. If you do not paste the cut item back in before you exit the Viewer dialog box, the control is deleted. Any control that you cut is displayed in the *Cut Buffer Contents* field.

Paste inserts a cut control back into the list, in *front* of the selected control.

Detail displays the detailed settings for each control (see Control Details).

Compile is a shortcut to close the Viewer dialog, open the Compile dialog, and compile the macro.

Cancel cancels *only* cut/paste modifications. After you accept a change in an individual

control's dialog box, *that* change can *not* be undone from the Control Viewer. Use the individual control's *Cancel* button *before* you leave *that* dialog box.

Control Details

When you click *Detail* in the View Controls dialog box, the program displays a list of the currently defined controls that includes all of the settings for each control.

Because a control is not actually deleted when you cut it from the Viewer, the detail list contains all controls defined when you opened the Viewer. When you dismiss the Viewer (deleting any cut controls), the detail list updates.

The detail list does not automatically update as you make changes to control order or control settings. To force an update without closing the Viewer, click **Refresh** .

```
                        Detail:2

Controller detail list:

Control:  Stretch
          Direction:      Both
          Length:         length
          Offset:         100.0 mil

Control:  Stretch
          Direction:      Both
          Length:         width
          Offset:         50.0 mil

Control:  Repeat
          Direction:      Parallel
          Parallel
            Number:       number
            Distance:     width+space


   OK              Refresh          Print
```
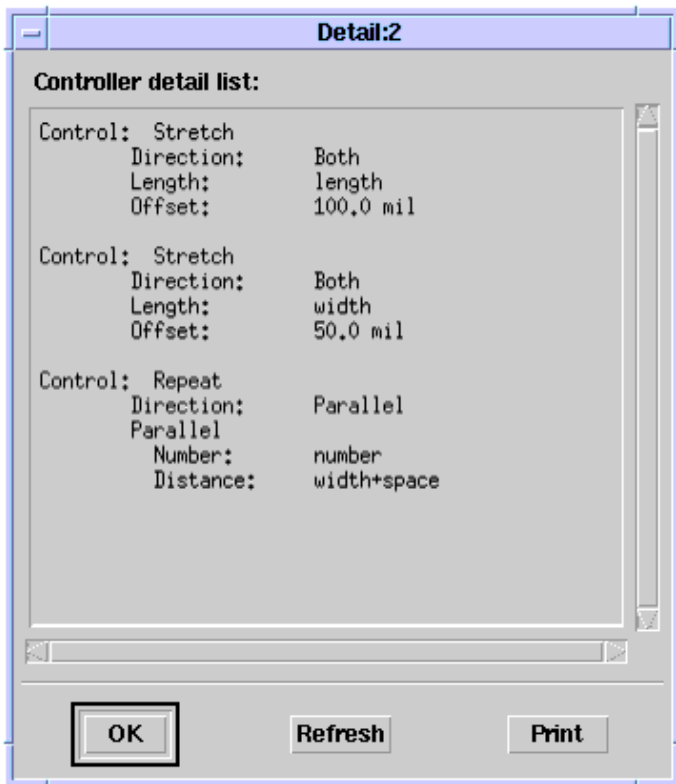
Compile a shortcut to close the Viewer dialog, open the Compile dialog, and compile the macro.

Editing a Control

Use the View Controls dialog box to edit an existing control:

1. Select the desired control.
2. Click **Edit** .
3. When the definition dialog box opens for the selected control, edit the control definitions. You can change parameter values, change the shapes operated on by the

control, or change the control line that the control uses as a reference

4. Click OK to accept the new settings.

> **ⓘ Note**
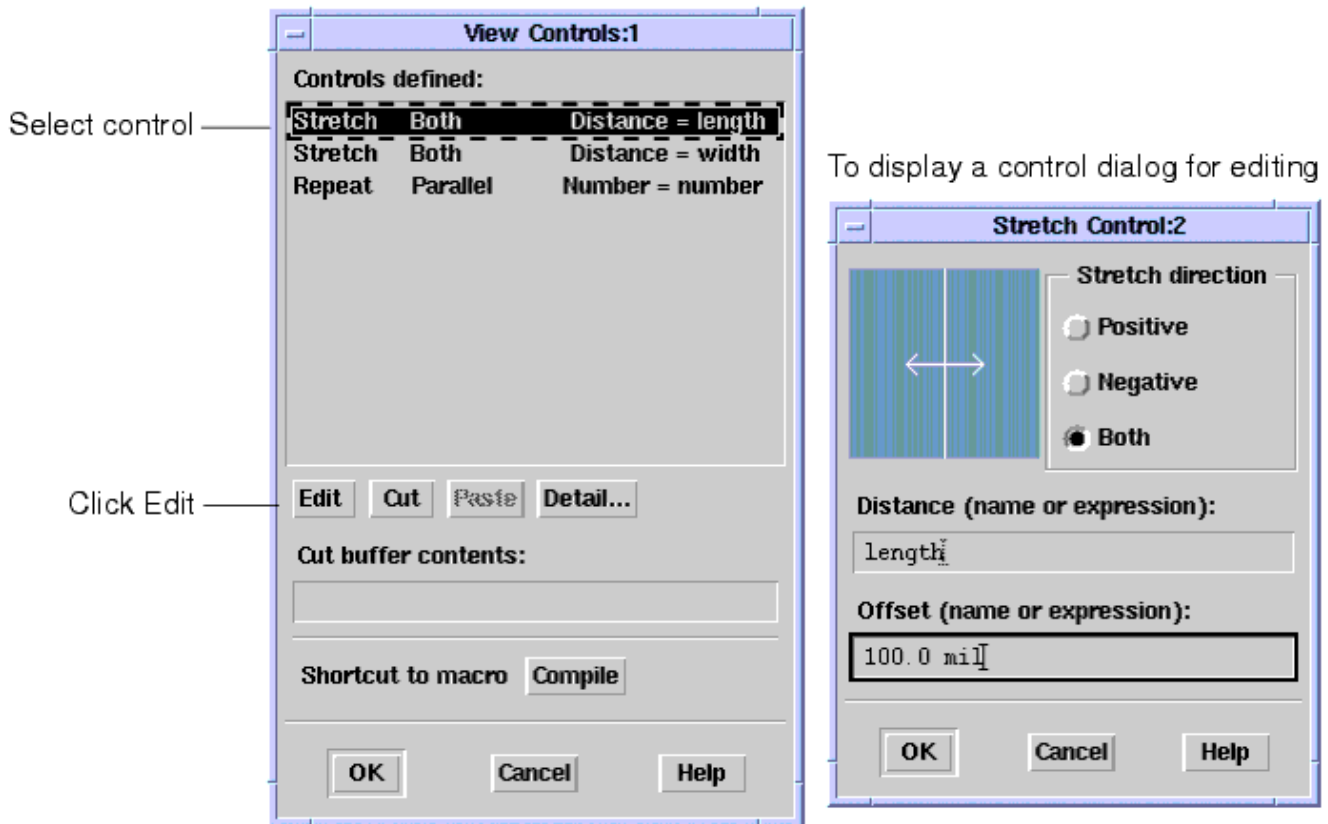> To cancel editing on a control, use the individual control's *Cancel* button *before* you leave *that* dialog box. Remember, after you accept a change in an individual control's dialog box, that change can *not* be undone from the Viewer; the *Cancel* button in the Viewer cancels *only* cut/paste modifications.

Compile Shortcut

After the initial PAM has been defined, a very common use-model is:

1. Open the Viewer dialog.
2. Make various changes to the controls under development.
3. Exit the Viewer dialog.
4. Open the Compile dialog.
5. Save the design.
6. Compile.
7. Exit the Compile dialog.
8. Test the latest version of the PAM.

When operating in this mode, steps 3 through 6 are little more than busy-work. The Compile button in the Viewer dialog performs these four steps for you, saving the extra mouse clicks and getting the results of the compile without delay.

Temporary Control Variables

A number of the controls have locally defined variables which are assigned by the control while it is executing. These variables can be used in the equations defined for the controls to make the calculations simpler.

The convention of a leading underscore (_) for global variables is the same as is used by the ADS Analog RF Simulator (ADSsim).

Polar Control

- *_angle* As calculated by the Angle expression for the current step.
- *_radius* For use in the Offset field only; as calculated by the Radius expression for the current step.
- *_angle_start* As calculated by the Start expression for the current step.
- *_angle_stop* As calculated by the Stop expression for the current step.
- *_angle_step* As calculated by the Step expression for the current step.
- *_angle_i* Step number that increments from 0 to $\dfrac{\_angle\_stop - \_angle\_start}{\_angle\_step}$.

User-Defined Control

- _cline_data A two-dimensional array of two points (as in [[x1, y1], [x2, y2]]) which define the construction line (if included) for this control.
- _shape_type The type of shape being operated on. One of the following pre-defined AEL constants:
  PAM_PATH_TYPE
  PAM_POLYGON_TYPE
  PAM_POLYLINE_TYPE
  PAM_PORT_TYPE
  PAM_TEXT_TYPE
- _shape_data The current (x,y) data points for the shape being operated on.
- _shape_init The initial (x,y) data points for the shape being operated on.
- _shape_list The current list of sets of (x,y) data points (the result of a Repeat or Polar control) for the shape being operated on.
- _shape_layer The current layer ID of the shape being operated on.

Second Spiral Example

While the model in the section *First Spiral Example* (gcc) generates a nice looking square spiral, the model may not be one of the standard spiral models that would have simulation data to go with it. This section takes the first spiral example another step and modifies it to generate a more useful model.

The basic premise of the first model is that the vertex points computed along the spiral are the vertex (corners) of the resulting square spiral, but a little *adjustment* is needed to square things up. While this worked well, the model does have a couple of failings when we try to use it for a real-world model. First, the *adjustment* math is rather complex. Second, it is *very* hard to make adjustments to the end-points so that these line up on edges or adhere to other model constraints.

To address this concern, a different geometry definition is explored. Instead of the computed spiral points being the vertex points, the computed points are defined as points *on the sides* of the square spiral. Then the vertex points are the perpendicular intersection of two sides lying on two successive compute points.

A Second Geometry Lesson

Once again we must consider geometry. We will start with two vectors of a given radius and angle (*r1,a1* and *r2,a2*). These vectors correspond to two of the computed points on the spiral. We know the angle and radius of each one. The end of each line lies on one leg of the final spiral, and the ends meet the leg at an angle of 90°. What we need to compute is the vector *r3,a3* to where these two sides intersect, and then the actual *x,y* point. For a square spiral, several more of these angles will also be 90°, but we will not assume that in this example, so we can create a general case that could be used to generate a spiral of any number of sides.
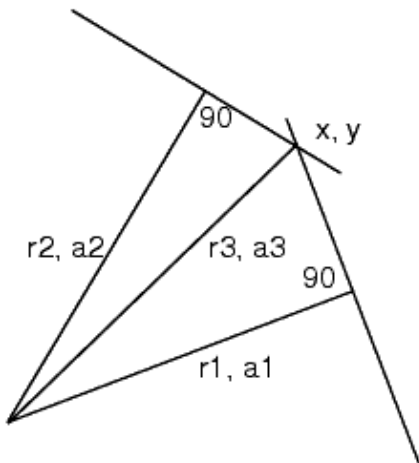
Now we'll take this general case geometry and formalize it a bit so it is easier to see the relationships between the various angles and sides. Note that even though it has been drawn such that the *r1,a1* vector is on the X-axis, this will not be true for the general case, so we must take care in the assumptions that we make.
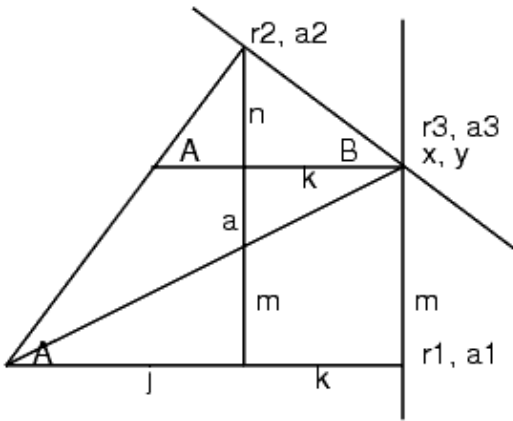
We can start with the known values and work to a solution for the *r3,a3* vector and then the values for (*x,y*):

1. The first given vector is of length and angle: *r1,a1* .
   The second given vector is of length and angle: *r2,a2* .
2. This makes the angle between them $A = a2 - a1$
   .

3. The definition of *cos()* says that $\cos(A) = \dfrac{j}{r2}$
   , $j = r2 \times \cos(A)$
   .

4. The definition of *sin()* says that $\sin(A) = \dfrac{a}{r2}$
   , so $a = r2 \times \sin(A)$
   .

5. *B* is part of a right triangle, so $B = 90 - A$
   .

6. $r1 = j + k$
   , so $k = r1 - j$
   . Substitute for j and we get $k = r1 - r2 \times \cos(A)$
   .

7. The definition of *tan()* says that $\tan(B) = \dfrac{n}{k}$
   , so $n = k \times \tan(B)$
   , which, substituting for *k* expands to $n = (r1 - r2 \times \cos(A)) \times \tan(90 - A)$
   .

8. $m = a - n$
   . Substitute for *a* and *n* , and $m = r2 \times \sin(A) - (r1 - r2 \times \cos(A)) \times \tan(90 - A)$
   . Substitute for *A* , and
   $\imath = r2 \times \sin(a2 - a1) - (r1 - r2 \times \cos(a2 - a1)) \times \tan(90 - a2 - a1$

9. We now have the two sides of the triangle that will give us the *r3,a3* vector: *r1* and *m*
   . The definition of a triangle says that $r3^2 = r1^2 + m^2$
   , so $r3 = \sqrt{r1^2 + m^2}$
   (using *m* from step 8).
   $$\tan(a3) = \dfrac{m}{r1}$$

10. And again,

$$a3 = \operatorname{atan}\left(\frac{m}{r1}\right)$$

,

(again using m from step 8). But that's only the angle between vector 1 and vector 3, the actual angle will be $a3 = \operatorname{atan}\left(\frac{m}{r1}\right) + a1$.

So, for any given $r1,a1$ and $r2,a2$ we can use the calculations above (m from step 8, $r3$ from step 9, and $a3$ from step 10) to generate an $r3,a3$ to the point of intersection. There will need to be some special case handling for the end-points, for example, making the final spiral start and stop on the first and last vectors of the spiral (rather than being a point on the side).

Using the Equations

The next question is how to represent all this in the polar control. We have rather long equations based on the current and previous radius/angle vectors (implying the need to save the current vector for the next iteration) as well as special case processing for the first and last points. While it may be possible to do all this within the limits of the fields in the polar control, it is much easier to do it as new functions defined in AEL and called from within the polar control.

Define Controls

We start by defining the initial controls needed to create the shape. Define a width control to operate on the initial path:

Control:	Width
Change:	Absolute
Width:	width

Next, define a polar control with the function calls in place. In the next section we will define the actual functions in AEL. Define the polar control as:

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	PI*2*turns
Step:	PI/(sides/2); set_angle_info(0.0,PI*2*turns,PI/(sides/2))
Radius:	init_radius + (width + space) * _angle / (2*PI)
X Offset:	compute_x(_angle_i, _radius, _angle)
Y Offset:	compute_y(_angle_i, _radius, _angle)

We set the variables *start* , *stop* , and *step* the same way they were set in the spiral example in the section *First Spiral Example* (gcc) The one addition to the angle definition section is the call to the function *set_angle_info()* . The purpose of this function is to pass these three values to the AEL code so they can be saved for use in future calculations.

Create an AEL File

We must create an AEL file to hold the function definitions that we need. The AEL file begins with some global variables and the definition of *set_angle_info()* .

```
decl angle_start, angle_stop, angle_step;
decl angle_n;
decl r1;          // last saved radius
decl a1;          // last saved angle
//   r2           // current radius from the polar controller
//   a2           // current angle from the polar controller
decl r3;          // new radius
decl a3;          // new angle
**********************************************************
**********************************************************
defun set_angle_info(start, stop, step)
{
angle_start = start;
angle_stop  = stop;
angle_step  = step;
angle_n     = (angle_stop - angle_start) / angle_step;
r1 = a1     = 0;
}
```

The function saves the three value, computes the number of angle steps, and clears the last radius/angle values (which will be used later). We add to this AEL file later.

Calculate the Radius

The radius calculation is basically the same as used in the previous example. Since the radius points lie on the sides rather than on the vertex points (corners) we do not need to do the adjustments that were required in the previous example (starting at *A Short Geometry Lesson* (gcc)). In addition, we have added a new variable (*init_radius*) so we can control the radius of the first turn and therefore the overall size. Also note that in order for the *init_radius* variable to work as intended, the original graphics must be centered on the origin (0,0) which is the center of rotation for the polar control. This way, any value for *init_radius* is in effect moving the initial shape off the origin before starting to sweep it in the spiral form.

Enter the X and Y Offsets

We now have a computed radius and angle (*r2,a2*) so we can proceed with the adjustments defined by the equations above. We will use the *X Offset* and *Y Offset* fields to make the needed function calls. Of course, since the offset fields provide a delta to apply to the computed radius and angle, we will actually have to return the difference between the *r2,a2* and *r3,a3* vectors in the function calls. We will also have to convert from polar to cartesian coordinates since the vectors are in polar coordinates but the *X* and *Y Offset* are in cartesian coordinates. Given a radius and angle the basic conversion formulas are:

$$x = r \times \cos(a)$$

$$y = r \times \sin(a)$$

But since we want the difference between the two vectors, we get:

$$c = r3 \times \cos(a3) - r2 \times \cos(a2)$$

$$y = r3 \times \sin(a3) - r2 \times \sin(a2)$$

As shown above in the polar control, we call two functions: *compute_x()* and *compute_y()*
. As parameters we pass the angle count (which iteration we are on) and the *r2,a2* vector.
The support code that we will now add to our AEL file appears on the following page.

```
***********************************************
****************************************************
defun compute_x(idx, r2, a2)
{
decl x;
decl m;
if (idx == 0)              // first point
       return(0);
if (idx == angle_n + 1) // last point
       return(r1*cos(a1) - r2*cos(a2));
m = r2*sin(a2-a1)-(r1-r2*cos(a2-a1))*tan(PI/2-a2+a1);
r3 = sqrt(r1*r1+m*m);
a3 = atan(m/r1)+a1;
x = r3*cos(a3) - r2*cos(a2);
return(x);
}
****************************************************
****************************************************
defun compute_y(idx, r2, a2)
{
decl y;
if (idx == 0) {          // first point
       r1 = r2;
       a1 = a2;
       return(0);
       }
if (idx == angle_n + 1) // last point
       return(r1*sin(a1) - r2*sin(a2));
r1 = r2;
a1 = a2;
y = r3*sin(a3) - r2*sin(a2);
return(y);
}
```

The *compute_x()* function is called first. It handles the special case code for the first and
last points, and then computes *m* , which is then used to compute the *r3,a3* vector. We
then convert to a delta-X for return to the polar control. The *compute_y()* is called next.
It, too, checks for the first and last points, then saves *r2,a2* as *r1,a1* (previous vector) for
the next iteration. Finally, it converts the delta-Y for return to the polar controller.

Place the AEL file containing the code for the three defined functions (*set_angle_info()* ,
compute_x() , and *compute_y()*) in the file user_defined_fun.ael in the
$HOME/hpeesof/de/ael or $HPEESOF_DIR/custom/de/ael directory so that these functions
are loaded at run-time and are available when the model is inserted. Placing the ael file in
either of these directories will require an additional configuration var setting:

  • SITE_AEL in $HPEESOF_DIR/custom/config/de_sim.cfg pointing to
    $HPEESOF_DIR/custom/de/ael

- USER_AEL in $HOME/hpeesof/config/de_sim.cfg pointing to $HOME/hpeesof/de/ael

The Results
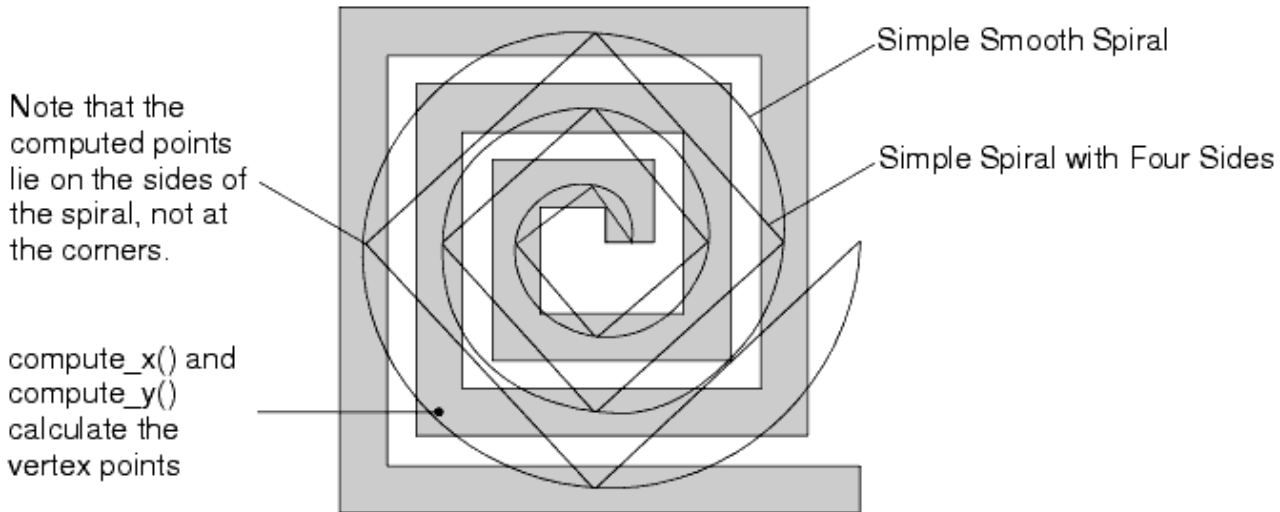
Here is an example inserted with the following parameters:

init_radius =	20.0 mil
sides =	4
turns =	3
width =	25.0 mil
space =	15.0 mil

Two additional polar controls were added to this model to help demonstrate how the geometry works. The first addition is a simple smooth spiral:

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	PI*2*turns
Step:	PI/16.0
Radius:	init_radius + (width+space) * _angle / (2*PI)
X Offset:	0.0 mil
Y Offset:	0.0 mil

The second is the same simple spiral but with only four sides, which was the basis of the model in the section *First Spiral Example.* (gcc)

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	PI*2*turns
Step:	PI/(sides/2)
Radius:	init_radius + (width + space) * _angle / (2*PI)
X Offset:	0.0 mil
Y Offset:	0.0 mil

Note that the
computed points
lie on the sides of
the spiral, not at
the corners.

compute_x() and
compute_y()
calculate the
vertex points

Simple Smooth Spiral

Simple Spiral with Four Sides

Create a Rectangular Spiral

Suppose we do not want a square spiral, but rather a rectangular spiral where we can
control the size of both the major and minor axis. The previous examples use the basic
idea of making the radius a function of angle so that as the angle increases so does the
radius. In other words, if
r = (constant) produces a circle, then
r = f(angle) produces a spiral.

We added additional math to normalize the angle, provide control over the size of the
spiral (as a function of width and space), and control the initial size of the spiral (by using
init_radius).

But if instead of a circle as the basic fundamental shape, what if we used an ellipse? We
could then modify it to increase its two axis as a function of angle as we did with the
circle.

The polar definition for an ellipse is:

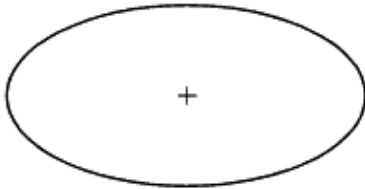$$r = \sqrt{\frac{a^2 \times b^2}{a^2 \times \sin(t)^2 + b^2 \times \cos(t)^2}}$$

where t = theta (an angle)

Create an Ellipse

Start by simply implementing the above to create a simple ellipse.

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	PI*2*turns
Step:	PI/16.0
Radius:	sqrt(a*a*b* b/(a*a*sin(_angle)*sin(_angle)+b *b*cos(_angle)*cos(_angle)))
X Offset:	0.0 mil
Y Offset:	0.0 mil

where:

turns =	1
a =	100.0 mil
b =	50.0 mil

Make an Elliptical Spiral

As we said above, a circle is:
r = (init_radius)
or
r = f(init_radius)
Where the function *f* does not actually modify the constant. Next we modified the constant so that it would increase in size as a function of the angle:

$$\cdot = f(\text{init\_radius} + (width + space) \times \text{\_angle}/(2\pi))$$

As stated above an ellipse is a more complicated function in terms of *a* and *b* :
r = f(a, b)
Where a and *b* serve the same purpose as *init_radius* . So, if we increase them as a function of angle in the same way, we should get the elliptical spiral we want:
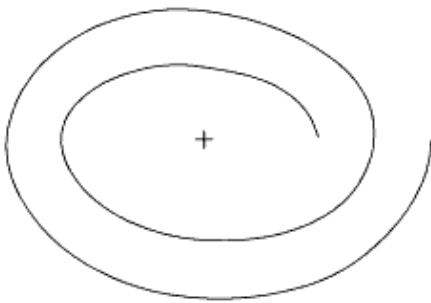
$$\cdot = f(a + (width + space) \times \text{\_angle}/(2\pi))$$

$$\text{\textit{b}} + (width + space) \times \text{\_angle}/(2\pi)$$

where *f()* is the above definition of an ellipse.
So now we have:

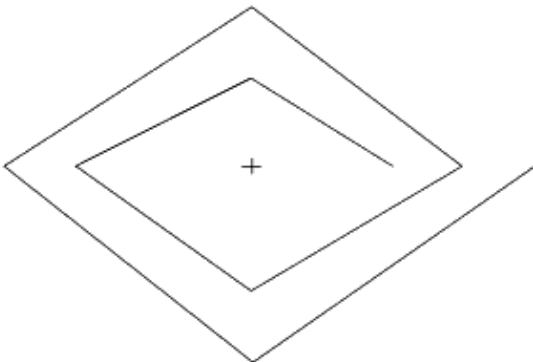| Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	PI*2*turns
Step:	PI/16.0
Radius:	sqrt(pow(a+(width+space)*_angle/(2*PI),2)* pow(b+(width+space)*_angle/(2*PI),2)/ (pow(a+(width+space)*_angle/(2*PI),2)* pow(sin(_angle),2)+ pow(b+(width+space)*_angle/(2*PI),2)* pow(cos(_angle),2)))
X Offset:	0.0 mil
Y Offset:	0.0 mil

where:

| turns =	1
a =	100.0 mil
b =	50.0 mil
width =	25.0 mil
space =	25.0 mil

Reduce the Number of Sides

Next we reduce the number of sides to four, and we have our points *on* the sides of the spiral.

Add Function Calls

Control:	Polar
Delete Ends:	True
Units:	Radians
Start:	0.0
Stop:	PI*2*turns
Step:	PI/2.0; set_angle_info(0.0, PI*2*turns, PI/2)
Radius:	sqrt(pow(a+(width+space)*_angle/(2*PI),2)* pow(b+(width+space)*_angle/(2*PI),2)/ (pow(a+(width+space)*_angle/(2*PI),2)* pow(sin(_angle),2)+ pow(b+(width+space)*_angle/(2*PI),2)* pow(cos(_angle),2)))
X Offset:	compute_x(_angle_i, _radius, _angle)
Y Offset:	compute_y(_angle_i, _radius, _angle)
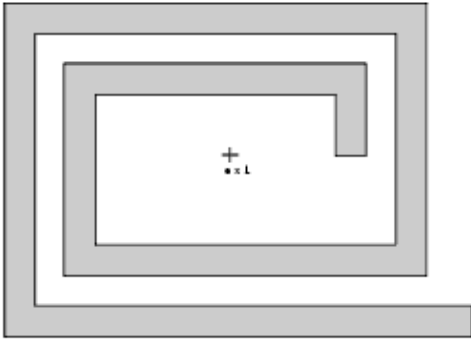
Basic PAM Example

Example: Creating a Multi-Coupled Line

The information in this example is presented with the assumption that you already know how to insert and work with shapes in a Layout window. If you do not, refer to the section *Defining Artwork* (gcc)

The Graphical Cell Compiler uses primitive graphical shapes to build AEL macros that create models. For this example, the source artwork comprises a rectangle and two pins (connectors). You add two control lines (construction lines) and define three controls:

- A stretch control for length that acts on the rectangle and the two pins.
- A stretch control for width that acts only on the rectangle.
- A repeat control for the number of lines to create that acts on the rectangle and the two pins.

> **ⓘ Note**
> The section *Code Walkthrough* (gcc) contains a walkthrough of the code that is generated to produce this model.

The steps to create a multi-coupled line are:
Step 1 Defining the Artwork
Step 2 Defining the First Stretch Control
Step 3 Defining the Second Stretch Control
Step 4 Defining a Repeat Control
Step 5 Viewing the Defined Controls
Step 6 Compiling the Macro
Step 7 Defining Parameter Default Values
Step 8 Using the New Component

Step 1 Defining the Artwork

The first step in the process is to define the source artwork.

1. In the Main window, open a new project (**File** > **New Project**).
2. In the new project, open a Layout window (**Window** > **Layout**).
3. Using the relative cursor readout (Differential coordinates) displayed at the bottom of the Layout window, insert a 100 x 50 mil rectangle (**Insert** > **Rectangle**).
4. Add two ports, one at the each end of the rectangle (**Insert** > **Component** > **Port**) and place two construction lines through the center of the rectangle, one horizontal and one vertical (**Insert** > **Construction Line**). Construction lines provide a reference for the controls that you define.
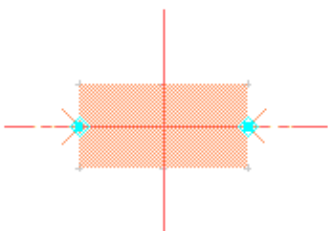
Step 2 Defining the First Stretch Control

Next, define a stretch control for length that acts on the rectangle and the two pins.

1. Select the **rectangle**, **both ports**, and the **vertical construction line**.
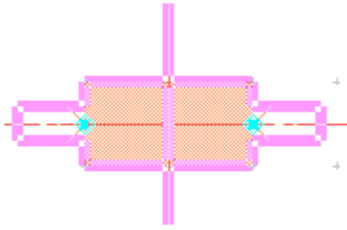
2. Select **Macro** > **Stretch** to open the Stretch Control Definition dialog box. Notice that the Stretch Orientation is shown as Positive, with respect to the vertical reference line.
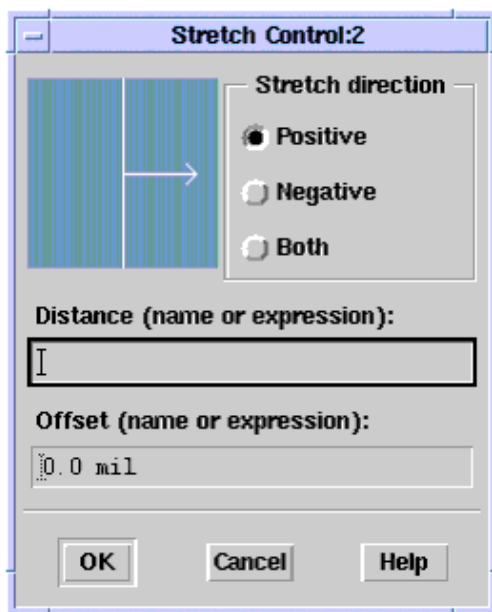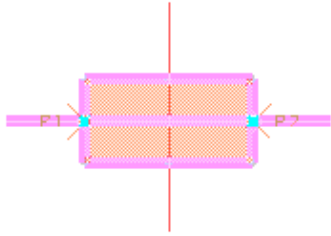

3. Select **Both** so that the stretch orientation goes both positive and negative. This produces a symmetrical stretch on each side of the reference line.
4. In the Distance field, enter the word **length** to define the name of the component parameter that adjusts the symmetrical stretch of the rectangle.
5. In the Offset field, enter **100.0 mil**.
   Offset compensates for the original size of the shape, making it possible for you to create a PAM using a shape with some size (so it is easy to work with), and then use the PAM without needing to know the initial size of the shape. In this example, you are creating a PAM using a rectangle of length 100. When you set the offset at 100, you are offsetting the original length. Then when you use the PAM and enter a length of 50, you get a rectangle with a total length of 50 rather than 150.
6. Click OK to accept the settings.

> **ⓘ Note**
> For details on the Stretch control, see *Stretch Control.* (gcc)

Step 3 Defining the Second Stretch Control

1. Select the **rectangle** and the **horizontal construction line**.

2. Select **Macro** > **Stretch**.
3. Choose the Stretch direction **Both** to produce a symmetrical stretch on each side of the reference line.
4. In the Distance field, enter the word **width**.
5. In the Offset field, enter **50.0 mil**.
6. Click OK to accept the settings.

Step 4 Defining a Repeat Control

Now you set a repeat control for the number of lines to create for acting on the rectangle and the two pins.

1. Select the rectangle, both connectors, and the vertical construction line.
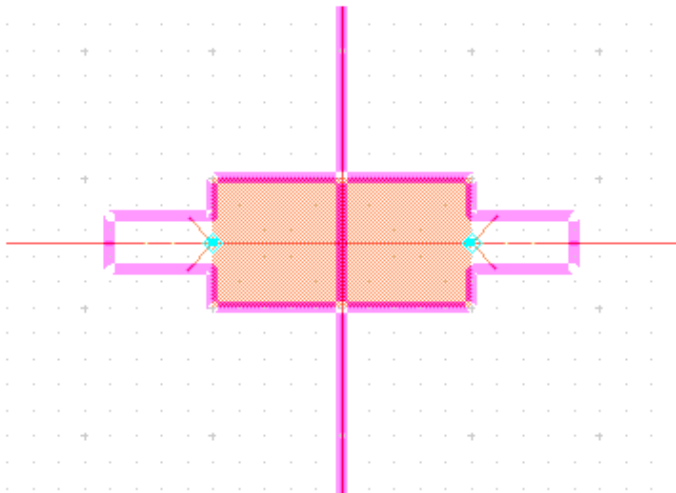
2. Select **Macro** > **Repeat** to open the Repeat Control dialog.

Using the Repeat Control Definition dialog, you can define parameters that duplicate the shape. The selections in the Repeat Direction area define the direction in which copies of the shape are placed:
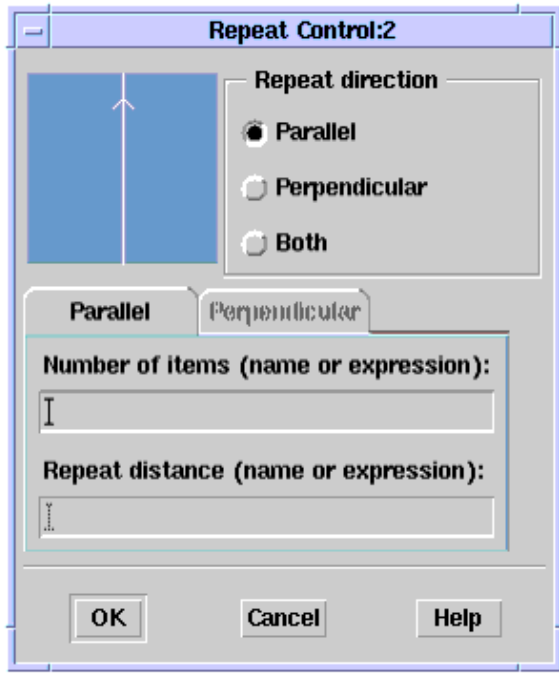
- Parallel places copies parallel to the selected reference line.
- Perpendicular places copies perpendicular to the selected reference line.
  In this case, leave **Parallel** selected.

3. In the Number of Components field, enter the word **number** to define the component parameter that controls the number of times the shape is duplicated.
4. In the Repeat distance field, enter **width+space** to create the component parameter space.
   The repeat distance is the distance from the beginning of one copy of the shape to the beginning of the next (including the space between the shapes).
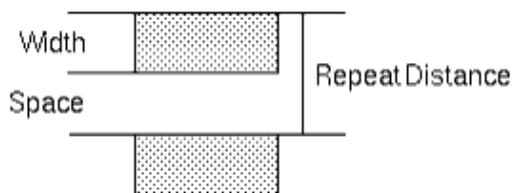
> ⚠️ **Caution**
> When defining a component parameter, do not use the word *step* or any other word that AEL recognizes as a variable/reserved word. Using such a word generates an error when you try to use the PAM. For more information, see *GCC Error Messages* (gcc) and *AEL* (ael).
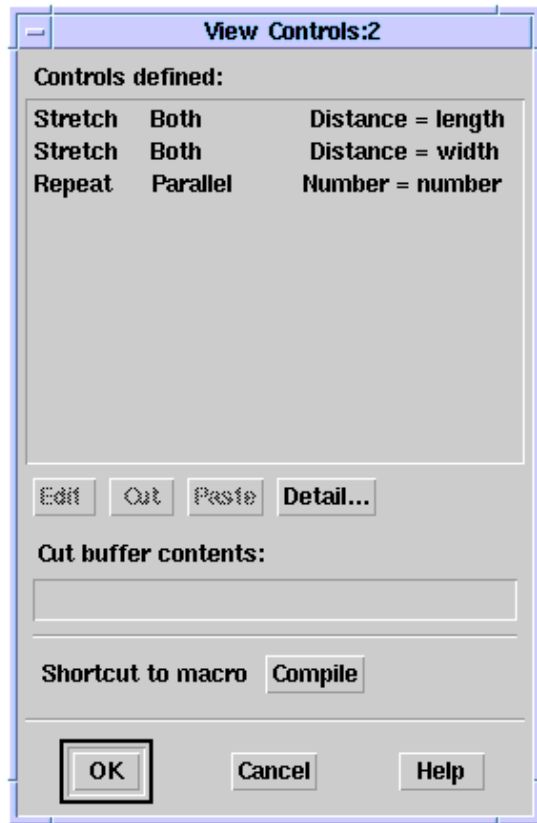
5. Click OK to accept the settings **.**

> ℹ️ **Note**
> For details on this control, see *Repeat Control.* (gcc)

Step 5 Viewing the Defined Controls

You can view the contents of the controls that you defined, as well as the detail of each control.

1. Select **Macro** > **View/Edit** to open the View Controls dialog box.

View Controls:2

Controls defined:

Stretch	Both	Distance = length
Stretch	Both	Distance = width
Repeat	Parallel	Number = number

Edit Cut Paste **Detail...**

Cut buffer contents:

Shortcut to macro Compile

OK   Cancel   Help

The three controls you set in the previous steps are displayed in the View Controls dialog: two stretches in both directions (with respect to the reference line) and a repeat in the Parallel direction (with respect to the reference line).
The program executes the controls in the order in which these are listed in this dialog. Selecting a control activates the editing capabilities. In this example, leave the controls as set.

2. Select **Detail** to open the Detail window. Each control is listed, along with the defined parameters.

```
 ┌─────────────────────────────────────────────────────┐
 │ ▄                       Detail:2                     │
 ├─────────────────────────────────────────────────────┤
 │  Controller detail list:                            │
 │  ┌──────────────────────────────────────────────┐▲  │
 │  │ Control:  Stretch                             ││  │
 │  │         Direction:     Both                   ││  │
 │  │         Length:        length                 ││  │
 │  │         Offset:        100.0 mil              ││  │
 │  │                                               ││  │
 │  │ Control:  Stretch                             ││  │
 │  │         Direction:     Both                   ││  │
 │  │         Length:        width                  ││  │
 │  │         Offset:        50.0 mil               ││  │
 │  │                                               ││  │
 │  │ Control:  Repeat                              ││  │
 │  │         Direction:     Parallel               ││  │
 │  │         Parallel                              ││  │
 │  │            Number:     number                 ││  │
 │  │            Distance:   width+space            ││  │
 │  │                                               ││  │
 │  │                                               ││  │
 │  │                                               ││  │
 │  │                                               ││  │
 │  │                                               ││  │
 │  │                                               ││  │
 │  │                                               │▼  │
 │  └──────────────────────────────────────────────┘   │
 │  ◁▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▷       │
 │                                                     │
 │   ┌────────┐       ┌─────────┐      ┌─────────┐     │
 │   │   OK   │       │ Refresh │      │  Print  │     │
 │   └────────┘       └─────────┘      └─────────┘     │
 └─────────────────────────────────────────────────────┘
```
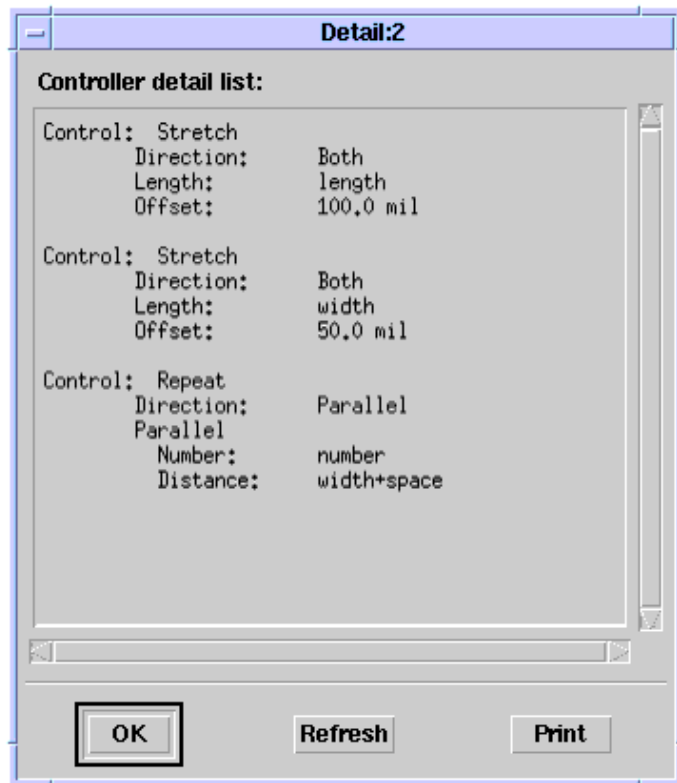
3. In the Detail dialog, click OK to dismiss the dialog.
4. In the View Controls Dialog box, click OK to dismiss the dialog.

> ⚠ **Caution**
> Always define Stretch controls first, then define Repeat controls. After you have completed this example, go back and switch the order of the controls (using Cut/Paste in the View Controls dialog). Then recompile and experiment with the modified component to see what happens when a shape is repeated *before* it is resized (stretched). Because of the way the program handles copies, you can see duplicates of the original rectangle. To stretch duplicates, stretch the original, then duplicate the stretched original. For more information, see *Defining Controls.* (gcc)

> ⓘ **Note**
> For details on the View Control dialog box, see the section *Viewing Controls.* (gcc)

Step 6 Compiling the Macro

After you have defined the controls, you can compile the macro.

Several names are associated with a PAM:

Design name is the name of the source design; that is, the name of the layout in which you create the source artwork.

Output file name is the name used for the AEL file that defines the macro. The default name is < *design name* >_art.ael.

Model name is the name that you select from a library to use the macro in a new layout. The default model name is < *design name* >.

To compile a macro:

1. Select **File** > **Save Design** to save the source design in the Layout window.
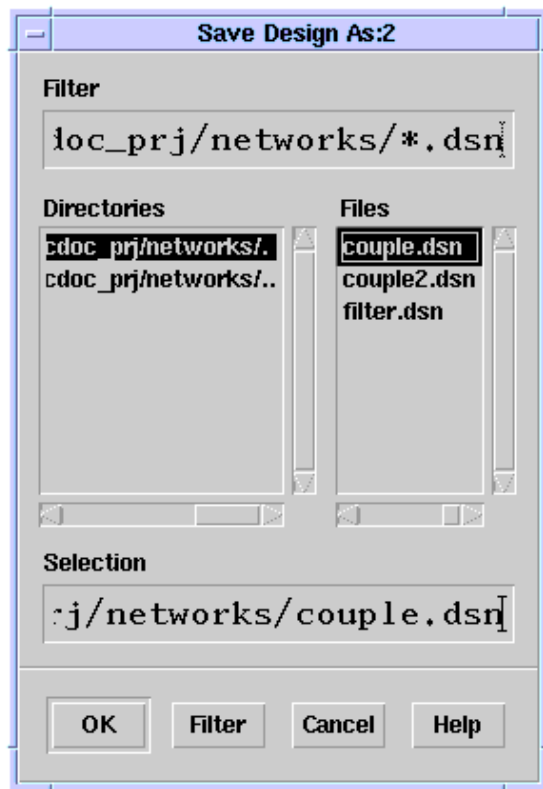
Note that this *does not* save the PAM. The Compile Messages area displays the message:

Design saved as: *< path >/< design name >*.

2. Select **Macro** > **Compile** to open the Graphical Cell Compiler dialog **.**
3. Select **Compile**.

The Compile Messages area displays the progress and results as the PAM compiles.

> **ⓘ Note**
> For details on this dialog box, see the section *Compiling a Macro.* (gcc)

Step 7 Defining Parameter Default Values

After you have compiled the macro, you can set the parameter default values and save the macro.

1. In the Graphical Cell Compiler dialog box, select **Design/Parameters** to open the Design Definition dialog.
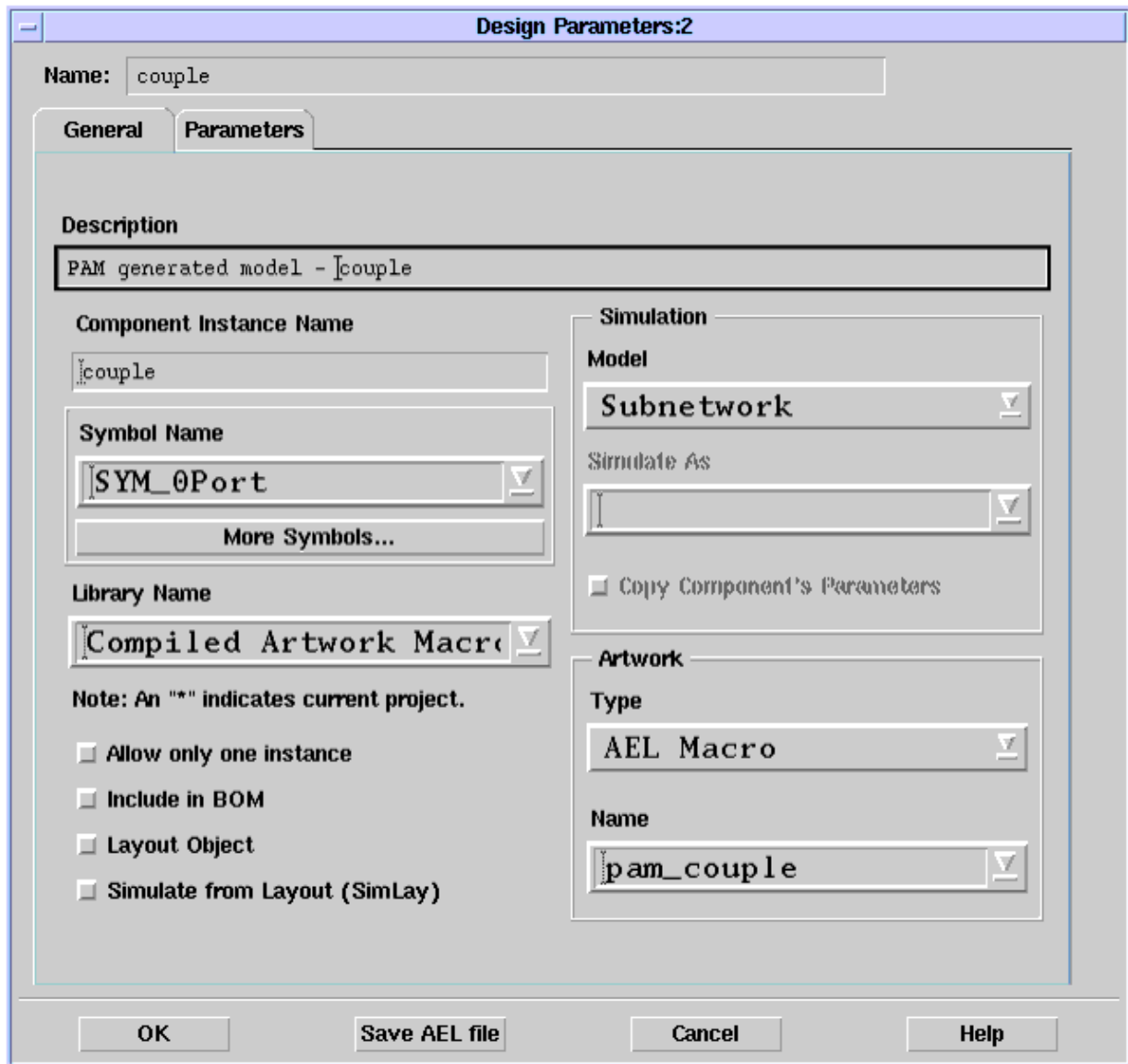
The Model Name that you defined in the PAM Compiler dialog box appears as the Name and as the Component Instance Name. A description is provided automatically, as is the default library name for PAMs. The name and type of artwork are entered automatically under Artwork.

2. Select the **Parameters** tab. The parameters you defined in a previous step are listed in the **Select Parameter** field.
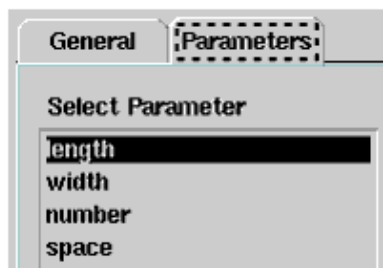
3. The length parameter is selected. In the Default Value field, enter **100.0 mil**.

4. Select the parameter **width**, and in the Default Value field, enter **40.0 mil**.

5. Select the parameter **number**, and in the Default Value field, enter **2**.
6. In the Parameter Type field, select **Unitless** from the drop-down list.

> ℹ️ **Note**
> If this field is *not* specified as unitless, the model may not appear when you try to use the PAM.

7. Select the parameter **space**, and in the Default Value field, enter **25.0 mil**.
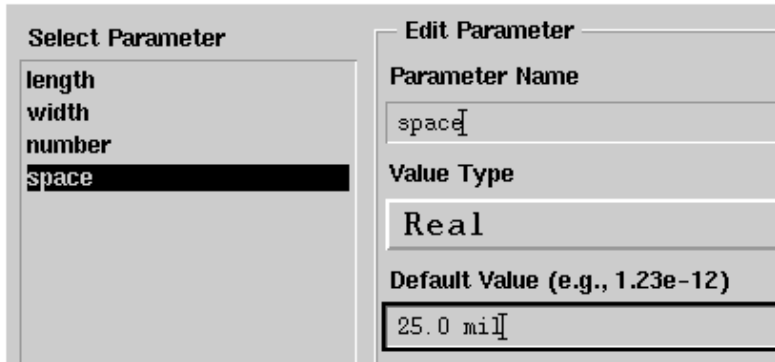
8. Select **Save AEL file**, then click OK to save the AEL file.

9. Click OK to close the Design Definition dialog box.
10. Click OK to close the Graphical Cell Compiler dialog box.

> **ℹ Note**
> For details on this dialog box, see the section *Compiling a Macro.* (gcc)

Step 8 Using the New Component

Now that you have created the new component (PAM), you can use it in a layout.

1. In the Main window, open a new layout window (**Window** > **New Layout**).
2. In the new Layout window, choose **Insert** > **Component** > **Component Library**.
3. In the Library List dialog box, select **Compiled Artwork Macros** to view the Component Library List. The component you have just defined is listed.
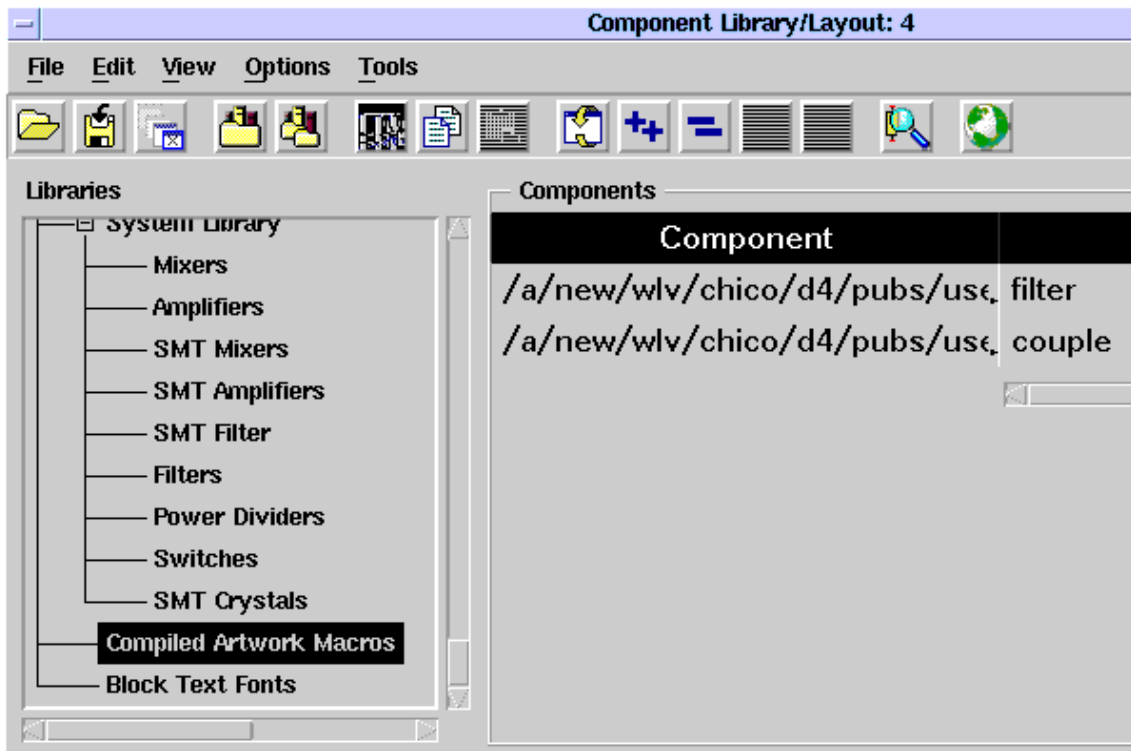
> ✔ **Hint**
> For a shortcut, you can type the component name into the component history pulldown on the toolbar.

4. Select the component. If the Edit Component Parameters dialog opens, click OK to accept the default parameters.

Instance Name (name[<start:stop>])

```
couple1
```

Select Parameter

length (Real, e.g. 1.25)

```
length=100.0 mil
width=40.0 mil
number=2
space=25.0 mil
```

```
100.0          mil      ▽
```
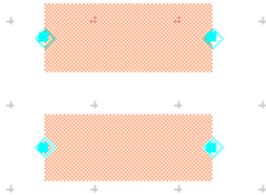
Equation Editor...

Optimization/Statistics/DOE Setup...

✗ Display parameter on schematic

| Add | Cut | Paste | Component Options... |

```
length : Generated parameter: length
```

5. Move the cursor to the Layout window and place the component. Then cancel the command by clicking the Cancel Command button on the tool bar.

6. Close the Component Library dialog.