



Defining and Developing Libraries

Product Version 23.1
September 2023

© 2024 Cadence Design Systems, Inc.
Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522.

All other trademarks are the property of their respective holders.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information. Cadence is committed to using respectful language in our code and communications. We are also active in the removal and replacement of inappropriate language from existing content. This product documentation may however contain material that is no longer considered appropriate but still reflects long-standing industry terminology. Such content will be addressed at a time when the related software can be updated without end-user impact.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Contents

1		14
Defining and Developing Libraries Overview		14
The Library Development Process Model		16
Library Development Tasks		16
Library Development Tools		16
2		21
Library Development Process		21
Library Development Tasks		23
Creating Libraries		23
Related Topics		24
Library Files		24
Related Topics		25
Library Development Tools		25
3		28
Library Padstacks Overview		28
Library and Layout Padstacks		30
Library Padstacks		30
Layout Padstacks		30
Related Topics		31
Padstacks and Pins		31
Standard Pad Shapes		32
Photoplot Pad Data		34
Defining Library Padstacks		36
Preparing to Define Padstacks		36
Using the Padstack Editor		37
Recording a Padstack Script		41
Managing Padstack Data Using XML Files		41
Updating Layout Padstacks		53
Custom Pad Shape Symbols		54
Suppressing Unused Padstacks		55
Purging Unused Padstacks		56
Creating and Using Structures		57

4		58
Working with Symbols		58
Working with the Symbol Mode		58
Symbol Types		58
Symbol and Drawing Files		60
The Symbol Library		61
Legal Classes for Symbol Types		62
Creating a Symbol		62
Adding Areas		63
Adding Pins		64
ETCH/CONDUCTOR and Vias in Symbols		64
Mapping STEP Models		65
Creating Package Symbols		68
Package Symbol Elements		68
Prerequisites for Creating a Package Symbol		70
Guidelines for Creating Package Symbols		70
Defining a Package Symbol		71
Defining Symbol Heights		71
Defining Component Heights with Properties		75
Creating Mechanical Symbols		78
Types of Mechanical Symbols		79
Guidelines for Creating Mechanical Symbols		79
Creating a Board Outline		81
Creating Format Symbols		82
Library Format Symbols		82
Guidelines for Creating Format Symbols		83
Creating Flash Symbols		84
Choosing a Design Methodology for Negative Planes		85
Converting Flash Symbols When Migrating Databases		86
Flash Symbols in Padstack Designer		87
Treatment of Nonconforming Symbols		87
MDA Format Output Files		88
Defining a Flash Symbol		88
Updating Symbols		89
5		90
Checking Symbols Automatically		90

Related Topics	90
Configuring the check symbol Command	90
Globals File	90
Rule Table File	91
Developing Symbol Check Rules	92
Installing Custom Rules	95
Predefined Rules for Checking Symbols	97
REFDES Checks	97
COMPONENT VALUE Checks	98
DEVICE TYPE Checks	99
TOLERANCE Checks	100
USER PART NUMBER Checks	101
GEOMETRY Checks	102
Reports	103
Global Variables and Values	104
6	110
Preparing Device Files	110
Checking a Device File	112
Reviewing the dev_check.log File	112
Creating a Device File	117
Specifying Multiple Functions in a Device File	117
Specifying Definition Properties in a Device File	119
Device File Records	120
Device File Format	121
Syntax and Field Descriptions	122
Guidelines for Creating a Device File	129
7	130
Using Technology and Parameter Files	130
Working with Tech Files	130
Accessing Parameter Files	132
Exporting Parameter Files	132
Importing Parameter Files	132
Accessing Tech Files	133
Exporting Tech Files	133
Importing Tech Files	133
Comparing Tech Files to Designs	133

Uprevng Tech Files	134
Locking Constraint Sets	135
Technology Constraints File	136
Working with Parameter Files	137
Parameter File Syntax	137
8	139
Generating Libraries	139
Creating a Clipboard Library	141
Setting the CLIPPATH Environment	142
Creating Libraries from Existing Designs	143
Creating Device and Symbol Files with Batch Commands	143
9	144
APD: Using LEF/DEF Files (APD XL)	144
Planning the Die Rings for an IC Layout Already Underway	144
Prototyping Die Pins for an IC Not Yet Laid Out in an SP&R Tool	145
Exporting Changes from APD to an IC Design	145
Related Topics	145
About Exchange Format Files	145
LEF and DEF File Fields	146
LEF Macro Data Identifiers	151
LEF Macro Pin Identifiers	152
LEF File Site Identifiers	154
LEF/DEF Standard Component Orientation Syntax	155
About the Condensed Macro Library Format File	157
About the Library Definition File	162
Condensed Macro Library Creation Messages	164
Information	164
Error	164
Log File	164
Exporting Data with the LEF/DEF Interface	165
Importing Data with the LEF/DEF Interface	168
LEF/DEF Message Generation	169
Import Errors	169
Export Errors	171
LEF Library Manager Messages	173
Information	173

Warning	173
Error	173
Using the LEF Library Manager	175
10	177
APD: Using the Package Designer Symbol Editors	177
Feature Set	177
BGA Editing Flows	180
Multiple Grids and Pin Number Patterning	183
How Multiple Grids Work	184
Pin Pattern Numbering with Multiple Grids	185
The Item Information Window	185
Package Designer Editors: Operating Parameters	187
11	191
Creating Jumper Package Symbols	191
Jumper Package Symbol Elements	191
Prerequisites for Creating a Jumper Package Symbol	191
Creating a Jumper Package Symbol	192
Defining the Drawing Type	192
Defining the Via List	192
Adding Vias	193
Adding Reference Designators	193
Creating Package Boundary	193
12	196
STEP Model Support	196
STEP Models	196
Environment Variables for STEP Model Support	196
Related Topics	197
Exporting Board Drawings to a STEP Model	198
Mapping Devices and Symbols to STEP Models	201
Mapping Mechanical Symbols to STEP Models	208
Viewing STEP Models in 3D Canvas	211
Appendix A: Package/Component Symbol Library	213
Capacitors	213
cap300	213
cap400	214

cap600	215
dipcap	215
smdcap	215
cap196	216
cap1000	216
cap1500	216
capck05	216
capck06	217
capck60	217
capck62	218
case17-02	219
ck12-10pf	219
ck13-10pf	220
ck14-10pf	220
ck15-10pf	220
ck16-10pf	220
ck17-10pf	221
cy10	221
cy15	221
cy20	221
Resistors	222
res400	222
res500	222
res1000	223
res800	223
resadj	224
smdres	224
SIPs	225
sip6	225
sip8	226
sip10	227
sip12	228
sip30	228
Connectors	230
conn6	230
conn9	230
conn10	231

conn20	232
conn26	233
conn50	233
multiconn30	233
multicon43	234
ibmconn	234
db9	234
db15	235
db25	235
eurocon	235
Crystals	237
crys11mhz	237
crys14	238
Diodes	239
do5	239
do13	239
do35	241
do41	241
dio400	242
dio500	243
Potentiometer	244
pot	244
Test Point	246
tp	246
Switches	246
dipswitch	246
switch	248
DIPs	248
dip32_6	248
dip4_3	249
dip6_3	249
dip8_3	249
dip10_3	250
dip14_3	250
dip16_3	251
dip18_3	252
dip18_4	253

dip20_3	254
dip20_4	255
dip20_6	256
dip22_3	257
dip22_4	257
dip22_6	258
dip24_3	259
dip24_4	259
dip24_6	260
dip26_3	260
dip28_3	261
dip28_6	262
dip40_6	263
dip48_6	264
dip52_6	265
dip64_6	266
dip68_6	268
Jumpers	269
jumper1	269
jumper2	269
jumper3	270
jumper4	270
jumper5	271
jumper8	271
jumper14	272
jumper16	272
Pin Grid Arrays	273
pga68	273
pga84	274
pga100	275
pga101	276
pga120	277
pga124	278
pga132	279
pga132_ci	280
pga132-x	281
pga133	282

pga156	283
pga156_x	284
pga172	284
pga176	286
PLCCs	287
picc18	287
plcc20	288
plcc28	289
picc32	290
plcc44	291
plcc48	292
picc52	293
picc68	294
picc84	295
icc20	296
icc24	297
icc28	298
icc24	299
icc48	300
iccs18	300
iccs22	301
iccs24	303
iccs28	303
iccs32	304
SOICs	305
soic8	305
soic14	306
soic16	306
soic16w	307
soic20	309
soic20w	309
soic24	310
soic24w	311
soic28w	312
sol120	314
Transistors	314
sot23	314

sot89	315
to3	315
to5	316
to12	316
to18	317
to39	317
to46	318
to52	318
to92	319
to107	319
to126	320
to126h	320
to126v	321
to204aa	322
to220abh	323
to220abv	323
to220h	324
to220v	325
Flat Packs	326
flat14	326
flat16	327
flat18	329
flat20	331
flat24	333
flat28	335
cpfp68	336
quadflat24	336
ZIPs	337
zip16	337
zip20	338
Appendix B: PCB Editor, Mechanical Symbol Library	341
Card Outlines	341
ibm	341
multibus	342
euros	343
eurod	344

Mounting Holes	345
mtq125	345
mtq156	346
mtq250	347
Appendix C: PCB Editor, Format Symbol Library	349
Target	349
target	349
Drawing Formats	349
asizev	350
asizeh	351
bsize	352
csize	353
dsize	354
esize	355

Defining and Developing Libraries Overview

The first step in a typical PCB physical design process is creating libraries, which are a collection of graphic symbols that represent packages, mechanical elements, drawing formats, and custom pads and padstacks. Library development is the process of updating the libraries of padstacks and symbols that are part of the installation of your layout editor.

The library consists of padstack, symbol, and device files in the library directory:

```
<install_dir>/share/pcb/pcb_lib
```

The library directory contains two subdirectories:

- symbols
- devices

The `symbols` subdirectory contains:

- Padstack files that define physical pad sizes and shapes. The characteristics of each padstack are contained in a `.pad` file, which contains:
 - Padstack name
 - Top, bottom, and internal pad data
 - Solder paste and film mask data
 - Drill hole information
- Symbol files for:
 - Packages that are physical representation of components, such as dual in-line packages, resistors, capacitors, and edge connectors
Package symbols have a reference designator label and at least one pin with a pin number.
 - Mechanical elements that are usually non-electrical, such as board outlines, mounting holes, plating bars, and card ejectors
Mechanical-only fixtures with drill holes have pins with no pin numbers. Mechanical symbols do not have a reference designator label.

- Drawing formats of standard drawing sizes that include borders, title blocks, notes, revision blocks, and other types of drawing information

The `devices` subdirectory contains:

- Device files that define logic information for certain component types

The library consists of several types of files, each identified by different file extensions:

Padstacks	.pad
Custom pad shapes	.dra, .ssm
Package symbols	.dra, .psm
Mechanical symbols	.dra, .bsm
Format symbols	.dra, .osm
Flash symbols	.dra, .fsm
Devices	.txt

The Library Development Process Model

- Update the libraries
 - Add new library padstacks and symbols
 - Add unique (that is, custom) pads used in padstack definitions that require a unique pad shape instead of one of the standard geometries
 - Edit padstacks and symbols
- Create libraries containing
 - Project-specific symbols
 - Device files containing logic information for any unique package symbols used in designs
 - Technology files consisting of specific design rules and constraints that can be applied to other designs

Library Development Tasks

When creating libraries for your layout editor:

- Check and review the supplied library.
- Create padstack library.
You must define padstacks before you create package symbols because each pin in a package symbol must have an associated padstack.
- Create package symbol .psm library.
- Create mechanical symbols (.bsm).
- Create format symbols (.osm).
- Create and check device files.
- Store the library files in the appropriate directory.

Library Development Tools

The following tools are available for creating, editing, and updating library components:

- **Pad Designer**

The Pad Designer lets you create and edit padstacks and save them to a design, to a library, or to both at once. You can run the Padstack Designer as a stand-alone tool or using the *Tools – Padstack – Modify Library Padstack* (`editpadlib`) menu command in your layout editor.

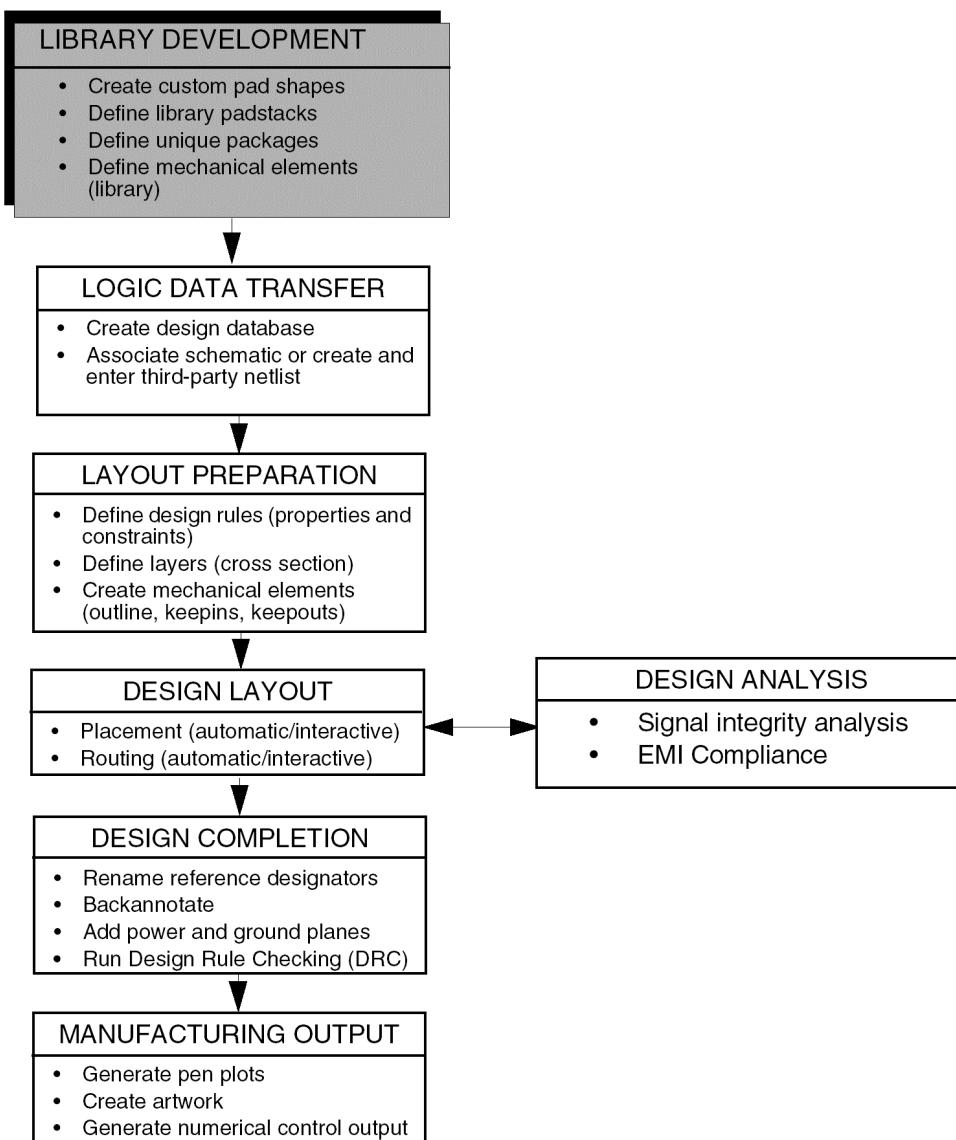
- **Symbol Editor**

The layout editor lets you create and/or edit symbols from one of five symbol modes:

- Package (or Package Wizard)
- Mechanical
- Format
- Shape
- Flash

Library padstack definition and symbol creation occur at the beginning of the design flow, as shown in the following figure.

Library Development in a Design Flow



You can define new libraries based on the libraries that are associated with existing designs. This feature is useful for:

- Creating a library for a design that originates from a different CAE or physical layout system, for which the original library is not provided
- Creating a library that is compatible with the padstacks and symbols that are already in a design, such as when the design is from an earlier library revision

- Recovering libraries that may have been lost, by obtaining the library data from a design that contains the correct library data
- Creating a clipboard library that contains elements from designs or symbol drawings

You can create libraries from existing designs by extracting:

- Device files
- Padstacks
- Symbols

You do this by using the `dlib` command to obtain device files, padstack definitions and symbol definitions from an existing layout, or by running the `create_devices` and `create_sym` batch programs to obtain device files and symbol definitions from an existing layout.

 Creating new libraries based on existing designs occurs late in a design flow or following its completion.

Defining and Developing Libraries

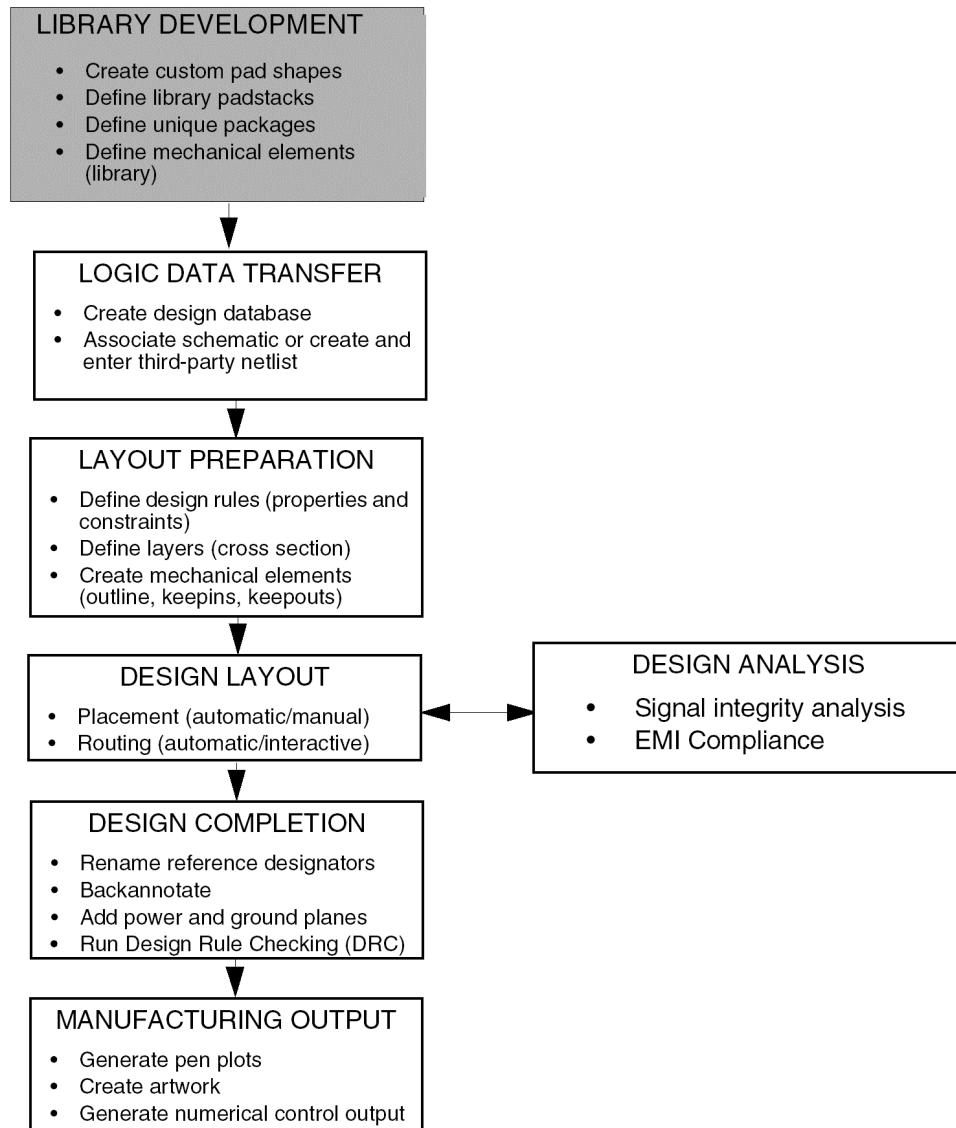
Defining and Developing Libraries Overview--The Library Development Process Model

Library Development Process

The first step in a typical PCB physical design process is to create libraries, which are collections of graphic symbols representing packages, mechanical elements, drawing formats, and custom pads and padstacks.

To develop libraries, you update the libraries of padstacks and symbols that are installed with the layout editor. Library padstack definition and symbol creation occur at the beginning of the design flow, as follows.

Library Development in a Design Flow



Library Development Tasks

The library development process consists of these tasks:

- Update the libraries provided with your layout editor.
 - Review the libraries
 - Add new library padstacks and symbols
 - Add unique (that is, custom) pads used in padstack definitions that require a unique pad shape instead of one of the standard geometries
 - Edit padstacks and symbols
- Create libraries containing:
 - Project-specific symbols
You must define padstacks before you create package symbols because each pin in a package symbol must have an associated padstack.
 - Device files containing logic information for any unique package symbols used in designs
 - Technology files consisting of specific design rules and constraints that can be applied to other designs

Creating Libraries

Later in a design flow or following its completion, you can define new libraries based on the libraries that are associated with existing designs. This feature is useful for:

- Creating a library for a design that originates from a different CAE or physical layout system, for which the original library is not provided
- Creating a library that is compatible with the padstacks and symbols that are already in a design, such as when the design is from an earlier library revision
- Recovering libraries that may have been lost, by obtaining the library data from a design that contains the correct library data
- Creating a clipboard library that contains elements from designs or symbol drawings

You can create libraries from existing designs by extracting:

- Device files

- Padstacks
- Symbols

Choose *File – Export – Libraries* (`dlib` command) to obtain device files, padstack definitions and symbol definitions from an existing layout, or by running the `create_devices` and (*File – Create Symbol*) `create_sym` batch commands to obtain device files and symbol definitions from an existing layout.

Related Topics

- [create_devices](#)
- [create_sym](#)

Library Files

The editor library consists of padstack, symbol, and device files supplied in the library directory:

`<install_dir>/share/lib/pcb/pcb_lib`

The library directory contains two subdirectories: `devices` and `symbols`.

The `devices` subdirectory contains device files that define logic information for certain component types.

The `symbols` subdirectory contains:

- Padstack files that define physical pad sizes and shapes. The characteristics of each padstack are contained in a `.pad` file, which contains:
 - Padstack name
 - Top, bottom, and internal pad data
 - Solder paste and film mask data
 - Drill hole information
- Symbol files for:
 - Packages that are physical representation of components, such as dual in-line packages, resistors, capacitors, and edge connectors
Package symbols have a reference designator label and at least one pin with a pin

number.

- Mechanical elements that are usually non-electrical, such as board outlines, mounting holes, plating bars, and card ejectors
Mechanical-only fixtures with drill holes have pins with no pin numbers. Mechanical symbols do not have a reference designator label.
- Drawing formats of standard drawing sizes that include borders, title blocks, notes, revision blocks, and other types of drawing information

The editor library consists of several types of files, each identified by different file extensions:

File Type	Extension
Padstacks	.pad
Custom pad shapes	.dra, .ssm
Package symbols	.dra, .psm
Mechanical symbols	.dra, .bsm
Format symbols	.dra, .osm
Flash symbols	.dra, .fsm
Devices	.txt

Related Topics

- [create_sym](#)

Library Development Tools

The layout editor supplies the following tools for creating, editing, and updating library components:

- Pad Designer

The Pad Designer lets you create and edit padstacks and save them to design, to a library, or to both at once. You can run the Padstack Designer as a standalone tool or choose *Tools – Padstack – Modify Design Padstack* ([pateditdb](#) command) and *Tools – Padstack – Modify Library Padstack* ([pateditlib](#) command) as you modify padstacks.

- **Symbol Editor**

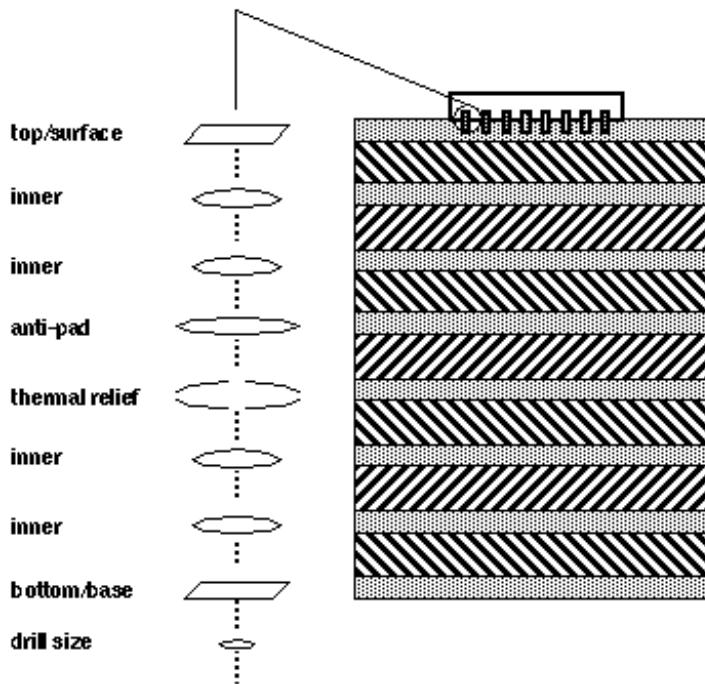
You can create and edit symbols from one of five symbol modes:

- Package (or Package Wizard)
- Mechanical
- Format
- Shape
- Flash

Library Padstacks Overview

Each symbol pin in a design must have a padstack associated with it. The padstack describes how the symbol pin connects to each layer in the design. You define new library padstacks open using the [padstack_editor](#) command.

Padstack Definition Expanded on an Eight-Layer Board



A padstack is a file that contains the following information for each layer:

- Pad size and shape
- Drill size and drill display figure

A padstack also describes the following information for the TOP and BOTTOM layers:

- Soldermask
- PasteMask

- Filmmask

A padstack can contain up to 32 user-defined mask layers. A padstack can also contain Numerical Control (NC) drill data, which the layout editor uses to create drill drawings, described in the *Preparing Manufacturing Data* user guide in your documentation set.

Library and Layout Padstacks

The layout editor categorizes padstacks as follows:

Library Padstacks

Library padstacks are padstack definitions contained in the editor library or a user-defined library directory.

The layout editor supplies a set of standard padstacks in the editor library. You can create new library padstacks with the Pad Editor.

Library padstacks are generic in that they define pad data for beginning (TOP) and ending (BOTTOM) layers. In addition, library padstacks have one default set of pad data that can apply to all internal layers.

You can also add other layers to a library padstack definition. The layout editor uses the pad data for these layers only when those same layers are defined in the layout.

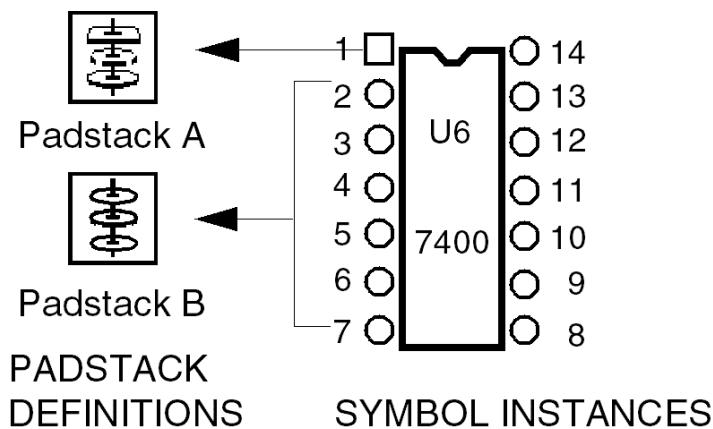
The first time the layout editor uses a library padstack in a design, it copies the generic pad data for internal layers from the library to all internal layers defined for the design. Also, as you add new ETCH/CONDUCTOR subclasses to a design, the editor adds the generic pad data for internal layers to each padstack for each new layer.

Layout Padstacks

A layout padstack is a padstack definition associated with a pin or via in a design.

Once a design contains padstack definitions, those padstacks are considered layout padstacks. Choose *Tools – Padstack – Modify Design Padstack* (`pateditdb` command) to define different sets of pad data for internal layers in layout padstacks.

Pins share layout padstack definitions. All pins with the same padstack name refer to the same padstack definition stored in the layout, as the following figure shows:



Related Topics

- [pateditdb](#)

Padstacks and Pins

In the layout editor, each pin in a symbol must have an associated padstack name. When you create a symbol, you add pins to the package symbol drawing. As you add each pin, the editor finds the library padstack, copies the padstack definition into the symbol drawing, and displays the padstack graphics. For this reason, you must define library padstacks before creating package symbols.

When you create a symbol, the layout editor stores the padstack name for each pin in the library symbol, but not the actual padstack data. When you add a symbol to a design for the first time, the editor copies the padstack data from the padstack library and the symbol data from the symbol library.

The editor locates the padstack library by using the padstack library path (*PADPATH*) and the package symbol library by using the symbol library path (*PSMPATH*) set in the global or local environment file.

Once a padstack has at least one instance in a design, the layout editor makes all subsequent references to that padstack in the design and not the padstack in the library.

Standard Pad Shapes

The layout editor displays pads using standard geometric shapes. On the Layers tab of the Pad Designer, you can choose from the pad types illustrated below in photoplots.

Pad Type	Description	Graphic Displayed
Regular	Positive pads (black) with a regular shape (circle, square, rectangle, oblong, octagon), flashed on positive layers only. Through hole pads require regular pad geometries be defined for every layer. Surface mount pads require only Top and related Top mask information. Non-standard shaped pads are available as pad shapes and as pad flashes. If you are using your own custom shape, choose shape, which is used for any definition that is not a circle, a square, a rectangle, oblong or an octagon. A Shape symbol for the geometry of the pad must be created manually using the Symbol Editor.	
	<ul style="list-style-type: none">• Null• Circle• Square• Oblong• Rectangle• Rounded Rectangle• Chamfered Rectangle• Octagon• Donut• n-Sided Polygon	

Thermal relief	<p>Used instead of regular pads to reduce the amount of heat absorbed through connect lines or shapes during the manufacturing process.</p> <p>Thermal reliefs can be:</p>	
	<ul style="list-style-type: none"> A negative pad on a positive copper area (shape). The thermal relief may be plotted as a regular pad flash (circle) with two lines. 	
	<ul style="list-style-type: none"> A positive pad on an embedded metal layer that distributes a voltage, such as power or ground. 	
Anti-pads	<p>Negative pads (clear, surrounded by black), usually a circle, to prevent connection of a pin to an embedded metal layer.</p>	
Shapes	<p>Custom pads created as symbols and added to a padstack either upon initial creation or when editing a design.</p>	

Flash	User-defined name of aperture for Gerber flashing of unique pad shapes. You can use the Symbol Editor's Add – Flash command to aid you in creating the negative thermal relief flash. You specify the inner and outer diameter sizes, the spoke width, the number of spokes, and the spoke angle. The center dot section can be used to create a filled circle that will graphically locate the center point of the flash. A thermal relief is created as a series of filled shapes located on the class ETCH/CONDUCTOR, subclass Top. You do not have to use the Add – Flash command when creating your thermal relief. You can manually draw any number, size and shape of filled shapes. Be sure to create all graphics on the class ETCH/CONDUCTOR, subclass Top.	
-------	---	--

Each pin can have any pad type (regular, thermal relief, anti-pads, and custom shapes) defined on each ETCH/CONDUCTOR layer of the design. For negative artwork layers, the layout editor uses thermal reliefs and anti-pads. For positive artwork layers, the layout editor uses just three regular pads. This means a pin can be put anywhere on the design, and the layout editor photoplots the correct pad, whether the location is in an open area or inside a filled shape.

Photoplot Pad Data

When you create plots, the layout editor uses the padstack information to write the photoplot pad data for a pin on a particular layer. The layout editor determines the pad types for photoplotting as follows:

How the Layout Editor Determines Pad Types for Photoplotting

If the Pin Is...	Uses Pad Type...
Not in a filled area on a layer being photoplotted, such as a shape or rectangle	Regular pad on positive artwork
Inside a filled area on a layer being photoplotted and the pin shares the same net as the filled area	A thermal relief pad created by positive artwork, or a thermal relief pad drawn by positive artwork

Inside a filled area on a layer being photoplotted, but the pin does not share the same net as the filled area

An anti-pad created by positive and negative artwork

 You can enable the suppression of unconnected internal pads during plotting in the Options tab of Pad Editor for designs created in earlier releases.

Defining Library Padstacks

You define new library padstacks using **Padstack Editor**, which you open using the `padstack_editor` command:

1. [Preparing to Define Padstacks](#), described below.
2. [Using the Padstack Editor to create and save library padstacks](#).

Preparing to Define Padstacks

Before you define new library padstacks:

- Check manufacturer specification sheets and verify design requirements.
- Gather the dimensions, physical data, logical data, manufacturing requirements, and documentation requirements for the padstacks you are defining.
- Check that the padstacks are not already in the editor library. To display a list of available library padstack definitions, choose **File – Open** in the Pad Editor.

Use the following table as a guide to pad selection:

Pad Type	Interpretation
h109p	hole; 109 Mils; plated
h109u	hole; 109 Mils; unplated
m43b	multi-bus connector; 43 Mils; BOTTOM mount
m43t	multi-bus connector; 43 Mils; TOP mount
p50c32	pad; 50 Mils; circle shape; 32 Mil drill size
p50s30	pad; 50 Mils; square shape; 30 Mil drill size
pga	pin grid array
s25_48	surface mount pad; 25 x 48 Mils pad size; TOP mount
s25_48b	surface mount pad; 25 x 48 Mils pad size; BOTTOM mount

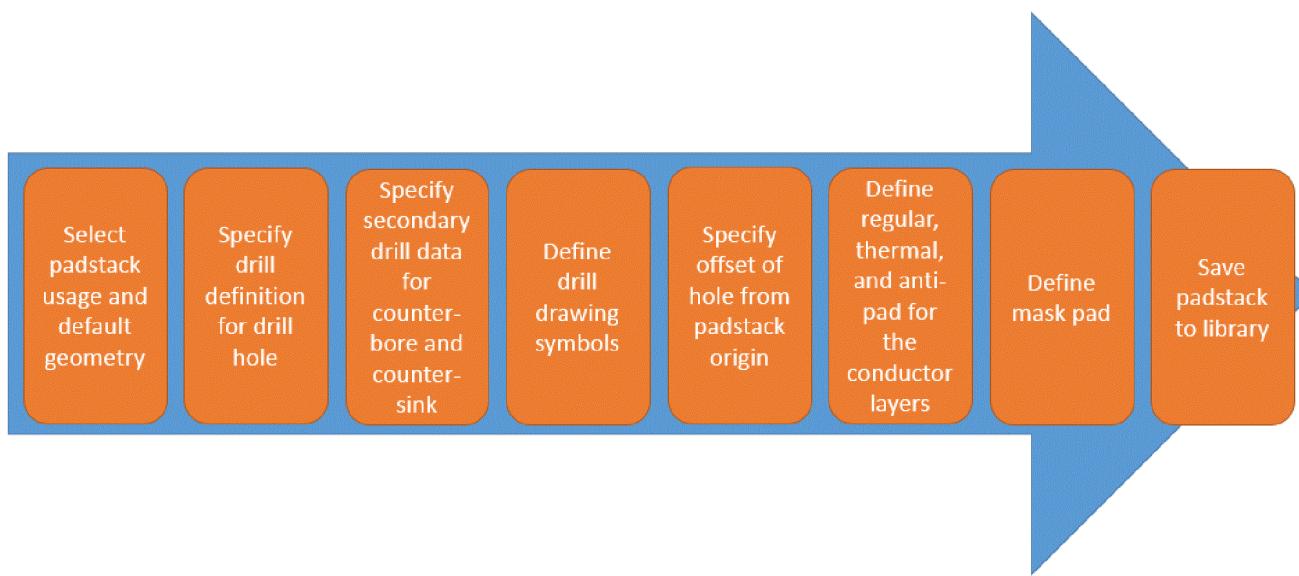
sq55	pad; square shape; 55 Mils
via	---

- Determine photoplot requirements such as flash names, NC Drill data, and offset requirements.
- Create any custom (unique) pad shapes by choosing *Shape – Add Polygon* (`shape add` command), *Shape – Add Rectangle* (`shape add rect` command) or *Shape – Add Circle* (`shape add circle` command).

Using the Padstack Editor

After you have prepared all the data necessary to define padstacks, open the Pad Editor to create padstacks and save them to a library. Use the `padstack_editor` command from

`<installation_directory>/tools/bin, .`



Types of Padstacks You Can Create

You can create different types of padstack in the Pad Editor. Based on the intended use, you can create 12 types of padstack that are defined in the Pad Editor.

- Through: Refers to a padstack with pads on all ETCH/CONDUCTOR layers. Select *Thru Pin*.

- Blind/Buried Via: Indicates the padstack is blind (between the surface and internal layers) or buried (between internal layers). Select *BBvia*.
- Microvia: Refers to Blind/Buried vias used in high density interconnect boards and packages. Once placed, design rules examine touch (via tangency) and coincidence (via stacking) of the padstack's pads and other design objects. Select *Microvia*.

 Select *Lock layer span* in the Options tab to prevent the layer expansion for bbvias when a new signal/plane layer is inserted. For example, a BBvia exists in a design that starts at layer_4 and ends on layer_5. A new layer, say Layer_4A, is inserted between layer_4 and layer_5. The via will start on layer_4, but end on Layer_4A.

- Holes: Refers to holes for fixtures. Depending upon the usage, select one of *Slot*, *Mechanical Hole*, *Tooling Hole*, or *Mounting Hole*.
- Fiducial: Refers to fiducial markers. Select *Fiducial*.
- SMD: Indicates the padstack is for SMD pins. Select *SMD Pin*.
- Bond Finger: Refers to padstacks for wire bond fingers. Select *Bond Finger*.
- Die Pad: Refers to padstacks for die pads. Select *Die Pad*.
- Undefined refers to a padstack where neither pads nor drill hole is defined.



Internal Layers

You can choose to enable the suppression of all unconnected pads on internal layers, as defined on the *Layers* tab, by selecting *Suppress unconnected int. pads;legacy artwork* on the *Parameters* tab. If this option is not selected, you cannot suppress any pads during photoplotting.

Units

You should choose the units and accuracy to suit your design environment, because all fields that use measurement refer to this value. Remember that these units are converted to board units and accuracy, which may result in rounding when you apply the units to a design.

Drill Information

You can specify drill information such as slot or hole type, finished hole or slot size, plating, drill rows and columns, counter-bore or counter-sink drilling operations, drill drawing symbols, and offsets from padstack origin in the Drill, Secondary Drill, Drill Symbol, and Drill Offset tabs. You can also define staggered drills and non-standard drilling for processes used to create the hole in the padstack other than a standard drill bit, other boring type processes may be designated for the hole in the padstack.

The drill figure and characters do not display when a padstack initially loads to a design. Slots are represented as objects in the database. Choose from either an oval or rectangular option. A slot figure and its respective x and y dimension are proportionally scaled to the actual slot size. A slot report, generated using the Tools - Reports menu command, lists slot type, location, and rotation in CSV/HTML format. NC Route, if executed, detects slots on the board and writes them out to the `<design>.rou` file.

To associate positive and negative drill tolerance to any plated or non-plated slot or circle drill, enter values directly to the padstack symbol. Tolerance information appears in the `.drl` file. The *Drill Character Field* field associated with the drill/slot symbol has been increased from one to three places.

Non-standard drill options for circle drills include *Laser*, *Plasma*, *Punch*, *Wet/dry etching*, *Photo Imaging*, *Conductive Ink Formation*, or *Other*. This effectively separates non-standard drills into a separate `.drl` file, making it easier for the fabricator to process incoming data from OEMs.

You can define multi-drill patterns to drill holes in addition to default row and column arrangement. Custom and polar drill arrangements provide flexibility to add different hole patterns. Multi-drill pattern reduce the need of using separate via arrangements inside of pad boundary. Additionally, it avoids accidental deletion of stitching vias inside pad boundary with padstack-defined drill pattern.

Layers

Depending upon the type of padstack, you can define or edit regular, thermal or anti-pad for different layers in the Design Layers tab. You can insert, delete, or copy layers and the pad definitions.

For the mask layers, which can also contain up to 32 user-defined mask layers, you can define regular pads in the Mask Layers tab.

Editing Padstacks

Once a design contains padstack definitions, those padstacks are considered layout padstacks. To modify padstacks within the design, use *Tools – Padstack – Modify Design Padstack* ([pateditdb](#) command). Use *Tools – Padstack – Modify Library Padstack* ([pateditlib](#) command) to modify a padstack from your library.

Recording a Padstack Script

If you have padstacks that share similar specifications, you can automate the process of entering padstack data by choosing *File – Script* in the Pad Editor.

The script feature lets you record the entries you make on the Pad Editor in a script file. When you want to define new padstacks that share similar specifications, you can replay the script file and edit the new padstacks as necessary (to assign new padstack names, make a few changes to the padstack specifications, and so on).

Managing Padstack Data Using XML Files

Library designers sometimes use in-house developed software to create padstacks and then transfer padstack definitions to *Pad Editor* through SKILL, scripts or manual entry.

To exchange the data with external library creation tools, *Pad Editor* uses a bi-directional mechanism where you can import and export padstack definitions through an XML file. The export process saves the padstack definition in an XML-formatted (.pxml) file. The XML file once modified in an external tool can be imported back to Pad Editor and saved as .pad file.

Modifying padstack definition in an XML format provides an easy way to generate and update padstack definitions individually or through a batch process.

Padstack XML files use .pxml as the file extension. The padstack XML format is driven by a Document Type Definition file or DTD file. Use this file as a reference when creating padstack XML-formatted files. This file is available in the installation hierarchy at
`<installation_hierarchy>\share\pcb\xml-formats\cdn_padstack.dtd`

 Do not modify padstack definition file. It will fail import of XML data in Pad Editor.

Sample File

Following is an example of an XML-formatted padstack file for a rounded rectangle surface mount pad.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE padstack>
<padstack>
  <padstackname>SMD_PIN</padstackname>
  <padstackusage>SMD_PIN</padstackusage>
  <units>MILS</units>
  <accuracy>1</accuracy>
  <drillinfo>
    <slothole>
      <holetype>NONE</holetype>
    </slothole>
  </drillinfo>
  <pad>
    <type>REGULAR_PAD</type>
    <layer>BEGIN_LAYER</layer>
    <figure>CHAMFERED_RECTANGLE</figure>
    <width>16.0</width>
    <height>36.0</height>
    <corner>5.0</corner>
    <ul>Y</ul>
    <lr>Y</lr>
  </pad>
  <pad>
    <type>REGULAR_PAD</type>
    <layer>TOP_SOLDER_MASK_PAD</layer>
    <figure>CHAMFERED_RECTANGLE</figure>
    <width>16.0</width>
    <height>36.0</height>
    <corner>5.0</corner>
    <ul>Y</ul>
    <lr>Y</lr>
  </pad>
</padstack>
```

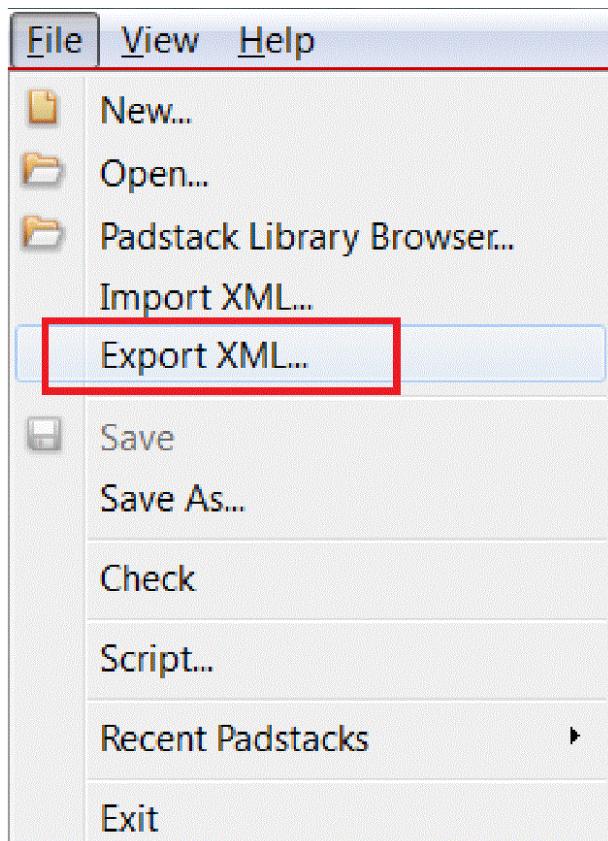
You can find another examples of padstack XML files at
`<installation_hierarchy>\share\pcb\examples\padstack_xml.`

The export process saves the padstack definition to an XML-formatted file. Library designers can

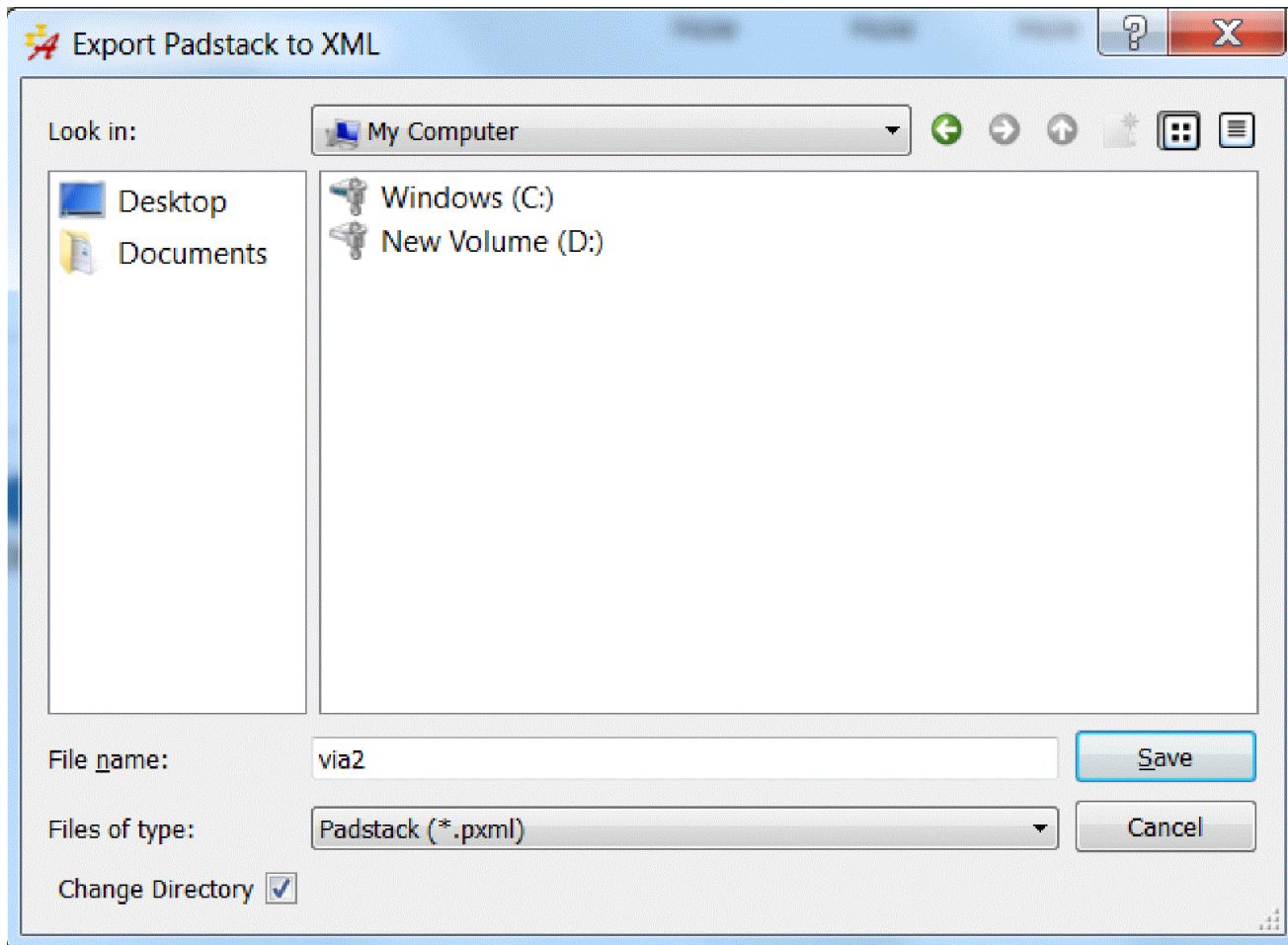
then modify or update these XML files in other tools.

Exporting Padstack to XML from UI

Open the padstack in *Pad Editor* and choose *File – Export XML*.



Warnings are displayed in the *Padstack Errors*. Click Close and then Yes to save the data with warnings. The *Export Padstack to XML* dialog box appears.



Browse the location to export the padstack definition to an XML (.pxml) file. The default name of the XML file is the name of the padstack.

Exporting to XML from Command Line

You can also export padstack definitions from the system command window. The command provides options to export the data either to a single XML file that contains details for all the library padstacks or to a separate XML file for each padstack.

To Separate XML File Using Pad File Name

Enter the following command in the system command window:

```
padstack_editor -xo via.pad
```

The command process a single .pad (for example, via.pad) file and output a .pxml (for example,

`via_pad2xml.pxml`) file at the same location where `.pad` file exist. The errors and warnings occurred during the export process are saved in a log (for example, `via_pad2xml.log`) file, which is created with the input `.pad` file name.

To Single XML File Using Wildcard

Enter the following command in the system command window:

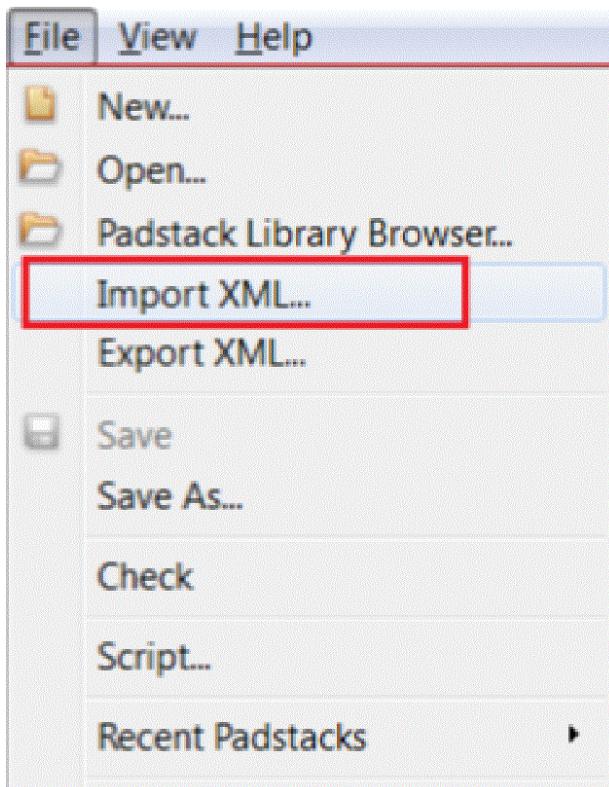
```
padstack_editor -xo *.pad
```

The command process multiple `.pad` files and creates a single `.pxml` file as an output. The errors and warnings occurred during the export process for each `.pad` file are saved in a single log file, which is created with the first input `.pad` file name.

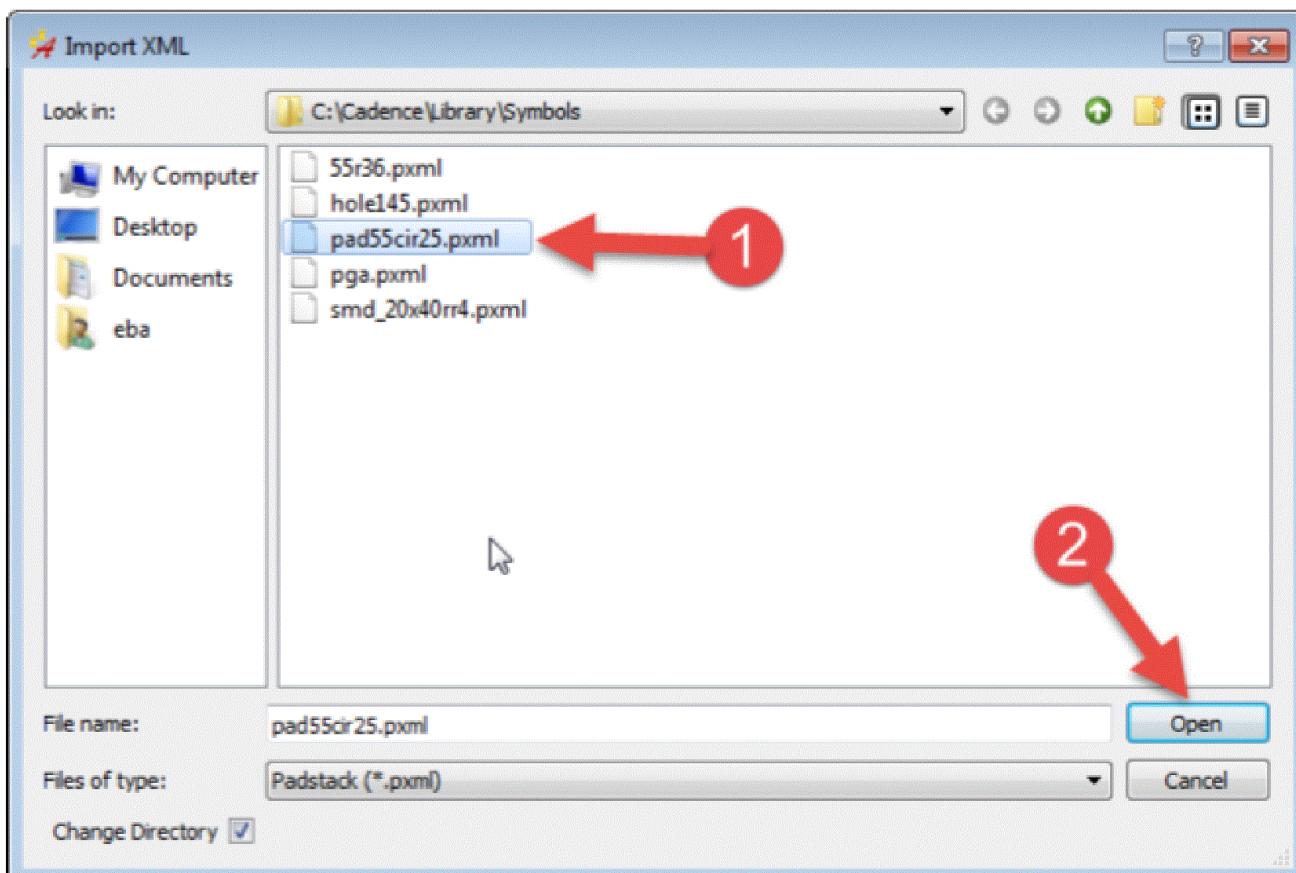
The import process generates padstacks by reading data from an XML file. Library designers can update an existing padstack library by importing the padstack data defined in an XMLformatted file(`.pxml`) into *Pad Editor*.

Importing XML from UI

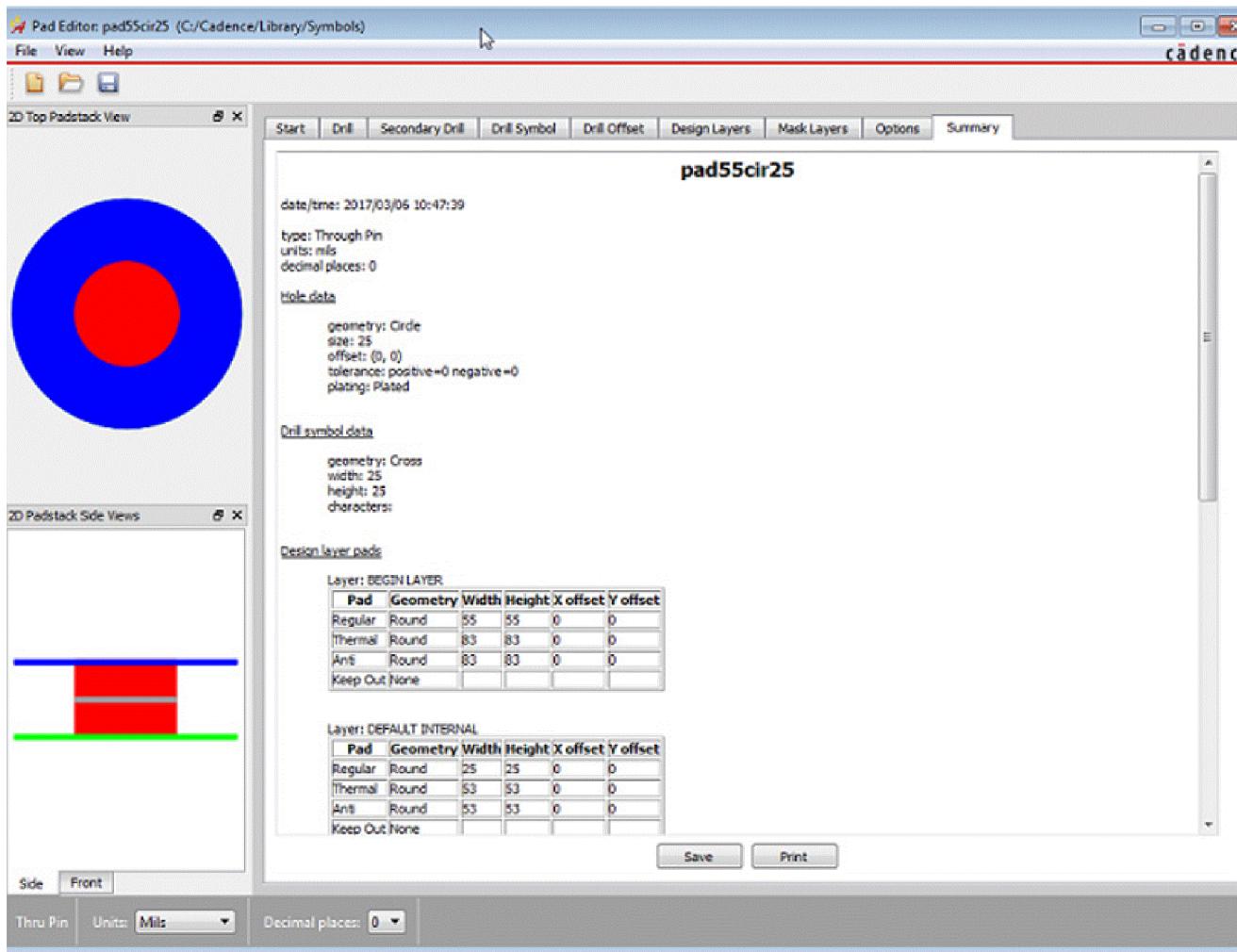
The padstack data defined in XML format is imported in the *Pad Editor* using *File – Import XML*.



Browse and select .pxml file in the *Import XML* dialog box.



The *Pad Editor* is updated with the padstack name and definition from the data defined in the XML file. You can review the results in the *Summary* tab in the *Pad Editor* and save the padstack (.pad) file.



On saving the padstack definition, the *Pad Editor* performs the checks and reports warnings and errors in the *Padstack Error* dialog box.

The import function also supports multiple padstack definitions from a single XML (.pxml) file. For each padstack definition, a separate .pad file is created by the import process.

The default template of multiple padstacks XML type definition file `cdn_padstack_multi.dtd` is available at `<installation_hierarchy>\share\pcb\xml-formats`.

Importing XML from Command Line

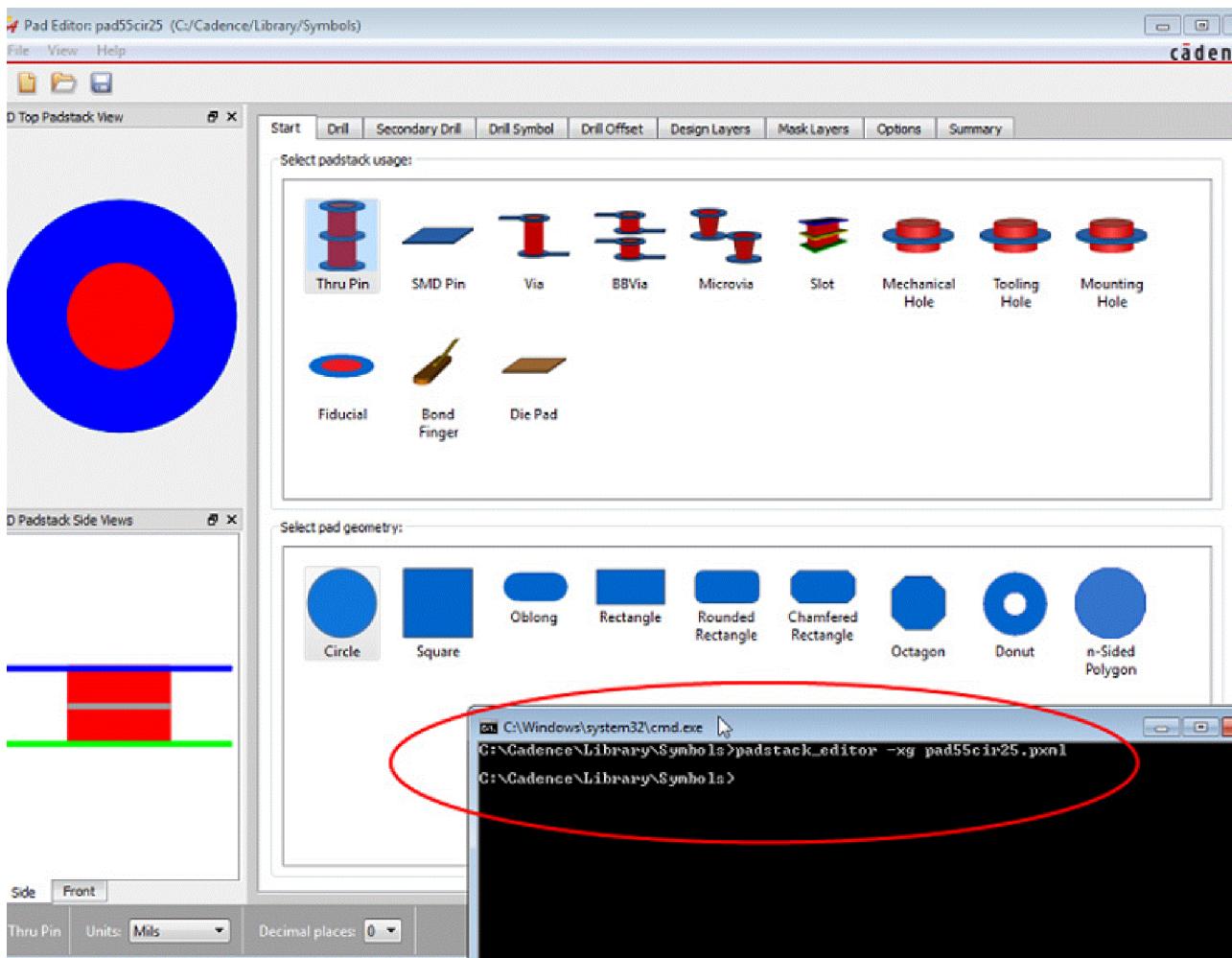
You can also create padstack definitions from the system command window. There are two ways of using command line options.

Creating Padstack in Graphical Mode

To generate padstack definition from an XML file and review it in *Pad Editor*, enter the following command in the system command window:

```
padstack_editor -xg <filename.pxml>
```

The command opens *Pad Editor* and displays the padstack data for review.

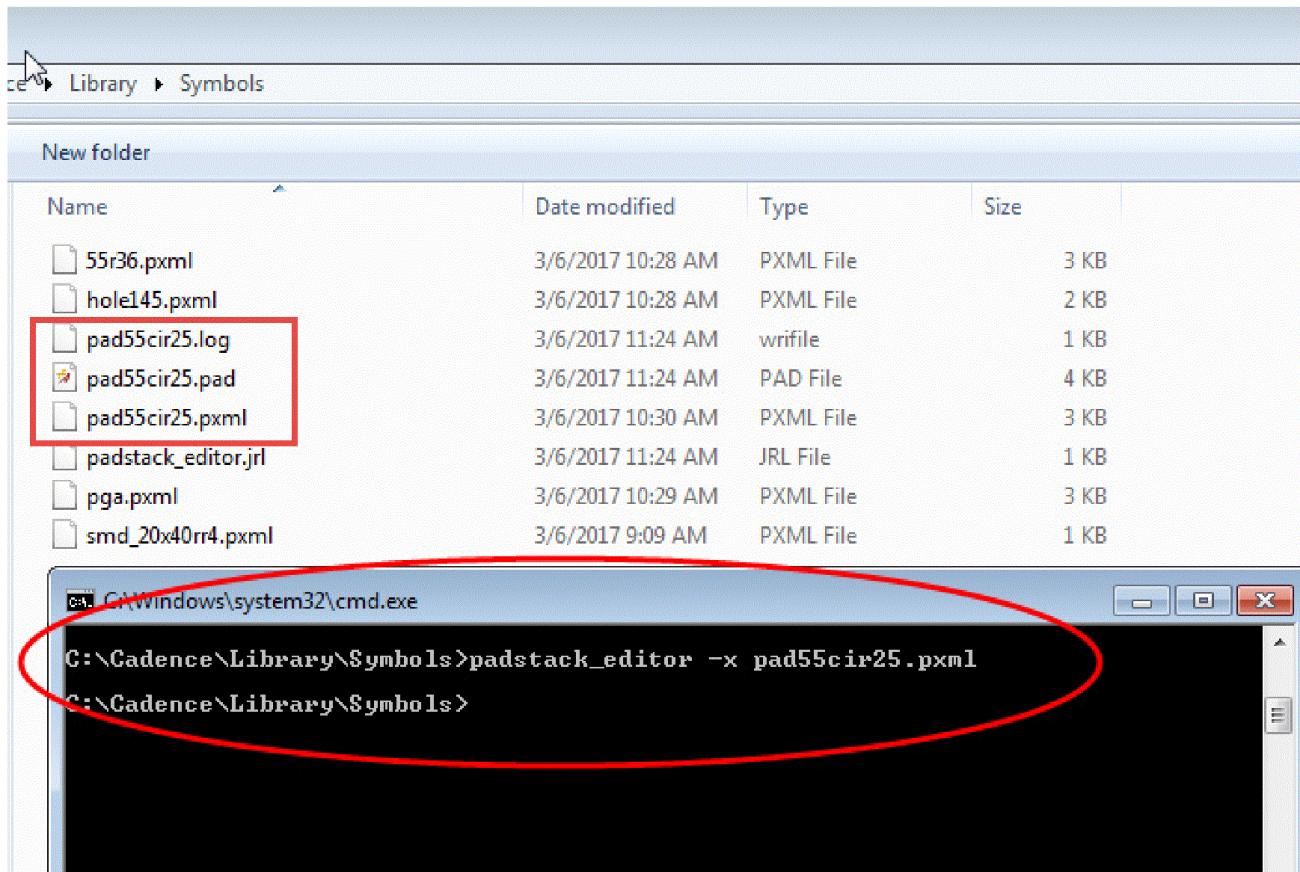


Creating Padstack in Non-graphical Mode

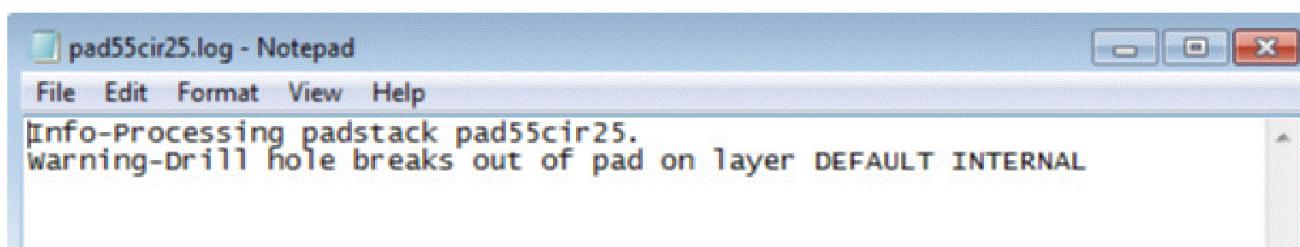
To generate padstack definition from an XML file without viewing the *Pad Editor*, enter following command in the system command window:

```
padstack_editor -x <filename.pxml>
```

The command creates padstack definition file without launching the *Pad Editor* and reports warning or errors in a log file.



If errors occurs during the import process, the padstack definition will not be created. You need to correct the XML file before importing.



Creating or Updating Padstacks in Batch Mode

There are two ways to do this task in non-graphical mode:

Using a Single XML File

To process a single XML file that consists of multiple padstack definitions, enter the following command in the system command window:

```
padstack_editor -x <filename>.pxml
```

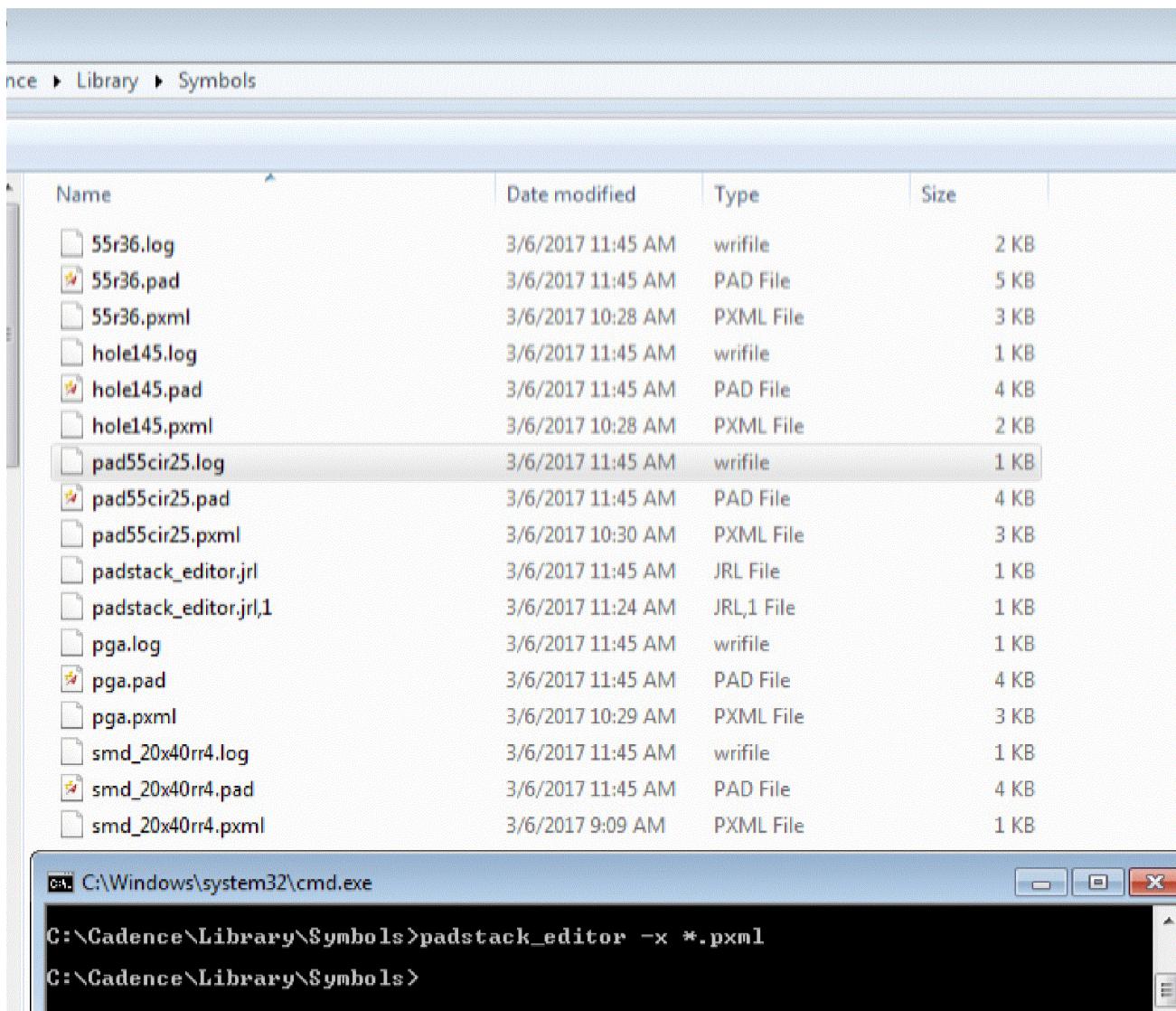
For each padstack definition, the command creates a separate .pad file. The errors and warnings occurred during the import process are recorded in a single log file which is created with the input XML file name.

Using Wildcard

To process multiple XML files using wildcard, enter the following command in the system command window:

```
padstack_editor -x *.pxml
```

For each .pxml file, the padstack definition and log file is created in the library.



The command overwrites padstack definitions and log files for the padstacks that are already in the library. If there is any error in the XML file, the command only creates the log file and does not update the padstack (.pad) file.

Updating Layout Padstacks

To ensure that a design uses the latest version of the padstacks in the library, the layout editor lets you refresh layout padstacks. You can do this interactively by choosing *Tools – Padstack – Refresh (refresh_padstack command)* or by running the `refresh_padstack` batch command described in the *Allegro PCB and Package Physical Layout Command Reference*.

When you refresh padstacks, you can either refresh all the padstacks in a design or just the ones in a padstack list file.

List files are ASCII text files that end in the `.lst` extension and contain the names of padstacks that can be updated. Use a text editor to create the padstack list file, and place the file in the current working directory.

The following file format conventions apply:

- Provide only one padstack name on each line.
- Use either uppercase or lowercase letters. The layout editor always stores padstack names in uppercase but can read mixed case in this file.
- Do not include leading or trailing white space. The layout editor removes it if it appears in the file.

Sample Padstack List File

This is a sample padstack list file:

```
pad93cir58d  
soj  
via  
smd25-94pf
```

The `refresh_padstack.log` file records the refresh padstack processing.

Custom Pad Shape Symbols

Suppressing Unused Padstacks

The *Suppress Unconnected Pads* option available with the *Manufacture – Artwork* ([film param](#)) command, can be used on a per artwork film basis to suppress the display of unused inner layer pads of pins and vias on specified layers.

Suppresses inner-layer pins or vias that are unconnected to a cline or plane on layers you specify using the [xsection](#) command. Pads on the top or bottom layer are never considered unused, even if they lack connections, nor are outer layers of a blind/buried via instance, which are preserved. Mechanical pins are ignored by pad suppression as are pins and vias with Fixed internal layer padstacks in the Pad Editor. Pads are added when a connection occurs to a pin or via, and removed when the connection is deleted. Pads cannot be suppressed on any layer that requires negative artwork, as the pad is required to create a negative artwork void.

Purging Unused Padstacks

The layout editor lets you remove unused padstacks from the list of available padstacks for a design. You can purge all unused padstacks or all derived padstacks by choosing *Tools – Padstack – Modify Library Padstack* (`padeditdb` command).

Derived padstacks are those you create by modifying existing padstacks. You may have unused derived padstacks if you restore derived pads to their original state. The derived padstack names are listed as available padstacks in the *Padstack Selection* dialog box until you purge them.

Creating and Using Structures

Working with Symbols

This chapter provides information on package, mechanical, and format symbols. As with padstack files, the layout editor supplies a symbol library containing symbols that you can copy into designs or that you can modify for specific needs. You can also create custom symbols. Creating symbols, while not tied directly to any specific part of a design flow, normally occurs at the beginning of the design process.

Working with the Symbol Mode

You work in the symbol mode to create and modify symbols. You can enter symbol editing mode in one of the following ways:

- Choose *File – Open* ([open command](#)) and choose a filename with a `.dra` extension.
- Choose *File – New* and choose one of the symbol drawing types from the *Drawing Type* list box in the *New Drawing* dialog box.

The editor workspace then changes from layout to symbol mode. The Symbol mode contains a subset of the layout mode's commands, and also has commands used exclusively with symbols.

Symbol Types

The layout editor symbols are categorized by type. All symbols are derived from a drawing file that is later converted into the appropriate symbol type. The following symbol types are available in the layout editor:

- Package
Package symbols are the physical representation of electronic devices such as dual in-line packages (DIPs), edge connectors, resistors, capacitors, and transistors. Each package symbol contains pins (with pads and possible drill holes) that act as attachment points for connecting ETCH/CONDUCTOR added during interactive or automatic routing.
- Mechanical
Mechanical elements in a design include items such as tooling holes, outlines, and card ejectors. Since these elements are non-electrical, they contain no reference designators or pin

numbers. Mechanical elements can also include keepin or keepout areas for routing, packages, and vias.

- Flash

Flash symbols are pads that you create for photoplotting using standard apertures.

- Format

Format symbols consist of manufacturing drawing formats such as sheet drawings in various sizes, and corporate legends and logos.

- Padstacks

Padstacks are also symbols; however, padstacks have their own editor you use to create and modify padstacks.

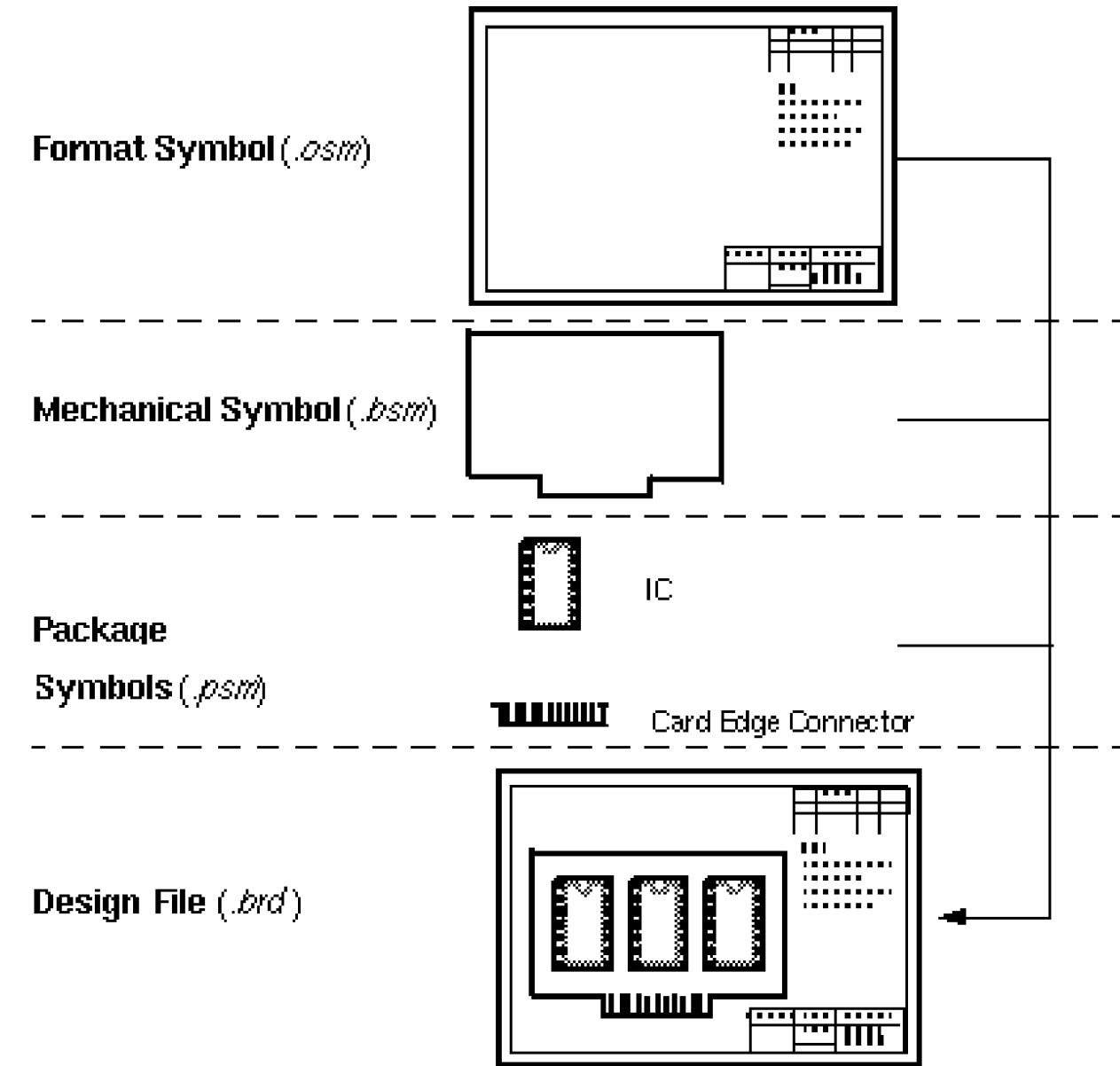
- Shapes

Shapes are another type of symbol; however, you create them by choosing *Shape – Add Polygon* (`shape add` command), *Shape – Add Rectangle* (`shape add rect` command) or *Shape – Add Circle* (`shape add circle` command) and then edit by choosing *Shape – Select Shape or Void* (`shape select` command). The symbol mode does not support dynamic shapes. To edit traditional static shapes or voids, use commands available from the *Shape – Manual – Void* menu (`shape void polygon`, `shape void circle`, `shape void rectangle`, `shape void copy`, `shape void move`, or `shape void delete` commands).

See the *Preparing for Layout* user guide in your documentation set for additional information on using shapes.

See the following figure for an example of packagepart, mechanical, and format symbols.

Symbols and their Relation to the Design File



Symbol and Drawing Files

A symbol in the layout editor comprises two files:

- The drawing file (.dra)

- The symbol file (*.sm)

The drawing file is the result of all the commands that you choose from the *Add* and *Edit* menus of the symbol mode. Additionally, you choose commands from the Layout menu to add a layer of electrical characteristics to the drawing, in the form of connections, pins, labels, and constraints.

When you first create a new drawing file for a symbol, you specify its type (package, mechanical, and so forth). After editing the file, you convert it into a symbol with one of these extensions:

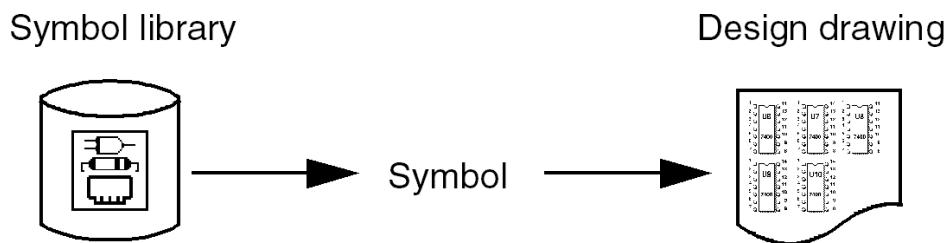
Symbol	Extension
Package	.psm
Mechanical	.bsm
Format	.osm
Shape	.ssm
Flash	.fsm

The layout editor automatically creates a symbol (*.sm file) every time you save a drawing (*.dra) when you are in the symbol mode. You no longer need to compile the symbol and save the drawing in two separate steps.

Set the environment variable `no_symbol_onsave` to restore the legacy behavior and allow the layout editor to compile the symbol and save the drawing in two steps. For additional information about setting environment variables, see [enved](#).

The Symbol Library

After creating a symbol, save it in a library to reuse it.



Once you load a library symbol into a design, the layout editor uses that symbol definition for future instances of that symbol. If that symbol does not exist in a design, the layout editor looks for it in the

library.

Legal Classes for Symbol Types

The following are legal classes for symbols:

	Package	Mechanical	Shape	Flash	Format
BOARD_GEOMETRY	yes	yes	no	no	yes
COMPONENT_VALUE	yes	yes	no	no	no
DEVICE_TYPE	yes	yes	no	no	no
DRAWING_FORMAT	no	no	no	no	yes
ETCH/CONDUCTOR	yes	yes	yes	yes	no
MANUFACTURING	yes	yes	no	no	yes
PACKAGE_GEOMETRY	yes	yes	no	no	no
PACKAGE_KEEPIN	no	yes	no	no	no
PACKAGE_KEEPOUT	no	yes	no	no	no
PIN	yes	yes	no	no	no
REFERENCE_DESIGNATOR	yes	yes	no	no	no
ROUTE_KEEPIN	no	yes	no	no	no
ROUTE_KEEPOUT	yes	yes	no	no	no
TOLERANCE	yes	yes	no	no	no
USER_PART_NUMBER	yes	yes	no	no	no
VIA_CLASS	yes	yes	no	no	no
VIA_KEEPOUT	yes	yes	no	no	no

Creating a Symbol

Open a new file of the desired symbol type (*Package*, *Mechanical*, *Format*, *Shape*, or *Flash*). Details for creating each type of symbol appear in the rest of this chapter.

 The Symbol mode inherits the dimensions of the associated design. You can specify different drawing parameters by choosing *Setup – Drawing Size*.

When you have created the symbol, choose *File – Create Symbol* (`create symbol`) in Symbol mode to convert the active drawing into a symbol. This compiles the artwork of a drawing, together with any electrical attributes that you specified into a binary file. The menu item and command are described in the *Allegro PCB and Package Physical Layout Command Reference*.

Adding Areas

Areas for symbols include route, via, and place keepouts and keepins, as well as place-bound rectangles, used to delineate a package boundary that specifies different height constraints of the placement area beneath components.

Keepouts demarcate an area on the symbol in which to avoid placing ETCH/CONDUCTOR, vias, or other symbols. Keepins demarcate an area on the symbol in which to guide (restrict) the placement of ETCH/CONDUCTOR, vias, or other symbols. Menu items and commands for areas include:

- *Setup – Areas – Component Keepin* (`keepin component` command)
- *Setup – Areas – Component Keepout* (`keepout component` command)
- *Setup – Areas – Component Height* (`component height` command)
- *Setup – Areas – Route Keepin* (`keepin router` command)
- *Setup – Areas – Route Keepout* (`keepout router` command)
- *Setup – Areas – Route Wire Keepout* (`keepout wire` command)
- *Setup – Areas – Via Keepout* (`keepout via` command)
- *Setup – Areas – Shape Keepout* (`keepout via` command)
- *Setup – Areas – Probe Keepout* (`keepout probe` command)
- *Setup – Areas – Gloss Keepout* (`keepout gloss` command)

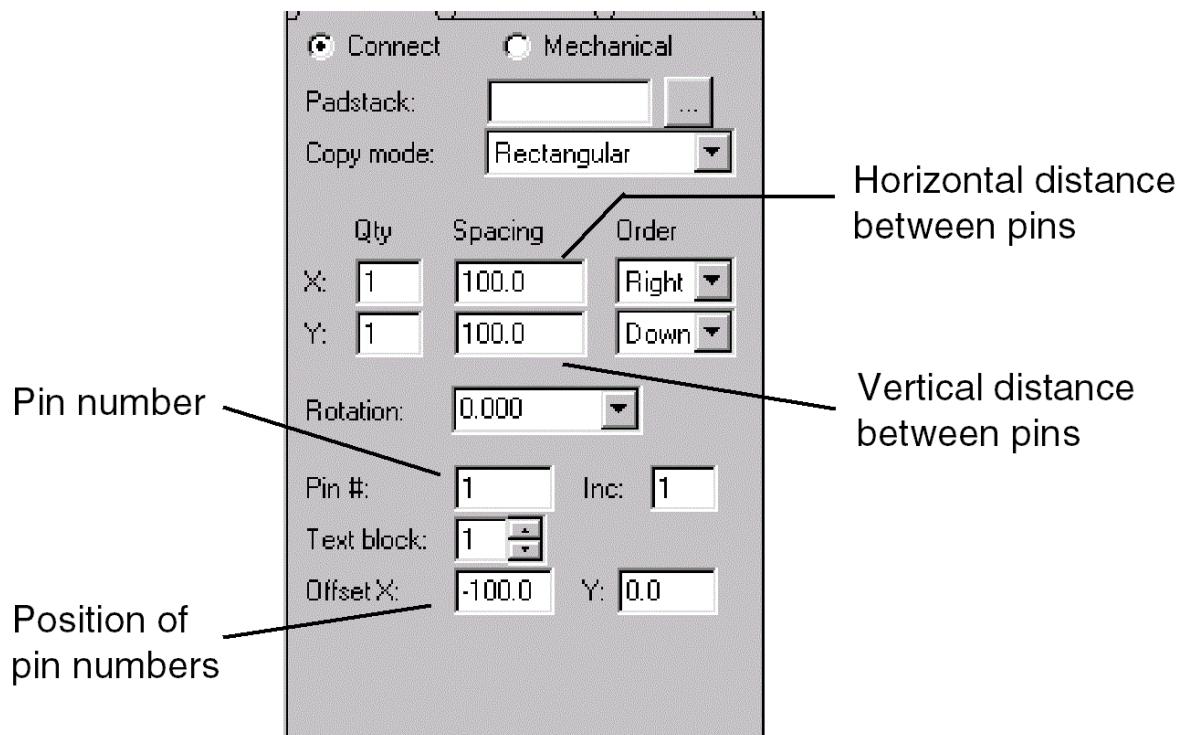
- *Setup – Areas – Photoplot Outline* (`keepin photo` command)
- *Setup – Areas – Package Keepin* (`keepin package` command)
- *Setup – Areas – Package Keepout* (`keepout package` command)
- *Setup – Areas – Package Boundary* (`keepout probe` or in the symbol mode to create place-bound rectangles)

 The layout editor automatically chooses the correct class/subclass based on the command that you choose.

Adding Pins

You can add pins to package and mechanical symbols. Package symbol pins have pin numbers; mechanical symbol pins do not.

From the symbol mode, choose *Layout – Pins* (`add pin` command) to initiate pin placement via parameters that you specify in the Options tab.



ETCH/CONDUCTOR and Vias in Symbols

You can add shapes, ETCH/CONDUCTOR, and vias to a symbol drawing. When you add a symbol to a design, any ETCH/CONDUCTOR and via elements in the symbol are added as well. However, you should consider the following when adding symbol elements to a design:

- To delete a layer from a design that has package symbols with an item, such as ETCH/CONDUCTOR, defined on that layer (subclass), delete the items defined on that layer before deleting the layer.
- Edit items not associated with a symbol either separately or move them through a group window operation.
- ETCH/CONDUCTOR text/lines, circles, arcs (unlike connect lines, these elements can never be connected to or associated with any nets) remain associated with their symbols. You can always move, copy, or delete ETCH/CONDUCTOR text/lines, circles, and arcs with the symbol. ETCH/CONDUCTOR text/lines, circles, and arcs are intended as constructs for ETCH/CONDUCTOR labeling, as an alternative to silkscreen.

 When used for electrical connectivity, these constructs may lead to problems with the ratsnest display and DRC.

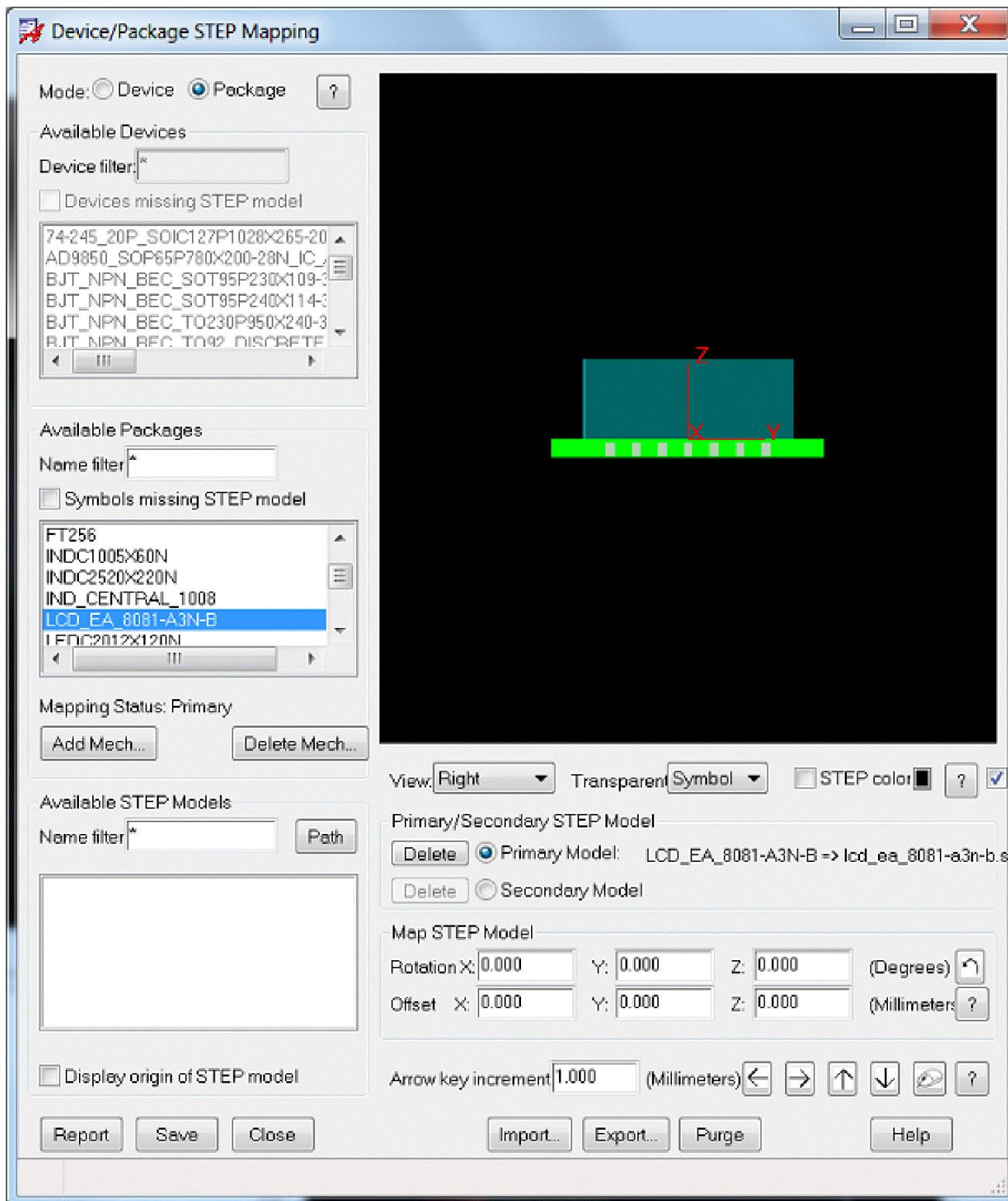
- The pin padstack layers remain part of the pin, and the pin always remains a part of the symbol.
- If a user-definable subclass exists in the symbol, you do not need to define the subclass in the design in order for the symbol to be added to the design.
When you add the symbol to the design, the subclass is automatically added to the design.
- You can create mirrored pairs of user-defined subclasses.
The graphics in these subclasses are interchanged whenever the symbol is mirrored.
These subclass pairs must be in the same class and are identified with identical names, except for the words TOP and BOTTOM. For example, a PACKAGE subclass named SPECIAL_DATA_TOP is moved to SPECIAL_DATA_BOTTOM. Likewise, the graphics in SPECIAL_DATA_BOTTOM are moved to SPECIAL_DATA_TOP.

Mapping STEP Models

The STEP models provides complete, detailed and accurate three-dimensional model representation of the components. You can map STEP models to package symbols and devices in the symbol editor.

To assign STEP models to symbols and devices you need to define the path of STEP model library. The `steppath` environment variable identifies the directory in which STEP model files (`.stp` and `.step`) are saved. You can set the path from *Setup – User preferences – Paths – Library*.

The PCB Editor provides a mapping tool to associate STEP models to devices, packages and mechanical symbols. This mapping tool associates primary and secondary STEP models to the symbols and devices. You can define offset and rotational information to correctly position the STEP models in 3D Canvas. The mapping data created is then instantiated into the symbol as a property.



To map the STEP model, select a STEP model from *Available STEP Models*. The models are displayed in the graphic pane.

To see the differences in origins overlay models, rotate and set X, Y, and Z offset values and save the mapping.

The mapping tool assigns two properties to the package symbol: PKGDEF_STEP_TRANSFORMATION and PKGDEF_STEP_FILE. These properties become part of the symbol/component definition and cannot be modified outside of the STEP Package Mapping utility.

You can also export the mapping data into an XML format to a .map file. There is another set of XML files called facet files that are also exported as mapping data into a .zip file (stepFacetFiles4Map.zip). The facet files have geometrical data required for 3D display. The mapping data files are saved at the location specified by the *step_mapping_path* and *step_facet_path* environment variables.

By importing you can re-use the mapping data into other designs. You need to extract all facet files into the destination directory. Multiple mapping files can be imported into a design.

Creating Package Symbols

Package symbols physically represent electronic devices in a design. They must correspond to the logic data loaded in the design database—pin numbers must match, and reference designators must exist. They can also carry manufacturing data, such as company part numbers, component values, and tolerances.

 Package/Component Symbol Library shows the package symbols available in the editor library.

Package Symbol Elements

Packages must have the following elements:

- At least one pin

Pins are connect points with associated pad geometry and pin number. Pins cannot be edited (moved or spun) unless the UNFIXED_PINS property has been attached to the symbol or the design in which the symbol resides.

- Component outline

A component outline is a geometric representation of the component and can consist of lines, arcs, circles, text, and notes.

- Reference designator

A reference designator is text that identifies a device in a design.

- At least one place-bound rectangle (user-defined or system default)

Place-bound rectangles are filled rectangles that define the package part boundary and govern placement restrictions. Placement tools use these rectangles for overlapping and mechanical restraints. DRC also uses them to check for violations of package-to-package keepin areas and keepout areas.

The following is a list of other elements that you can add to a package symbol during the symbol building process:

- Device type text (for the component device type)

- Component value (text for the component value)

- Tolerance (text for the component tolerance)

- Component height (text for the physical height of the component)

- Silkscreen information (text and lines for information on silkscreen layer of drawing)

- User part number (text for the package part number)

- Route keepout shape(s) identifying areas where ETCH/CONDUCTOR is not allowed

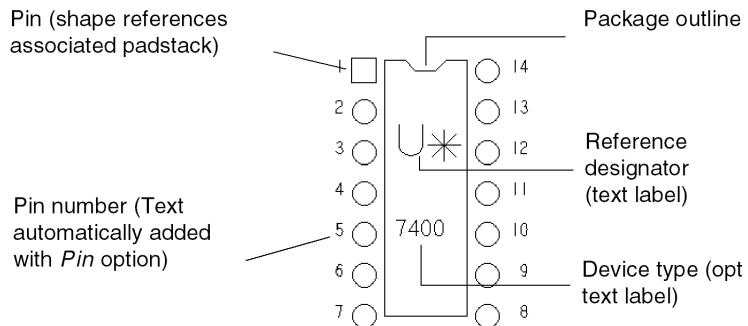
- Via keepout shape(s) identifying via keepout areas

- ETCH/CONDUCTOR (lines, arcs, rectangles, shapes, and/or text added to the symbol)

- Vias

The following figure shows a DIP14 package symbol drawing, which identifies the class of each element in the drawing. This drawing does not contain all possible package classes.

DIP 14 Package Symbol Drawing



Prerequisites for Creating a Package Symbol

Check the symbol library to see if the package symbol exists. If you need to create a package symbol, you should adhere to your company's standards for naming conventions, pad sizes, component origins and orientation, as well as any specific manufacturing criteria such as silkscreen and auto-insertion requirements. For each component, you should have the following physical data:

- Component body size
- Component origin
- Drill size
- Insertion machine clearances
- Pad size
- Pin-to-pin spacing

Create the library padstacks associated with the pins in the components using Padstack Editor, which you open using the `pad_designer` command.

Guidelines for Creating Package Symbols

When you create a package symbol:

- Build package symbols on the TOP of the drawing.
Choose *Edit – Mirror* (`mirror` command) to place symbols on the bottom side of the design during placement, which causes symbols to be mirrored as they are added to a design.

- Set up the text for titling and marking up the symbol.

Choose *Setup – Design Parameters* (`prmed` command) and choose the *Text* tab of the Design Parameter Editor to specify in the *Text Setup* dialog box the characteristics of each type of text (called text blocks) you can use in the design. Then choose one of these text blocks as the text default along with how text will be justified and the size of markers.

- If you are using automatic placement, you use the same origin in all the package symbols. The symbol origin determines how the package symbol is physically located in the design. For example, if the origin of the symbol is the center of the component, the pick location tells the layout editor to place the component center on that grid point. On the other hand, if the symbol origin is pin 1 of the component, the component is positioned so that pin 1 is at the location picked. Automatic placement uses the same origin to place all components during a placement session.

Defining a Package Symbol

You can create a package symbol in any of these ways:

- Choose *File – New* and choose Package Symbol wizard or run the *package symbol wizard* command to create a basic symbol using the wizard.
- Manually
- In batch mode.

Defining Symbol Heights

Height restrictions allow you to limit the boundaries of a package and define the dimensions of its keepout areas.

- You define the 3-D geometry of package boundaries using place-bound rectangles. You can use multiple place-bound rectangles to define different height constraints that describe placement space available under component bodies. You might need multiple rectangles to better define complex, asymmetrical symbols or to maximize placement space.
- A keepout is a restricted area. Height keepout areas allow package symbols whose height is below a minimum or above a maximum to be placed in that area. For example, if the height range of a keepout is 500 to infinity, package symbols whose height is less than or equal to 500 can be placed in it.

Package Height

The following describes package height values for a package keepout area and a place-bound rectangle.

Package keepout area:

- If a package keepout has no height value, then it means that the value of PACKAGE_HEIGHT_MIN is 0 and the value of PACKAGE_HEIGHT_MAX is INFINITY. Packages with heights in this range (the default setting) produce a DRC error.
- If a package keepout has only a minimum height value of MIN, this means that the value of PACKAGE_HEIGHT_MIN is MIN and the value of PACKAGE_HEIGHT_MAX is INFINITY and that the keepout flags a DRC if the package height falls within this 3-D space (MIN to INFINITY).
- If a package keepout has only a maximum height value of MAX, this means that PACKAGE_HEIGHT_MIN is 0 and PACKAGE_HEIGHT_MAX is MAX and that the keepout flags a DRC if the package height falls within this 3-D space (0 to MAX).
- If a package keepout has both MIN and MAX height values, then it means that PACKAGE_HEIGHT_MIN equals MIN and PACKAGE_HEIGHT_MAX equals MAX and that the keepout flags a DRC if the package height falls within this 3-D space (MIN to MAX).

Place-bound rectangle:

- If a place-bound rectangle has a height value of MAX, this means that values of PACKAGE_HEIGHT_MIN is 0 and PACKAGE_HEIGHT_MAX is MAX and that the package occupies the 3-D space (0 to MAX).
- If a place-bound rectangle has both MIN and MAX height values, this means that it occupies the space between the value of MIN for PACKAGE_HEIGHT_MIN and the value of MAX for PACKAGE_HEIGHT_MAX, which is in the 3-D space (MIN to MAX).

Typically when you specify MIN and MAX values for a place-bound rectangle, the symbol includes multiple place-bound rectangles with different restrictions.

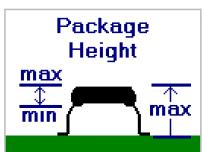
Package Height Example

For example, if the package keepout PACKAGE_HEIGHT_MAX is 1.7 MM and for the place bound PACKAGE_HEIGHT_MAX is 0.61 MM, then the exclusion area for the keepout is (0,1.7 MM) and the occupied area for the package is (0, 0.61 MM). Therefore, the package invades the exclusion area of the keepout and thus generates a DRC

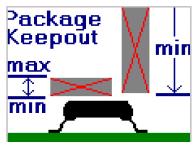
The following figure distinguishes between package height and package keepout height, as displayed in graphics that appear in the *Options* tab when you run the `package_height` command from the layout or symbol modes.

 The objects of measurement depend on the chosen class.

Graphic Representations of `package_height` Commands



Package height for a place-bound rectangle



Package height for a keepout

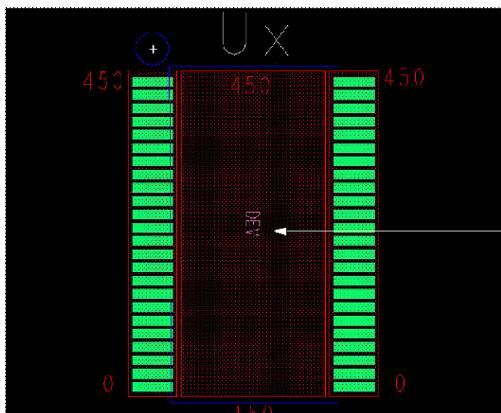
You can use the `package_height` command to update height restrictions on existing objects or to add new areas. The command defaults to updating heights on existing areas but, by choosing either *Add Rectangle* or *Add Shape* from the pop-up menu, you can add new areas. Choosing these options has the same effect as running the `add_rect` or the `add_fshape`, `shape add`, and `add_xshape` commands, after which you are prompted to add the appropriate height property.

 When drawing a component (other than a die) the Allegro 3D Viewer uses the place bound (or assembly bound) to draw the symbol's outline and the minimum and maximum height properties to give the component thickness above the substrate.

Multiple Place-bound Rectangles

The following figure shows how you use three place-bound rectangles (shown in gray) to define the height of a symbol. Two place-bound rectangles define the height parameters (0 to 450) of the pins. A third place-bound rectangle defines the height range of the body of the component (150 to 450). Since the body begins only at 150, there is space under this component from 0 to 150 where a smaller component can be placed.

Component Symbol with Three Place-Bound Rectangles



Symbol height is the z-dimension (depth) of the package symbol. By default, the layout editor sets the maximum height for all unlabeled package symbols to 150 mils.

To ensure that the unit measurements assigned to symbols do not conflict with the unit measurements on the design or module, package symbol height in the layout editor versions 14.0 and later is stored in the database as representations of two properties, PACKAGE_HEIGHT_MIN and PACKAGE_HEIGHT_MAX. You can attach these properties to rectangle and shape objects on class/subclass:

- PACKAGE_KEEPOUT/ANY (ALL, TOP, or BOTTOM)
- PACKAGE_GEOMETRY/PLACE_BOUND_BOTTOM
- PACKAGE_GEOMETRY/PLACE_BOUND_TOP

If you attempt to place objects containing the package height properties on other class/subclass layers, an error message appears in the status window.

i Previous to version 14.0, package height constraints were stored as attached text, causing problems when changing design units and loading symbols whose height constraint design units were dissimilar from those of the rest of the design. Associating package heights with properties eliminates such problems by updating the package height values when the overall design units are changed and by converting the units of symbols you load to those of the design.

The manner in which package height restrictions are handled in versions prior to 14.0 databases (that is, by way of attached text) are converted to the new method (property definitions) when you:

- Open a pre-14.0 database in 14.0 (and subsequent versions)
- Place a symbol in a 14.0 or later database

- Uprev the database using the `uprev` batch command.

Defining Component Heights with Properties

The PACKAGE_HEIGHT_MIN, PACKAGE_HEIGHT_MAX, and HEIGHT properties are methods to define component heights, using either a symbol-driven or a Part Table File (PTF) package height approach. Both methods may be employed in the same design.

Symbol-Driven Package Height Model

When multiple heights within the same symbol are required, assign the PACKAGE_HEIGHT_MIN and PACKAGE_HEIGHT_MAX properties to component definitions or to package keepout areas and place-bound shapes on the PLACE_BOUND_TOP or PLACE_BOUND_BOTTOM subclasses. To use this symbol (.psm) driven package height model, choose either *Setup – Areas – Package Height* (`package_height` command) or *Edit – Properties* (`property edit` command).

A package symbol can have multiple place-bound shapes, and each of them can have a unique PACKAGE_HEIGHT_MIN or PACKAGE_HEIGHT_MAX value. A MAX value defines the height of the package, and the MIN value defines a clearance under a part of the package. Allegro also audits the PACKAGE_HEIGHT_MIN and MAX values when determining if component-to-component or component-to-keepout violations exist.

If a property assigned to place-bound is:	then...
PACKAGE_HEIGHT_MAX	the symbol fills that area between the surface and the maximum height specified in the property value
PACKAGE_HEIGHT_MIN	the symbol fills the area above the minimum height specified in the property value, and a clearance exists between the surface and the minimum value.
PACKAGE_HEIGHT_MIN and PACKAGE_HEIGHT_MAX	the symbol fills that area above the minimum value and below the maximum value.

When evaluating the PACKAGE_HEIGHT_MIN or PACKAGE_HEIGHT_MAX properties, the first value found applies in the following order of precedence:

- Place-bound shape or rectangle

- Component definition
- Default package height (MAX value only)

PTF Package Height Model

The HEIGHT property also controls package height and can be sourced from the Allegro Design Entry HDL Part Table File (PTF). For discrete parts, whose physical footprints are identical except for height variations due to multiple manufacturers, use the PTF package height model, which minimizes design disruption as front-end librarians may already be using this property for IDF support.

When creating the physical footprint, ensure that no PACKAGE_HEIGHT_MAX property is assigned to place-bound shapes. Only those symbols whose height is driven from the schematic require this change. (Any existing HEIGHT properties assigned to package symbols take precedence.)

To allow the DRC system to use the component-definition HEIGHT property driven from the PTF, choose *File – Import– Logic* (`netin` command) to map the component-definition HEIGHT property currently used by the IDF interface to the PACKAGE_HEIGHT_MAX property on the component definition.

Values for the HEIGHT and the PACKAGE_HEIGHT_MAX properties may be specified in any legal length unit: The HEIGHT property value is maintained in user units in the database; the PACKAGE_HEIGHT_MAX value is converted to the current board design units.

Because the HEIGHT property is defined as a component property in Allegro it may be passed forward to Allegro from an Allegro Design Entry HDL netlist. Its value cannot be changed in the Allegro database as it is device and netlist driven.

Define the HEIGHT property in one or more of the following locations. When the design is packaged, Packager XL applies the first HEIGHT value found in the following order of precedence.

- as a body property in the symbol definition
- in the part table as either a key or injected property
- the `chips.prt` file as a body property

However, the component may have only one HEIGHT property value. If the component's actual height is irregular, the varying heights of its profile cannot be described using a HEIGHT property, and component-to-component or component-to-package-keepout DRC audits ignore the HEIGHT property's value.

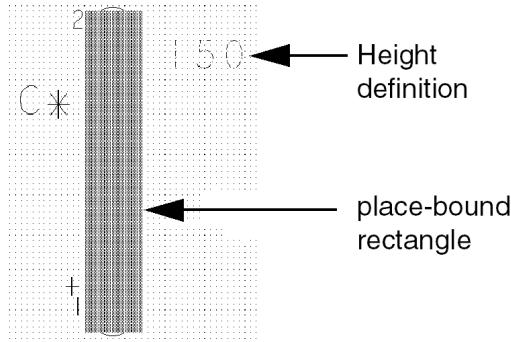
The `idf_out` utility passes a component's HEIGHT value when both of the following conditions are met:

- No PACKAGE_HEIGHT_MIN and/or PACKAGE_HEIGHT_MAX properties are attached to the place-bound shapes defined in the component instance's symbol
- The environment variable `idf_ignore_comp_height` is not set

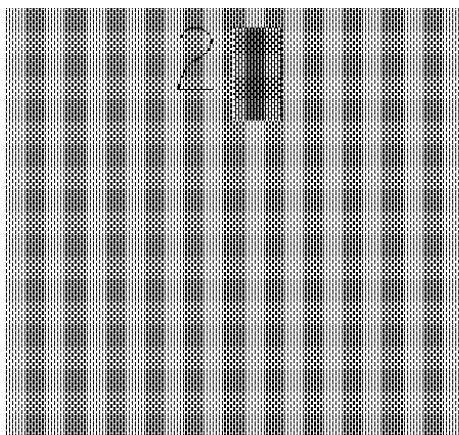
Adding Symbol Height Text to a Place-Bound Rectangle

You can attach a symbol height label to a place-bound rectangle that specifies a height range, which consists of a maximum and minimum height value that defines the z-dimension (depth) of the package.

If you attach only one height value, the layout editor assumes the specified value is the maximum height value and 0 is the minimum. Symbol height is only used with package symbols.



You can choose a location right on the rectangle, as shown below.



Location picks are shown in the status bar. You are automatically put into the attach text function mode, as shown in the status area. The layout editor prompts you to enter a text string.

 If you want to change the symbol height text for the package bound after the component is placed, choose *Edit – Text* (`text edit` command).

Related Topics

- [Package/Component Symbol Library](#)
- [Padstack Editor](#)
- [mirror](#)
- [prmed](#)
- [package symbol wizard](#)
- [Creating a Package Symbol Manually](#)
- [Creating a Package in the Batch Mode](#)
- [uprev](#)
- [package_height](#)
- [property edit](#)
- [netin](#)
- [idf_out Batch Command](#)
- [text edit](#)

Creating Mechanical Symbols

Mechanical symbols physically represent mechanical elements in a design.

 [PCB Editor: Mechanical Symbol Library](#) shows the mechanical symbols in the editor library.

Types of Mechanical Symbols

A mechanical symbol can be one of the following:

- An element relating to manufacturing, such as
 - Design outline
 - Tooling corners
 - Plating bars
 - Mounting holes
 - Dimensions
 - Areas (shapes) to be photoplotted
 - Areas not to be glossed
- Keepin and keepout areas
 - Package
 - Package
 - Route keepin
 - Route keepout
 - Via keepout
- Pins without pin numbers (for mounting holes)
- Non-connecting ETCH/CONDUCTOR elements, such as logos on ETCH/CONDUCTOR layers

You can create each mechanical element as a separate symbol and store the symbol in a library for use in different designs. You use *File – Create Symbol* (`create symbol` command) in the symbol mode to convert a mechanical symbol drawing `.dra` into a mechanical symbol binary file `.bsm`.

Guidelines for Creating Mechanical Symbols

When you create a mechanical symbol, follow these guidelines:

- Create mechanical elements on their related classes:

BOARD	Includes elements relating to manufacturing: design outline, tooling corners, plating bar, dimensions, targets, and mounting holes. You create various geometries by placing basic element types, lines, arcs, text, rectangles, and shapes, into subclasses of the appropriate name.
ETCH/CONDUCTOR	Non-connecting ETCH/CONDUCTOR elements only. For example, to add a company logo on an ETCH/CONDUCTOR layer.
PACKAGE	A single unfilled shape inside which you can place package symbols.
PACKAGE ETCH/CONDUCTOR	Any number of filled shapes. Each shape defines an area where no package symbols may be placed.
ROUTE KEEPIN	A single unfilled shape inside which ETCH/CONDUCTOR you can route.
ROUTE KEEPOUT	Shape identifying areas where ETCH/CONDUCTOR may not be placed.
VIA KEEPOUT	Shape identifying areas where vias may not be placed.
PINS	Identifies pins that do not have pin numbers. These pins are used to define mounting holes and registration holes in the layout geometry.
MANUFACTURING	Shapes identifying areas of the layout to be photoplotted; shapes identifying areas of the layout not to gloss.

- Start a mechanical symbol drawing (an outline, for example) with some mechanical element, such as a tooling hole, in the lower left corner, at 0,0. This helps maintain the relationship between the symbol drawing origin and the mechanical drawing when placing the mechanical symbol in another design.
- If you have mechanical elements (such as design outlines) that are non-standard and unique to a particular design, and have no mounting holes (pins), create such mechanical elements as part of the design .brd file, by choosing *Add – Line* ([add line](#)) or *Add – Rectangle* ([add rect](#)) commands.

Creating a Board Outline

Before creating a board outline, refer to [Guidelines for Creating Mechanical Symbols](#). For procedural information, see *File – Create Symbol* (`create symbol` command) in the *Allegro PCB and Package Physical Layout Command Reference*.

Creating Format Symbols

With the layout editor, you can create various format elements in format symbol drawings for standard drawing sizes. You can edit or customize these format symbol elements for unique situations. You can also store these format symbols in a library so that you can easily access standard ones for designs without having to re-create these symbols for each design.

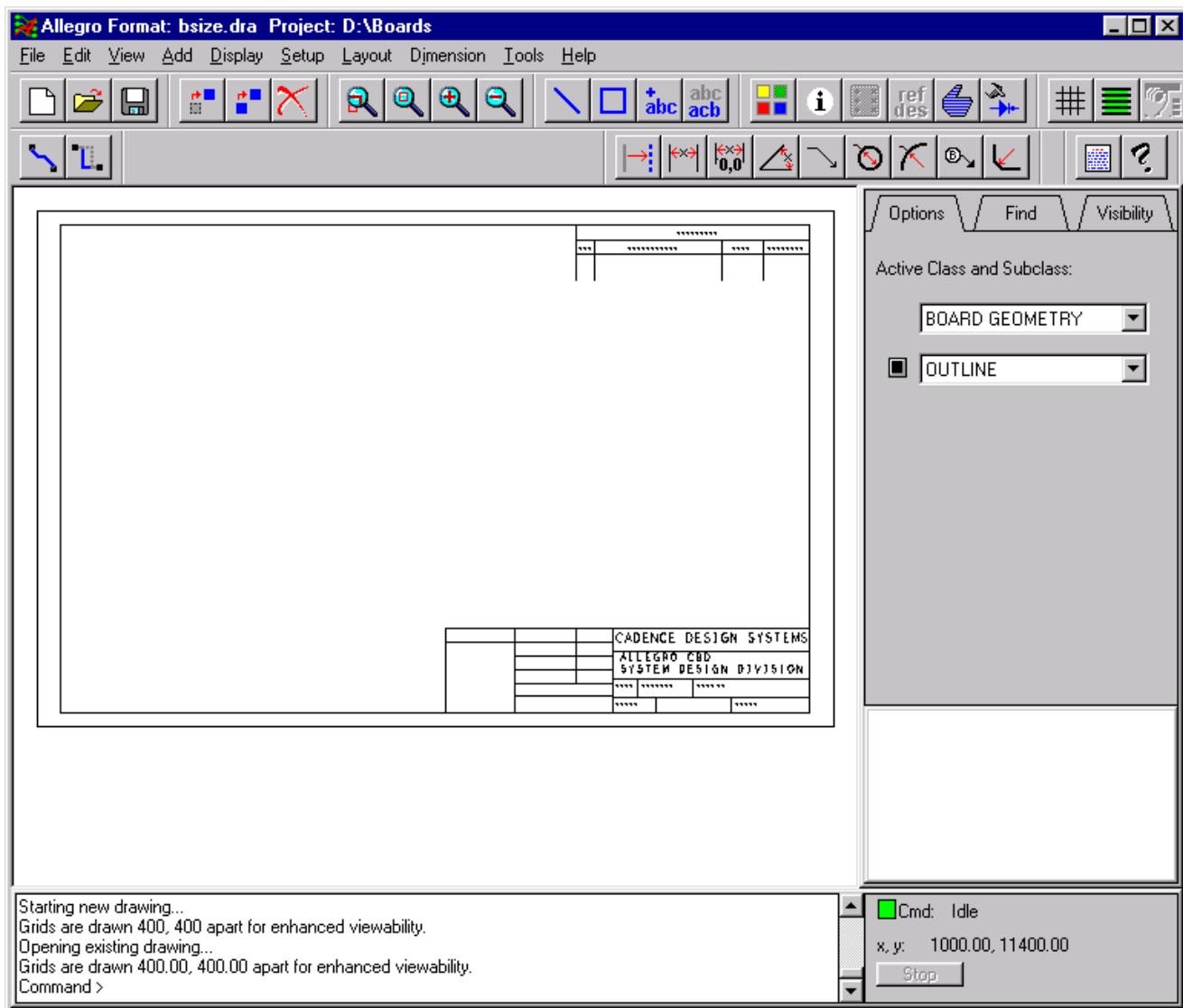
Library Format Symbols

Format symbols in the the layout editor library are standard drawing formats that contain the following:

- Borders (outline)
- Title blocks
- Revision blocks

The the layout editor library contains a separate format symbol for each standard drawing size. Format symbols have an .osm file name extension.

The following sample format symbol shows the format elements included in a format symbol. This format has a border, a revision block, and a title block for a B size drawing.



Guidelines for Creating Format Symbols

When you create format symbols that can be used for a variety of designs:

- Create one symbol for each drawing size.
Format drawings are not scalable, so plan to create one for each drawing size used by your company. When you create a new drawing, the layout editor offers the four standard sizes—A, B, C, and D—as options.
- Create format elements on their related subclasses.
- Include all the format elements that might be needed for the design, then use the visibility controls for each subclass to determine which elements to display.

- Before you convert the drawing to a symbol, verify the origin of the symbol. Choose an origin that facilitates placing the format into the design drawing.

Use the message bars at the bottom of the editor window to see what text has been typed in, the x,y location of the cursor in the active window, or what command is currently active.

Related Topics

- [PCB Editor: Format Symbol Library](#)
- [create symbol](#)

Creating Flash Symbols

Flash symbols are pads that you create for photoplotting using standard apertures. For raster formats (Gerber RS-274X, Barco DPF, and McDonald Dettwiler MDA), you can include definitions for all apertures in the artwork file. For standard apertures and pads-as-shapes, you can derive the aperture definition from the design.

For aperture flashes, however, the design does not contain information about the geometry required for flashes referenced in the padstack data if the design has not been flash-converted from pre-Release 14.0 versions of the editor software (as described in [Choosing a Design Methodology for Negative Planes](#)). In designs that have not been flash-converted, the layout editor displays only a circle with a cross-wire for the flash.

In new or flash-converted designs, information is stored as flash (.fsm) symbols. Designs created in Release 14.0 (and later) or flash-converted from earlier designs contain geometry information on all thermal flash symbols.

You can create flash symbols that display in WYSIWYG mode in three ways:

- As a new flash symbol in the symbol mode

Choose *File – New* ([new](#) command) to create a flash symbol such as a thermal pad for Raster formats. The layout editor saves flash symbols to the symbol library and appends the file name that you specify with an .fsm extension.

- From an existing shape symbol .dra file in the symbol mode

Choose *Setup – Design Parameters* ([prmed](#) command), the *Design* tab of the Design Parameter Editor, and *Flash* in the *Drawing Type* section. When you save the change with the *create symbol* command, the .dra is saved as a flash symbol.

- From existing flash .bsm files

Use the `flash_convert` command when updating pre-14.0 designs for use with later editor versions.

The character limit for flash names is 18.

For procedural information, see *File – Create Symbol* (`create symbol` command) in the *Allegro PCB and Package Physical Layout Command Reference*.

Choosing a Design Methodology for Negative Planes

The following sections describe the processes required to design databases with negative planes. Beginning with version 14.0, you can choose an old or new methodology for designing with negative planes.

Methodology Differences

Designs created in the layout editor starting with version 14.0 come up automatically in the new methodology; older databases open in the pre-14.0 manner. You must run the conversion program `flash_convert` on all older databases that you open in 14.0 or later versions, regardless of which methodology in which you want to work, to convert .bsm flash files to .fsm files. (See [Converting Flash Symbols When Migrating Databases](#) for details.) When you convert from one methodology to the other, .bsm files referenced locally or in the PSMPATH environment variable are converted to .fsm files and then imported into the design.

 See [Treatment of Nonconforming Symbols](#) for instances when you may not want to run `flash_convert`.

Important: Once you have migrated a pre-14.0 database to the new methodology, it cannot be converted back to the old style.

You can choose one of the following methodologies in which to work.

Old Style Methodology

In this mode, flash information is not stored in the design, as is the case in pre-14.0 versions of the layout editor. Rather, it is stored in the library referenced by the PSMPATH environment variable. This style lets you work with flash files as you did in earlier versions for artwork, display, and plotting.

- For pre-14.0 databases in the old methodology, run `flash_convert` on the design to update flash symbols from `.bsm` to `.fsm` files.
- For new databases created in the old methodology, you need to:
 - Set the environment variable `OLD_STYLE_FLASH_SYMBOLS`.
 - Run `flash_convert` on any `.bsm`-formatted flash symbols referenced in the new design.

New Style Methodology

This is the default style for new databases.

 See [Treatment of Nonconforming Symbols](#) for instances when you may not want to run `flash_convert` on certain new designs.

- Run `flash_convert` if you import any `.bsm`-formatted flash symbols into the design then use the `refresh symbol` command to update the flash symbols.

See [Updating Symbols](#) about refreshing symbols.

Converting Flash Symbols When Migrating Databases

Flash symbols in databases are referenced as `.fsm` files. The `flash_convert` command lets you migrate older databases to the new methodology. You can choose to define flash symbols interactively for the database you are currently working in, or for one or more designs in a project hierarchy.

In most cases, `.bsm` symbols in a database are not referenced after running `flash_convert`. This is true regardless of the methodology you are using. The exception is that the `artwork` and `load gerber` commands reference a `.bsm` file if an `.fsm` file cannot be found.

Flash Symbols in Padstack Designer

As long as you use the same name for flash names used in a padstack and flash symbols in a library of shape symbols, you can use the same thermal relief padstacks in either design methodology, because Padstack Designer maintains pad neutrality with respect to .fsm files. In instances where both flash symbol names and shape symbol names are referenced, the methodology in which you are working determines which symbol name takes precedence. If you are working in the old style, the shape figure is referenced; if in the new style, the flash figure is referenced using the PSMPATH environment variable.

Actions that you take in Padstack Designer have no affect on the conversion processes described previously.

Treatment of Nonconforming Symbols

In cases where the creation of .bsm files has been made using line segments rather than shapes or where voided shapes are used in the board symbol, `flash_convert` is unable to migrate a design's .bsm files to .fsm files. Where this occurs, options are available, based on the methodology you are using.

Using the Old Methodology

- Use `flash_convert` to migrate all shape-based, non-void .bsm files to .fsm files.
Artwork uses the unconverted .bsm flash when unable to find the .fsm.

Using the New Methodology for Unconverted Antipad Flashes

1. Create in the local directory of the database dummy .bsm files of any shape geometry for each antipad flash.
2. Run `flash_convert` on the database.
The thermal flashes are converted to dummy .fsm files and loaded into the database in WYSIWYG mode.
3. Delete the dummy .bsm and .fsm files.
The database now contains thermal flashes in WYSIWYG mode. However, artwork does not detect unconverted antipad flashes, so it uses the existing flash.bsm files.

MDA Format Output Files

For films that include antipads and thermal flashes, MDA format requires two artwork files. For example, when you specify a film named 5v for a layer that contains antipads or thermal flashes, the layout editor generates the following files:

- 5v.art
- 5v_s.art

MDA format uses paint and scratch commands. The `_s` suffix is for the file with the scratch commands.

Defining a Flash Symbol

To successfully convert an older database to the new methodology or to use new padstacks in new databases, you must provide flash symbol information; that is, you must have defined padstack flashes in `.fsm` files. For thermal flashes without `.bsm` files, you must create flash symbols that are used as thermal pads on a negative plane.

For procedural information, see *File – Create Symbol* (`create symbol` command) in the *Allegro PCB and Package Physical Layout Command Reference*.

Updating Symbols

To ensure that you are using the latest symbols in a design, the layout editor lets you update symbols and symbol padstacks from a library. You can do this interactively in the editor's layout mode by choosing *Place – Update Symbols* (`refresh_symbol` command), *Tools – Update Symbols* in editor's symbol mode, or by running the `refresh_symbol` batch command from an operating system prompt.

The `refresh.log` file records refresh symbol processing.

Related Topics

- [refresh symbol](#)
- [refresh_symbol](#)

Checking Symbols Automatically

During the development of physical symbols, such as component footprint symbols, you verify the existence of symbol elements, element layer definition, properties, and other criteria manually. However, manual verification can inadvertently add or omit some of these elements, and you might overlook such errors.

To automate the verification of physical symbols, choose *Tools – Check Symbol* (`check_symbol` command).

Related Topics

- [check symbol](#)

Configuring the check symbol Command

You can customize the *Tools – Check Symbol* menu in the following ways:

- Modify the values of the global variables. These variables contain information such as the error messages. The global variables and its values are stored in the `rule_check_globals.il` file. This file is also called the Globals file.
- Add rules against which checks can be run. The rules are written in the AXL-SKILL language and then added to the `rule_check_tables.il` file. This file is also called the Rule Table file.

Globals File

The `rule_check_globals.il` file contains global variables whose values you can change using any text editor. This file is an AXL-SKILL command file that defines one function, `(_PAC_setUserGlobals())`, for the assignment of global variables.

While you should not change the names of the global variables, you can change the values for these variables, which lets you define error types for specific checks. There are some global variables that have recommended values and should not be changed but are available in case you want to customize error severity.

You can also change messages that appear in this file.

This file must be located in the directory defined by the `SKILLPATH` environment variable. If stored elsewhere, the symbol check uses the default values.

Rule Table File

The `rule_check_tables.il` file defines and assigns rule classes, rules that fall into those classes, and what action to take when a rule is marked for execution. The `rule_check_tables.il` file is an AXL-SKILL program file that defines one function,

(`_PAC_setUserFormTreeData()`), for defining the rules tree in the *Physical Symbol Attributes Check* dialog box. You can edit this file using any text editor.

This file must be located in the directory defined by the `SKILLPATH` environment variable. If stored elsewhere, the symbol check uses the default definitions.

The format of this file is a series of embedded lists. The outermost list is the one that is passed to the `check symbol` command. The next tier of lists is the group or rules class. The first item in this list is the name of the rules class followed by a number of list pairs. This lowest level of lists contains the name of the rule followed by the AXL-SKILL function call.

The syntax is as follows:

```
list(  
    list( ["group_name1"]  
        list(  
            list(["check_name1"] ["function_call"])  
            list(["check_name2"] ["function_call"])  
        )  
    )  
)
```

The following example displays a sample Rule Table file:

```
Defun( _PAC_SetUserFormTreeData  
list(  
list( "REFDES Checks"  
    list( '("REFDES Exist" "check_refdes_exist(file_ptr)")  
    ' ("REFDES Text Size" "check_refdes_blocksize(file_ptr)")  
    ' ("REFDES Subclass" "check_refdes_subclass(file_ptr)")  
)  
)  
list( "COMPONENT VALUE Checks"  
    list( '("COMPONENT VALUE Exist" "check_compvalue_exist(file_ptr)")  
    ' ("COMPONENT VALUE Text Size" "check_compval_blocksize(file_ptr)")  
    ' ("COMPONENT VALUE Subclass" "check_compval_subclass(file_ptr)")  
)  
)  
list( "DEVICE TYPE Checks"  
    list( '("DEVICE TYPE Exist" "check_device_exist(file_ptr)")  
    ' ("DEVICE TYPE Text Size" "check_device_blocksize(file_ptr)")  
    ' ("DEVICE TYPE Subclass" "check_device_subclass(file_ptr)")  
)
```

```
)  
list( "TOLERANCE Checks"  
    list( '("TOLERANCE Exist" "check_prttol_exist(file_ptr)")  
        '("TOLERANCE Text Size" "check_prttol_blocksize(file_ptr)")  
        '("TOLERANCE Subclass" "check_prttol_subclass(file_ptr)")  
    )  
)  
  
list( "PART NUMBER Checks"  
    list( '("PART NUMBER Exist" "check_prtnum_exist(file_ptr)")  
        '("PART NUMBER Text Size" "check_prtnum_blocksize(file_ptr)")  
        '("PART NUMBER Subclass" "check_prtnum_subclass(file_ptr)")  
    )  
)  
  
list( "GEOMETRY Checks"  
    list( '("SILK SCREEN Geometry" "check_silkscreen_geometry(file_ptr)")  
        '("ASSEMBLY Geometry" "check_assembly_geometry(file_ptr)")  
    )  
)  
  
list( "Reports"  
    list( '("PIN Location Report" "report_pin_location(file_ptr log_file)")  
    )  
)
```

Developing Symbol Check Rules

The `check symbol` command lets you create rules and add them to the command. While writing the rules, you need to follow certain guidelines. These pertain to:

- Choosing the Language for the Source Code
- Deciding on Function Inputs
- Deciding on Function Returns
- Writing to the Marker File
- Writing to the Log File

Choosing the Language for the Source Code

You must use the AXL-SKILL extension language available in the layout editor to develop the source code for symbol rules. You can create as many functions as needed, but there must be one function call that the `check symbol` command calls, which requires the passing of at least one variable into the function to record information into the markers file. A secondary variable might be passed to record information in the log file.

Deciding on Function Inputs

There are two variables that need to be provided to the rule set function call:

- The `file_ptr` variable is used to pass information to the markers file.
- The `log_file` variable is the file pointer to the `<design_name>_symchk.log` file generated when the checks are run.

Deciding on Function Returns

The rule function call must return a list of two elements:

- The number of errors found.
- The number of warnings found during the rule check.

```
return( list( errors warnings))
```

Writing to the Marker File

The marker file (`<design_name>_symchk.mkr`) contains messages and location-specific information for any element found in the drawing that may be an error, warning, or message. You call the `_PAC_report_errors()` function to write to the marker file.

The complete syntax is as follows:

```
_PAC_report_errors( file_ptr <error_type> <short_message> <long_message> <object_kind> <object_name> <parent_name> <design_name> )
```

The function takes the following parameters:

<code>file_ptr</code>	The pointer to the output port for the markers file.
<code>error_type</code>	An integer value that states what type of message is being displayed. Typically, you use one of the global variables, such as <code>ERROR</code> , <code>WARNING</code> , and so on.
<code>short_message</code>	An ASCII string that gives a brief description of the issues being reported. This is a single-string entry.
<code>long_message</code>	An ASCII string that gives a detailed description of the issue being reported. This is a single-string entry.
<code>object_kind</code>	An ASCII string for the name of the element type. The element type is either <code>PIN</code> or <code>TEXT</code> .

<i>object_name</i>	An ASCII string that denotes the X/Y location and the class and subclass description of the item. The string should follow the following format: <code>x.x:y.y=class/subclass</code> For example: <code>1.000:1.250=PACKAGE GEOMETRY/ASSEMBLY_TOP</code>
<i>parent_name</i>	An ASCII string that describes the element type and X/Y location. This is the format for the string: <code>item@(x.x y.y)</code> For example: <code>PIN@ (3.450 4.623)</code>
<i>design_name</i>	An ASCII string that describes the drawing or drawing element

An example of the `_PAC_report` errors is given below.

Assume that the following global variables are defined in the Global file:

```
TEST_CASE_ERROR = ERROR
TEST_CASE_SHORT = "Test Element missing"
TEST_CASE_LONG = "Missing Add Test Element at origin.\n Please add."
```

The function call would be:

```
_PAC_report_errors( file_ptr TEST_CASE_ERROR TEST_CASE_SHORT TEST_CASE_LONG "TEXT"
"0.00:0.00=PACKAGE_GEOMETRY/DISPLAY_TOP"
"CIRCLE@(0.00 0.00)" "test_symbol.dra")
```

Writing to the Log File

You can write to the log file (`<design_name>_symchk.log`) using the standard `fprintf` functions available in AXL-SKILL. The pointer variable to the output port for the log file is `log_file`. You must pass this pointer into the function.

For example:

```
Defun( your_function  ( file_ptr log_file)
Prog( ()
Code
if( errors < 1 then
 if(warnings < 1
```

```
fprintf( log_file "This Check Passed\n")
else
  fprintf( log_file "This check has warnings\n")
);end-if
else
  fprintf( log_file "This check has errors\n")
); end-if
return( list( errors warnings))
))
```

Installing Custom Rules

Installing the rules that you have created involves two activities:

- Modifying the user-defined global variable, if required.
- Including the rules in the check symbol command so that they appear in the `check symbol` dialog box.

Modifying the User-Defined Global Variables

The Global variable for setting messages and error/checking type values should be created in the `rule_check_globals.il` file. This is a central global variable value repository for all checking functions. Formats can be observed in the file and may be copied or altered as required.

Do not change the values for the ERROR, WARNING, and INFO definitions.

Do not remove or change the name of existing GLOBAL variables defined in this file. You can change values, but do not change the variable names.

Including Rules in the `check symbol` Command

To include a new custom rule, store the SKILL program in a directory to which all those who use the program have at least read access. Then, load the SKILL file upon opening the Symbol Editor. This can be done through the `allegro.ilinit` file. The command or function should then be added in the `rules_check_table.il` file.

This file is a function that returns a list of lists. The format of the list can not be changed. The first list level is the inclusion all lists, with the next level being the category or rules class definition. The last level of lists is lists of paired values, the rule name, and the specific SKILL function call for that rule.

```
List(
  list( category1
    list( rule_name1 skill_function1)
```

```
list( rule_name2 skill_function2)
)
list( category2
    list( rule_name1 skill_function1)
    list( rule_name2 skill_function2)
    list( rule_name3 skill_function3)
)
list( category3
    list( rule_name2 skill_function2)
    list( rule_name1 skill_function1)
)
)
```

Example

```
list( "misc"
    list( "Text Rotation" "check_text_rotation( file_ptr )")
    list( "Print All Text" "print_out_all_text( file_ptr log_file )")
)
```

Saving the Custom Rules

The SKILL program file created with the custom rules set should be saved in a directory that users that require use of the rules have access to. The same location as the current rule set and globals file reside would be a good suggestion.

Initializing the Custom Rules

The Allegro tool environment uses an `allegro.ilinit` file to locate SKILL program files. This file exist in the `pcbenv` directory. Edit this file, and add a line at the bottom of the file using the load function call. A literal path may be required.

```
load("my_test_program.il")
```

Related Topics

- [Global Variables and Values](#)

Predefined Rules for Checking Symbols

The `check symbol` command includes seven sets of rules, which appear in the *Physical Symbol Attributes Check* dialog box:

- REFDES Checks
- COMPONENT VALUE Checks
- DEVICE TYPE Checks
- TOLERANCE Checks
- USER PART NUMBER Checks
- GEOMETRY Checks
- Reports

REFDES Checks

This contains three rules that check for the existence, text block size, and the layer definition for each reference designator in the package or mechanical symbol.

REFDES Exist

This rule checks for the existence of the Reference designator in the footprint. The check status is defined by the global variable as an error. For this rule, the following global variables are defined:

```
TEXT_REFDES_ERROR = ERROR  
  
TEXT_REFDES_SHORT = "No REFDES Text in Symbol"  
  
TEXT_REFDES_LONG = "No REFDES Text in Symbol \n REFDES Text Has Not Been Defined in this Symbol"
```

REFDES Text Size

This rule checks for the text block size of the reference designator in the footprint. The Text Size is verified using the global variable `TEXT_REFDES_TEXT_BLOCK`, which is set to the default of 3. The check status is defined by the global variable as an error. For this rule, the following global variables are defined:

```
TEXT_REFDES_TEXT_BLOCK = "3"  
  
TEXT_REFDES_SIZE_ERROR = ERROR  
  
TEXT_REFDES_SHORT = "Incorrect Text Size For REFDES Text"  
  
TEXT_REFDES_LONG = "Text Size for REFDES is Incorrect"
```

REFDES Subclass

This rule checks for the CLASS/SUBCLASS layer where the REFDES text should be in the footprint symbol. The CLASS/SUBCLASS is verified using the global variable *TEXT_REFDES_SUBCLASS*, which is set to the default value `list("REF DES/ASSEMBLY_TOP" "REF DES/SILKSCREEN_TOP")`. The check status is defined by the global variable as an error. For this rule, the following global variables are defined:

```
TEXT_REFDES_SUBCLASS = list("REF DES/ASSEMBLY_TOP" "REF DES/SILKSCREEN_TOP")  
  
TEXT_REFDES_SUBCLASS_ERROR = ERROR  
  
TEXT_REFDES_SUBCLASS_SHORT = "Incorrect REFDES Text Subclass Assignment."  
  
TEXT_REFDES_SUBCLASS_LONG = "Subclass Assignment For REFDES Text Not\\non an Allowed Layer."  
  
TEXT_REFDES_SUBCLASS_MISSING = "REFDES Text Missing Subclass Definition"
```

COMPONENT VALUE Checks

This contains three rules that check for the existence, text block size, and the layer definition for the component value in the package or mechanical symbol.

COMPONENT VALUE Exist

This rule checks for the existence of the component value text in the footprint. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_COMP_VALUE_ERROR = WARNING  
  
TEXT_COMP_VALUE_SHORT = "No PART VALUE Text in Symbol"  
  
TEXT_COMP_VALUE_LONG = "No PART VALUE Text in Symbol \\n COMPONENT VALUE Text Has Not Been  
Defined in this Symbol"
```

COMPONENT VALUE Text Size

This rule checks for the text block size of the component value in the footprint. The Text Size is verified using the global variable *TEXT_COMP_VALUE_TEXT_BLOCK*, which is set to the default of 3. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_COMP_VALUE_TEXT_BLOCK = "3"  
  
TEXT_ COMP_VALUE _SIZE_ERROR = WARNING  
  
TEXT_ COMP_VALUE _SHORT = "Incorrect Text Size For COMPONENT VALUE Text"  
  
TEXT_ COMP_VALUE _LONG = "Text Size for COMPONENT VALUE is Incorrect"
```

COMPONENT VALUE Subclass

This rule checks for the CLASS/SUBCLASS layer where the component value text should be in the footprint symbol. The CLASS/SUBCLASS is verified using the global variable *TEXT_COMP_VALUE_SUBCLASS*, which is set to the default value `list ("COMPONENT VALUE/ASSEMBLY_TOP")`. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_COMP_VALUE_SUBCLASS = list ("COMPONENT VALUE/ASSEMBLY_TOP" )  
  
TEXT_ COMP_VALUE _SIZE_ERROR = WARNING  
  
TEXT_ COMP_VALUE _SUBCLASS_SHORT = "Incorrect COMPONENT VALUE Text Subclass Assignment."  
  
TEXT_ COMP_VALUE _SUBCLASS_LONG = "Subclass Assignment For component Value Text Not\nnon an  
Allowed Layer."  
  
TEXT_ COMP_VALUE _SUBCLASS_MISSING = "COMPONENT VALUE Text Missing Subclass Definition"
```

DEVICE TYPE Checks

This contains three rules that check for the existence, text block size, and layer definition for the device type in the package or mechanical symbol.

DEVICE TYPE Exist

This rule checks for the existence of the device type text in the footprint. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_DEVICE_ERROR = WARNING  
  
TEXT_DEVICE_SHORT = "No DEVICE Text in Symbol"  
  
TEXT_DEVICE_LONG = "No DEVICE Text in Symbol \n DEVICE Text Has Not Been Defined in this Symbol"
```

DEVICE TYPE Text Size

This rule checks for the text block size of the device type in the footprint. The text size is verified using the global variable *TEXT_DEVICE_TEXT_BLOCK*, which is set to the default value of "3". The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_DEVICE_TEXT_BLOCK = "3"  
  
TEXT_ DEVICE _SIZE_ERROR = WARNING  
  
TEXT_ DEVICE _SHORT = "Incorrect Text Size For DEVICE Text"  
  
TEXT_ DEVICE _LONG = "Text Size for DEVICE is Incorrect"
```

DEVICE TYPE Subclass

This rule checks for the CLASS/SUBCLASS layer where the device type text should be in the footprint symbol. The CLASS/SUBCLASS is verified using the global variable TEXT_DEVICE_SUBCLASS, which is set to the default value `list("DEVICE TYPE/ASSEMBLY_TOP")`. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_DEVICE_SUBCLASS = list("DEVICE TYPE/ASSEMBLY_TOP" )  
  
TEXT_DEVICE_SUBCLASS_ERROR = WARNING  
  
TEXT_DEVICE_SUBCLASS_SHORT = "Incorrect Subclass Assignment For DEVICE Text."  
  
TEXT_DEVICE_SUBCLASS_LONG = "Subclass Assignment For DEVICE Text Not\\non an Allowed Layer."  
  
TEXT_DEVICE_SUBCLASS_MISSING = "DEVICE Text Missing Subclass"
```

TOLERANCE Checks

This contains three rules that check for the existence, text block size, and layer definition for component tolerance in the package or mechanical symbol.

TOLERANCE Exist

This rule checks for the existence of the tolerance text in the footprint. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_PART_TOL_ERROR = WARNING  
  
TEXT_PART_TOL_SHORT = "No TOLERANCE Text in Symbol"  
  
TEXT_PART_TOL_LONG = "No TOLERANCE Text in Symbol \\n TOLERANCE Text Has Not Been Defined in  
this Symbol"
```

TOLERANCE Text Size

This rule checks for the text block size of the tolerance in the footprint. The text size is verified using the global variable `TEXT_PART_NUMBER_TEXT_BLOCK`, which is set to the default value of 3. The check status is defined by the global variable as a warning. For the rule, the following global variables are defined:

```
TEXT_PART_TOL_TEXT_BLOCK = "3"  
  
TEXT_PART_TOL_SIZE_ERROR = WARNING  
  
TEXT_PART_TOL_SHORT = "Incorrect Text Size For TOLERANCE Text"  
  
TEXT_PART_TOL_LONG = "Text Size for TOLERANCE is Incorrect"
```

TOLERANCE Subclass

This rule checks for the CLASS/SUBCLASS layer where the tolerance text is to be in the footprint symbol. The CLASS/SUBCLASS is verified using the global variable *TEXT_PART_TOL_SUBCLASS*, which is set to the default value `list("TOLERANCE/ASSEMBLY_TOP")`. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_PART_TOL_SUBCLASS = list("TOLERANCE /ASSEMBLY_TOP" )  
  
TEXT_PART_TOL_SIZE_ERROR = WARNING  
  
TEXT_PART_TOL_SUBCLASS_SHORT = "Incorrect TOLERANCE Text Subclass Assignment."  
  
TEXT_PART_TOL_SUBCLASS_LONG = "Subclass Assignment For TOLERANCE Text Not\nnon an Allowed Layer."  
  
TEXT_PART_TOL_SUBCLASS_MISSING = "TOLERANCE Text Missing Subclass Definition"
```

USER PART NUMBER Checks

This rule class contains three rule sets that check for the existence, text block size, and layer definition for the part number in the package or mechanical symbol.

USER PART NUMBER Exist

This rule checks for the existence of the part number text in the footprint. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_PART_NUMBER_ERROR = WARNING  
  
TEXT_PART_NUMBER_SHORT = "No USER PART NUMBER Text in Symbol"  
  
TEXT_PART_NUMBER_LONG = "No USER PART NUMBER Text in Symbol \n PART NUMBER Text Has Not Been  
Defined in this Symbol"
```

USER PART NUMBER Text Size

This rule checks for the text block size of the component value in the footprint. The text size is verified using the global variable *TEXT_PART_NUMBER_TEXT_BLOCK*, which is set to the default value of `3`. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_PART_NUMBER_TEXT_BLOCK = "3"  
  
TEXT_PART_NUMBER_SIZE_ERROR = WARNING  
  
TEXT_PART_NUMBER_SHORT = "Incorrect Text Size For USER PART NUMBER Text"  
  
TEXT_PART_NUMBER_LONG = "Text Size for USER PART NUMBER is Incorrect"
```

USER PART NUMBER Subclass

This rule checks for the CLASS/SUBCLASS layer that the part number text should be in the footprint symbol. The CLASS/SUBCLASS is verified using the global variable *TEXT_PART_NUMBER_SUBCLASS*, which is set to the default value `list("USER PART NUMBER /ASSEMBLY_TOP")`. The check status is defined by the global variable as a warning. For this rule, the following global variables are defined:

```
TEXT_PART_NUMBER_SUBCLASS = list("USER PART NUMBER /ASSEMBLY_TOP")  
  
TEXT_PART_NUMBER_SIZE_ERROR = WARNING  
  
TEXT_PART_NUMBER_SUBCLASS_SHORT = "Incorrect PART NUMBER Text Subclass Assignment."  
  
TEXT_PART_NUMBER_SUBCLASS_LONG = "Subclass Assignment For PART NUMBER Text Not\nnon an Allowed Layer."  
  
TEXT_PART_NUMBER_SUBCLASS_MISSING = "PART NUMBER Text Missing Subclass Definition"
```

GEOMETRY Checks

This rule class contains two rule sets that check for the existence of geometric data in the package or mechanical symbol.

SILK SCREEN Geometry

This rule checks for the existence of the arcs, lines, and shapes defined on the PACKAGE GEOMETRY class and TOP and bottom silkscreen subclasses in the footprint. The check status is defined by the global variable as an error. For this rule, the following global variables are defined:

```
SILKSCREEN_GEOMETRY_ERROR = ERROR  
  
SILKSCREEN_GEOMETRY_SHORT = "No Silk Screen Geometry in Symbol"  
  
SILKSCREEN_GEOMETRY_LONG = "No Silk Screen Geometry in Symbol\nSilk Screen Graphical Data Will Not Exist"
```

ASSEMBLY Geometry

This rule checks for the existence of arcs, lines and shapes defined on the PACKAGE GEOMETRY class, and the ASSEMBLY_TOP and ASSEMBLY_BOTTOM subclasses. The check status is defined by the global variable as an error. For this rule, the following global variables are defined:

```
ASSEMBLY_GEOMETRY_ERROR = ERROR  
  
ASSEMBLY_GEOMETRY_SHORT = "No Assembly Geometry in Symbol"  
  
ASSEMBLY_GEOMETRY_LONG = "No Assembly Geometry in Symbol\nAssembly Graphical Data Will Not Exist"
```

Reports

This check allows report data to be generated as output. This contains one rule that reports the pin number, the XY location and the padstack name for each pin in a symbol drawing. This rule is hard-coded because check status INFO and cannot be modified.

Global Variables and Values

Variable Name	Type	Default Value
ERROR	Integer (Do not change)	40
WARNING	Integer (Do not change)	30
INFO	Integer (Do not change)	20
ASSEMBLY_GEOMETRY_SHORT	String (short message)	"No Assembly Geometry in Symbol."
ASSEMBLY_GEOMETRY_LONG	String (long message)	"No Assembly Geometry in Symbol \n Assembly Graphical Data Will Not Exist"
ASSEMBLY_GEOMETRY_ERROR	Integer	ERROR
SILKSCREEN_GEOMETRY_SHORT	String (short message)	"No Silk Screen Geometry in Symbol."
SILKSCREEN_GEOMETRY_LONG	String (long message)	"No Silk Screen Geometry in Symbol\n Silk Screen Graphical Data Will Not Exist"
SILKSCREEN_GEOMETRY_ERROR	Integer	ERROR
TEXT_REFDES_SHORT	String (short message)	"No REFDES Text in Symbol."
TEXT_REFDES_LONG	String (long message)	"No REFDES Text in Symbol\n REFDES Text Has Not Been Defined in This Symbol."
TEXT_REFDES_SIZE_SHORT	String (short message)	"Incorrect Text Size For REFDES Text."
TEXT_REFDES_SIZE_LONG	String (long message)	"Text Size For REFDES is Incorrect."

TEXT_REFDES_SUBCLASS_SHORT	String (short message)	"Incorrect REFDES Text Subclass Assignment."
TEXT_REFDES_SUBCLASS_LONG	String (long message)	"Subclass Assignment For REFDES Text Not \n on an Allowed Layer."
TEXT_REFDES_SUBCLASS_MISSING	String	"REFDES Text Missing Subclass Definition"
TEXT_REFDES_ERROR	Integer	ERROR
TEXT_REFDES_BLOCK_ERROR	Integer	ERROR
TEXT_REFDES_SUBCLASS_ERROR	Integer	ERROR
TEXT_COMP_VALUE_SHORT	String (short message)	"No COMPONENT VALUE Text in Symbol."
TEXT_COMP_VALUE_LONG	String (long message)	"No COMPONENT VALUE Text in Symbol \n COMPONENT VALUE Text Has Not Been Defined in This Symbol."
TEXT_COMP_VALUE_SIZE_SHORT	String (short message)	"Incorrect Text Size For COMPONENT VALUE Text."
TEXT_COMP_VALUE_SIZE_LONG	String (long message)	"Text Size For COMPONENT VALUE is Incorrect."
TEXT_COMP_VALUE_SUBCLASS_SHORT	String (short message)	"Incorrect Subclass Assignment For COMPONENT VALUE Text."
TEXT_COMP_VALUE_SUBCLASS_LONG	String (long message)	"Subclass Assignment For COMPONENT VALUE Text Not \n on an Allowed Layer."
TEXT_COMP_VALUE_SUBCLASS_MISSING	string	"COMPONENT VALUE Text Missing Subclass Definition"
TEXT_COMP_VALUE_ERROR	Integer	WARNING
TEXT_COMP_VALUE_BLOCK_ERROR	Integer	WARNING
TEXT_COMP_VALUE_SUBCLASS_ERROR	Integer	WARNING
TEXT_DEVICE_SHORT	String (short message)	"No DEVICE Text in Symbol."

TEXT_DEVICE_LONG	String (long message)	"No DEVICE Text in Symbol \n DEVICE Text Has Not Been Defined in This Symbol."
TEXT_DEVICE_SIZE_SHORT	String (short message)	"Incorrect Text Size For DEVICE Text."
TEXT_DEVICE_SIZE_LONG	string	"Text Size For DEVICE is Incorrect."
TEXT_DEVICE_SUBCLASS_SHORT	String (short message)	"Incorrect Subclass Assignment For DEVICE Text."
TEXT_DEVICE_SUBCLASS_LONG	String (long message)	"Subclass Assignment For DEVICE Text Not \n on an Allowed Layer."
TEXT_DEVICE_SUBCLASS_MISSING	string	"DEVICE Text Missing Subclass Definition"
TEXT_DEVICE_ERROR	Integer	WARNING
TEXT_DEVICE_BLOCK_ERROR	Integer	WARNING
TEXT_DEVICE_SUBCLASS_ERROR	Integer	WARNING
TEXT_PART_NUMBER_SHORT	String (long message)	"No USER PART NUMBER Text in Symbol."
TEXT_PART_NUMBER_LONG	String (long message)	"No USER PART NUMBER Text in Symbol \n PART NUMBER Text Has Not Been Defined in This Symbol."
TEXT_PART_NUMBER_SIZE_SHORT	String (long message)	"Incorrect Text Size For PART NUMBER Text."
TEXT_PART_NUMBER_SIZE_LONG	String (long message)	"Text Size For USER PART NUMBER is Incorrect."
TEXT_PART_NUMBER_SUBCLASS_SHORT	String (short message)	"Incorrect Subclass Assignment For USER PART NUMBER Text."
TEXT_PART_NUMBER_SUBCLASS_LONG	String (long message)	"Subclass Assignment For USER PART NUMBER Text Not \n on an Allowed Layer."
TEXT_PART_NUMBER_SUBCLASS_MISSING	string	"USER PART NUMBER Text Missing Subclass Definition"

TEXT_PART_NUMBER_ERROR	Integer	ERROR
TEXT_PART_NUMBER_BLOCK_ERROR	Integer	ERROR
TEXT_PART_NUMBER_SUBCLASS_ERROR	Integer	ERROR
TEXT_PART_TOL_SHORT	String (short message)	"No TOLERANCE Text in Symbol."
TEXT_PART_TOL_LONG	String (long message)	"No TOLERANCE Text in Symbol \n PART TOLERANCE Text Has Not Been Defined in This Symbol."
TEXT_PART_TOL_SIZE_SHORT	String (short message)	"Incorrect Text Size For TOLERANCE Text."
TEXT_PART_TOL_SIZE_LONG	String (long message)	"Text Size For TOLERANCE is Incorrect."
TEXT_PART_TOL_SUBLCASS_SHORT	String (short message)	"Incorrect Subclass Assignment For TOLERANCE Text."
TEXT_PART_TOL_SUBCLASS_LONG	String (long message)	"Subclass Assignment For TOLERANCE Text Not \n on an Allowed Layer."
TEXT_PART_TOL_SUBCLASS_MISSING	string	"TOLERANCE Text Missing Subclass Definition"
TEXT_PART_TOL_ERROR	Integer	WARNING
TEXT_PART_TOL_BLOCK_ERROR	Integer	WARNING
TEXT_PART_TOL_SUBCLASS_ERROR	Integer	WARNING
TEXT_REFDES_SUBCLASS	List (string class/subclass)	list("REF DES/ASSEMBLY_TOP" "REF DES/SILKSCREEN_TOP")
TEXT_COMP_VALUE_SUBCLASS	List (string class/subclass)	list("COMPONENT VALUE/ASSEMBLY_TOP")
TEXT_DEVICE_SUBCLASS	List (string class/subclass)	list("DEVICE TYPE/ASSEMBLY_TOP")
TEXT_PART_TOL_SUBCLASS	List (string class/subclass)	list("TOLERANCE/ASSEMBLY_TOP")
TEXT_PART_NUMBER_SUBCLASS	List (string class/subclass)	list("USER PART NUMBER/ASSEMBLY_TOP")

TEXT_REFDES_TEXT_BLOCK	String (text block size)	"3"
TEXT_COMP_VALUE_TEXT_BLOCK	String (text block size)	"3"
TEXT_DEVICE_TEXT_BLOCK	String (text block size)	"3"
TEXT_PART_NUMBER_TEXT_BLOCK	String (text block size)	"3"
TEXT_PART_TOL_TEXT_BLOCK	String (text block size)	"3"

Preparing Device Files

Device files are ASCII text files that provide logic information for unique components. During design database creation, links device files to related package symbols to obtain a complete description of these unique components.

Device files provide the layout editor with the following information:

- Package configuration (for example, DIP and SIP)
- Placement class of the package (for example, IC, discrete)
- Number of pins in the component package
- Electronic description of the component pins (for example, logical use and pin swap information)
- Number of functions in the physical package
- How functions are mapped to package slots
- Pin use (how pins are used)—for example, input pins, output pins, and bidirectional pins
- How the logical function pins correspond to the physical pins of the package
- Which pins are tied to power and ground
- Definition properties, such as alternate symbols that can be used instead of the primary package symbol during placement

You need device files only when you are passing the electrical configuration to the layout editor with a third-party netlist, and you are not using Concept for schematic entry. (For details, see the *Transferring Logic Design Data* user guide in your documentation set. The layout editor uses a netlist to determine which electrical components the package symbols represent and how signals are connected. The netlist lists the logic functions (NAND gates, resistors, capacitors, connect pins, and so on) or electrical components of the design and their interconnections.

The netlist contains various sections that require the names of the device files for a design:

- The \$PACKAGES section identifies the components to be added to the design.
For each user-created package symbol in the \$PACKAGES section, provide the name of the device file corresponding to the package symbol and the reference designators for the device type.

 You can also assign reference designators by choosing *Logic – Assign Refdes* ([assign refdes](#) command) and *Tools – Assign Refdes* (for Allegro PCB Performance Option L only), described in the *Allegro PCB and Package Physical Layout Command Reference*.

- The \$FUNCTIONS section identifies the devices with functions.
You must provide the device file names in the \$FUNCTIONS section of the netlist.

The netlist also carries property information for components, functions, or nets. For details about netlists, see the *Transferring Logic Design Data* user guide in your documentation set.

Because the layout editor requires device file information when creating the design database, create device files before choosing *File – Import – Logic* ([netin](#) command) to create the design database. This command is described in the *Allegro PCB and Package Physical Layout Command Reference*.

Create device files using a text editor on your system. You can obtain much of the information for device files from manufacturers' product specification data books.

Checking a Device File

Before you create a design database, use the `dev_check` command to make sure the device files correspond to the correct package symbols. The *Allegro PCB and Package Physical Layout Command Reference* describes the command.

Reviewing the `dev_check.log` File

The names of the devices and their matching package names are generated in the `dev_check.log` file. If you specify only one device file in the `dev_check` command, only the information for that file appears in the log.

The log file identifies which devices have errors by printing

```
*** ERROR: <message>
```

beneath each device file that contains an error, and provides the total number of errors at the end of the `dev_check.log` file.

Error Messages

The following is a list of common errors that you might find in the log file:

- An error occurred while finding the symbol
This can indicate a problem with the symbol. For example, there could be a database error.
- The symbol is not found
Check to see if a `.psm` file was created. Also check the environment path and naming conventions for the symbol and the device file.
- The symbol has an extra pin that the device file does not
The pin number of the extra pin is provided so that you can check the pin numbers.
- The device has an extra pin that the symbol does not
The pin number of the extra pin is provided so that you can check the pin numbers.
- The symbol does not have a reference designator
The symbol is missing a reference designator.

Sample `dev_check.log` File

```
(-----)
( DEVICE FILE CHECKER )
(
( Drawing : dev_check.brd )
( Date/Time : Tue Sep 28 15:54:24 1993 )
(-----)
Checking device Z8581_1, with symbol DIP18_3.
Checking device Z8001_52_1, with symbol PLCC52.
Checking device XTAL_2, with symbol CRYSTAL.
Checking device WR_ENABLE_1, with symbol SOIC20.
Checking device TERMINATOR_33, with symbol SIP8.
Checking device SIPRES_4_7K, with symbol SIP10.
Checking device RESISTOR_4_7K, with symbol SMDRES.
Checking device IOBUF_1, with symbol SOIC20.
Checking device INTERRUPT_1, with symbol SOIC20.
Checking device HM6116_2_1, with symbol SOIC24.

Checking device HM4864_2_1, with symbol SIP30.

Checking device ECON_1, with symbol ECON62_100.

Checking device DIPRES_33, with symbol SOIC16.

Checking device CLEAR_1, with symbol SOIC20.

Checking device CAPACITOR_470PF, with symbol SMDCAP.

Checking device CAPACITOR_0_01UF, with symbol SMDCAP.

Checking device 74LS51_1, with symbol SOIC14.

Checking device 74LS393_1, with symbol SOIC14.

Checking device 74LS373_1, with symbol DIP20_3.

Checking device 74LS373_1, with symbol SOIC20.

Checking device 74LS373_1, with symbol SOIC20.

Checking device 74LS373_1, with symbol SOIC20_PE.

***ERROR: Symbol not found.

Checking device 74LS32_1, with symbol SOIC14.

Checking device 74LS245_1, with symbol SOIC20.

Checking device 74LS157_1, with symbol SOIC16.

Checking device 74LS138_1, with symbol SOIC16.
```

```
Checking device 74LS08_1, with symbol SOIC14.  
Checking device 74LS04_1, with symbol SOIC14.  
Checking device 74LS04_1, with symbol DIP14_3.  
Checking device 74LS04_1, with symbol SOIC14_PE.  
Checking device 74F74_1, with symbol DIP14_3.  
Checking device 74F02_1, with symbol DIP14_3.  
Checking device 74F00_1, with symbol DIP14_3.  
Checking device 2716_1_1, with symbol SOIC24.  
1 Error(s) occurred.  
Tue Sep 28 15:54:19 2005      Page 1  
Allegro NETLIST IN Log File  
=====  
Netlist File Name: '/usr1/QA pcb/devices/dev_check.net' Layout File Name:::  
'/usr1/QA pcb/devices/dev_check.brd'  
=====  
$PACKAGES  
!2716_1_1 ; U1  
!74f00_1 ; U2  
!74f02_1 ; U3  
!74f74_1 ; U4  
!74ls04_1 ; U5  
!74ls08_1 ; U6  
!74ls138_1 ; U7  
!74ls157_1 ; U8  
!74ls245_1 ; U9  
!74ls32_1 ; U10  
!74ls373_1 ; U11  
!74ls393_1 ; U12
```

```
!74ls51_1 ; U13
!Altos ; U14
^
ERROR: Cannot find device file for 'ALTOS'.
-----
!capacitor_0_0luf ; U15
!capacitor_470pf ; U16
!clear_1 ; U17
!devices ; U18
^
ERROR: Cannot find device file for 'DEVICES'.
-----
!dipres_33 ; U19
!econ_1 ; U20
!hm4864_2_1 ; U21
!hm6116_2_1 ; U22
!interrupt_1 ; U23
!iobuf_1 ; U24
!resistor_4_7k ; U25
!sipres_4_7k ; U26
!terminator_33 ; U27
!wr_enable_1 ; U28
!xtal_2 ; U29
!z8001_52_1 ; U30
!z8581_1 ; U31
$END
=====
End of NETLIST IN Syntax/Logic Check
```

```
=====
Total Netlist Warnings = 0.

Total Netlist Errors = 2.

Tue Sep 28 15:54:19 1993      Page 1

Parsing device file: '/usr1/QA pcb/devices/2716_1_1.txt'.

=====
(DEVICE FILE: 2716_1_1 - used for device: '2716-1-1')

PACKAGE SOIC24

CLASS IC

PINCOUNT 24

PINORDER '2716-1-1' '-CS' '-OE' 'A<0>' 'A<10>' 'A<1>' 'A<2>' 'A<3>' 'A<4>'  
'A<::5>' 'A<6>' 'A<7>', 'A<8>' 'A<9>' 'Q<0>' 'Q<1>' 'Q<2>' 'Q<3>' 'Q<4>' 'Q<5>'  
'Q<6>' 'Q<7>' VPP

PINUSE '2716-1-1' IN TRI TRI TRI TRI  
T:::RI TRI TRI IN

FUNCTION G1 '2716-1-1' 18 20 8 19 7 6 5 4 3 2 1 23 22 9 10 11 13 14 15 16 17 21

GROUND GND ; 12

POWER VCC ; 24

PACKAGEPROP MAX_POWER_DISS '.5'

END
```

Creating a Device File

1. Change to the directory where you are placing the device file.
2. Use a text editor, such as vi or Notepad, to write the device file.
3. To include comment information at the beginning of the file, enter a beginning parenthesis followed by the text and a closing parenthesis.

For example:

(device file for 7400)

4. Enter the device file records as described in [Device File Records](#).

For details about multiple functions, see [Specifying Multiple Functions in a Device File](#). If you want to include definition properties for the device, see [Specifying Definition Properties in a Device File](#).

5. it is recommended that you end the file with the following keyword:

END

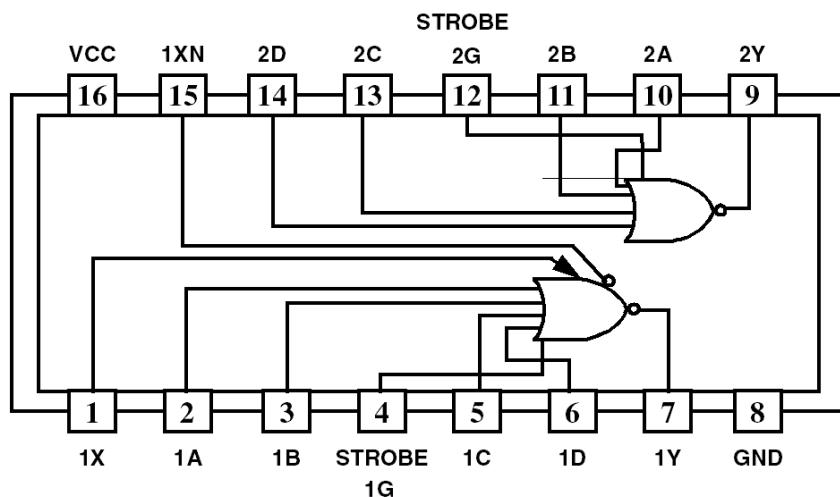
6. Save the file.

Specifying Multiple Functions in a Device File

If a device has different functions, you must define additional PINORDER, PINUSE, PINSWAP, and FUNCTION records in the device file for each different function, so that the layout editor can determine which function slot to assign.

For example, in the following figure, all functions are of the same type, 7400 NAND2 gates. The layout editor assigns function slots.

7423 Component



If you specify only 7423 in the PINORDER and FUNCTION lines of the device file for the package shown, the layout editor cannot determine which 7423 function is referenced—the four-input NOR gate or the expanded NORX gate.

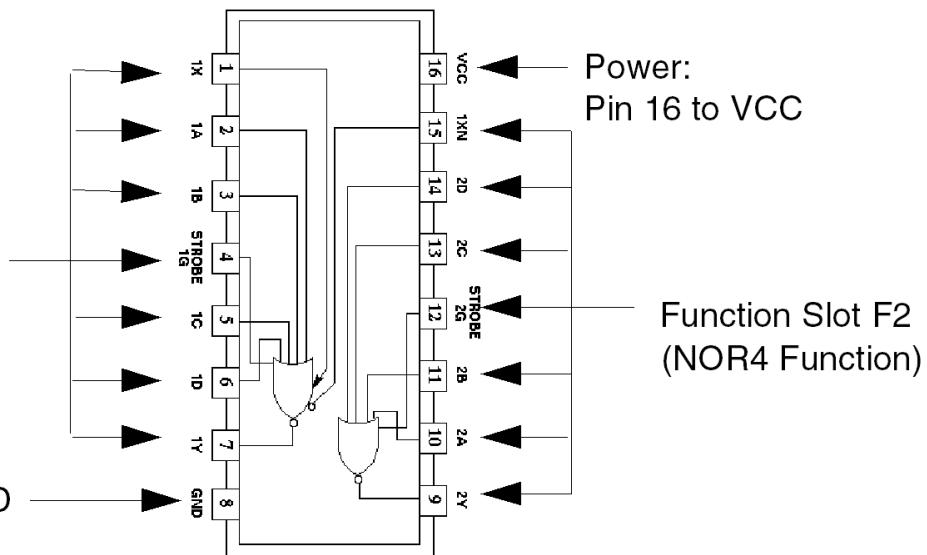
You must define two different PINORDER, PINUSE, PINSWAP, and FUNCTION records, as the following figure shows.

Two Functions in a Device File

```
(Device file for 7423)
PACKAGE DIP16
CLASS IC
PINCOUNT 16
PINORDER NOR4X A B C D G X XN Y
PINUSE NOR4X IN IN IN IN IN IN IN OUT
PINSWAP NOR4X A B C D
FUNCTION F1 NOR4X 2 3 5 6 4 1 15 7

PINORDER NOR4 A B C D G Y
PINUSE NOR4 IN IN IN IN IN OUT
PINSWAP NOR4 A B C D
FUNCTION F2 NOR4 10 11 13 14 12 9

POWER VCC; 16
GROUND GND; 8
END
```



Specifying Definition Properties in a Device File

You can assign properties to a device by creating a PACKAGEPROP property record for each property in the following format:

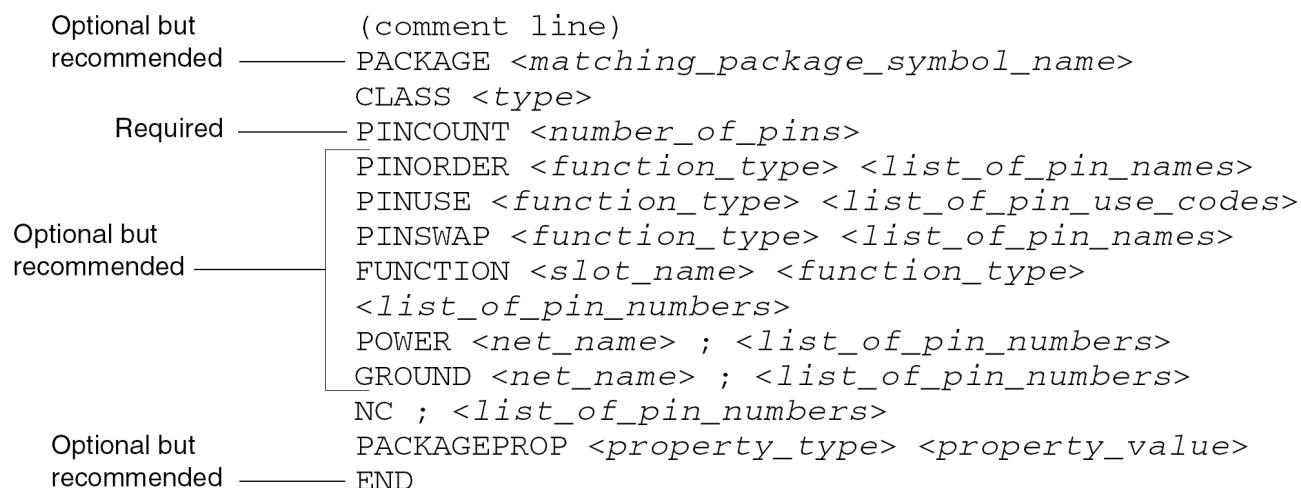
```
PACKAGEPROP <property_name> <property_value>
```

< <i>property_name</i> >	Specifies the property you are assigning to the device. These are the property names, described in the rest of this section:
	<ul style="list-style-type: none">• From the VOLT_TEMP_MODEL property group, you can use MAX_POWER_DISS.
	<ul style="list-style-type: none">• DEVICE_LABEL
	<ul style="list-style-type: none">• TOL
	<ul style="list-style-type: none">• VALUE
	<ul style="list-style-type: none">• PART_NUMBER
	<ul style="list-style-type: none">• INSERTION_CODE
	<ul style="list-style-type: none">• TERMINATOR_PACK
	<ul style="list-style-type: none">• HEIGHT
	<ul style="list-style-type: none">• ALT_SYMBOLS
< <i>property_value</i> >	Specifies the value of the property you are defining. See the property descriptions for information about values.

Device File Records

A device file consists of separate lines (records) that provide the logical data for the device. Each line contains a keyword followed by one or more data fields. A keyword is an attribute that describes the device, and fields are units of information that further define the keyword. The following shows the format of a device file.

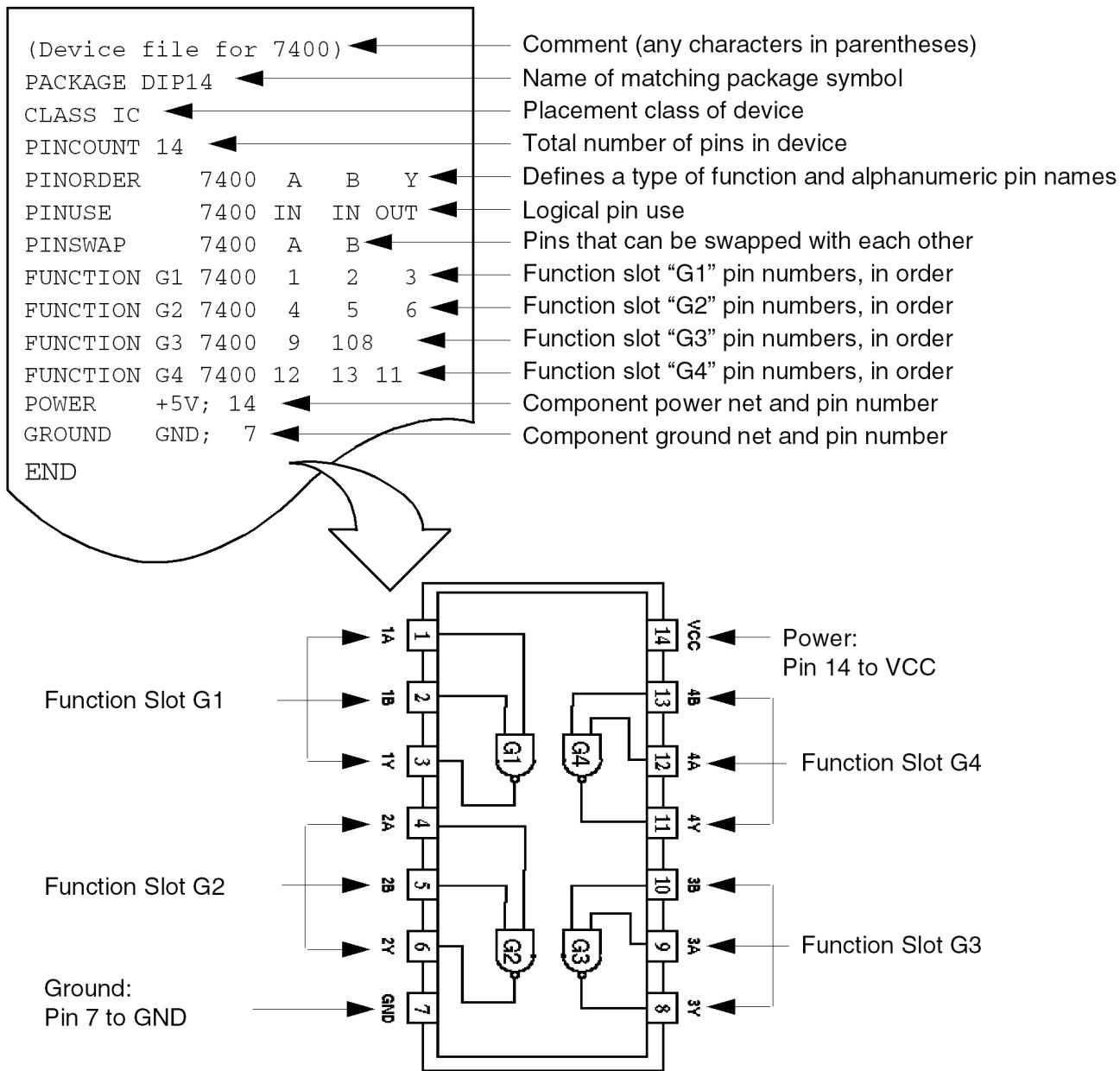
Device File Format



The records not labelled as required or recommended are optional.

The following figure shows how each line in a sample device file is used in the layout editor to construct a physical package. This example shows how a 7400 maps the functions to slots and the pin names to pin numbers.

Device File for 7400 Component



Syntax and Field Descriptions

This section describes each of the device file records and their syntax.

PACKAGE

Optional, but highly recommended. The PACKAGE record describes the physical package for the device. The layout editor uses this information during interactive and automatic placement.

PACKAGE <name>

<i>name</i>	The package symbol name to which the device corresponds. If there is a mismatch, automatic placement fails.
-------------	---

CLASS

Optional, but highly recommended. The CLASS record lets you place the device by class type.

CLASS <type>

<i>type</i>	The type of class: <ul style="list-style-type: none">• IC: Integrated circuit• IO: Input/output—connectors• DISCRETE
-------------	--

If the device file does not contain a pin order section, the layout editor uses the class to determine how many functions are contained in the device. If you specify class *IO*, the layout editor sees each pin as a function. If you specify class *IC*, the layout editor considers all pins on the device, except for power and ground pins, to be in one function.

Test prep uses the class to determine which holes are defined as test point sites. Class is also used during placement.

PINCOUNT

Required. The PINCOUNT record tells the layout editor the total number of pins (connect points) on the physical package. This number is used in interactive and automatic placement.

PINCOUNT <number_of_pins>

<i>number_of_pins</i>	The total number of pins on the device. The pincount is the total number of all pins, including mounting holes. For example, a 20-pin connector with 2 mounting holes has a pincount of 22.
-----------------------	---

PINORDER

Optional. The PINORDER record contains information about a unique function type in a device. You must also supply PINUSE, PINSWAP, and FUNCTION records to fully define the function.

```
PINORDER <function_type> <list_of_pin_names>
```

<i>function_type</i>	The name of the function.
<i>list_of_pin_names</i>	A list of pins associated with the function. Separate each pin name with a space or tab. Pin names can be alphanumeric. Each pin name corresponds to a pin of the function.

If you have a device with multiple functions like 7431, the device file must contain multiple PINORDER statements that identify each function.

PINUSE

Optional. The PINUSE record defines the logical use of each pin within a gate. You must specify a pin use code for each pin listed in the PINORDER record.

```
PINUSE <function_type> <list_of_pin_use_codes>
```

<i>function_type</i>	The name of the gate.
----------------------	-----------------------

<i>list_of_pin_use_codes</i>	A list of the types of pins used in the gate: <ul style="list-style-type: none"> • <code>IN</code>: Input • <code>OUT</code>: Output • <code>BI</code>: Bidirectional • <code>TRI</code>: Tristate—sending, receiving, or held at some state • <code>OCA</code>: Open Emitter • <code>OCL</code>: Open Collector • <code>POWER</code> • <code>GROUND</code> • <code>NC</code>: Not connected internally • <code>UNSPEC</code>
------------------------------	---

Separate each pin type with a space or tab.

Pinuse codes are required when using more advanced features in the toolset. Pinuse codes apply for Allegro PCB SI L, XL, or GXL and other applications (such as scheduling) that merely need information as to whether the pin is an output or input, or to determine a default simulation model. The layout editor uses the logical information during scheduling and terminator assignment. Test prep also uses PINUSE information.

Scheduling considers OUT, OCA, OCL, NC, and TRI as drivers; IN, UNSPEC, POWER, GROUND as loads. The pin type BI can be either a load or a driver, depending on other PINUSE codes on the net. For example, if no other driver exists on a given ECL net, the BI pin is considered the driver. If a driver exists on the net, then the BI pin is located between the driver and the load.

The OCL and OCA pinuse codes can be used for devices that tie directly to a certain leg of the output transistor and therefore require an external resistor. They can also be used for devices tied together to represent Wired logic configurations (Wired-AND or Wired-OR depending on positive or negative logic). The following table describes their function:

PINUSE	PINTYPE	Description	Default SI Model	SI Model Type
OCL	OC	Open Collector	Open Drain	Open PullUp
OCA	OE	Open Emitter	Open Source	Open PullDown

PINSWAP

Optional. The PINSWAP record tells the interactive and automatic swapping routines which pins inside a gate are legal for swapping.

```
PINSWAP <function_type> <list_of_pin_names>
```

function_type	The name of the gate. Must match the gate defined in the pinorder record.
list_of_pin_names	A list of pins inside the gate that can be swapped.

The layout editor uses the PINSWAP record during interactive and automatic swapping to determine which pins inside a gate are available for pin swapping.

If a gate has more than one group of swappable pins, enter a pinswap record for each group of swappable pins to the device file.

FUNCTION

Optional. The FUNCTION record identifies the slots available in the package for the function type described in the PINORDER record.

```
FUNCTION <slot_name> <function_type> <list_of_pin_numbers>
```

slot_name	The name of the gate. Must match the gate defined in the PINORDER record.
function_type	The name of the physical slot.
list_of_pin_numbers	A list of alphanumeric pin numbers in the device. The pin numbers must match the pin numbers on the package symbol that corresponds to this device. Separate each pin number with a space.

One symbol on the schematic corresponds to one function in the device file. If you do not enter a PINORDER record, the layout editor assumes that for each non-power and non-ground pin there is a schematic symbol for CLASS IO devices.

For CLASS IC and DISCRETE devices, the layout editor assumes there is one symbol showing all pins in the device, except for power and ground pins. For example, the previous figure shows a 7400 device file. The corresponding schematic symbol for the 7400 device file would contain four separate gates, each with three pins. However, if the symbol were a 14-pin CLASS IO connector, the corresponding schematic would contain 12 one-pin symbols.

POWER

Optional. The POWER record ties the pins of a device to a particular power signal. The layout editor uses this power information when you use the `netin` command to create a database. The net name given is the default net for the defined power pins. This net is assigned to these pins if there is no net specified for the pins in the netlist.

```
POWER <net_name> ; <list_of_pin_numbers>
```

net_name	The default net for the power pins listed. The net is assigned to these pins if you do not specify a net for pins in the netlist.
list_of_pin_numbers	A list of alphanumeric pin numbers in the device. Separate each pin number with a space or tab.

GROUND

Optional. The GROUND record ties the pins of a device to a particular ground signal. The layout editor uses this ground information when you use the `netin` command to create a database. The net name given is the default net for the defined ground pins. This net is assigned to these pins if there is no net specified for the pins in the netlist.

```
GROUND <net_name> ; <list_of_pin_numbers>
```

net_name	The default net for the ground pins listed. The net is assigned to these pins if you do not specify a net for pins in the netlist.
list_of_pin_numbers	A list of alphanumeric pin numbers in the device. Separate each pin number with a space or tab.

NC

Optional. The NC (no connect) record defines pins on a device that will never be connected.

```
NC ; <list_of_pin_numbers>
```

list_of_pin_numbers	A list of alphanumeric pin numbers in the device. Separate each pin number with a space or tab.
---------------------	---

PACKAGEPROP

Optional. The PACKAGEPROP record(s) identifies definition properties that apply to the device.

END

Optional, but highly recommended. This ends the device file description. If you do not enter an END statement, the layout editor issues a warning during the `netin` process.

Related Topics

- [Specifying Multiple Functions in a Device File](#)
- [Specifying Definition Properties in a Device File](#)

Guidelines for Creating a Device File

You can create a device file by choosing *File – Create Device* (`create device` command) in the Symbol Editor, or you can observe the following guidelines when creating device files:

- Follow DOS naming conventions and use the file extension `.txt`.
- Enter text in upper- or lowercase letters.
- Add any comments as the first line in the file, enclosed in parentheses.

The layout editor ignores comments. Include the file name as a comment.

- Separate fields in a record with spaces, unless the syntax for a record specifies a different separator.
- Enter each record on a separate line.
- Enclose a text string that contains non-alphanumeric characters in single quotes.
- End the device file with an END statement to avoid receiving a warning.

Related Topics

- [create device](#)

Using Technology and Parameter Files

Both technology and parameter files are essential components in the process of leveraging reusable design information during the database creation stage of the design flow. Technology (tech files) are used to enter cross-section, drawing and constraint settings into the database while parameter files are used to enter settings for global and application-based functions.

Working with Tech Files

Technology files, also called tech files, contain the following types of neutral design data:

- Drawing parameters (includes units and design extents)
- Layout cross section
- DRC modes
- Spacing, physical, electrical, and design constraints
- User property definitions

Tech files are eXtensible Markup Language (XML)-based files. The tech file syntax is described in the XML Schema Definition (.xsd) files located at:

`<installation_directory>/share/pcb/consmgr/schemas`

The `dcf.xsd` file is the top-level definition for both the constraints (`.dcfx`) and technology (`.tcfx`) files.

 The schema definitions may change with newer releases.

If you have families of designs that share similar technologies, you should create design-specific tech files pertaining to these families. You can create these tech files by exporting the rules from an existing design or using one of the physical editors to set up the required design rules in an empty design and then export the data. The tech file name should describe the rules contained within the file.

You can have tech files for specified design information. For example, you can have one set of tech files containing only stack-up information; another set, constraints; and a final set, corporate user property definitions.

When using tech files, if you find that a constraint value does not contain an explicit unit, then it is assumed that the value is in the design units specified in the header of the tech file. If the tech file is read into a design with different units, the values are converted to the current Allegro database design units.

Cadence recommends that you place tech files in a central location using the CDS_SITE strategy (`<CDS_SITE>/pcb/tech`). By locating technology files in a centralized directory, you promote the sharing of design rules across similar designs. Refer to the *Getting Started User Guide* in your product documentation for additional information on CDS_SITE.

Tech files are located using the TECHPATH environment variable. You can manage this variable in the User Preferences Editor (`enved` command).

Tech files use the `.tcf` extension. Your layout tool also supports pre-Release 16.0 tech files using the `.tech` extension. It automatically uprevs the pre-Release 16.0 tech files to the Release 16.0 `.tcf` files. Tech files created in the current release are forward-compatible with future releases (an uprev process may be required). Tech files are not backward-compatible. For example, you cannot import a technology file that you created in or upreved to Release 15.0 into a Release 14.0 design.

Accessing Parameter Files

Using any Allegro design database, you can import or export parameter files.

Exporting Parameter Files

You can create a parameter file from an existing design. Use one of the following methods to export the file:

- *File – Export – Parameters* (`param out` command)
- `techfile` batch command

After export, view error messages and other process information in the current or last generated `param_write.log` report.

Importing Parameter Files

When you initially begin a design, import the `.prm` file from a centrally located corporate library, or your local working directory. The environment variable PARAMPATH determines the path of library-based files.

Use one of the following methods to import the file:

- *File – Import – Parameters* (`param in` command)
- The new board wizard (`layout wizard`)
- `techfile` batch command

If a parameter record of the same name already exists in the design database, the `.prm` file overwrites the existing record when you import. When no parameter name exists, a new record is created. After import, view error messages and other process information in the current or last generated `param_read.log` report.

Accessing Tech Files

Using any Allegro design database, you can import, export, or compare tech files.

Exporting Tech Files

You can create a tech file from an existing design. Use one of the following methods to export the file:

- *File – Export* menu command in Constraint Manager
- *File – Export* menu command (`techfile out`) in one of the physical design editors
- `techfile` batch command

Currently, the Constraint Manager export command offers more options than the other methods.

Importing Tech Files

You can import a tech file into an existing design. Use one of the following methods to import the file:

- *File – Import* menu command in Constraint Manager
- *File – Import* menu command (`techfile in`) in one of the physical design editors
- The PCB Editor board wizard (`layout wizard`)
- `techfile` batch command

During import, if an error exists in the tech file, the tool continues reading the file, and writing warning and error messages, but does not create an updated design. After import, check the `techfile.log` for any warnings or errors.

In *Overwrite* mode, importing a technology file overwrites any values that already exist in the design. If a constraint does not exist in the design, it is added.

Currently, the Constraint Manager import command offers more options than the other methods.

Comparing Tech Files to Designs

Comparing a tech file to a design can help determine if the values in a design conform to the intended values residing in the tech file, before you send the design to manufacturing.

The `techfile.log` records the values of the file and the design for side-by-side comparison. Only the constraints specifically contained in the tech file are checked against their counterparts in the design. The `techfile.log` also contains any warnings or errors encountered while reading the tech file.

Use one of the following methods to compare a tech file to a design:

- *Tools – Technology File Compare* menu command in one of the physical design editors (`techfile compare`)
- `techfile` batch command
- *File – Import* menu command in Constraint Manager

Upgrading Tech Files

If you created tech files in previous versions of Allegro, you can uprev them to the latest XML version (.tcf). The tool automatically uprevs the tech file when you import it but you may want (for performance reasons) to update all your tech files to the latest version by using the `techfile` batch command.

DRC analysis modes are not layer-specific in the new format. The tool updates the database as follows:

- If any layer is set to ALWAYS, the tool sets all layers to ALWAYS.
- If any layer is set to BATCH, the tool sets all layers to BATCH.

Uprev restrictions include the following:

- Comments in the old version of the tech file (.tech) do not appear in the new file.
- Pre-Release 16.0 tech files do not formally support segmented data. For example, in Release 15.7 you could manually edit a tech file so that it contained only the user-defined property section. When you uprev this tech file, it will be updated to a full tech file and you will need to manually edit the result to restore it to its original intent.

 Tech files are not backward-compatible. You cannot import a technology file for a newer release into an older release.

Locking Constraint Sets

The tech file supports XML tag called `<objectFlag>` that determines whether all values in a physical, spacing, or electrical constraint set in a design are locked from editing. Importing a tech file is the only way to lock, and later unlock, these items.

- Setting the XML tag value to `fObjectReadOnly` in a tech file marks the constraint set as Read only or locked.
- Setting the XML tag value to `fObjectNOTReadOnly` in a tech file marks the constraint set as Editable or unlocked.

When a constraint set is locked, its values cannot be changed from within the layout editor.

In an Electrical domain you can override a locked constraint value by setting the corresponding property on particular design elements.

⚠ In the Physical and Spacing domains, if any one of the constraint sets is locked; all design element overrides in that domain are ignored, even if the override was present prior to importing the locked constraint set.

A locked constraint set can only be changed by importing a tech file.

⚠ On importing a tech file, the constraint values are updated, irrespective of the `<objectFlag>` setting in the tech file.

Examples of Locked and Unlocked Items in Tech Files

Read only or locked:

```
</attribute>  
  
<objectFlag>fObjectReadOnly</objectFlag>
```

Editable or unlocked:

```
</attribute>  
  
<objectFlag>fObjectNOTReadOnly</objectFlag>
```

⚠ By default, the XML tag value is set to '`fObjectNOTReadOnly`' in an exported tech file.

Technology Constraints File

The `.tcf` file supports all the information in the `.tech` file. No design specific information such as nets, xnets or buses are saved to the tech file.

Section	Description
Drawing	Consists of design units, design extents, and origin. The drawing extents and origin are used only for new boards.
Cross-section	Defines the design stackup.
User-defined constraint definitions	Contains any property dictionary entries.
opFlags	Specifies the DRC analysis mode setting for all constraints.
Constraints	Contains all objects, for example, CSets, Net Classes, and Regions, and their constraints.

Working with Parameter Files

Database parameter files contain customized parameter records, which are those that have a single instance in the database and include the following types of customized settings:

- Design settings (such as global values and grid settings)
- Artwork
- Text size settings
- Application or command parameters (includes auto rename, auto assignment, auto silkscreen, global dynamic fill, autovoid, export logic, drafting, gloss line fattening, gloss dielectric generation, Options window tab settings, test prep, automatic placement, auto swap, thieving, backdrill, interactive flow planner, and Signoise analysis)

Global values are those such as dynamic fill; grid settings; artwork format; and Xhatch style, line width, spacing, and angle, for example.

Parameter File Syntax

Parameter files are eXtensible Markup Language (XML)-based files. The database parameter file syntax is described in the Document Type Definition (DTD) file located at:

<cdsroot>/share pcb/xml-formats/parameter.dtd

Database parameter files use the .prm extension and are located using the PARAMPATH environment variable in the User Preferences Editor, available by choosing *Setup – User Preferences* ([enved](#) command).

Generating Libraries

You can define new libraries based on libraries from existing Allegro layout editor designs. This is useful for:

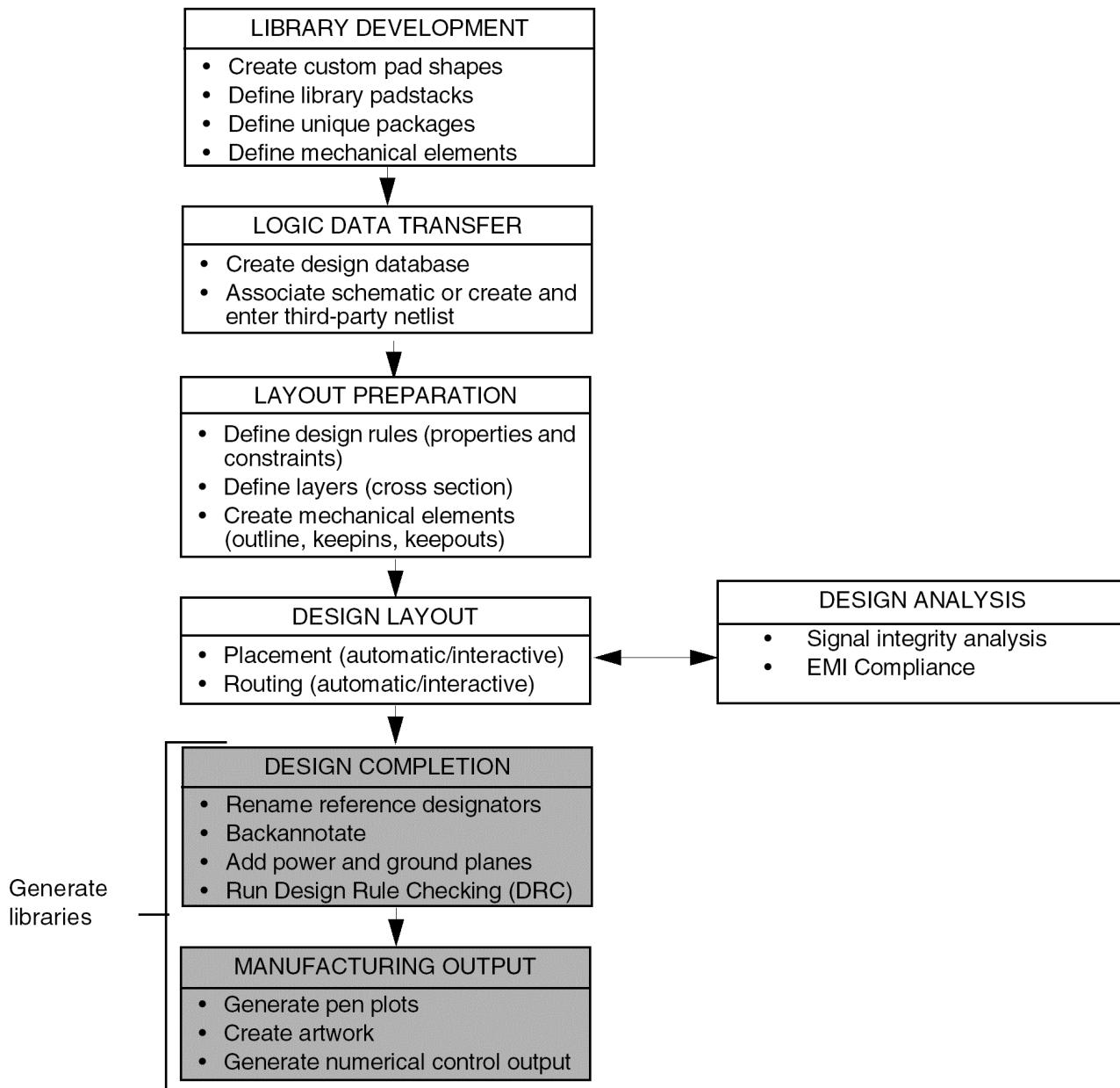
- Creating a library for a design originating from a different CAE or physical layout system, where the original library is not provided
- Creating a library that is compatible with the padstacks and symbols already in a design, when the design is from an earlier library revision
- Recovering libraries that may have been lost by obtaining the library data from a design that contains the correct library data
- Creating a clipboard library that contains elements from designs or symbol drawings

You can create libraries from existing designs by extracting:

- Device files
- Padstacks
- Symbols

Creating new libraries based on existing designs occurs late in a design flow or following its completion, as follows.

Generating New Libraries in a Design Flow



Creating a Clipboard Library

You can create a library (directory) of clipboard files for use in designs and symbol drawings. These are elements you can place in such a library:

- Connect lines
- Connect points
- Connect line segments (including connect arc segments)
- Drafting symbols
- Filled rectangles
- Groups
- Lines
- Line segments (including arc segments)
- Package, mechanical, and format symbols
- Pins
- Rectangles
- Shapes
- Text
- Vias

These are the elements you cannot place in clipboard files:

- Components
- DRC violations
- Figures
- Functions
- Nets
- Ratsnest lines

You can copy and paste these elements between designs or symbol drawings by choosing *File – Import – Sub_Drawing* (`clppaste` command) and *File – Export – Sub_Drawing* (`clpcopy` command),

described in the *Allegro PCB and Package Physical Layout Command Reference*.

Setting the CLIPPATH Environment

The clippath environment variable in the the layout editor global or local environment file identifies the directory in which clipboard elements are stored. The default setting for the clippath environment variable is the current directory from which the layout editor is run, as indicated by a period:

```
set clippath = .
```

 You can check the current setting for the clippath variable by entering `set` at the command line. A list of the defined environment variables appears. See the *Getting Started with Physical Design* user guide in your documentation set for details on the various environment variable settings.

To change the directory in which the clipboard elements are stored, do one of the following:

- Temporarily change the clippath using the `set` command, described in the *Allegro PCB and Package Physical Layout Command Reference*.
- Change the clippath setting in the local `env` environment file. For details, see the *Getting Started with Physical Design* user guide in your documentation set.

Creating Libraries from Existing Designs

To obtain device files, padstack definitions, and symbol definitions from an existing design, choose *File – Export – Libraries* (`dlib` command) (dump libraries).

Creating Device and Symbol Files with Batch Commands

You can obtain device files and symbol definitions from an existing design by running these batch commands:

- The `create_devices` batch command creates device files.
- The `create_sym` batch command creates symbol definitions.

Related Topics

- [dlib](#)
- [create_devices](#)
- [create_sym](#)

APD: Using LEF/DEF Files (APD XL)

APD provides an interface that lets you import and export data using the Design Exchange Format (DEF) import and export commands. DEF—and associated Library Exchange Format (LEF) files—are industry-standard formats developed by Cadence Design Systems for representing digital IC implementation data. The `def in` command lets you import design data for integrated circuits from Cadence IC design tools as well as third-party IC tools that use standard-format DEF files. This feature lets you import data that defines the die, specifically IC size, I/O pads and buffers, die pin, netlist, and constraints. *File – Export – DEF* (`def out` command) lets you read the changes made to that data in APD back to an IC tool.

The LEF/DEF interfaces support:

- DEF import for existing die physical structure, netlist, and constraints
- LEF import for macro footprints with die pin locations and pad size or shape
- DEF export to IC tools for built-in die pin locations
- Export for modifications made to existing dies

 The following sections suggest how you typically use the import (`def in`) and export (`def out`) commands to import or export DEF data.

Planning the Die Rings for an IC Layout Already Underway

You use the `def in` command to import the I/O pad ring and associated buffers, constraints, and electrical simulation data from a Synthesis, Place, and Route (SP&R) tool. APD automatically creates layer information based on data in *Setup – Cross-section* (`xsection`). You make changes to the die pin based on your packaging requirements, then export to the IC tool using the `def out` command.

Prototyping Die Pins for an IC Not Yet Laid Out in an SP&R Tool

Create the die pin layout using both the Die Generator (`die generator` command) and the Die Editor (`die editor` command). Run the LEF Pin Parameters command to select COVER BUMP LEF macro types to use, if desired, to implement the pads for the die pins in the IC tool database. Select the macros from a list of available LEF files. When you complete your die pin planning, use the `def out` command to export the prototype die pin layout to your SP&R tool for use as a template for your IC.

Exporting Changes from APD to an IC Design

You use the `def out` command to export changes that you made to the position of die-pin locations and net assignments. The IC design into which the DEF file is then imported incorporates changes only to those elements which you changed in APD; other elements in the IC design database remain unchanged.

Related Topics

- [def in](#)
- [def out](#)
- [xsection](#)
- [die editor](#)

About Exchange Format Files

To use the *File – Import/Export – DEF* menu commands, you should be familiar with the structure of exchange format files. DEF files work with LEF files. Both file types are written in ASCII-based description language. The `def in` and `def out` commands require the presence of four file types: LEF and DEF files, which are IC-generated, and Library Definition Files and Condensed Macro Library files, which are APD-generated.

- Library Exchange Format (.lef)
LEF files contain mandatory and optional fields that define reusable components that are used in an IC design as well as macros that define I/O pads for die pins.
- Design Exchange Format (.def)

DEF files define the instances of the macros from one or more LEF files as components. They also include netlist data and define placement for the macro instances, as well as optional physical pins that are an alternative way to implement pads for die pins.

- **Library Definition File (.ldf)**

A library definition file defines libraries and the paths to the LEF files defined in them. If you have multiple libraries, you can have an associated .ldf file for each one or a single .ldf for multiple libraries. Library definition files are created using the LEF Library Manager.

- **Condensed Macro Library (.cml)**

The .cml file stores LEF data for those pins of a macro that impact your component design.

One .cml file is automatically created for each LEF file. The LEF Library Manager, discussed later in this chapter, helps you create and maintain these files.

LEF and DEF files must contain the following types of information to best define an IC:

- Physical descriptions of the die outline and die pin pads
- Netlist information
- Component properties
- Constraint data

 LEF and DEF files do not contain component substrate information of the type found in some Cadence component design tools. Before importing a DEF file, you should set the following parameters:

- Approximate drawing size, design units type, and accuracy (*Setup – Design Parameters* (prmed), *Design* tab)
- Layer stackup (*Setup – Cross-section* (xsection)). You can also set up layer information in the New Design Wizard accessed using the new command.

The following section provides information on LEF and DEF files and their components. For additional information about LEF/DEF files, go to <http://www.openeda.org> and download the latest version of the LEF/DEF specifications.

LEF and DEF File Fields

LEF and DEF files are composed of information contained within fields. To successfully import data, the following mandatory fields must be present:

In DEF Files	In LEF Files
VERSION	VERSION
NAMESCASESENSITIVE	NAMESCASESENSITIVE
BUSBITCHARS	MACRO (Macro data identifiers must contain SITE and PIN ----- The macro identifier must contain SITE and PIN -----)
DIVIDERCHAR	
DESIGN	
COMPONENTS	

DEF File Standard Fields

The LEF/DEF interface in APD does not read every standard field contained in a DEF file. In the following list of field names, mandatory fields are called out in bold text while fields that are read, if they are present in the file, are in italics. Ignored fields are in plain text.

DEF Field Name	Definition
VERSION	DEF file version statement
NAMESCASESENSITIVE	Statement indicating if the names given to fields are case sensitive or not. If case sensitive, indicated 'ON' otherwise 'OFF' (default).
BUSBITCHARS	Statement indicating what character is specified to differentiate a different bus line. Can be any pair of characters, the default is '['].
DIVIDERCHAR	Statement indicating hierarchy of names. Default character is '/'. <i>Design</i>
DESIGN	Name of the design
TECHNOLOGY	Specifies a technology name for the design
ARRAY	Gate Ensemble-specific
FLOORPLAN	A filename. Gate Ensemble-specific

UNITS	The divisor to obtain the units. For physical locations, the units are in microns. (Units are similar to accuracy, as understood in your component design tool.)
HISTORY	A one-line historical record of the design
PROPERTYDEFINITIONS	Lists all the properties used in the design
DIEAREA	Optional statement defining the bounding box of the die. List the bounding box as lower left point and upper right point.
ROW	Silicon Ensemble-specific
TRACKS	Silicon Ensemble-specific. Defines the grid for the design.
GCELLGRID	Silicon Ensemble-specific
DEFAULTCAP	Estimated wire capacitance for each net. Silicon Ensemble-specific.
CANPLACE	Gate Ensemble-specific. An array pattern. Overrides LEF.
CANNOTOCCUPY	Gate Ensemble-specific. An array pattern. Overrides LEF.
VIAS	Names and geometries of all vias in a design
REGIONS	A physical area where you can assign a component or group
COMPONENTS	Statement containing a list of all macro instances. First line of the component section indicates the number of components contained within.
PINS	Statement defining external pins. For a complete die, indicates logical pin names and their associated nets. Can optionally supply physical pin locations.
PINPROPERTIES	Property definitions assigned to a specific pin
BLOCKAGES	A routing blockage geometry
SPECIALNETS	Defines netlist connectivity for special nets (nets that use special routing rules). Often used to define power and ground nets.
NETS	Defines netlist connectivity for pins
IOTIMINGS	Pin timing simulation data

SCANCHAINS	
CONSTRAINTS	Net physical and electrical constraints
GROUPS	A group of components and placement info
BEGINEXT	User-defined fields

LEF File Standard Fields

The LEF/DEF interface in APD does not read every standard field contained in a LEF file. In the following list of field names, mandatory fields are called out in bold text while fields that are read, if they are present in the file, are in italics.

LEF Field Name	Definition
VERSION	LEF file version statement
NAMESCASESENSITIVE	Statement indicating if the names given to fields are case sensitive or not. If case sensitive, indicated 'ON' otherwise 'OFF' (default).
NOWIREEXTENSIONATPIN	Wires are not extended at the pins.
BUSBITCHARS	Statement indicating what character is specified to differentiate a different bus line. Can be any pair of characters, the default is '[]'.
DIVIDERCHAR	Statement indicating hierarchy of names. Default character is '/'.
MANUFACTURINGGRID	Value in microns for manufacturing grid. All polygon and line edges must lie on a whole number of manufacturing grids.
USEMINSPACING	Specifies minimum spacing for pins and/or obstructions
CLEARANCEMEASURE	Clearance distance for spacing rule checks
ARRAY	Gate Ensemble-specific

Defining and Developing Libraries

APD: Using LEF/DEF Files (APD XL)--Prototyping Die Pins for an IC Not Yet Laid Out in an SP&R Tool

FLOORPLAN	A filename. Gate Ensemble-specific
UNITS	The divisor to obtain the units. For physical locations, the units are in microns. (Units are similar to accuracy, as understood in your component design tool.)
PROPERTYDEFINITIONS	Lists all properties in the LEF file
ANTENNASIZE	Defines the gate area and pin diffusion data
{LAYER (<i>Nonrouting</i>)}	Defines non-routing layer physical information
LAYER (<i>Routing</i>)}	Defines routing layer physical information
{VIA}	Defines vias for non-default wiring
{VIARULE}	Defines special wiring not found in the VIA statement
VIARULE GENERATE}	Defines special wiring not found in VIARULE
NONDEFAULTRULE	Defines all regular wiring except default wiring
UNIVERSALNOISEMARGIN	Crosstalk definition field
EDGERATETHRESHOLD1	Crosstalk definition field
EDGERATETHRESHOLD2	Crosstalk definition field
EDGERATESCALEFACTOR	Crosstalk definition field
NOisetable	Crosstalk definition field
CORRECTIONTABLE	Crosstalk definition field
SPACING	Spacing rule for components on the same net
MINFEATURE	Gives the minimum feature size for the technology
DIELECTRIC	Dielectric constant for IC insulating material
IRDROP	Table for IR drop of wired OR gates
{SITE}	Placement grid for a family of macros
ARRAY	Gate Ensemble-specific; instantiates a pattern of sites

{MACRO macroname	Macro name
Macro data	Data defining the macro
{PIN}	Statement within a macro specifying all the parameters for a pin
{OBS}	Macro blockage
TIMING	Delay relationship of pins in a macro
END macroname	
BEGINEXT extension	Application-customized syntax

LEF Macro Data Identifiers

The following is the format of one macro. The macros that are collected for creating die pins are of CLASS PAD, ENDCAP, or COVER. All other types of macros are ignored. The origin of a macro is always 0, 0 and specifies the lower left-hand corner of the macro's bounding rectangle. The SIZE statement defines the size of the bounding rectangle. All other coordinates are relative to this point unless optionally translated using the ORIGIN statement.

```

MACRO macroName
*[CLASS{ COVER| RING| BLOCK
| PAD [INPUT |OUTPUT |INOUT |POWER |SPACER]
| CORE [FEEDTHRU | TIEHIGH | TIELOW | SPACER | ANTENNACELL]
| ENDCAP
{ PRE | POST | TOPLEFT | TOPRIGHT
| BOTTOMLEFT | BOTTOMRIGHT}
};]
[SOURCE {USER | GENERATE | BLOCK} ;]
[FOREIGN
foreignCellName [ pt [ orient]] ;]...
[ORIGIN pt ;]
[EEQ macroName ;]
```

```
[LEQ macroName ;]

*SIZE width BY height ;

[SYMMETRY {X | Y | R90}... ;]

## The following statement applies to Silicon Ensemble

## place-and-route.

*[SITE siteName ;]

## The following statement applies to Envia Gate Ensemble

## place-and-route.

{SITE sitePattern}...

[POWER powerConsumption ;]

*{PIN statement}...

[OBS statement]...

[TIMING statement]

[PROPERTY { propName propVal}...;]...

END macroName
```

LEF Macro Pin Identifiers

This section describes the syntax for a macro pin record. A LEF pin record is only described within macro records, as shown in the PIN statement in the Macro field format, above. The LEF/DEF translator requires that the shape field be undefined, meaning it has a default shape of a rectangle. The bounding points for the rectangle are listed.

Field Name	Definition
PinName	The name of the pin
STRUCTURE	Origin is the bottom left corner location of the pin
DIRECTION	Direction of the signal (INPUT, OUTPUT, TRISTATE, INOUT (bi-directional) or FEEDTHROUGH

*PIN pinName

Defining and Developing Libraries

APD: Using LEF/DEF Files (APD XL)--Prototyping Die Pins for an IC Not Yet Laid Out in an SP&R Tool

```
[FOREIGN foreignPinName ;]

[FOREIGN foreignPinName

*STRUCTURE [ pt [ orient]] ;]...

[LEQ pinName ;]

*[DIRECTION {INPUT | OUTPUT [TRISTATE] | INOUT | FEEDTHRU} ;]

[USE { SIGNAL | ANALOG | POWER | GROUND | CLOCK } ;]

*[SHAPE {ABUTMENT | RING | FEEDTHRU} ;]

[MUSTJOIN pinName ;]

[OUTPUTNOISEMARGIN statement ;]

[OUTPUTRESISTANCE statement ;]

[INPUTNOISEMARGIN statement ;]

[POWER powerConsumption ;]

[LEAKAGE current ;]

[CAPACITANCE pinCapacitance ;]

[RESISTANCE pinResistance ;]

[PULLDOWNRES resistance ;]

[TIEOFFR resistance ;]

[VHI voltage ;]

[VLO voltage ;]

[RISEVOLTAGETHRESHOLD voltage ;]

[FALLVOLTAGETHRESHOLD voltage ;]

[RISETHRESH capacitance ;]

[FALLTHRESH capacitance ;]

[RISESATCUR current ;]

[FALLSATCUR current ;]

[CURRENTSOURCE {ACTIVE | RESISTIVE} ;]

[IV_TABLES lowTableName highTableName ;]

{PORT [ portName]
```

```
[CLASS {NONE | CORE};] layerGeometries... END}...  
[PROPERTY { propName propVal}...;]...  
[ANTENNAPARTIALMETALAREA value [LAYER layerName] ;]...  
[ANTENNAPARTIALMETALSIDEAREA value [LAYER layerName] ;]...  
[ANTENNAGATEAREA value [LAYER layerName] ;]...  
[ANTENNADIFFAREA value [LAYER layerName] ;]...  
[ANTENNAMAXAREACAR value ;]  
[ANTENNAMAXSIDEAREACAR value ;]  
[ANTENNAPARTIALCUTAREA value [LAYER layerName];]  
[ANTENNAMAXCUTAREA value ;]  
END pinName
```

LEF File Site Identifiers

Field Name	Definition
SITE	The library site
CLASS	Distinguishes core sites from I/O sites
SIZE	Dimensions of the site in default (North) orientation

```
SITE siteName  
CLASS {PAD | CORE} ;  
SYMMETRY {X | Y | R90}...;  
SIZE width BY height ;  
[ROWMINSPACING value ;]  
[ROWABUTSPACING value [FLIP] ; ]  
END siteName
```

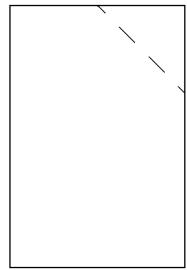
LEF/DEF Standard Component Orientation Syntax

Standard notation is used to represent rotations of objects in IC tools, shown below.

North (default orientation)	=	0 degrees
East	=	90 degrees
South	=	180 degrees
West	=	270 degrees

An 'F' (flipped) indicates that the component is mirrored in the *y* axis after rotation. This syntax is used in both LEF and DEF files. Figure 81515 provides a graphic illustration of this. Mirroring is performed after rotation.

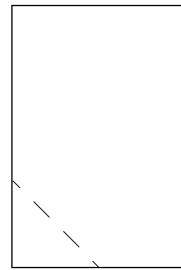
Component Orientation in LEF/DEF Files



N



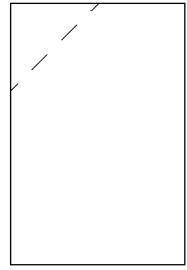
E



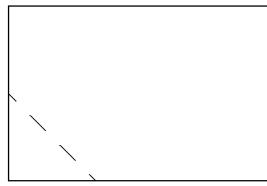
S



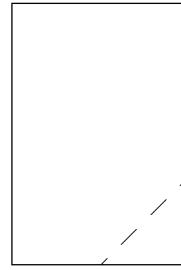
W



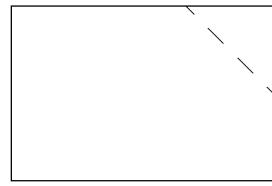
FN



FE



FS



FW

To convert an object that has a North orientation to a different orientation, apply the following

conversion:

Orientation Desired	CCW Rotation	Mirror axis
North	0	None
West	90	None
South	180	None
East	270	None
Flipped North	0	y-axis
Flipped West	90	y-axis
Flipped South	180	y-axis
Flipped East	270	y-axis

About the Condensed Macro Library Format File

The condensed macro library (.cml) file stores LEF macro pin data for each macro of class type PAD, ENDCAP, COVER, or COVER BUMP defined in a LEF file. It is designed to identify which shapes on a top metal layer are actual die pin pads. You create the .cml file by specifying pin information for each LEF file in the LEF Library Manager which automatically gives the .cml file the same name as the current library (for which you are defining pin information). Like their associated library files, .cml file names must be unique. Because APD generates them in ASCII format, you can later edit .cml files in spreadsheets or text editors.

Example of a Condensed Macro Library File

```
#File Created: Mon June 02 14:31:45 2004
VERSION 5.6
NAMECASESENSITIVE ON
MACRO    macroName1
EXTENTS  (700.00, 1000.00)
PIN M7  IO    PAD    OUT     ORIGIN(40.50,43.00)      DIM(65.00,70.00)
PIN M6  m6pin CPT    IN      ORIGIN(40.00,43.00)      DIM(65.00,70.00)      OFFSET(65.00,
70.00)    PADDIM(40.00,40.00)
END      macroName1
SITE     BUFFSZE
EXTENTS  (80,200)
END      BUFFSZE
END
# Options Section
OPTIONS
# LAYERS option
    LAYERS_OPTIONS
        LEF_LAYER    METAL7    1
        LEF_LAYER    METAL6    1
        LEF_LAYER    METAL5    1
        LEF_LAYER    METAL4    0
        LEF_LAYER    METAL3    0
        LEF_LAYER    METAL2    0
        LEF_LAYER    METAL1    0
    END      LAYERS_OPTIONS
# PAD options
    PAD_OPTIONS
        PAD_SIZE
        PAD_WIDTH_MIN    40.00
```

```
PAD_HEIGHT_MIN      40.00
PAD_WIDTH_MAX       100.00
PAD_HEIGHT_MAX      100.00
END      PAD_SIZE
PAD_ALLOWED_PAD_NAMES
PAD_NAME          PAD
PAD_NAME          IO
PAD_NAME          A
PAD_NAME          Z
END      PAD_ALLOWED_PIN_NAMES
PAD_ALLOWED_PIN_USE
PAD_USE           SIGNAL
PAD_USE           POWER
PAD_USE           GROUND
END      PAD_ALLOWED_PIN_USE
END      PAD_OPTIONS

#CPT OPTIONS
CPT_OPTIONS
  CPT_ALLOWED_CPT_NAMES
    CPT_NAME   m5vddb
    CPT_NAME   m5vddb1
    CPT_NAME   m5vddb2
    CPT_NAME   m5vddb3
  END CPT_ALLOWED_CPT_NAMES
  CPT_ALLOWED_USE
    CPT_USE   SIGNAL
  END CPT_ALLOWED_USE
END      CPT_OPTIONS

# AUTO_PAD options
AUTO_PAD_OPTIONS
  AUTO_PAD_NAME      m5vddb3
  AUTO_PAD_WIDTH    46.00
  AUTO_PAD_HEIGHT   46.00
  AUTO_PAD_OFFSET_X 20.00
  AUTO_PAD_OFFSET_Y 0.00
END      m5vddb3
  AUTO_PAD_NAME      DEFAULT
  AUTO_PAD_WIDTH    40.00
  AUTO_PAD_HEIGHT   40.00
  AUTO_PAD_OFFSET_X 0.00
  AUTO_PAD_OFFSET_Y 0.00
```

```

END      DEFAULT
END      AUTO_PAD_OPTIONS
# MACROS  option
MACRO_OPTIONS
MACRO_NAME_SPECIAL    macroName1
MACRO_PAD_NAME_SPECIAL      PIN      # used if the pin does not meet the size requirements
MACRO_PAD_NAME_SPECIAL      IO
END  macroName1
MACRO_NAME_SPECIAL        macroName2
MACRO_PAD_NAME_SPECIAL      PIN
MACRO_PAD_NAME_SPECIAL      IO
MACRO_CPT_NAME_SPECIAL     cpt1
END  macroName2
END      MACRO_OPTIONS

```

CML Header Identifier

VERSION	The version number of the .cml file format.
NAMESCASESENSITIVE	Indicates the names are case sensitive. Follows LEF syntax. ON OFF are valid values.
MACRO	Identifier for the macro name.
SITE	Indicates the information is for a site. The site information is automatically added when the LEF file is processed.
EXTENTS	Indicates the macro or site width and height as defined in the LEF file.
PIN	The identifier to indicate this line defines a pin. A macro pin may have the following fields:
LEF layer name	– The layer name for the pin.
Pin name	– The name of the macro pin as defined in the LEF file.
Type	– PAD.
Use	– The pin use as defined in the LEF file either IN,OUT,BI, or UNSPEC.

Origin	– The origin of the die pin.
Dimensions	– The die pin width and height.
Auto pad offset (optional)	– Indicates the offset for an auto created pad.
Auto pad dimensions (optional)	– Defines an auto created pad width and height.
END	Indicates the end of macro, site, technology or option information.
OPTIONS	Indicates the start of the options information section of the .cml file.
LAYERS _OPTIONS	Indicates the start of the layers options section.
LEF_LAYER	Indicates the start of the layer record and consists of: LEF layer name, layer usage (0 - not used for processing, ignore, 1 - processed as a die pad layer, 2 - processed as a routing layer).
PAD_OPTIONS	Indicates the beginning of the die pin options section.
PAD_OPTIONS_SIZE	Indicates the beginning of the PAD size options section.
PAD_WIDTH_MIN	Indicates the minimum pin width.
PAD_HEIGHT_MIN	Indicates the minimum PAD height.
PAD_ALLOWED_PAD_NAMES	Indicates the beginning of the allowed PADs section.
PAD_NAME	Indicates the PAD name.
PAD_ALLOWED_PIN_USE	Indicates the beginning of the allowed pin use section.
PAD_USE	Indicates the pin use.
CPT_OPTIONS	Indicates the beginning of the RDL connection point pin options section.
CPT_ALLOWED_CPT_NAMES	Indicates the beginning of the allowed connection points section.

CPT_NAME	Indicates the CPT name.
CPT_ALLOWED_USE	Indicates the beginning of the allowed pin use section.
CPT_USE	Indicates the CPT pin use.
AUTO_PAD_OPTIONS	Indicates the beginning of the pad options section.
AUTO_PAD_CPT_NAME	Indicates the CPT name that will have special size of the pad, otherwise it will have default size.
AUTO_PAD_CPT_WIDTH	Indicates the CPT pad width.
AUTO_PAD_CPT_HEIGHT	Indicates the CPT pad height.
AUTO_PAD_CPT_OFFSET_X	Indicates the CPT pad offset X.
AUTO_PAD_CPT_OFFSET_Y	Indicates the CPT pad offset Y.
MACRO_OPTIONS	Indicates the start of the macro options section.
MACRO_NAME_SPECIAL	Indicates the macro name.
MACRO_PAD_NAME_SPECIAL	Indicates the pin name.
END OPTIONS	Indicates the end of the options information section of the .cml file.

About the Library Definition File

A library definition file is an ASCII text file that defines libraries and the paths to the LEF files defined in them. (Your Cadence tool provides a LEF Library Manager that lets you build and edit library definition files dynamically.) If you have multiple libraries, you can have an associated `.ldf` file for each one or a single `.ldf` for multiple libraries. An automatically generated library definition file is created in your current working directory with the default file name, `default.ldf`.

Library definition files have the following syntax conventions:

- One statement per line.
- Blank lines are allowed.
- Keywords are the first non-whitespace character string on a line.
- Keywords and library names are not case-sensitive.
- No wildcards such as asterisks and question marks are allowed in the directory or file names.
- Library names in a library file must be unique. This includes libraries that are defined through keywords `INCLUDE` and `SOFTINCLUDE`.
- Files may be given with either an absolute or a relative path.

The format of a library definition file is as follows:

Example of a Library Definition File

```
DEFINE LibA  
  
LEFFILE C:/lib/Lef/pads.lef  
  
LEFFILE U:/lib/Lef/iomacros.lef  
  
INCLUDE U:/libs/chipio/user1.ldf  
  
SOFTINCLUDE U:/libs/chipio/toplevel.ldf  
  
UNDEFINE LibB
```

Keyword	Example	Description
DEFINE	libraryname1	Defines the name of the library.

LEFFILE	/home/filename.lef	Paths of one or more LEF files to include in the library. The statement must be immediately preceded by either the DEFINE statement or a previous LEFFILE statement.
INCLUDE	/home/lib/filename.ldf	Reads the specified library file and extracts the libraries defined within it.
SOFTINCLUDE	/home/lib/filename.ldf	Same as an INCLUDE statement except no error message is included if the file does not exist.
UNDEFINE	Libraryname1	Library is ignored.
#	#March 15/01	Symbols to define a comment statement.

UNDEFINE statements undefine only libraries that have been previously defined, as is the case in the example. The UNDEFINE statement is ignored if you insert it before the DEFINE statement.

Condensed Macro Library Creation Messages

Various types of messages may be displayed when you are creating condensed macro library files. The following are messages you may see, and a brief explanation of their cause.

Information

<filename> creation successful.

The .cml file was successfully created.

Reading <filename>.

Displayed for each LEF file processed for the .cml file.

<filename> creation aborted.

Displayed in the currently open dialog box when .cml file creation is aborted.

Updating <filename>.

There is a previous .cml file and new entries are being added to the file.

Error

Error: LEF file <filename> not found. File creation aborted.

If a LEF file cannot be found the .cml file creation process is aborted.

Log File

Any LEF warning messages are captured and placed in a log file named `lef_lib.log`.

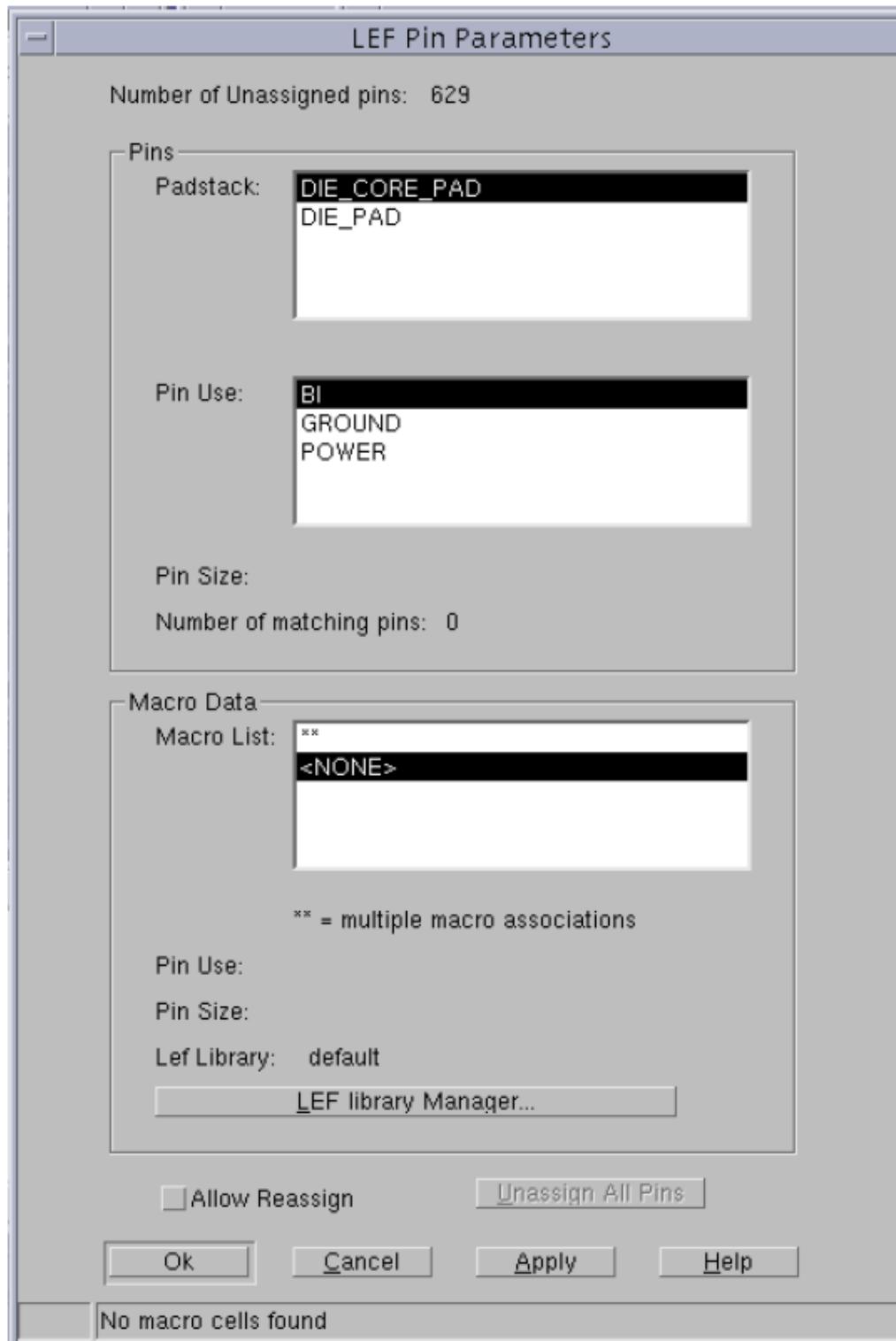
Exporting Data with the LEF/DEF Interface

The `def out` command exports data for a single design back to the IC tool. Elements of the die that you changed in any way in APD results in modifications in the IC tool; however, design elements that were not changed are ignored.

To successfully export a design back to the IC tool, you must ensure that all die pins created in APD are defined in a LEF file.

If you generate die pins in APD (as opposed to having imported them through `def in`), and need to send that data back to an IC tool as COVER BUMP macro instances, you must run *Edit – LEF Pin Parameters* (`lef pin param` command) before exporting your DEF file. Otherwise, APD exports them as physical pins.

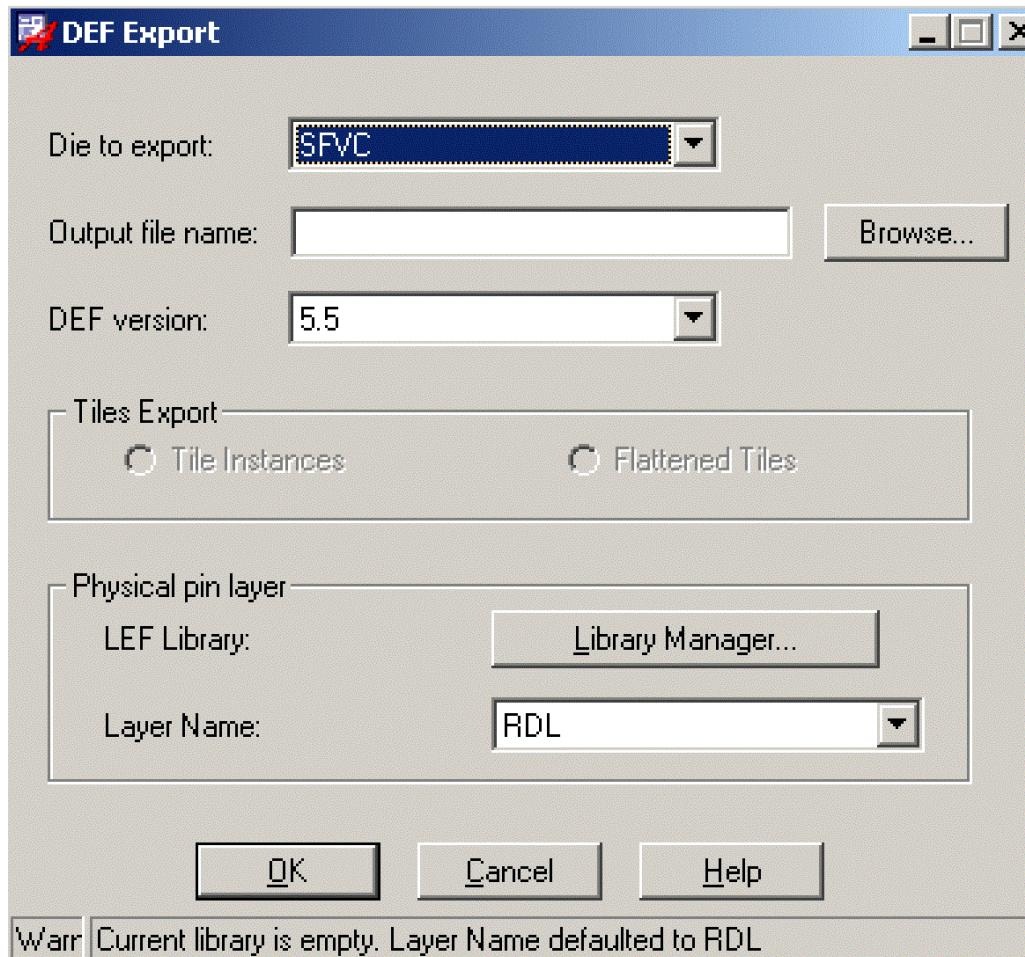
LEF Pin Parameter Dialog Box



This command ensures that die pins you created in APD are correctly associated with LEF macro cells of class COVER BUMP, as required by Cadence's IC design tool, First Encounter (as of version 3.3, First Encounter supports both COVER BUMP cells as well as DEF physical pins for die pins).

If your database contains unassigned die pins, and you attempt to export a DEF file, the `def out` command prompts you to run *Edit – LEF Pin Parameters* menu command. If you do not assign COVER BUMP cells to the pins, APD exports them as physical pins.

DEF Export Dialog Box



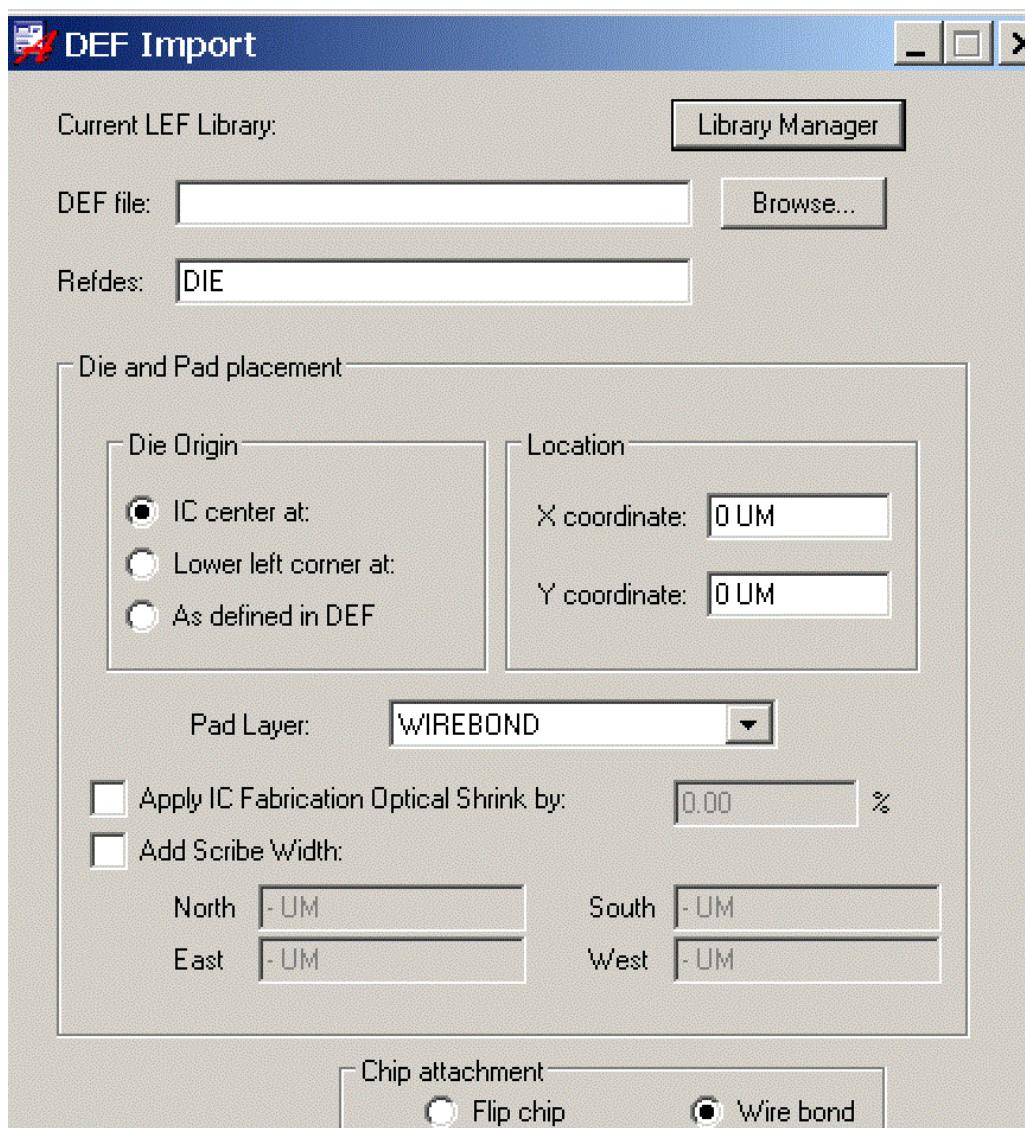
Importing Data with the LEF/DEF Interface

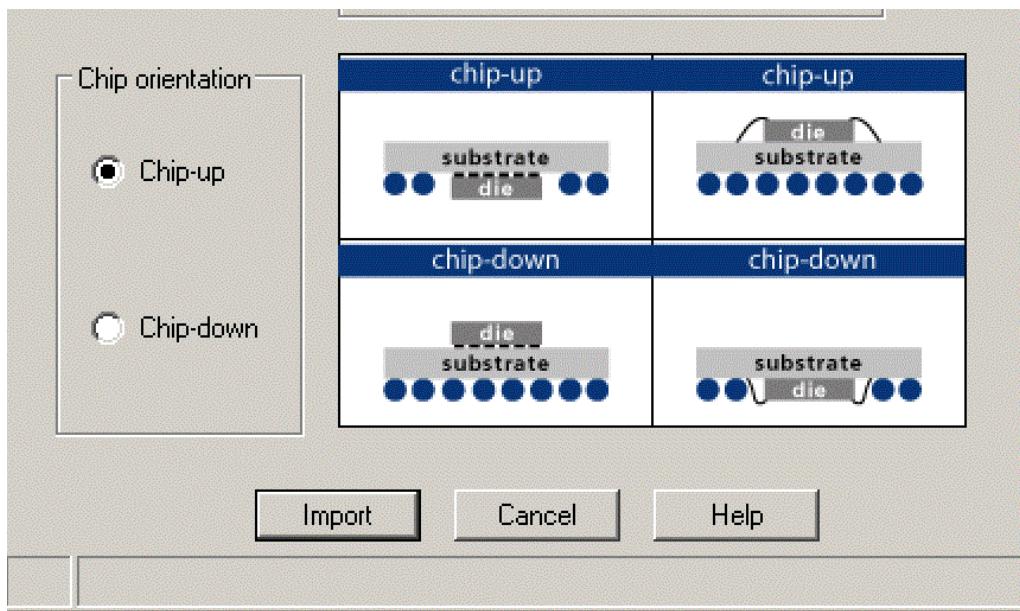
Procedures that detail the methodology for bringing design data into APD through a DEF file are described under *File – Import – DEF* (`def in` command). The procedures cover the methodologies for:

- Importing DEF files with an existing library definition.
- Creating a new library definition and condensed macro library file.

Once created, you do not have to create a new library definition file for every library; rather, you can create a new library in an existing .ldf file.

IC Import from DEF Dialog Box





Related Topics

- [Using the def in Command with an Existing Library Definition File](#)
- [Using def in with a New Library Definition File](#)

LEF/DEF Message Generation

The `def in` and `def out` commands may generate various types of messages when you attempt to import or export design data from a DEF file. The following are messages you may see during DEF data transfers, and a brief explanation of their cause.

Import Errors

Information

The following are information messages you may see during DEF import, and a brief explanation of their cause.

No logical pin data found. Using a sequential numbering scheme.

Displayed if the *PINS* field is not present in the DEF file. The *PINS* field specifies I/O names for use as pin names by other IC tools. Since this field is optional, a sequential numbering scheme for the pins is created in its absence. Macro instance names are not used, as they can be too long and may contain characters that are invalid for pin numbers.

DEF file Import successful.

Appears after a successful DEF file import.

Error

Error messages can occur for a variety of reasons; for example, if you indicate a die placement origin that puts the die partially outside the design extents. Messages stating there are no matching pins can indicate that you:

- Selected the wrong LEF files so there are no matching macros in the DEF file.
- Neglected to define I/O pads in the .cml file.
- Excluded a required macro.

The following are error messages you may see during DEF import, and a brief explanation of their cause.

Couldn't open input LEF file 'filename'.

A LEF file specified in a library cannot be opened for some reason (no such file, no read permissions, and so on).

Couldn't open input DEF file 'filename'.

The DEF file specified for import cannot be opened for some reason (no file, no read permissions, and so on).

Error: Select a LEF Library with at least one LEF file to import a DEF file.

There is no LEF library selected or the library selected does not specify any LEF files.

Error: No components defined as I/O found in 'filename'. Import aborted.

During DEF import, if no LEF macro definitions have been found to match the DEF component definitions, nothing is created.

Error: No macros defined as I/O found in 'filename'. Import aborted.

If no LEF macros are found to be class PAD, ENDCAP, or COVER in the specified library, nothing can be created and the import is aborted.

Error: No I/O pins found. Not created.

After importing the DEF file, if the macros for the design do not contain any I/O pins, then the design cannot be created.

Log File

Log files record any LEF/DEF import errors to the `lef_lib.log`, `defImport.log`, and `lefCmlOptions.log` files. If there are error messages logged, a Viewlog window appears

The log file writes the design name, the library name, and the selected LEF and DEF files, as well as the date and time of the transaction. At the end of the file, a *Log complete* message indicates that a complete log file was recorded. The following messages are generated in the log file:

```
Duplicate Macro name found. 'MacroName' defined in 'filename1' and 'filename2'. The macro defined in 'filename1' was used.
```

This warning is written to the log file if duplicate macro names are found in a library. If found, the macro name and the LEF files they were found in are recorded. The macro that resides in the file listed first is used.

Before reading a DEF file, a message is recorded indicating the path and filename of the file that is about to be read. This allows you to associate the file with the error message.

All the DEF parser errors are trapped and written to the log file. The error messages specify on what line of the file the error occurred and the last keyword identifier (token) that was read. An example of the format of the parser errors is:

```
***ERROR*** <parse error> at line 14  
Last token was <BEGIN>  
  
***ERROR*** <parse error> at line 18  
Last token was <V-PIN>  
  
***ERROR*** <parse error> at line 402  
Last token was <BEGINEXT>
```

Export Errors

The following are information messages that you may see during DEF export, and a brief explanation of their cause.

Information

DEF file export successful

This message appears when the data is successfully written to a DEF file.

Error

No die component found in the design. There is nothing to export.

This error appears if there are no die components in the design.

Couldn't create DEF file 'filename' for writing.

This error appears if the permissions for the specified file location is read-only.

If you attempt to export a DEF design to a nonexistent path or to a read-only directory, a standard warning appears.

Log File

DEF export errors are logged in the `def_export.log`. In the case of a fatal error during export, a Viewlog window alerts you to the condition. The log file writes the design name and the DEF file, as well as the date and time of the transaction. At the end of the file, a `DEF Writer finished exporting successfully` message indicates that a complete log file was recorded.

LEF Library Manager Messages

The LEF Library Manager may generate various types of messages when you are creating or editing libraries. The following are messages you may see and a brief explanation of their cause.

Information

There are no libraries specified in 'filename.ldf'

There are no libraries defined in the library file. This is not an error if this is a new library file that you are using the library manager to edit.

Warning

Warning: Include statements create a loop. File 'path/xxx.ldf' is circularly referenced.

A set of include statements creates a loop reference. For example, `first.ldf` has an include statement referencing `second.ldf` and `second.ldf` has an include statement referencing `first.ldf`. This creates an endless loop. The warning indicates that you must edit your library file. The last file that creates the loop is the filename and path that displayed in the warning.

Error

Error: Library file 'file.ldf' is read-only. Cannot modify library 'lib name' as it is defined in this file.

You tried to add or remove a LEF file of a library that is in a read-only file. This error is most likely to occur if you try to modify a library that has been indirectly specified through an include statement.

Error: Please select a unique LEF Library Name.

If a new library is created, it must have a unique name. If the name is not unique this error is displayed and you must select a different library name before it can be added.

Syntax Error: LEF 'filepath' is not associated with a library.

An `.ldf` file has a keyword assignment `LEFFILE` that is not preceded by the library `DEFINE` statement. The reading of the `.ldf` file is aborted.

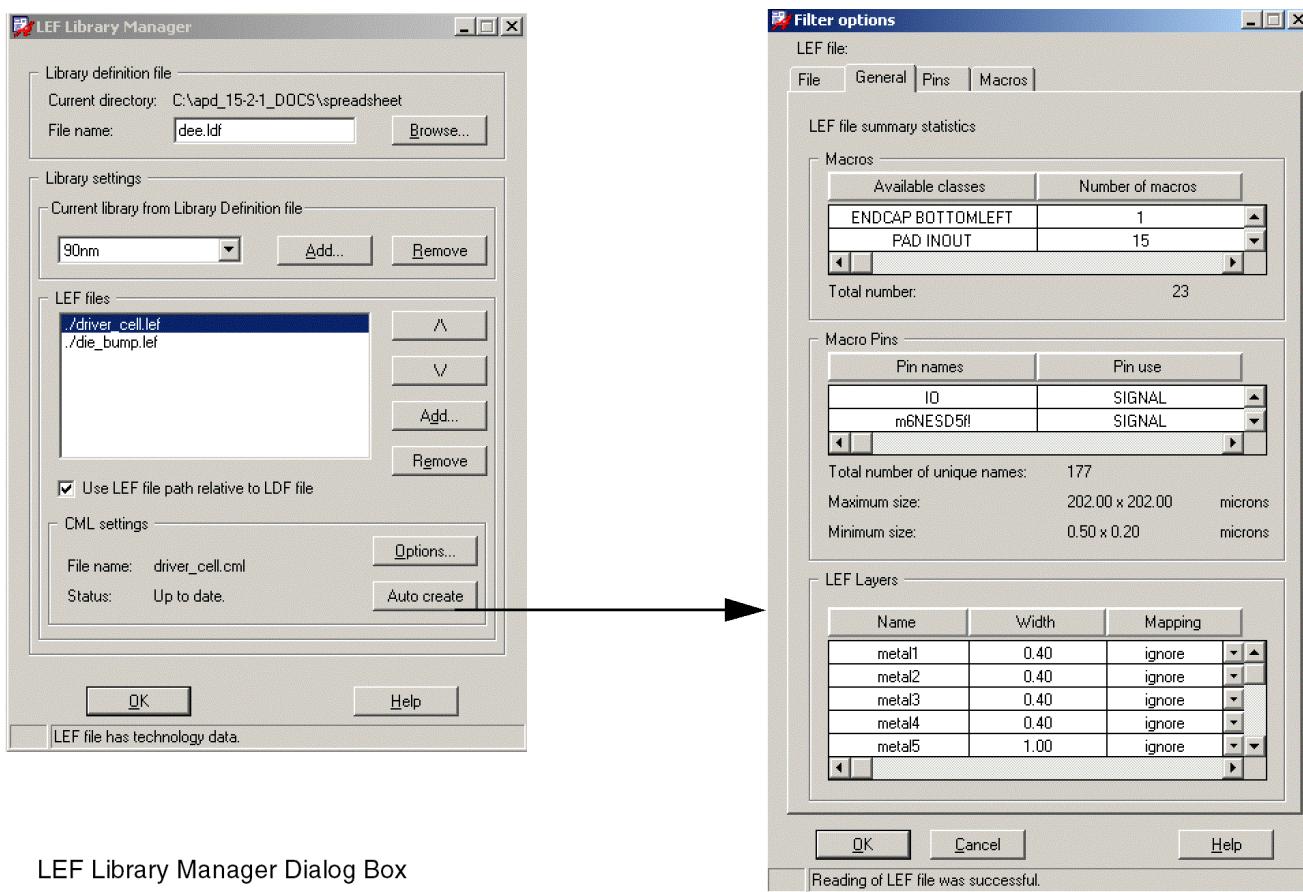
Error: Could not open library 'libraryname.ldf'. File not found.

You specified a filepath or filename that does not exist.

Using the LEF Library Manager

You can use the LEF Library Manager to build and edit library definition files as well as create and configure condensed macro library files by clicking the *Library Manager* button in the Import from DEF dialog box ([def in](#) command) or when you run *Setup – LEF Libraries* ([lef lib](#) command). From this dialog box, you can edit the I/O macros in your library's LEF files as well as configure all the elements within your .cml files. For details on using these dialog boxes to create and configure LEF and .cml files, see the [def in](#) command or [lef lib](#) command topics in the *Allegro PCB and Package Physical Layout Command Reference*.

LEF Library Manager Dialog Box Suite



LEF Library Manager Dialog Box

LEF Filter options Dialog Box
(as configured when run from the
APD window)

APD: Using the Package Designer Symbol Editors

Cadence provides editors that enable you to conveniently edit symbols and components from inside your packaging environment, eliminating the need to export design elements to other tools for editing. The Package Designer Symbol Editing Suite comprises:

- BGA Editor
- Die Editor (available only in APD XL)
- Tile Editor (available only in APD XL)

Although these editors share a common look and some common features, each is used for the design elements indicated by their names. For types of symbols that do not fall under the scope of these editors (such as mechanical, discretes, and so on), it is recommended that you use APD in symbol editor mode.

Feature Set

The following table describes the general capabilities of the individual editors.

Capabilities	BGA Editor	Die Editor	Tile Editor
Adds, deletes, copies, moves, swaps, and modifies pins.	•	•	•
Adds, deletes, copies, moves, swaps, and disbands tiles.		•	•
Places and unplaces tiles.		•	
Supports multiple hierarchical grid structures (add, delete, copy, modify) for laying out a floorplan of grids to control pin placement within the symbol. The highest priority grid is drawn first and overrides lower priority grids lying beneath it.	•	•	•

Supports grids where the pin pitch is turned off; applicable to hard macro blocks from DEF files or to designs where total freedom is required.	•	•	•
Allows mirroring and rotation of copy and move selections of items within a group relative to a selected origin.	•	•	•
Allows grid setting changes (numbering pattern, pitch, offset, and so on) during the active editing session.	•	•	•
Supports pin coloring based on current pin use.	•	•	•
Displays a heads-up window that provides information on elements over which you place the cursor. Optionally, it simultaneously highlights the element that is the object of the display.	•	•	•
Supports many pin numbering schemes, including sequential and customized.	•	•	•
Adds a pin number prefix to all pin numbers within the same grid.	•	•	•
Supports offset pin numbering, allowing the same scheme to be used in multiple grids without causing number duplication.	•	•	•
Allows pin numbering schemes that ignore unused grid points.	•	•	•
Supports rotated pins, both manually or automatically.	•	•	•
Supports border pin text label creation for single-grid symbols.	•	•	•
Allows automatic generated pin number text on every pin in a symbol, offset by a user-defined vector.	•	•	•
Supports placement of hard macro blocks into the target symbol.		•	•
Provides warnings and back-out options when a grid operation might result in pins being shifted, deleted, or renumbered.	•	•	•
Permits existing grids to other locations in a symbol.	•	•	•
Allows restriction of new pins and tiles within a grid.	•	•	•
Allows the disbanding of tiles during an active editing session.		•	•

Allows creation of new symbols and components during an active editing session.

•

•

BGA Editing Flows

This section provides a sampling of how you might use the BGA Editor in some common circumstances. These scenarios give only a glimpse of the many use models that you can create pairing good design practices and the flexibility of the BGA Editor.

 These flows are provided as examples only. The procedures associated with them provide instructions on running the editor for these scenarios only.

These flows apply to an existing BGA symbol under one or more of the following conditions:

- The BGA was created with the BGA Generator.
- The BGA was previously edited with the BGA Editor.

This ensures that a grid structure is in place at the time of the editing session. If it is not in place, create one.

Goal	Open the file containing the BGA that you want to edit.
User Action	Run <i>Edit – BGA</i> (<code>bga editor</code> command)

Deleting Balls From the Grid Array

Goal	Remove some balls from the corner areas to improve routability.
User Action 1	Click <i>Next</i> in the Component Selection dialog box.
Result	The editor advances to the component editing phase.
User Action 2	Click <i>Delete</i> in the <i>Action</i> frame of the Component Editing dialog box, <i>Pins</i> tab.
User Action 3	Either pick individual pins, go into the Temp Group mode, or window around the balls you want to delete.
Result	The selected pins disappear from the Design Window.
User Action 4	To complete the goal of deleting pins, click <i>Next</i> in the dialog box.
Result	The Final Verification dialog box appears, presenting you with further options as described in the dialog.

Changing the Padstack of the Core Balls

This flow assumes that you have an existing BGA with a defined grid open in the Design Window, and that the BGA Editor is open in pin-editing mode.

Goal	Change the padstack assignment of the core balls in the BGA.
User Action 1	Click <i>Next</i> in the Component Selection dialog box.
Result	The editor advances to the component editing phase.
User Action 2	Click <i>Modify</i> in the <i>Action</i> frame of the Component Editing dialog box.
Result	The <i>Attributes</i> , <i>Pin Use</i> , and <i>Netframes</i> of the dialog box become active.

User Action 3	Window around the core balls to select them for editing.
Result	The pins are highlighted and the <i>Attributes</i> , <i>Pin Use</i> , and <i>Net</i> frames of the dialog box are updated with information based on your selection. Double asterisks ** in the list fields of the dialog box indicate multiple object types in your selection group.
User Action 4	In the <i>Attributes</i> frame of the Component Editing dialog box, change the padstack, and click <i>Apply Changes</i> .
Result	The display of core balls changes to reflect your selection.
User Action 5	To complete the goal of changing padstacks, click <i>Next</i> in the dialog box.
Result	The Final Verification dialog box appears, presenting you with further options as described in the dialog box.

Assigning a New Net to a BGA Ball

This flow assumes that you have an existing BGA with a defined grid open in the Design Window, and that the BGA Editor is open in pin-editing mode.

Goal	Change the net assignment of a ball in the BGA.
User Action 1	Click <i>Next</i> in the Component Selection dialog box.
Result	The editor advances to the component editing phase.
User Action 2	Click <i>Modify</i> in the <i>Action</i> frame of the Component Editing dialog box.
Result	The <i>Attributes</i> , <i>Pin Use</i> , and <i>Net</i> frames of the dialog box become active.

User Action 3	Click the <i>Item Info</i> button in the dialog box.
Result	The Item Information "heads-up" display appears.
User Action 4	Make sure that the item type is set to <i>Pins</i> , then place your cursor over a pin.
Result	The pin number, net, padstack, and location of the pin are displayed in the window.
User Action 5	Click on the pin to select it for editing. Then, in the <i>Net</i> frame of the Component Editing dialog box, make your change and click <i>Apply Changes</i> .
Result	The net assignment of the ball changes to reflect your selection.
User Action 6	Confirm that the change occurred by placing the cursor over the ball and reading the new net name in the heads-up display.

Related Topics

- [bga editor](#)
- [Temp Group](#)

Multiple Grids and Pin Number Patterning

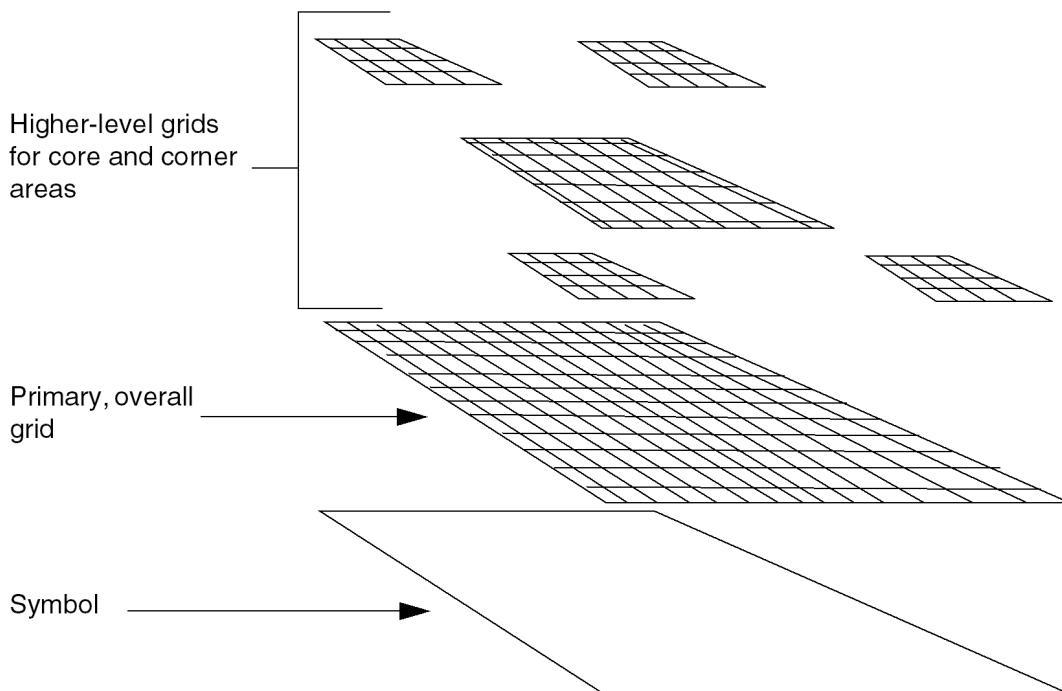
The new symbol editors enable you to specify multiple grids and grid-specific numbering schemes in the symbols that you edit. You can incorporate various pin patterns for the core and number of rings that make up your symbol.

How Multiple Grids Work

Multiple grid floorplans follow a hierarchical order with each grid assigned a priority. The foundation is a single grid that overlays the entire symbol. This grid has the lowest priority; each new grid that you add is assigned the next highest priority and takes precedence over the lower grids. This means that you cannot locate a design object to a point on one grid that is overlaid by a higher level grid.

In the following figure, the grid patterns comprise a base (or primary) grid that overlays the entire symbol, a core grid, and a set of corner grids. You would have defined the dimensions of the base grid as part of a standard drawing setup if you created your symbol in APD. To accommodate differing placement considerations, additional grids are windowed-in (added) for the core area of staggered power/ground pins; corner perimeter area grids address escape routing issues that may be more stringent than in the central perimeter areas covered by the base grid.

Sample Multiple Grid Layout



- i** Note that the use of multiple grids may have an adverse affect on tiles or pins that you add or move within the design. Also, to minimize the impact on performance that the visual representation of grids on dense designs would entail, grid points are not visible in the Design Window.

Pin Pattern Numbering with Multiple Grids

You can use different pin numbering schemes for every grid in your design. For example, you can choose a numbering approach for your core pattern that works best in a staggered, fully-populated layout area, and a different style that allows you to skip unused grid locations on the periphery of your design. You can label each numbering scheme separately to distinguish one from another, using alphanumeric prefixes, leading zeroes, and so on.

Pin renumbering is performed automatically whenever you complete add, delete, or move operations. This ensures that pin numbering is always up-to-date. In instances where you modify a pin, then cancel it (for example, if you use the *Oops* feature after adding a pin using a skip-unused-position scheme), you can force renumbering by using the *Redraw all pin numbers* button.

The Item Information Window

You can obtain information about the elements in your design any time during the editing process by clicking the *Item Info* button in the Component Editing dialog box. Selecting an *Item Type* from the pull-down list, then positioning the cursor over an instance of that type in your design causes the data associated with that object to appear in the Item Information window. You can view additional information by clicking *Display detailed info* (which opens a second information window) and highlighting the specified object by clicking *Highlight item*.

Item Information Window



Right-Button Pop-Up Support

The Package Designer Symbol Editors support operations available from the pop-up menu that displays when you right-click. Only operations pertinent to the mode you are in are active. The operations are:

<i>Cancel</i>	Exits the command without committing any changes.
<i>Oops</i>	<p>This operation varies according to where you are in an editing session:</p> <ul style="list-style-type: none"> • Lets you undo the last action of a completed editing operation, such as adding pins • Moves you back to the last action in a multi-stage operation. For example, if you selected pins to copy, but have not placed them, selecting <i>Oops</i> unselects the pins. • Backs you up to the previous symbol editor dialog box if you are not editing anything; for example, if you are in the finalization phase of the editing session.
<i>Next</i>	This operation has dual capabilities: in <i>Copy</i> mode, choosing <i>Next</i> clears your cursor of attached objects so you can select new objects to copy. In any other mode, <i>Next</i> moves you to the next stage of the editing process; for example, from component editing to finalization.
<i>Temp Group</i>	Allows you to choose multiple elements for simultaneous editing.
<i>Complete</i>	Completes the <i>Temp Group</i> operation and returns you to the main editing environment to continue your editing session.
<i>Reject</i>	Rejects the last <i>Temp Group</i> selection.
<i>Mirror</i>	Used in <i>Copy</i> and <i>Move</i> mode, this operation lets you mirror your selection in the y-axis. Used in combination with <i>Rotate</i> , <i>Mirror</i> , it is applied to the selection first. This option mirrors the group items geometry only; it does not change layers within a group.
<i>Rotate</i>	Used in <i>Copy</i> and <i>Move</i> mode, this operation lets you rotate your selection before placing it. The tool rotates only the selection, not its constituent parts, which you can rotate using the <i>Orientation</i> fields on the editing dialog box.
<i>Add more</i>	Active only when adding tiles, this operation lets you increase the number of tile instances attached to your cursor. You must click the right button while positioned over the editing dialog box to use this option.
<i>Remove some</i>	Active only when deleting tiles, this operation lets you decrease the number of tile instances attached to your cursor. You must click the right button while positioned over the editing dialog box to use this option.

Related Topics

- [Temp Group](#)

Package Designer Editors: Operating Parameters

To ensure that you obtain optimal results while using the Package Designer Editors, consider the following operating conditions.

- Symbols that you must edit:
 - Have a grid-based pin layout (or turn your grid to *free placement*, which diminishes the editors functionality)
 - Conform to JEDEC standards
 - Have the correct component class
- You can edit only one component at a time.
- You cannot edit a symbol if two symbol instances of the same symbol definition exist, because modifications to the symbol definition would propagate to all symbol instances.
- Components that you are editing for the first time and which were *not* created using one of the component generating tools initialize with a customized numbering pattern and the grid disabled. This condition does not apply to components created with the Die Generator, BGA Generator, or the New Design Wizard.
- You cannot edit the symbol until all the fields in the editors dialog boxes contain valid values.
- You cannot create new nets. You can only place pins on nets that existed prior to invoking the editors.
- You cannot assign the name of a new pin unless you are using a customized numbering scheme. The labelling scheme determines all pin names, which the editors create, assign, and maintain.

- Attributes you attach to a pin are not recreated if you delete the pin.
- You can undo (*Oops*) only the last operation that you performed.
- You cannot add metal, text, keepouts, or lines.
- Symbols and components must have at least one pin in order to be saved. This includes die pins in the Die Editor, BGA balls in the BGA Editor, or tile bumps in the Tile Editor.
- Pin numbers must be unique; otherwise symbol and component generation fail.
- All pin numbering schemes except *Customized* are controlled by the editor. User-control of numbering values is available only through the *Pin Numbering* frame of the Component Editing dialog box.
- You cannot display or hide pin number text on a per-pin or per-grid basis, only at the component level.
- Border text is not available when your pin numbering scheme is set to *Customized*, *Sequential*, if the pin pattern skips unused spaces, or if there are multiple grids.
- Metal connected to pins in the component that you are editing are not affected if you move the pins; that is, it is not moved as a unit with the pins.
- You cannot modify the symbol outline.
- You cannot add mechanical pins.
- You cannot add connect pins at off-grid locations.
- You cannot add more than one pin at the same location.
- Rotated symbols spin back to 0 degrees during the editing session. When you have completed, the rotation value is returned.
- Mirrored symbols are unmirrored during editing.
- Symbols created in versions prior to Release 15.2 that contained the `mirror_on` condition are uprevved with the `mirrored_geometry` attribute when they exit the edit process.



This does not automatically modify padstacks in the design to accommodate the new condition. You may need to manually modify padstacks of the same definition that appear elsewhere in the design.

Additionally, the mirror status of symbols that you create cannot be changed by way of interactive commands.

- The editors color code pins that you are editing based on pin use. The default color scheme is based on colors defined in the Color Dialog:

Power Red

Ground Green

Signal Blue

Unspecified/ No Connection Blue

Creating Jumper Package Symbols

This section provides information on how to create jumper package symbols. Jumpers represent hard wired connection that is used to connect a single net when etch connections are not possible.

Jumpers are used on single layer designs to solve routing problems. They are also used sometimes to control options on a board.

Jumper Package Symbol Elements

Jumper packages must have the following elements:

- Via Padstack
- Reference designator

Optionally, a jumper package may contain:

- Place-bound rectangle
- Component outline

 If a Place Bound shape is not defined in the jumper symbol, the tool uses the jumper symbol drawing extents when the jumper is placed in the design.

Prerequisites for Creating a Jumper Package Symbol

For each jumper, you should have the following physical data:

- Lead-to-Lead spacing
- Drill size
- Pad size

Creating a Jumper Package Symbol

All commands listed here are described in the *Allegro PCB and Package Physical Layout Command Reference*.

Defining the Drawing Type

1. Choose *File – New* and choose Package Symbol.
The New Drawing dialog box appears.
2. Enter a file name in the Drawing Name field.
3. Choose *Package symbol* from the Drawing Type list box, and choose *OK*.
4. Select *Setup – Design Parameters*.
The Design parameter Editor form will appear.
5. Select the *Design* tab on the *Design Parameters* form.
6. Select Jumper in the Drawing type field of the *Design Parameter* form, and choose *OK*.

Defining the Via List

1. Choose *Setup – Constraints – Physical*.
The Constraint Manager Physical worksheet is displayed.
2. In the Physical Constraint Set workbook, choose *All Layers*.
3. Scroll to the right and choose the column *Vias*.
4. Click in the *Vias* cell.
The *Edit Via List* dialog form appears.
5. Double-click on a via in the *Select a via from the library or the database* field.
The via populates the *Via list* field.
6. If more than one via is listed in the *Via list* field, then select the newly added via and click the *Up* button to move the via to the top of the *Via list* field.

Adding Vias

1. Choose *Layout – Connections*.
2. Move the cursor to the location where you want to place the through-hole via.
3. Choose one of these ways to add the via:
 - a. Double-click to automatically add a via.
 - b. Right-click and choose *Add Via* from the pop-up menu.

The via appears at the location where you clicked.

4. Right-click and choose *Next* from the pop-up menu.
5. Repeat previous steps to add the second via.
The distance between vias needs to match the jumper requirements.
6. Right-click and choose *Done* from the pop-up menu.

Adding Reference Designators

1. Choose *Layout – Labels – Refdes*.
2. Choose the *Subclass* and *Text* block on the *Options* tab.
3. Move the cursor to the location where you want to place the RefDes label.
4. Click to anchor the text box.
5. Enter the text at the command line.
6. Repeat previous steps to add additional text labels, such as *Silkscreen_Top*, to the symbol.
7. Right-click and choose *Done* from the pop-up menu.

Creating Package Boundary

1. Choose *Setup – Areas – Package Boundary*.
2. Set the *Class* and *Subclass* fields in the *Options* tab to *PACKAGE_GEOMETRY* and *PLACE_BOUND_TOP*.
3. Click to draw a polygon representing the area required for placement.
4. Right-click and choose *Done* to close the polygon.
The polygon is automatically filled solid.

STEP Model Support

STEP model support allows PCB designers to map package, device and mechanical symbols to 3D STEP models to view and verify the designs in the 3D viewing tool before manufacturing.

The STEP models provides complete, detailed and accurate three-dimensional model representation of the components. Using STEP models for critical components ensures that they are placed correctly and follow design constraints.

You can map STEP models in the PCB Editor and in the Symbol Editor. The mapped data is saved at the library and the design level. The mapping utility allows you to export the mapping formation along with the facet files for re-use or import into other designs. For reusing STEP mapping information into new designs, associate the STEP models with the symbol (.dra) files.

You can also export the board design as a STEP model that includes enclosures associated with the board for positioning and collision detection.

STEP Models

STEP stands for *Standard for the Exchange of Product model data* and is an international standard for data exchange between mechanical CAD/CAM/CAE models and PCB design tools. The STEP models describe graphical details for a physical part. The STEP file (.stp and .step) supports two formats, XML format and ASCII text format. The STEP model formats supported by PCB Editor for electrical and mechanical parts are AP203, AP214, and AP242.

You can get the STEP models through manufacturer sites, third-party part library solutions, and over the net.

Environment Variables for STEP Model Support

To assign STEP models to symbols and devices you need to define the path of STEP model library. The *steppath* environment variable identifies the directory in which STEP models are stored. You can set the path in the *User Preferences Editor* that is accessible from *Setup – User preferences – Paths – Library*.

To save the mapping data on export you need to set *step_mapping_path* environment variables. A set of facet files are also exported that are saved at the location defined by *step_facet_path* environment variable. The facet files represents the STEP geometry and assembly

information for 3D display. Prior to import mapping data, these variables should be set.

Related Topics

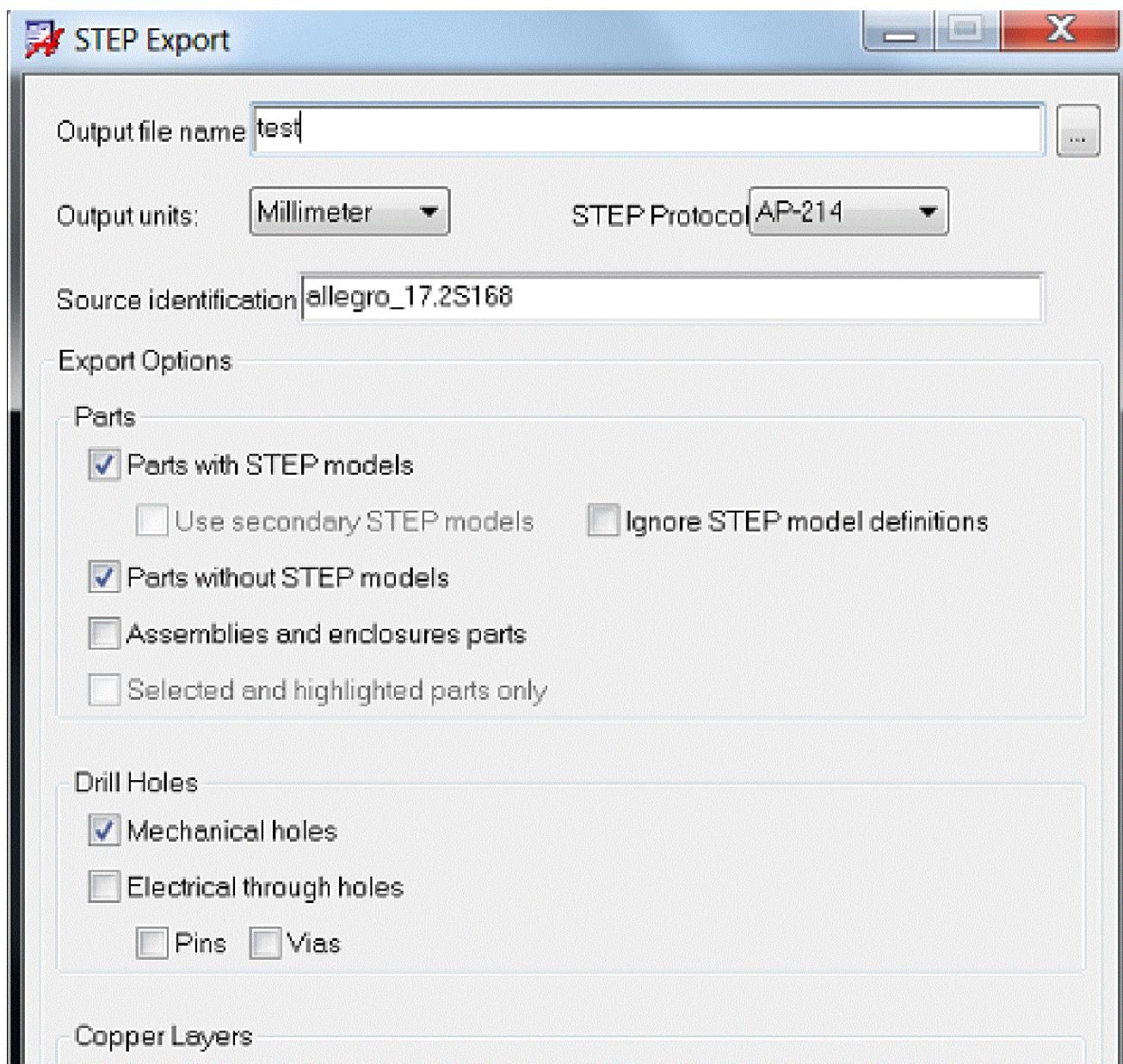
- [Mapping devices and symbols to STEP models](#)
- [Viewing STEP model in 3D Canvas](#)
- [Mapping mechanical symbols to STEP models](#)
- [Exporting a board design as a STEP model](#)

Exporting Board Drawings to a STEP Model

The STEP model support provides the ability to export a board drawing as a STEP model for use in a mechanical design environment.

STEP model export supports AP203, AP214 and AP242 protocols, standard units, external copper data, and various output options to minimize or maximize STEP model data.

Choose *File – Export – STEP* (`step out` command) to export physical design data to the STEP data format.





The STEP Export UI

Using STEP Export UI you can export:

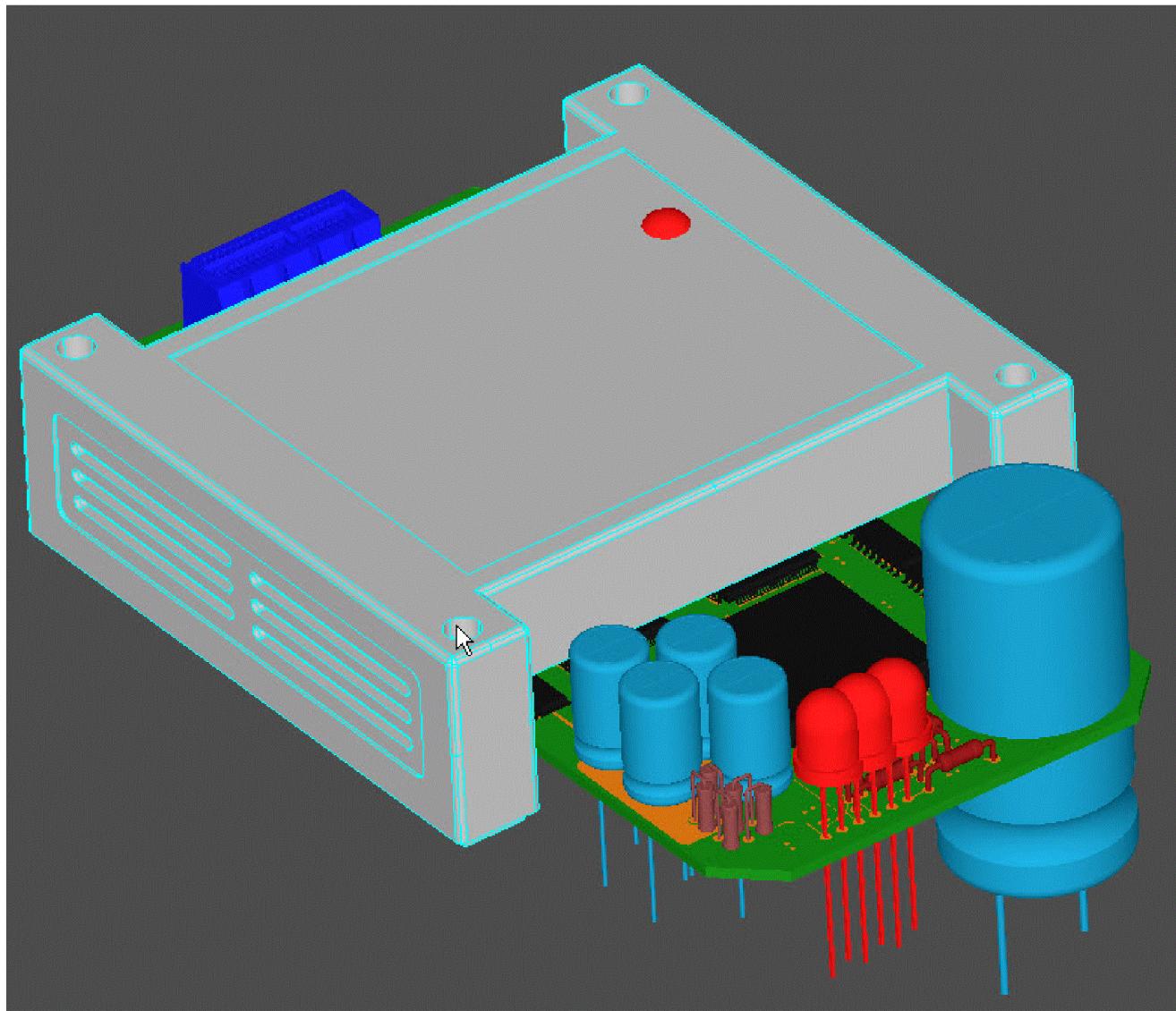
- parts with STEP models in the current design.
- parts without STEP models. If symbols without mapped STEP models exist in the design, the symbols are exported as defined by the PACKAGE GEOMETRY – PLACE_BOUND_TOP/BOTTOM.
- assemblies and enclosure parts. Exports the STEP3D_MECH models created in the STEP Package Mapping utility.
- mechanical and electrical through holes defined in the current design.
- external or internal traces, pads, and shapes on the ETCH/TOP and ETCH/BOTTOM layers.

⚠ This option may increase the size of step file.

- highlighted parts with and without STEP model assignments.
- secondary STEP models.
- bare physical board with mechanical holes and external traces.
- ignore STEP models on mapped symbols.

Exported step file gets package definitions from the library and is small in size. If you change package height and outline in the design and want to export the modified package definitions, set an environment variable *step_board_level_package_height*. This variable exports design-level package definitions for all the parts, but the size of the exported step file increases.

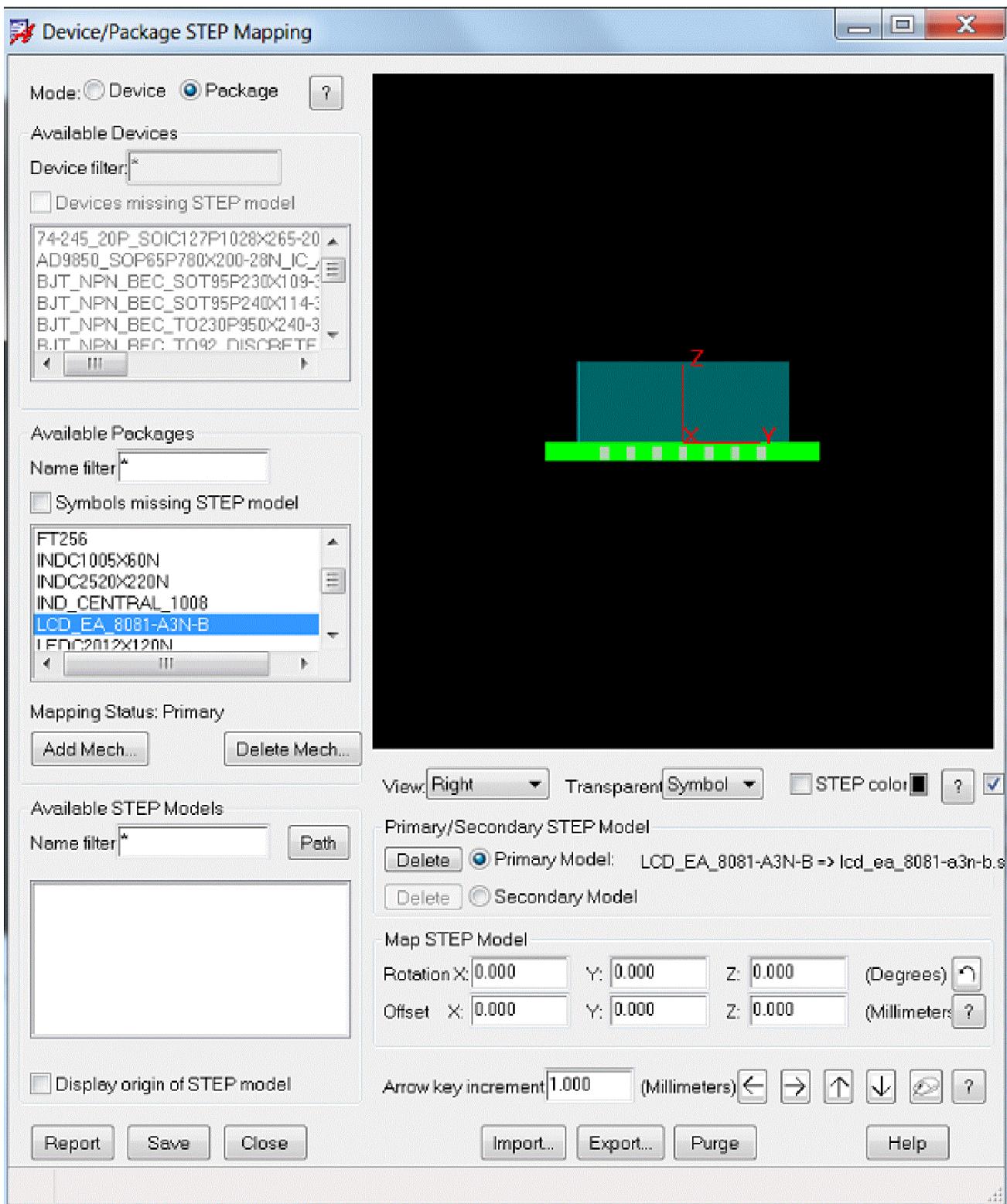
To view the exported STEP model file you can use the STEP model viewer.



Mapping Devices and Symbols to STEP Models

PCB editor provides a mapping tool *Device/Package STEP Mapping* to associate STEP models to package, device and mechanical symbols. This tool maps the STEP model name to the symbol or device and defines offset and rotational information to correctly position the STEP model in 3D Canvas. You can map both the primary and the secondary models. The mapping data created is then instantiated into the symbol or device as a property.

-  For a mapped or unmapped model to be displayed in Allegro 3D Canvas, the board or package symbol must contain a PACKAGE GEOMETRY class and PLACE_BOUND_TOP or PLACE_BOUND_BOTTOM subclass.



The Device/Package STEP Mapping UI

- lists all the devices and symbols in the current design
- adds or deletes mechanical symbol
- display mapping status for packages
- lists all the available STEP models as defined by *steppath* variable
- display origin of STEP models
- displays graphic pane for viewing package and STEP models
- provides various viewing options
- sets different transparent modes for better viewing
- overlays the package and STEP models
- hides board-section from display
- uses STEP model colors
- displays color legends for graphics
- defines primary and secondary STEP models
- sets the offset and rotation values
- provides mouse buttons control in the graphic pane
- provides arrow key movements to move the STEP models in the graphic pane
- deletes the primary and secondary STEP mapping
- displays a report that includes device name, package name, STEP model name, rotational and offsets mapping
- deletes STEP device/package mapping information and the STEP facet data from the database
- exports the mapping data along with facet information

To map the STEP model, select a device or a package from the *Available Devices or Available Packages* list and a STEP model from *Available STEP Models*. The selected model and package are displayed in the graphic pane. You can enable the checkbox *Display origin of STEP models* for viewing origin of STEP models in the graphic pane.

To see the differences in origins overlay models, rotate and set X, Y, and Z offset values. Use arrow keys to move the STEP models in left/right/up/down directions. You can enable mouse buttons actions for moving, panning and zooming the STEP models in the display panel. Save the mapping data in the current working directory or exporting it to some other location.

Exporting and Importing Mapping Data

Using *Device/Package STEP Package Mapping UI*, you can export the mapping data into and XML format to a .map file. There is another set of XML files called facet files that are also exported as mapping data into a .zip file(`stepFacetFiles4Map.zip`). These files have facet representation of the STEP geometry and assembly, required to view the STEP models in 3D Canvas. The mapping data files are saved at the location specified by the `step_mapping_path` and `step_facet_path` environment variables.

The mapping data can be re-used by importing it into another design. You can use *Import* button in the *Device/Package STEP Mapping UI*. On importing, the mapping data available in the .map and facet files are attached to the design.

-  Before importing, extract STEP facet files into the
`<working_directory>/stepFacetFiles4Map/`.

You can import multiple mapping files into a design.

Sample Mapping file

```
<?xml version = "1.0" encoding = "UTF-8"?>
<STEP-3D-MAPPING revision="A" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Header lengthUnit="MM"/>
  <Body>
    <MapItem package="CC5V-T1A">
      <StepModel type="PRIMARY" file="xtal_smt.stp">
        <Offset x="2.050000" y="0.000000" z="0.000000"/>
        <Rotation x="0.000" y="0.000" z="0.000"/>
      </StepModel>
    </MapItem>
    <MapItem package="CAP25X50MM-RAD">
      <StepModel type="PRIMARY" file="Cap25x50.STEP">
        <Offset x="581.500000" y="229.500000" z="0.000000"/>
        <Rotation x="0.000" y="0.000" z="0.000"/>
        <MapColor r="-1.000000" g="-1.000000" b="-1.000000"/>
      </StepModel>
    </MapItem>
    <MapItem package="CAP10X16MM-RAD">
      <StepModel type="PRIMARY" file="Cap10x16.STEP">
        <Offset x="175.100000" y="62.900000" z="9.000000"/>
        <Rotation x="0.000" y="0.000" z="0.000"/>
        <MapColor r="-1.000000" g="-1.000000" b="-1.000000"/>
      </StepModel>
    </MapItem>
  </Body>
</STEP-3D-MAPPING>
```

Properties for STEP Model Support

The mapping tool assigns two properties to the package symbol:
`PKGDEF_STEP_TRANSFORMATION`, and, `PKGDEF_STEP_FILE`. These properties become part of the symbol definition and cannot be modified outside of the STEP Package Mapping utility.

Select the symbol, right-click and select *Show Element*. The *Show Element* form displays both the properties.

The screenshot shows a Windows application window titled "Show Element". The window has a toolbar with icons for "Show Element", "Delete", "Print", and "Help". There is a search bar with a "Search:" field and two checkboxes: "Match word" and "Match case". The main content area displays the following text:

```
LISTING: 1 element(s)

< SYMBOL >

RefDes:      Y1
Symbol name: CC5V-T1A
origin-xy:   (12.0650 50.8000)
rotation:    0.000 degrees
not_mirrored

Attached text:
class        = REF DES
subclass     = ASSEMBLY_TOP
value        = Y1

Properties attached to symbol definition
PKGDEF_STEP_TRANSFORMATION = MM,2.050000,-0.750000,0.000000,0.000
                           ,0.000,0.000
PKGDEF_STEP_FILE   = xtal_smt.stp, 528568, 1355904560, 0, 0.000000,
                     0.000000, 0.000000, 738338
LIBRARY_PATH      = D:/Projects/STEP/STEP_workshop/symbols/cc5v-t1
                     .psm

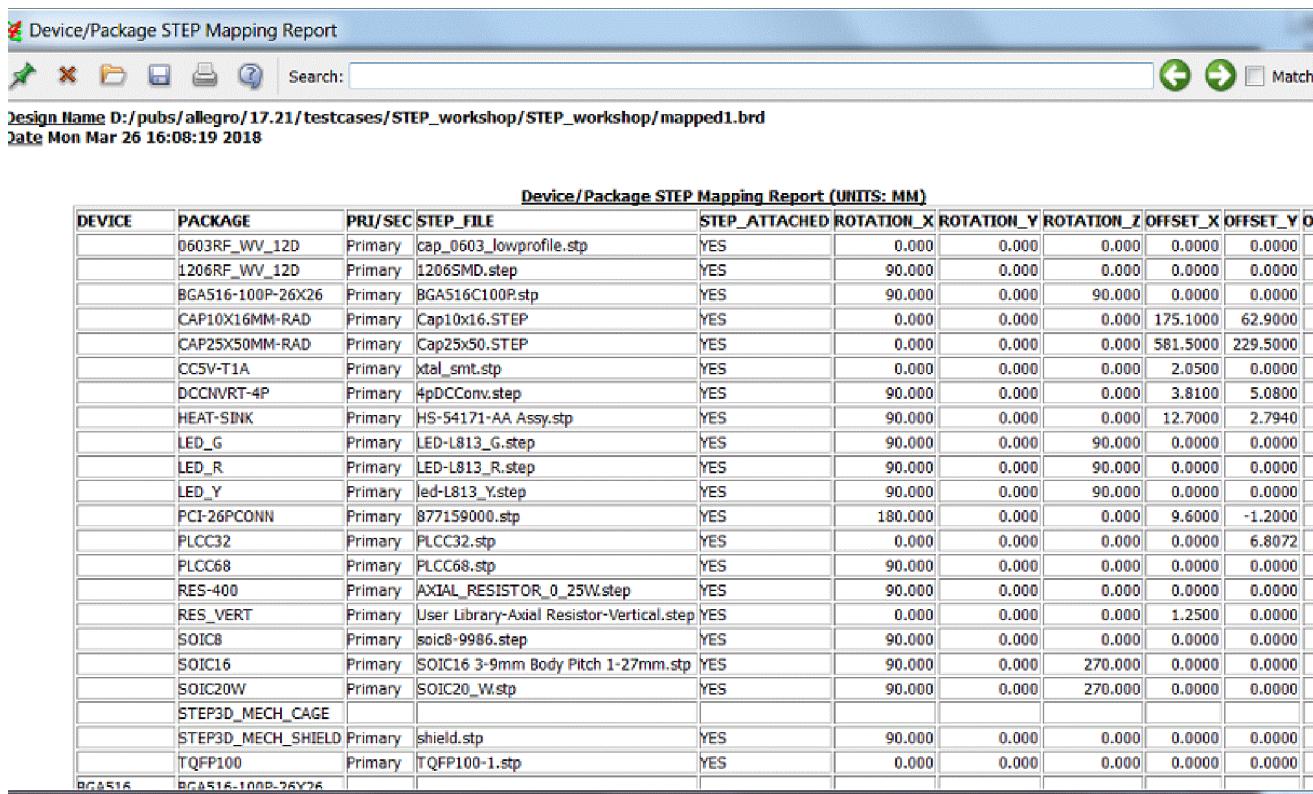
-----Component Instance Y1-----
Component Class:      DISCRETE
Device Type:          XTAL

Function(s):
Designator: TF-121
Type:      TMPDEV
Pin(s):    1, 2

Pin IO Information:
  Pin   Type   SigNoise Model      Net
  ---   ----   -----            ---
  1     UNSPEC           CLK53
  2     UNSPEC           CLK51
```

To delete the primary or secondary STEP model assignment using *Delete* button.

Assign STEP models to all the symbols and complete the mapping. Generate report to see the mapping details.



The screenshot shows a software interface titled "Device/Package STEP Mapping Report". The window includes a toolbar with icons for file operations (New, Open, Save, Print, etc.) and search functions. Below the toolbar, the "Design Name" is listed as "D:/pubs/allegro/17.21/testcases/STEP_workshop/STEP_workshop/mapped1.brd" and the "Date" is "Mon Mar 26 16:08:19 2018". The main area displays a table titled "Device/Package STEP Mapping Report (UNITS: MM)". The table has columns for DEVICE, PACKAGE, PRI/SEC, STEP_FILE, STEP_ATTACHED, ROTATION_X, ROTATION_Y, ROTATION_Z, OFFSET_X, OFFSET_Y, and OFFSET_Z. The data in the table lists various components and their corresponding STEP files and transformation parameters.

DEVICE	PACKAGE	PRI/SEC	STEP_FILE	STEP_ATTACHED	ROTATION_X	ROTATION_Y	ROTATION_Z	OFFSET_X	OFFSET_Y	OFFSET_Z
0603RF_WV_12D	Primary		cap_0603_lowprofile.stp	YES	0.000	0.000	0.000	0.0000	0.0000	
1206RF_WV_12D	Primary		12065MD.step	YES	90.000	0.000	0.000	0.0000	0.0000	
BGA516-100P-26X26	Primary		BGA516C100P.stp	YES	90.000	0.000	90.000	0.0000	0.0000	
CAP10X16MM-RAD	Primary		Cap10x16.STEP	YES	0.000	0.000	0.000	175.1000	62.9000	
CAP25X50MM-RAD	Primary		Cap25x50.STEP	YES	0.000	0.000	0.000	581.5000	229.5000	
CC5V-T1A	Primary		xtal_smt.stp	YES	0.000	0.000	0.000	2.0500	0.0000	
DCCNVRT-4P	Primary		4pDCCconv.step	YES	90.000	0.000	0.000	3.8100	5.0800	
HEAT-SINK	Primary		HS-54171-AA.Assy.stp	YES	90.000	0.000	0.000	12.7000	2.7940	
LED_G	Primary		LED-L813_G.step	YES	90.000	0.000	90.000	0.0000	0.0000	
LED_R	Primary		LED-L813_R.step	YES	90.000	0.000	90.000	0.0000	0.0000	
LED_Y	Primary		led-L813_Y.step	YES	90.000	0.000	90.000	0.0000	0.0000	
PCI-26PCONN	Primary		877159000.step	YES	180.000	0.000	0.000	9.6000	-1.2000	
PLCC32	Primary		PLCC32.stp	YES	0.000	0.000	0.000	0.0000	6.8072	
PLCC68	Primary		PLCC68.stp	YES	90.000	0.000	0.000	0.0000	0.0000	
RES-400	Primary		AXIAL_RESISTOR_0_25W.step	YES	90.000	0.000	0.000	0.0000	0.0000	
RES_VERT	Primary		User Library-Axial Resistor-Vertical.step	YES	0.000	0.000	0.000	1.2500	0.0000	
SOIC8	Primary		soic8-9986.step	YES	90.000	0.000	0.000	0.0000	0.0000	
SOIC16	Primary		SOIC16 3-9mm Body Pitch 1-27mm.step	YES	90.000	0.000	270.000	0.0000	0.0000	
SOIC20W	Primary		SOIC20_W.stp	YES	90.000	0.000	270.000	0.0000	0.0000	
STEP3D_MECH_CAGE										
STEP3D_MECH_SHIELD	Primary		shield.stp	YES	90.000	0.000	0.000	0.0000	0.0000	
TQFP100	Primary		TQFP100-1.stp	YES	0.000	0.000	0.000	0.0000	0.0000	
BGA516	Primary		BGA516-100P-26X26							

You can also map low resolution secondary STEP model to the same device and package symbols. The mapping tool assigns two properties to the package symbol for alternative STEP models: PKGDEF_ALT_STEP_TRANSFORMATION, and PKGDEF_ALT_STEP_FILE. These properties become part of the symbol definition and cannot be modified outside of the STEP Package Mapping dialog box.

Mapping Mechanical Symbols to STEP Models

With STEP model support you can view mechanical objects such as shields, fans, heat sinks and housings in 3D Canvas and visually check for any collisions or other placement issues.

Mapping Mechanical STEP Models

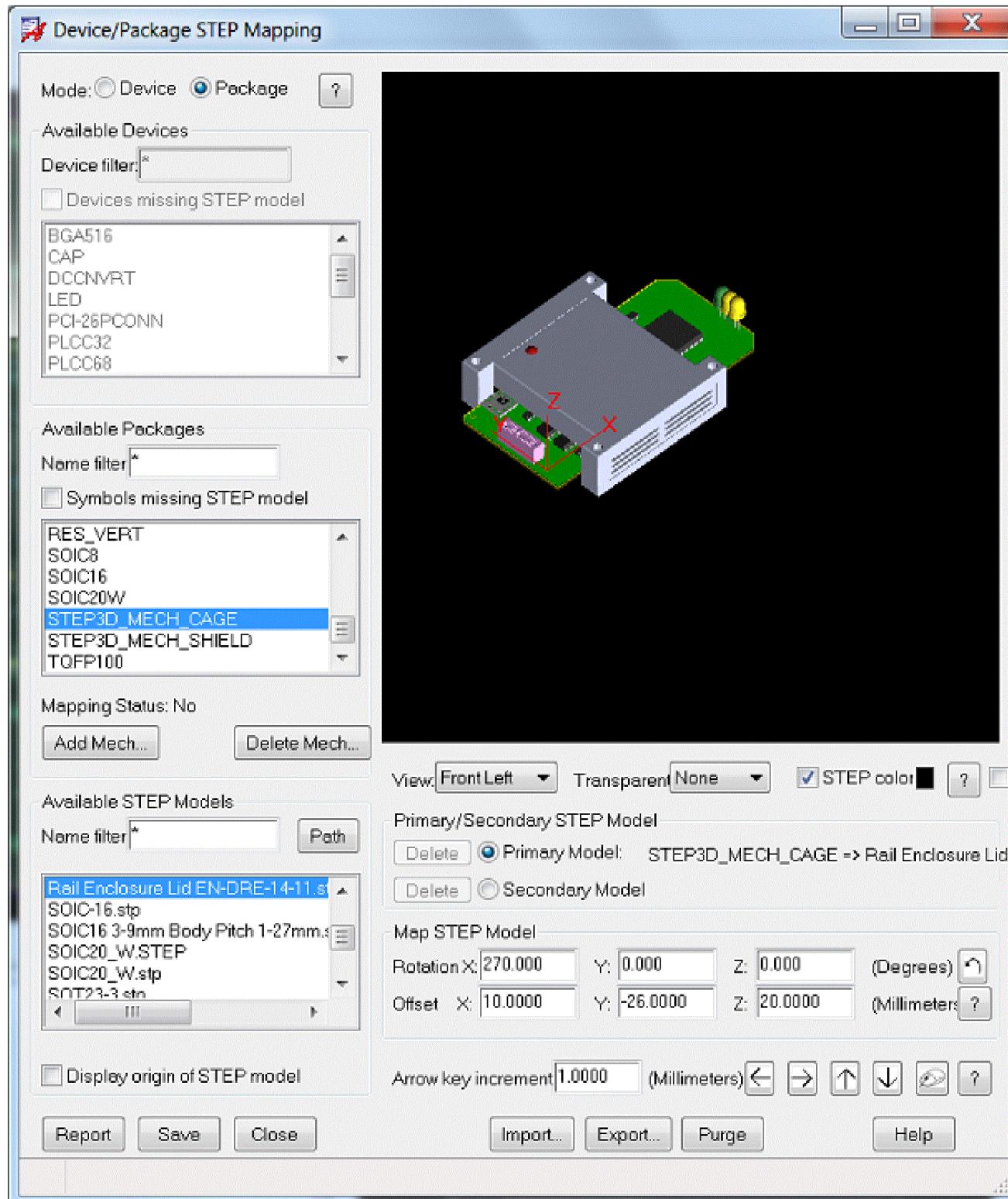
The *Device/Package STEP Mapping* dialog box lets you add/remove mechanical models and maps to STEP models. These mechanical symbols are stored at the same location as defined by `steppath` environment variable.

You can create a board or mechanical symbol that represents the mechanical model (enclosure) and map to the STEP models. This board symbol used for mapping is placed on the board drawing origin. The offset values defined in the mapping tool places the enclosure STEP model onto the proper location and orientation in the board drawing.

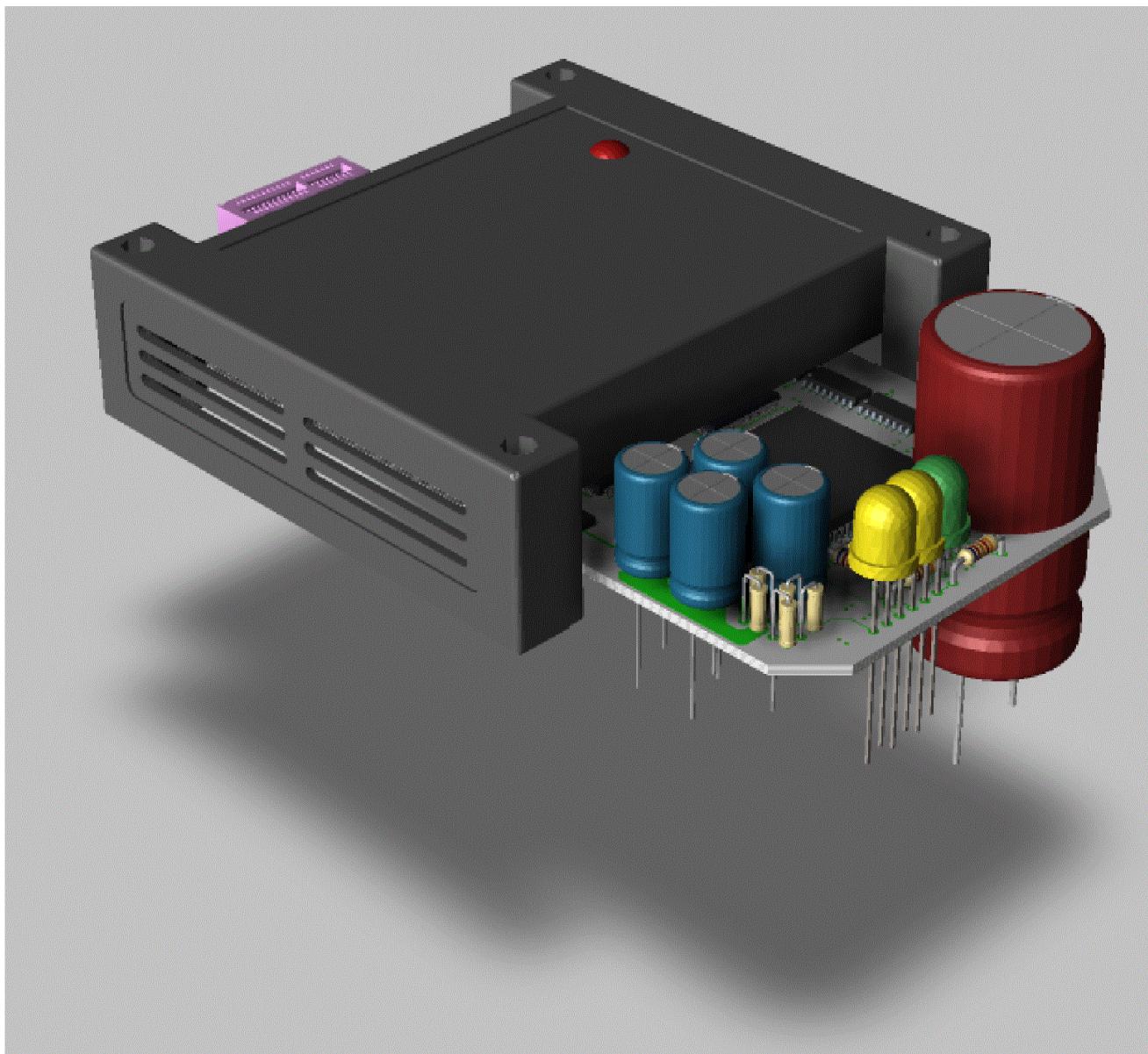
To improve the mapping performance for mechanical assembly or enclosure you can set the following variables in the *User Preferences Editor*.

- `step_ignore_all_electrical_packages`: to filter all the electrical packages.
- `step_display_resistors_capacitors`: to turn on the display of resistors and capacitors in the graphical area. By default, all the resistors and capacitors are filtered from the graphic pane.

In the following example, a board symbol `STEP3D_MECH_CAGE` is placed on the drawing origin and mapped to a STEP model defining an enclosure.



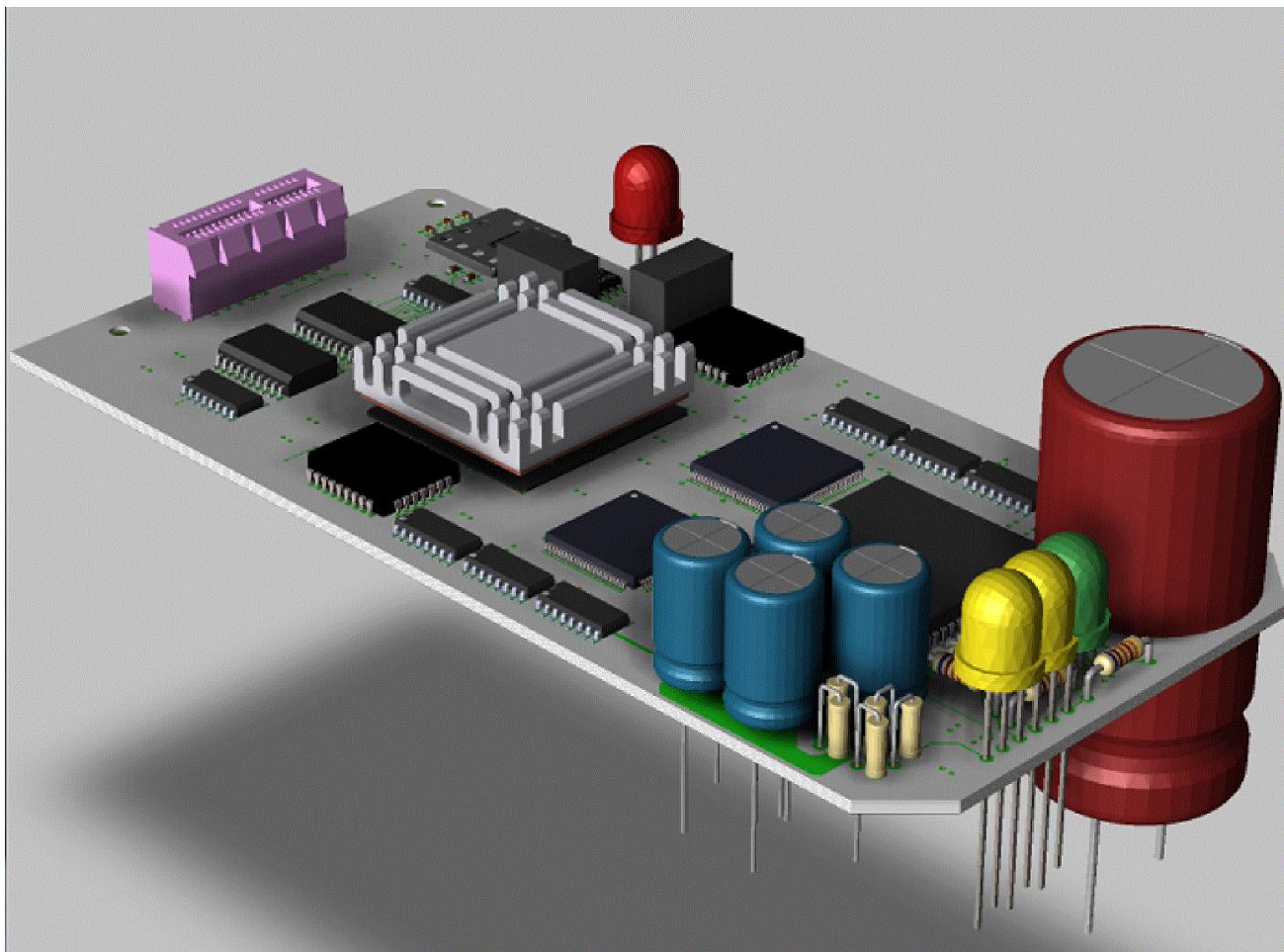
3D Canvas displays the mechanical model as below.



Viewing STEP Models in 3D Canvas

The STEP models provides more accurate view of symbols in Allegro 3D Canvas. You can visually check whether the symbol placement, position, and proximity to other symbols is proper and decide if a violation of design constraints occur.

When you view STEP models in 3D Canvas it first checks if the device mapping is available. If device mapping is not present then package mapping is used for STEP models.



⚠ If the STEP file has no associated color, 3D Canvas uses a dark grey as a default.

To view the secondary STEP model in 3D Canvas, enable the *Use secondary step models* option in 3D Canvas function in Display tab of the *Design Parameter Editor*, available by choosing *Setup – Design Parameters*.

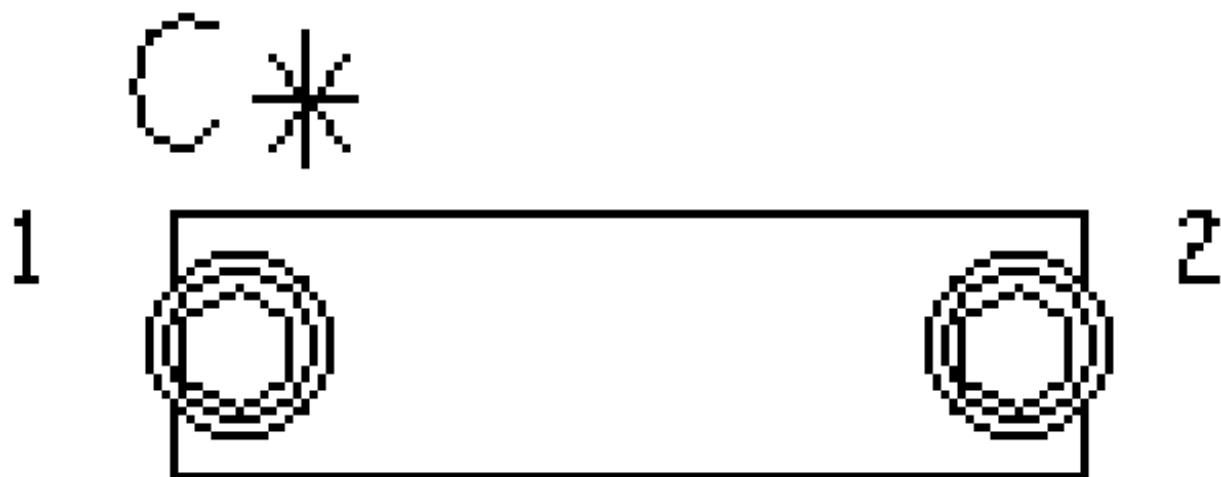
For more information, refer to *Allegro X PCB Editor 3D Canvas*.

Appendix A: Package/Component Symbol Library

This section illustrates some of the package/component symbols available in the library.

Capacitors

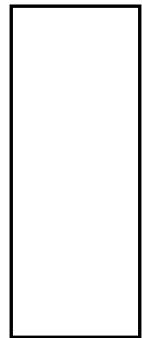
cap300



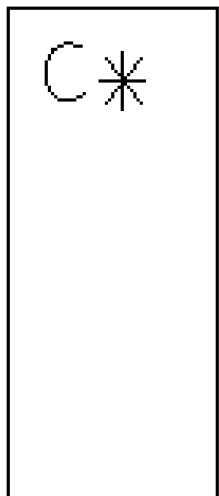
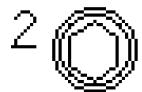
cap400



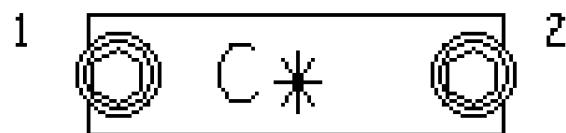
C *



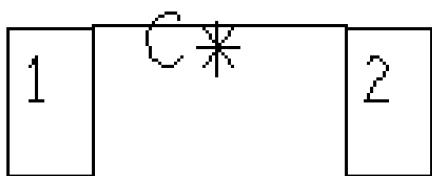
cap600



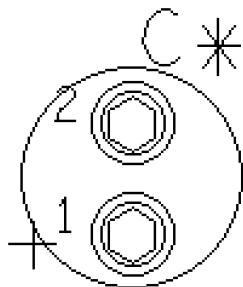
dipcap



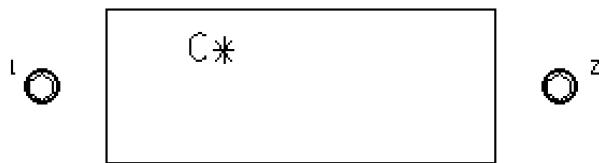
smdcap



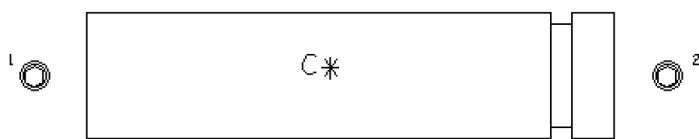
cap196



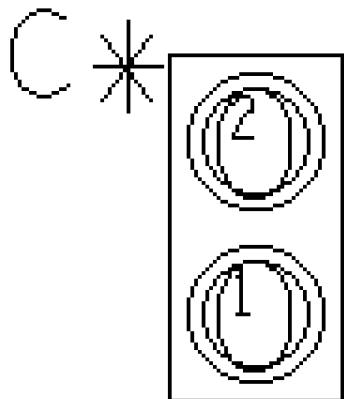
cap1000



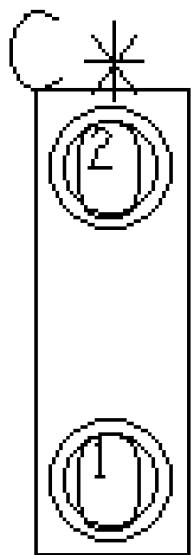
cap1500



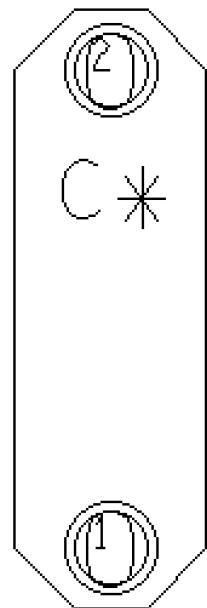
capck05



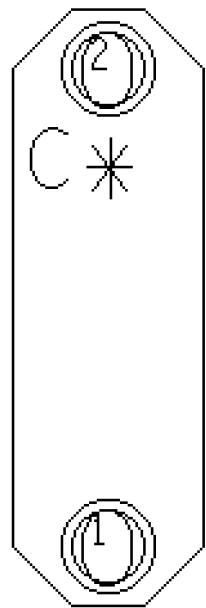
capck06



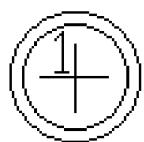
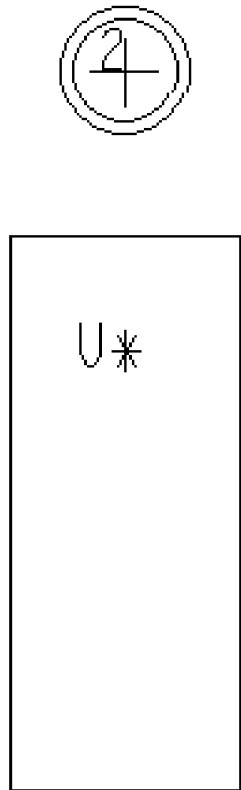
capck60



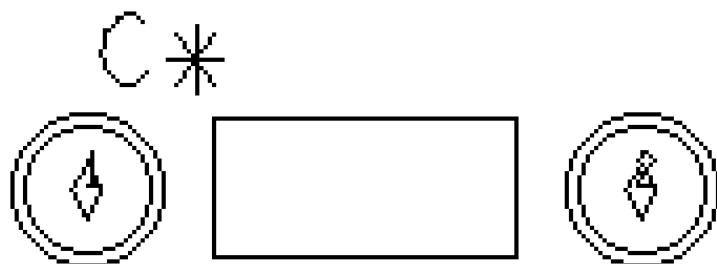
capck62



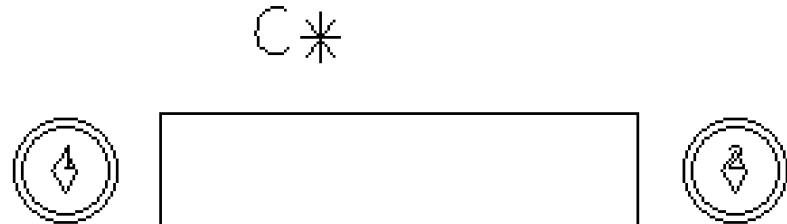
case17-02



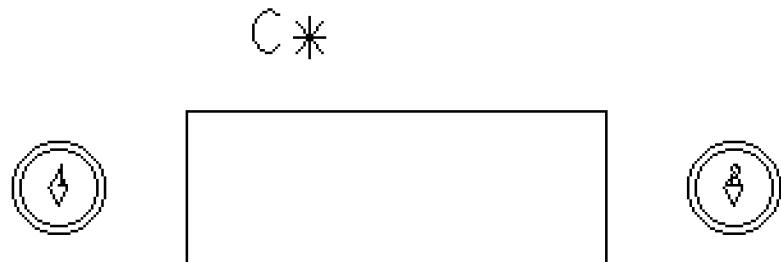
ck12-10pf



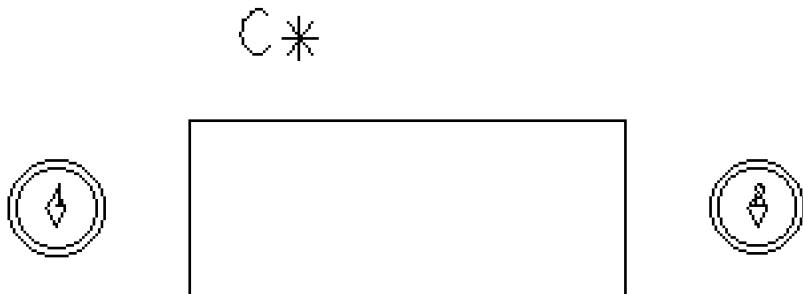
ck13-10pf



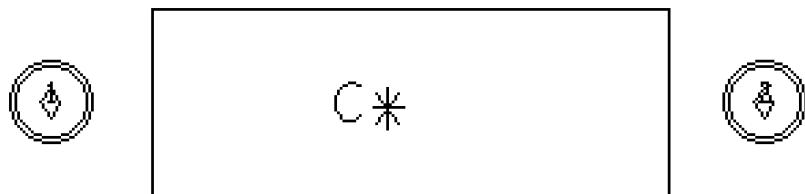
ck14-10pf



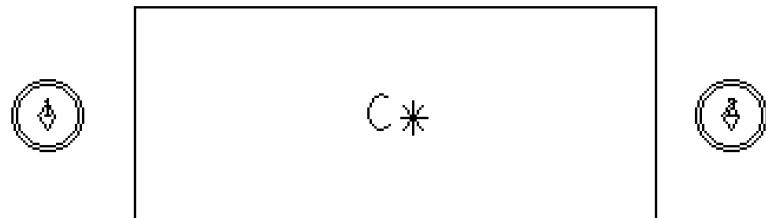
ck15-10pf



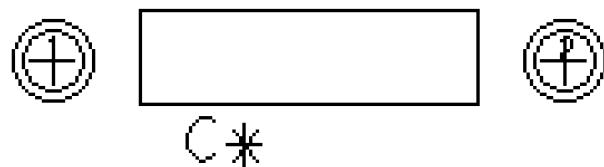
ck16-10pf



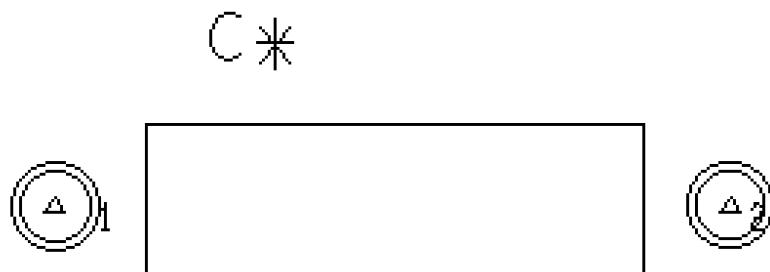
ck17-10pf



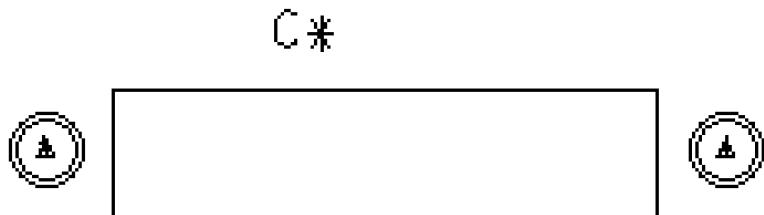
cy10



cy15

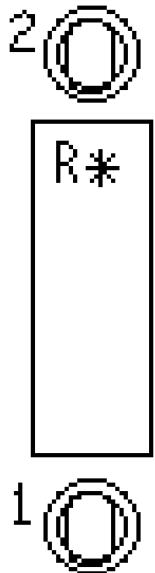


cy20

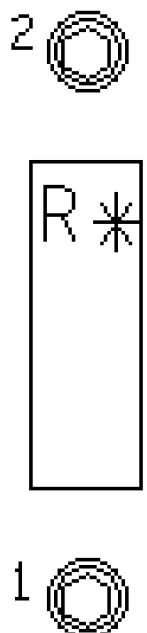


Resistors

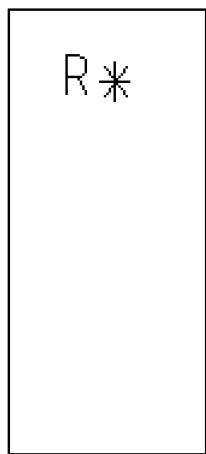
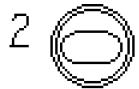
res400



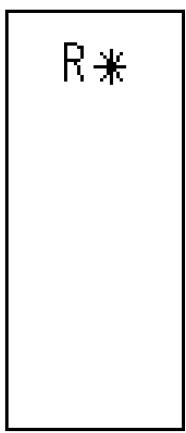
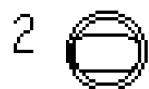
res500



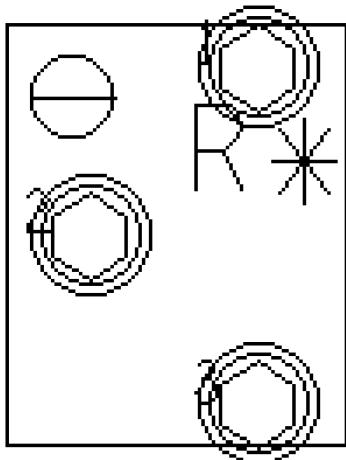
res1000



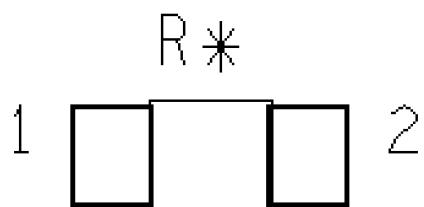
res800



resadj

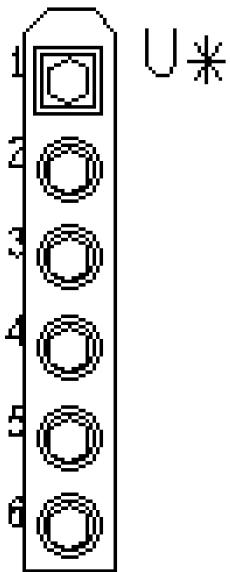


smdres

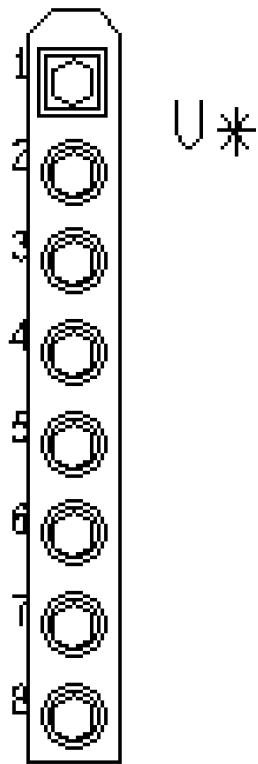


SIPs

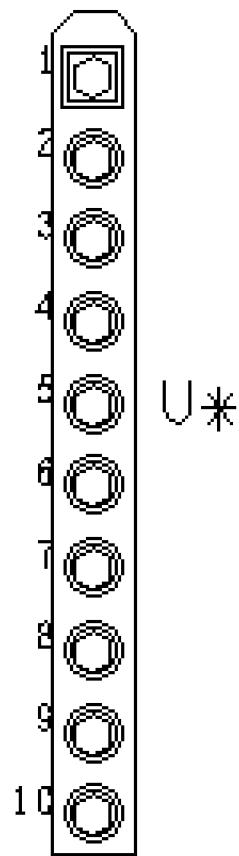
sip6



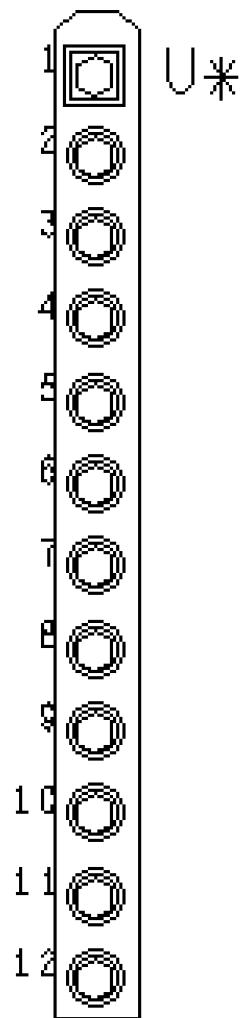
sip8



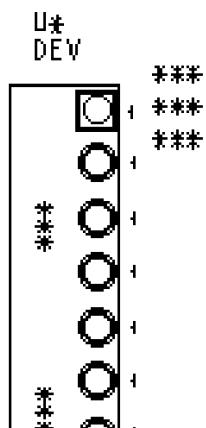
sip10

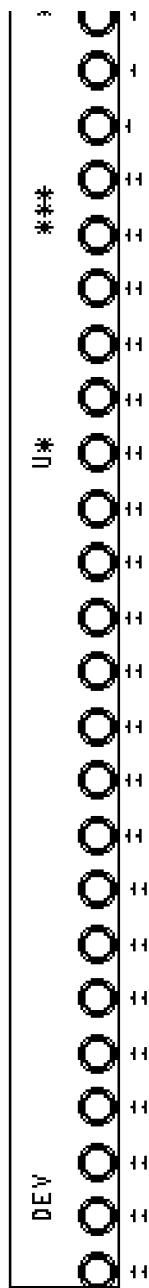


sip12



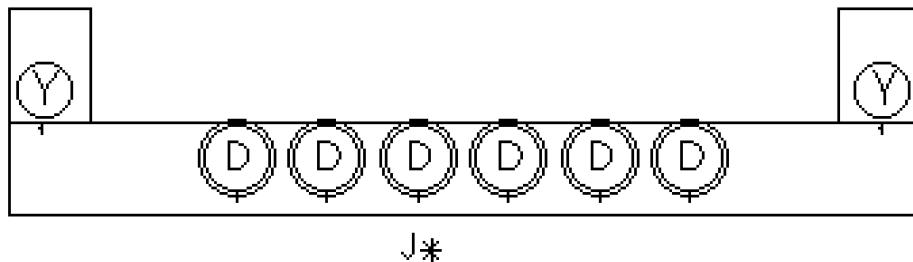
sip30



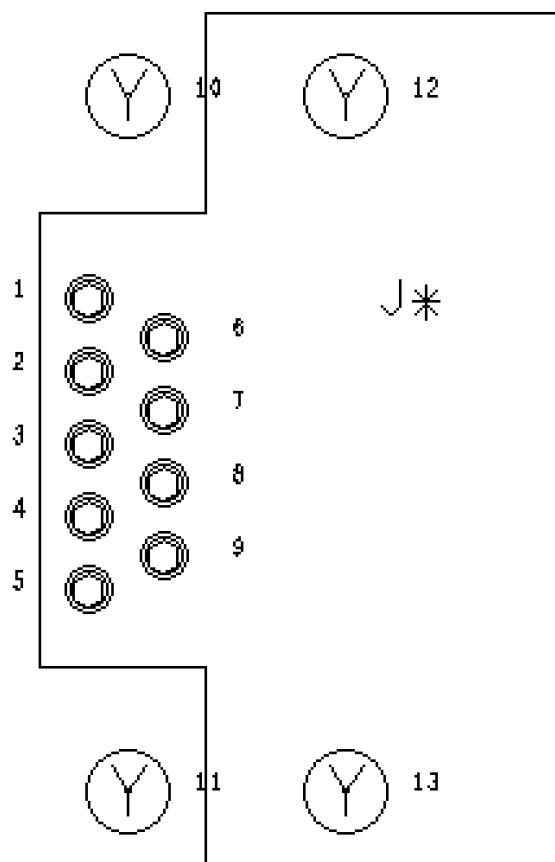


Connectors

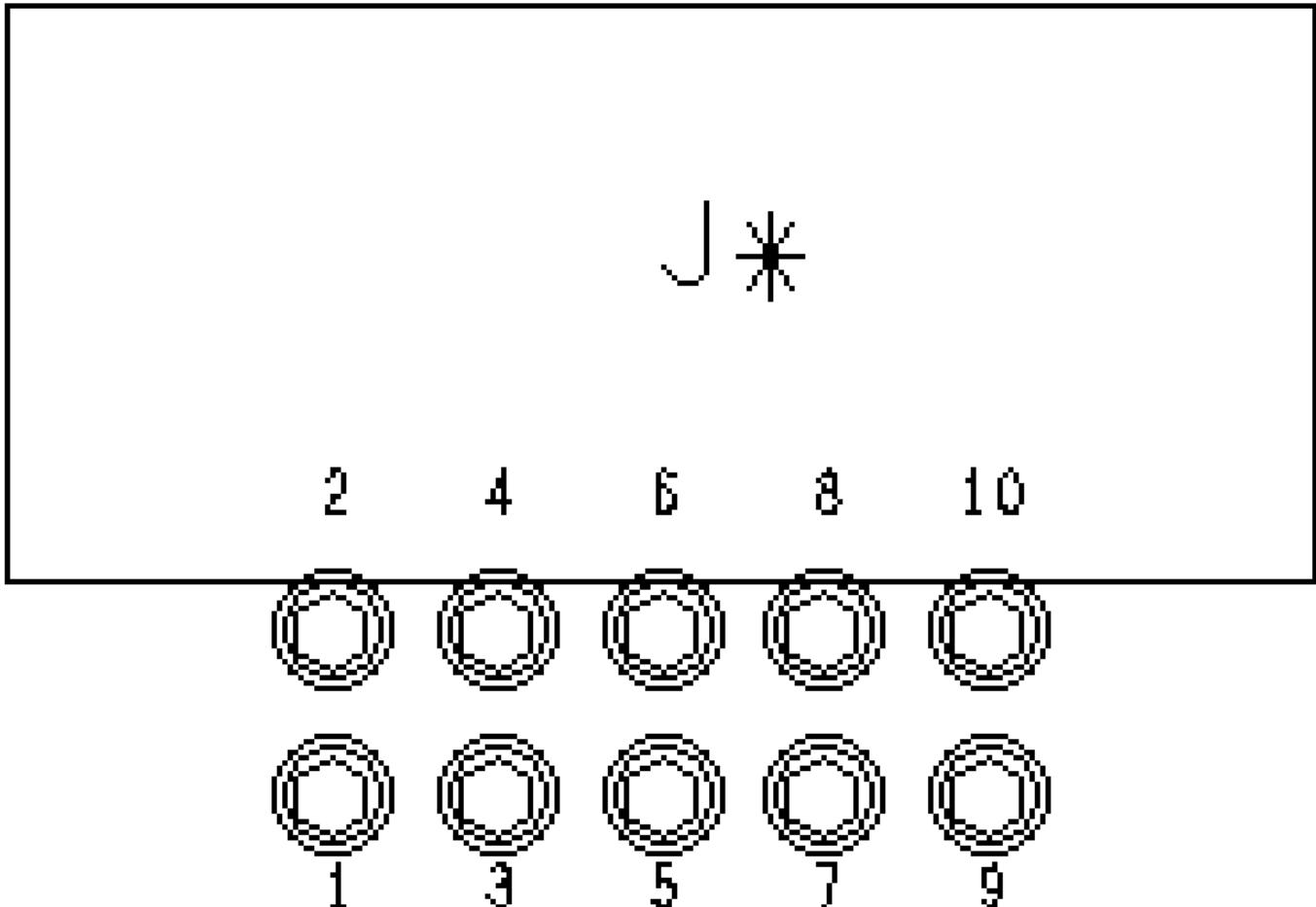
conn6



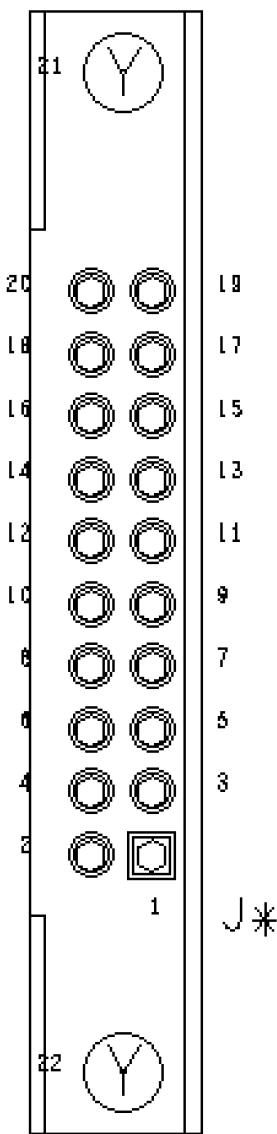
conn9



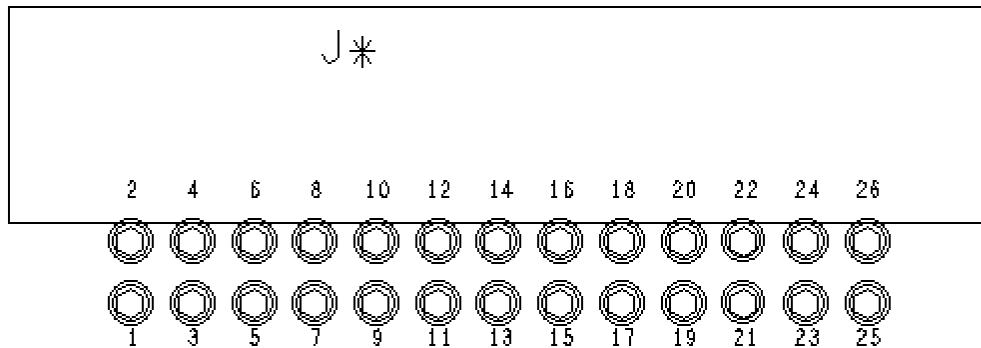
conn10



conn20



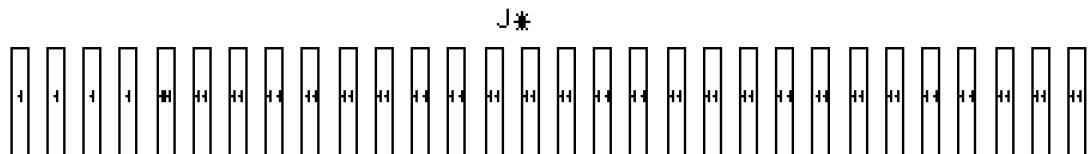
conn26



conn50



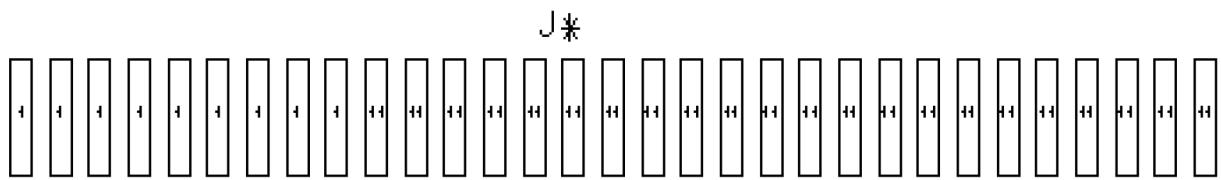
multiconn30



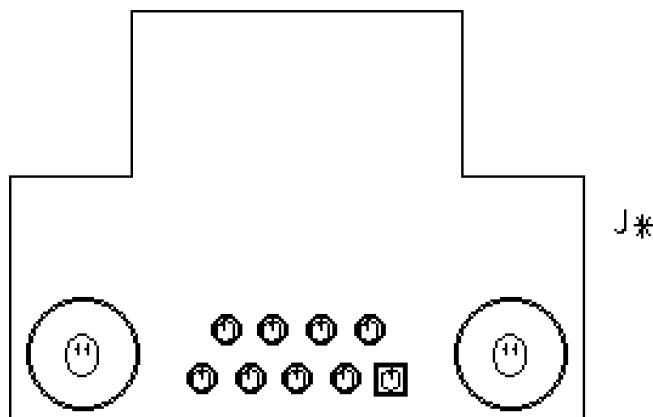
multicon43



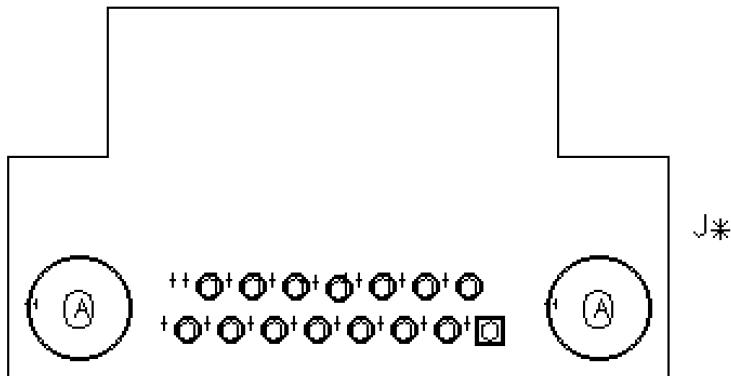
ibmconn



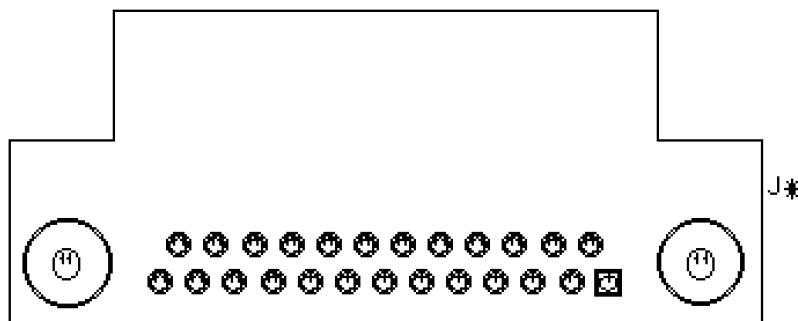
db9



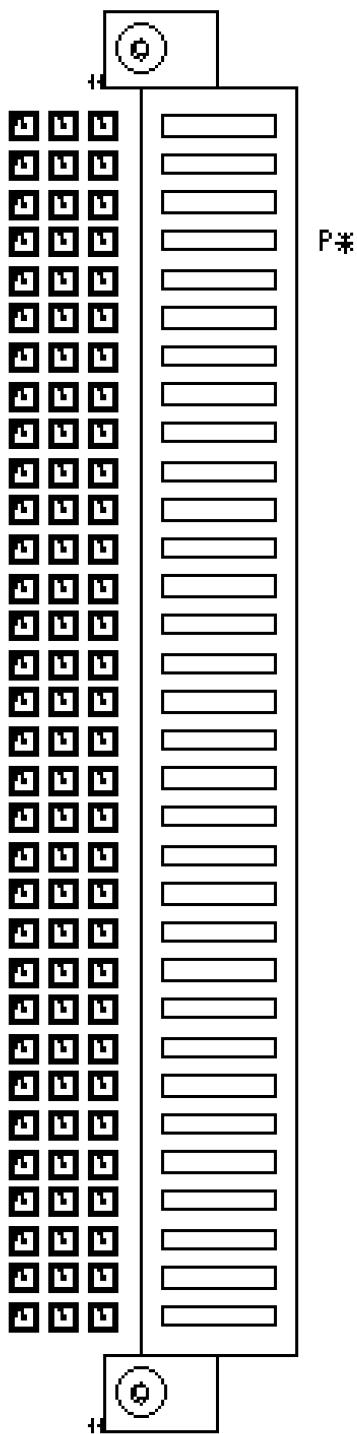
db15



db25

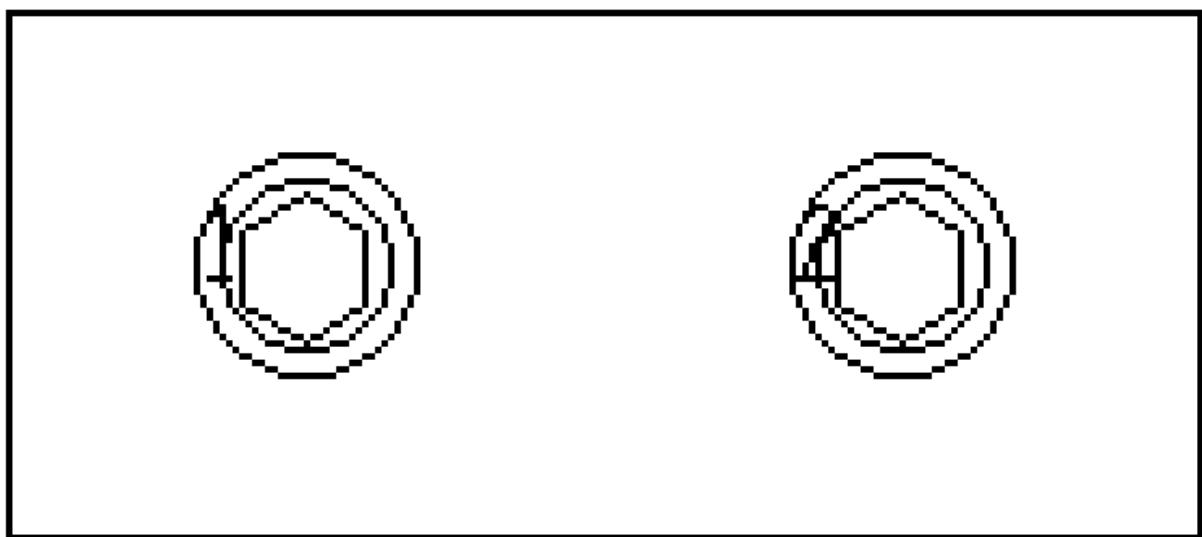


eurocon

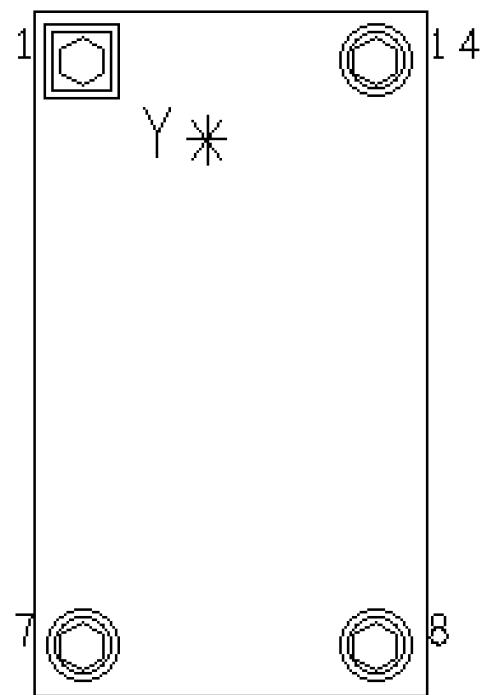


Crystals

crys11mhz

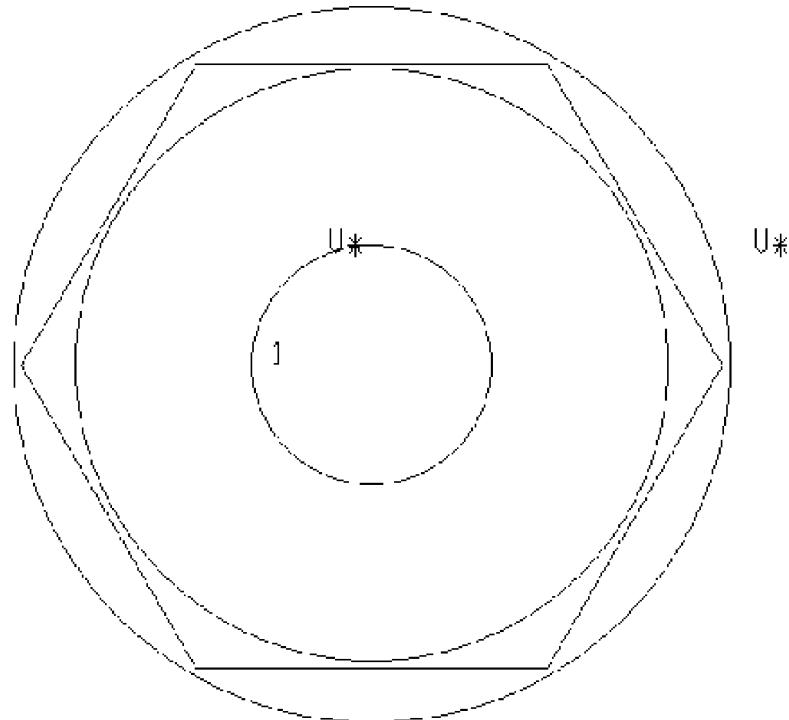


crys14

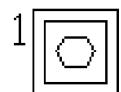
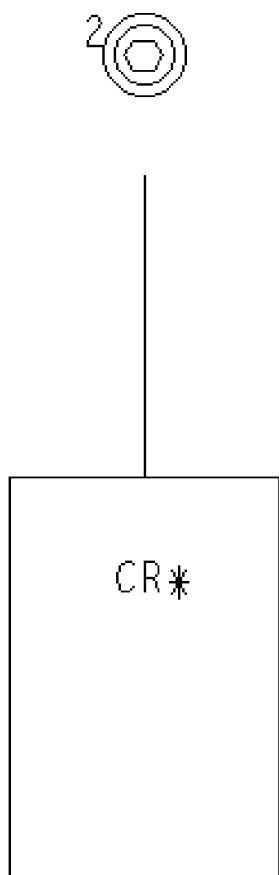


Diodes

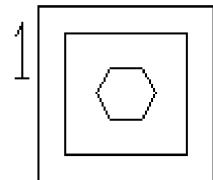
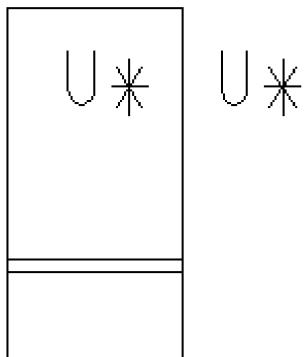
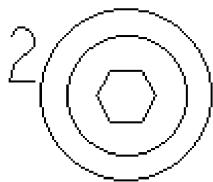
do5



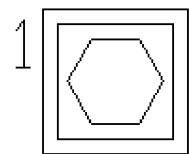
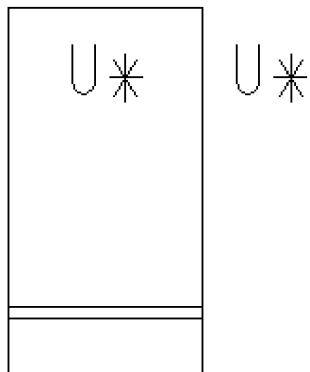
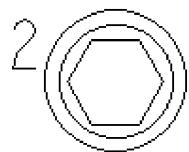
do13



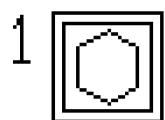
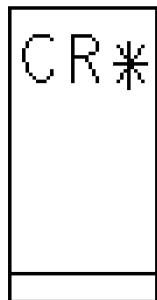
do35



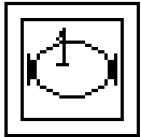
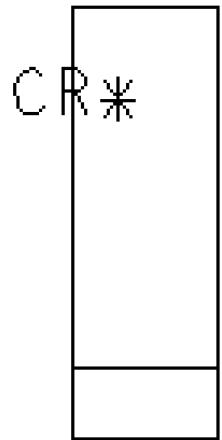
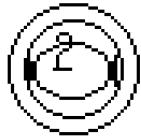
do41



dio400



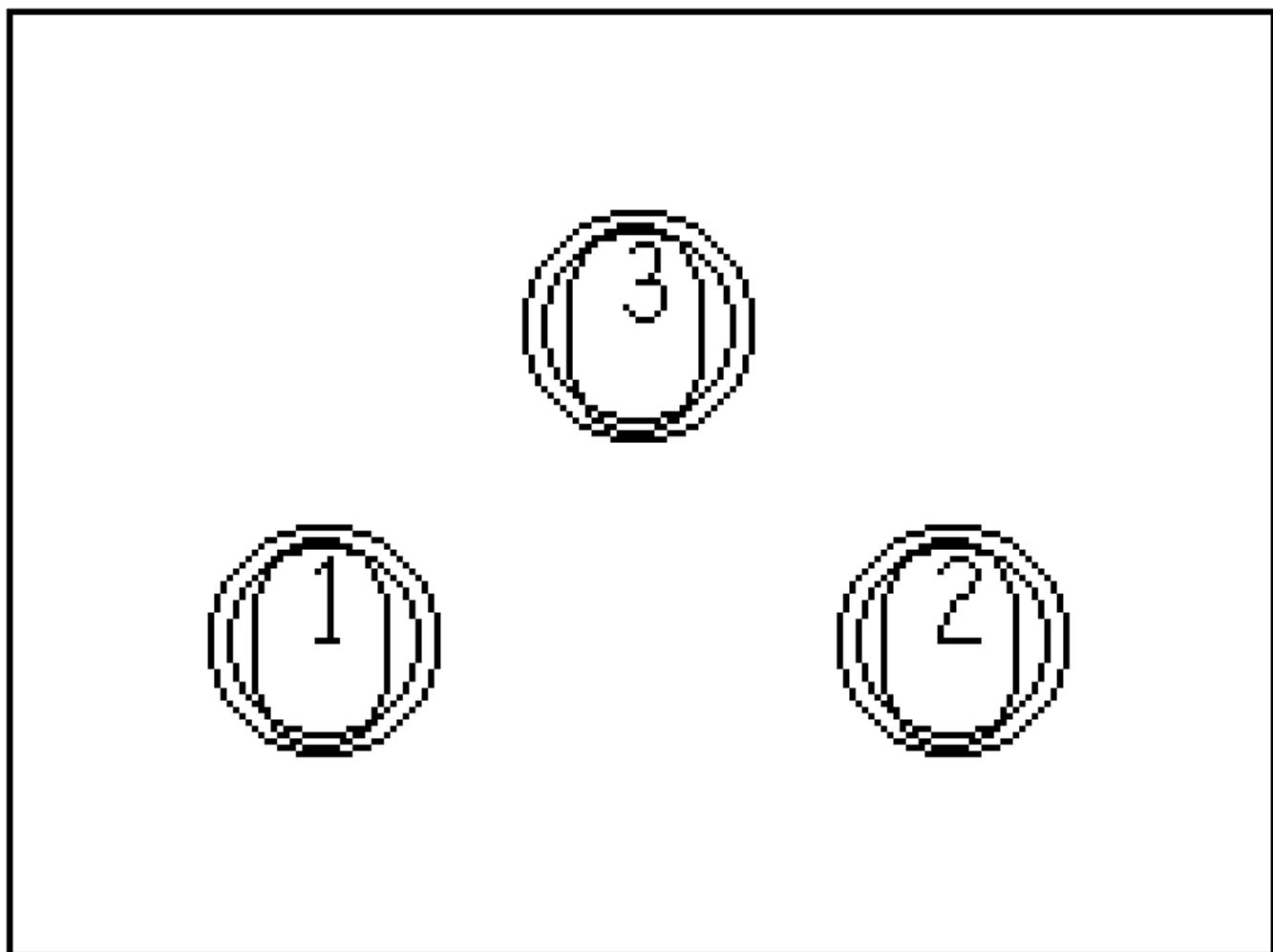
dio500



Potentiometer

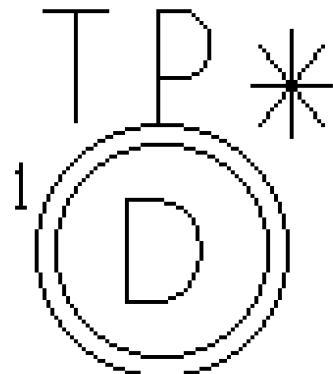
pot

R *



Test Point

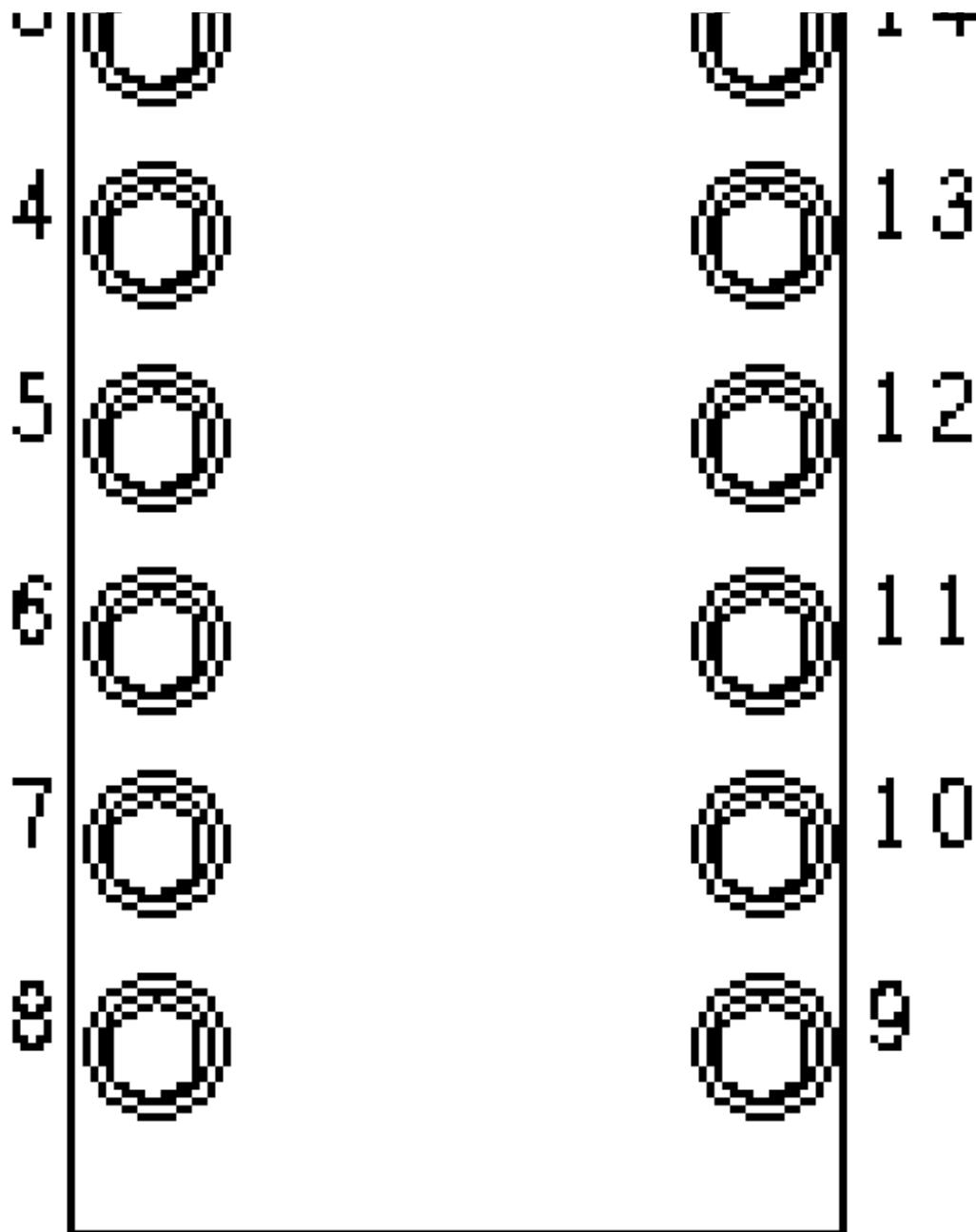
tp



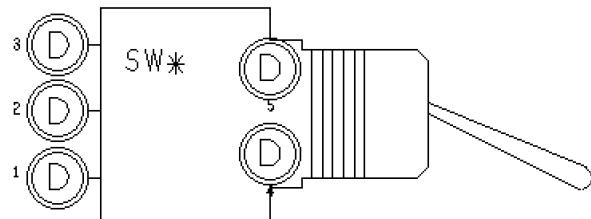
Switches

dipswitch



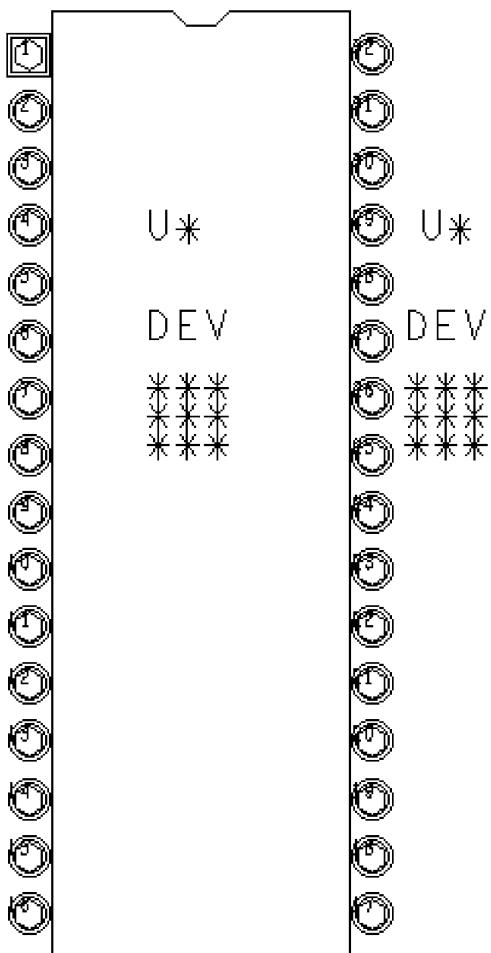


switch

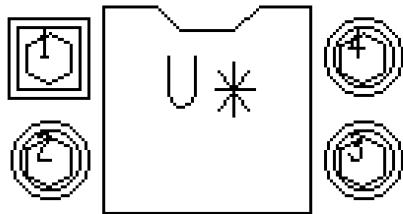


DIPs

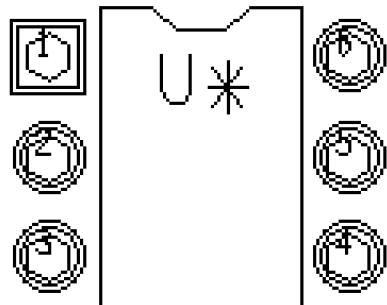
dip32_6



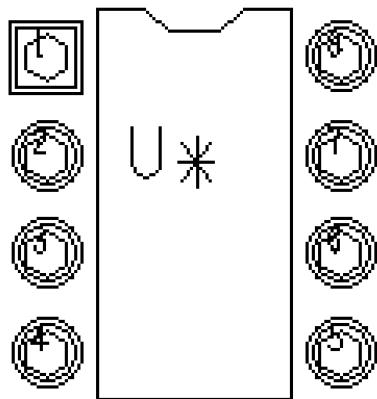
dip4_3



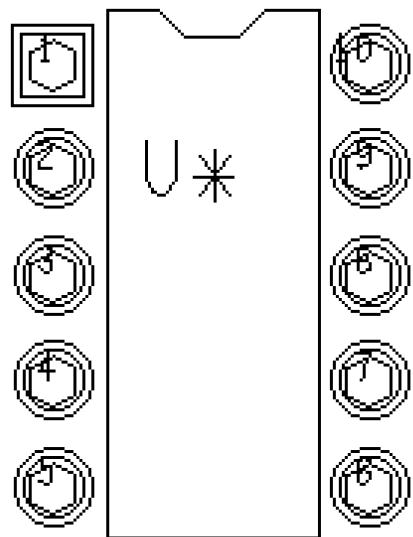
dip6_3



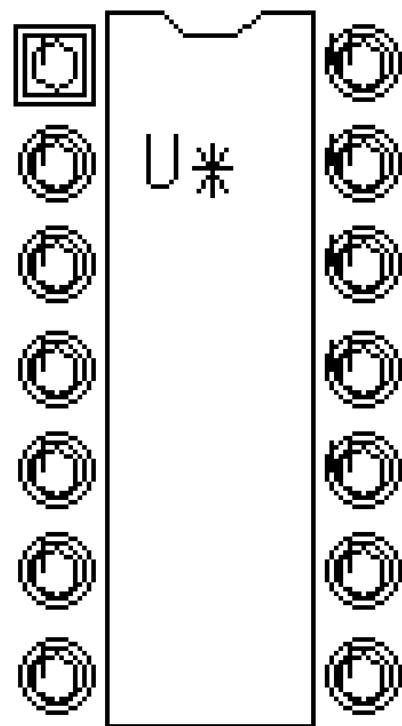
dip8_3



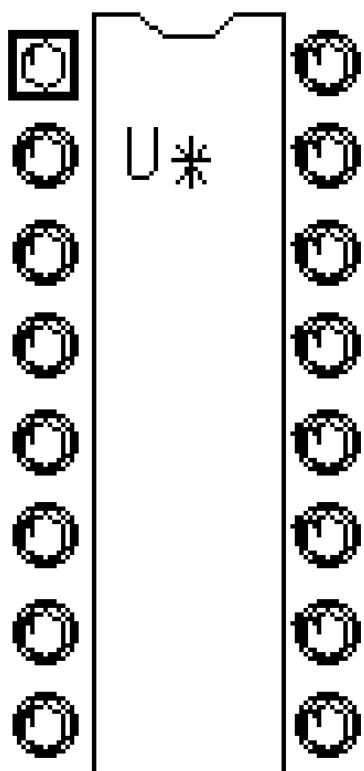
dip10_3



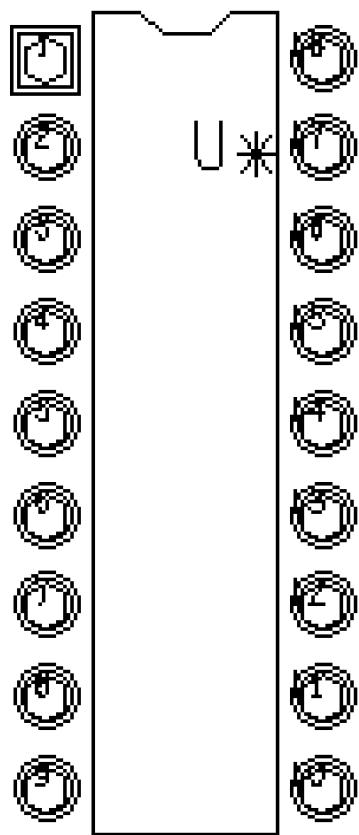
dip14_3



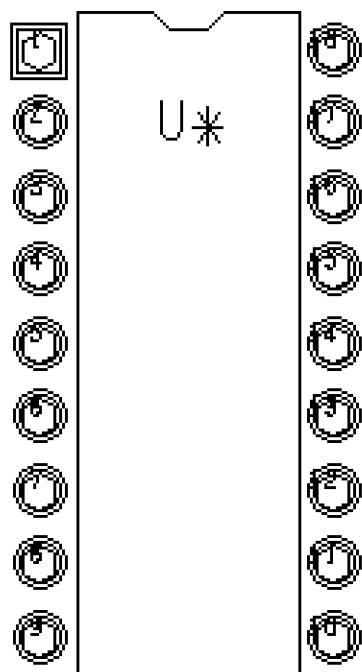
dip16_3



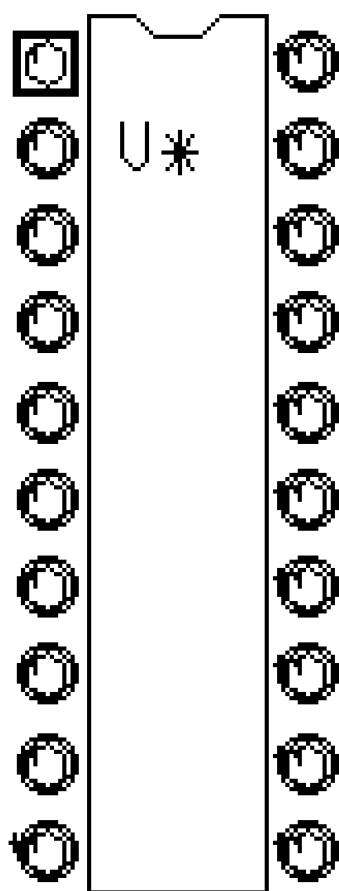
dip18_3



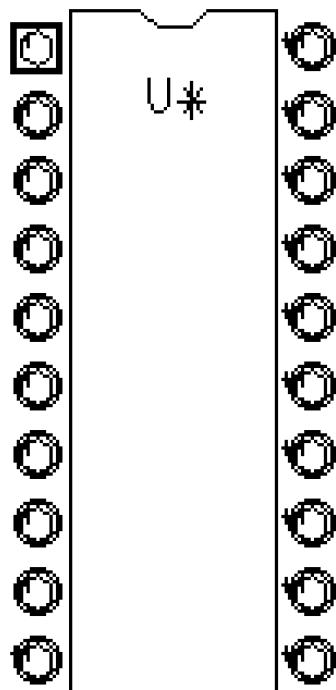
dip18_4



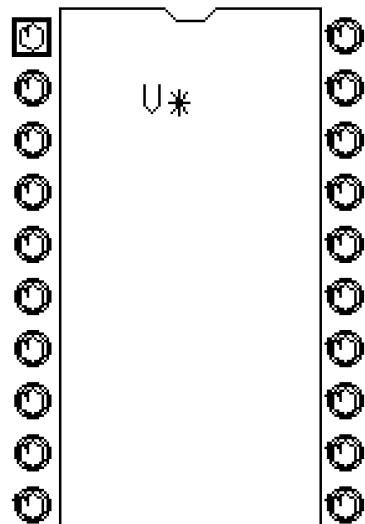
dip20_3



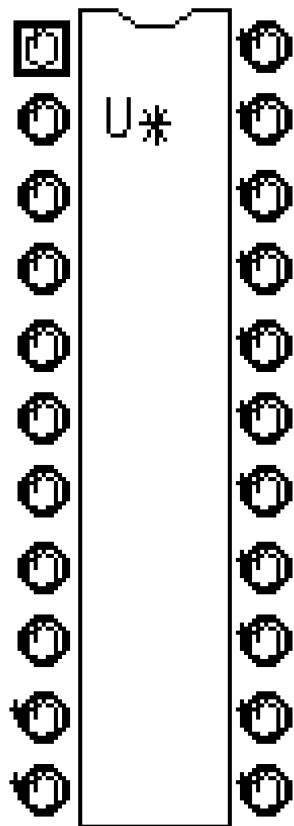
dip20_4



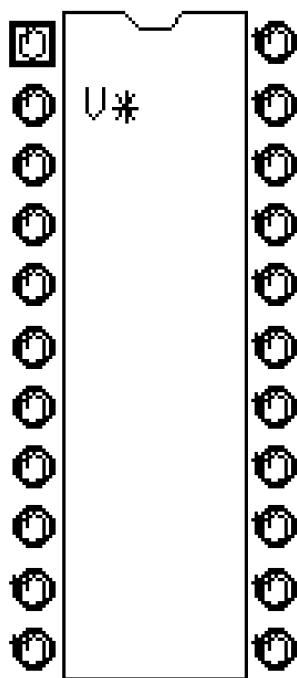
dip20_6



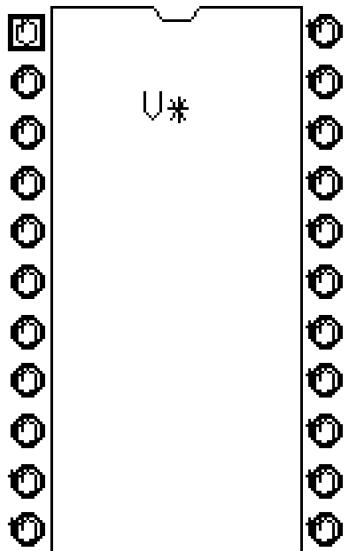
dip22_3



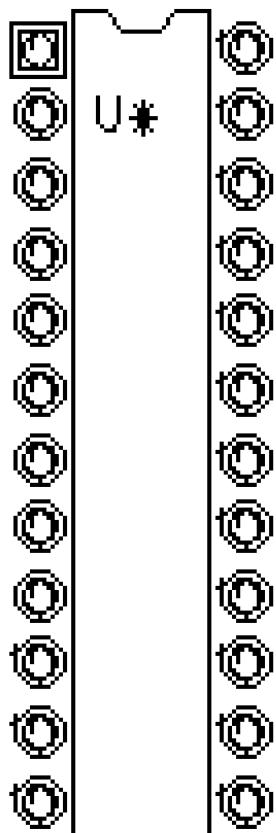
dip22_4



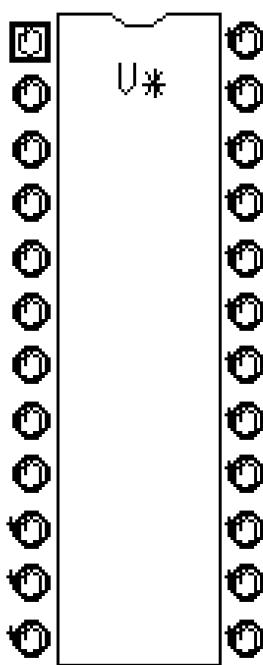
dip22_6



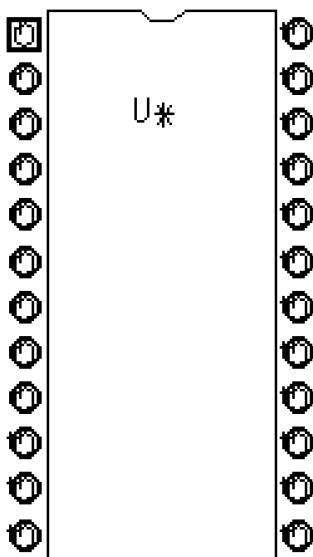
dip24_3



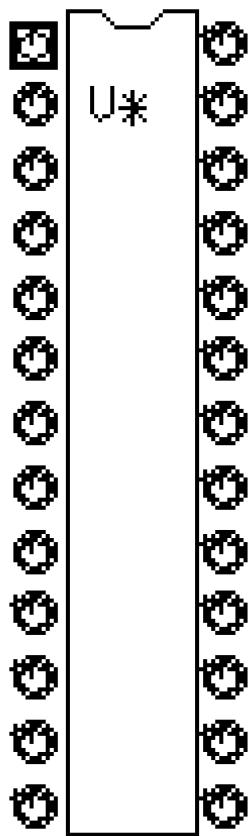
dip24_4



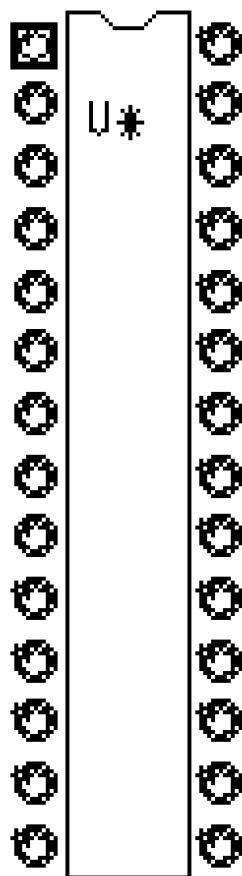
dip24_6



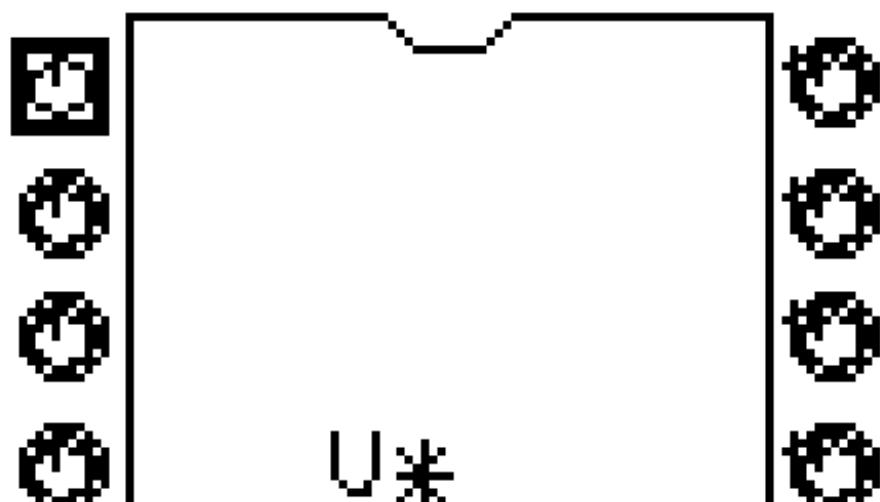
dip26_3

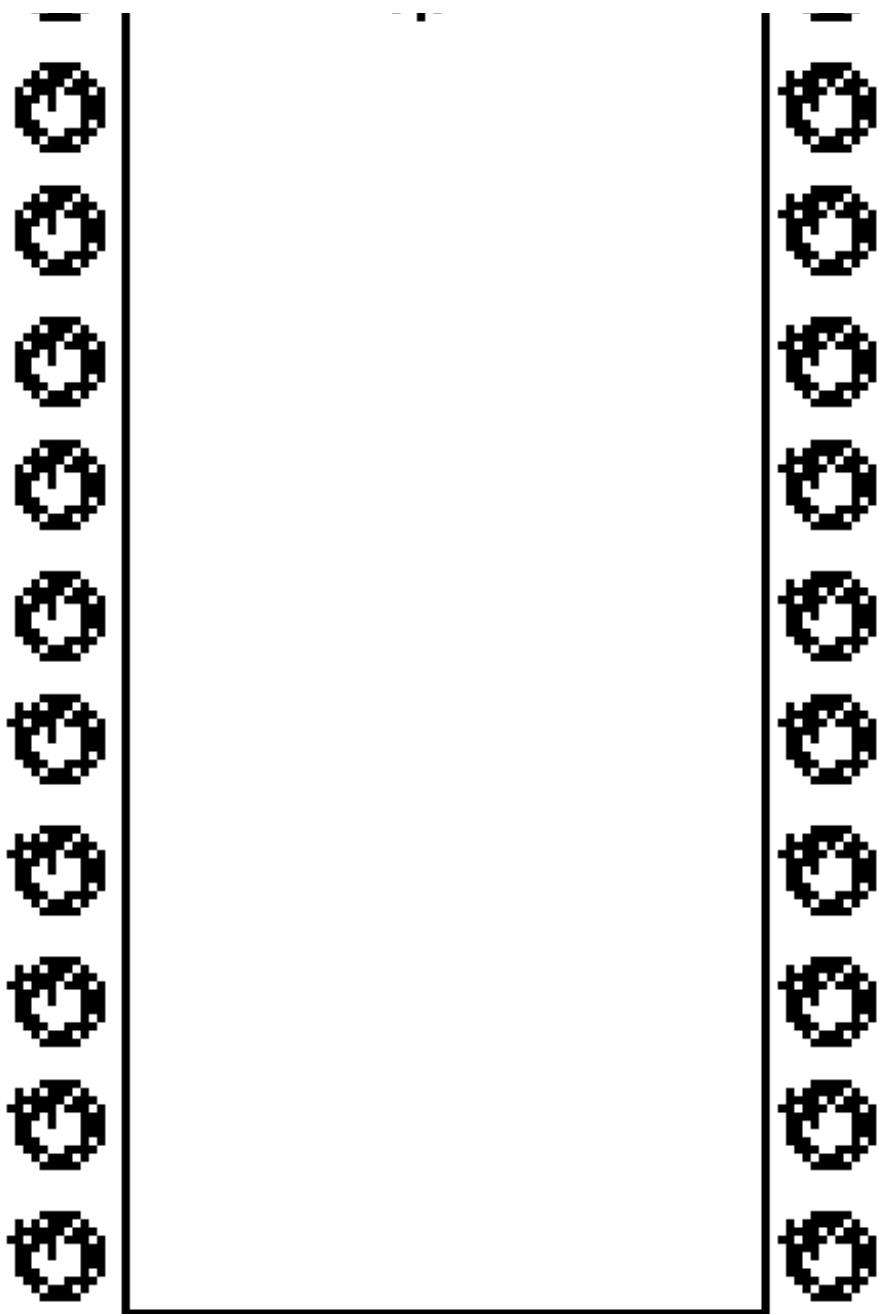


dip28_3

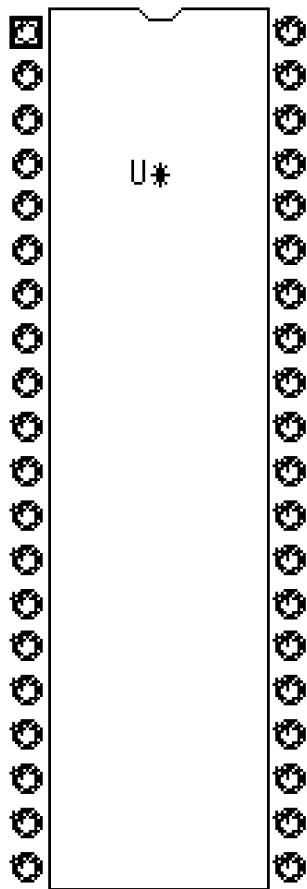


dip28_6

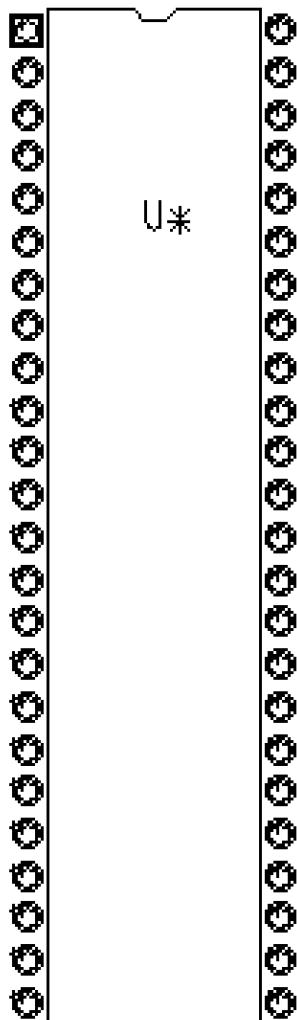




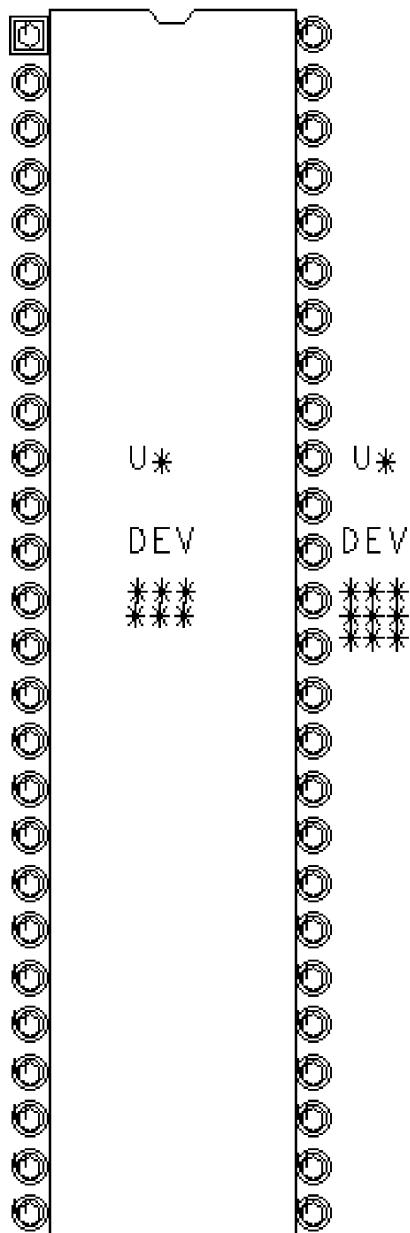
dip40_6



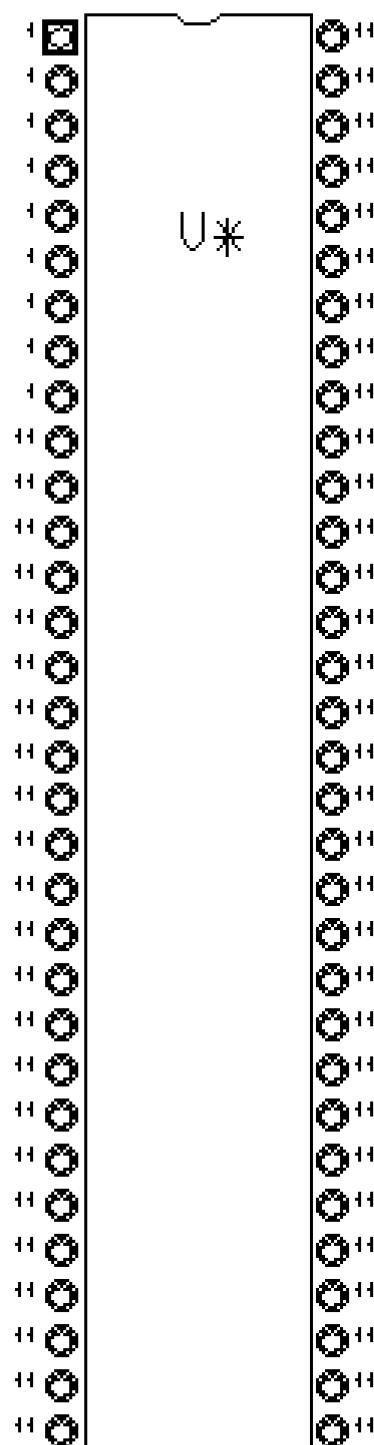
dip48_6



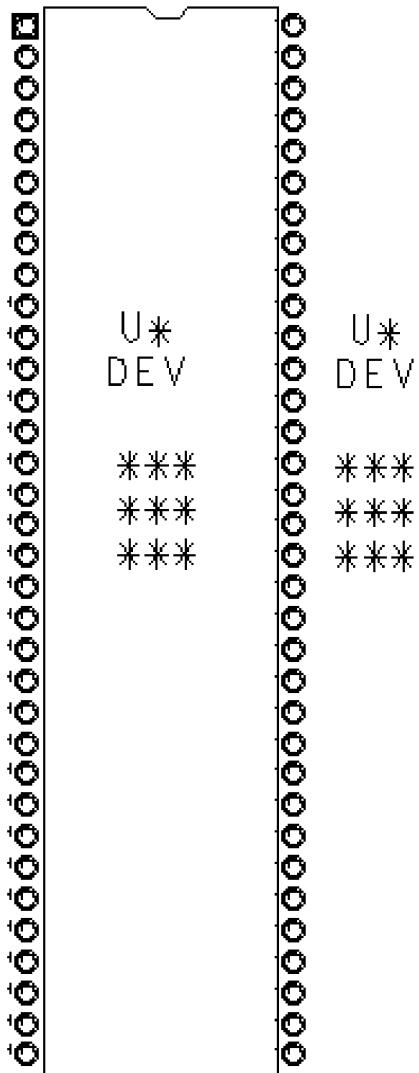
dip52_6



dip64_6

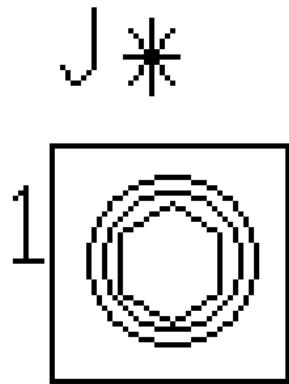


dip68_6

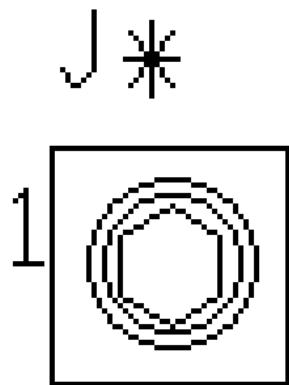


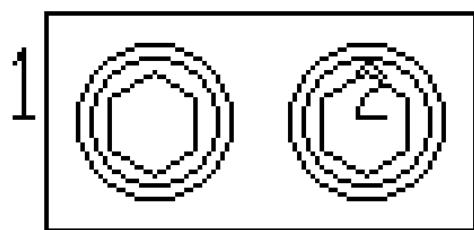
Jumpers

jumper1



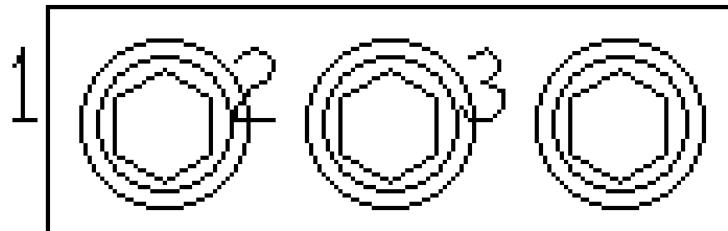
jumper2





J *

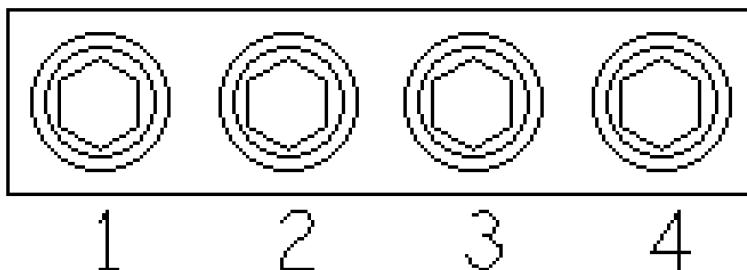
jumper3



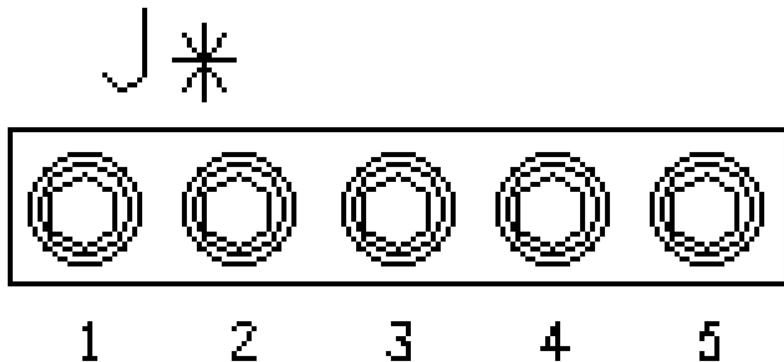
J *

jumper4

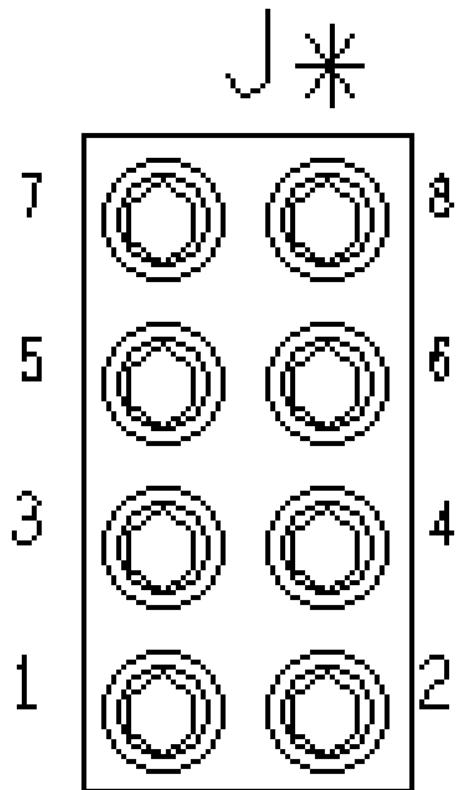
J *



jumper5

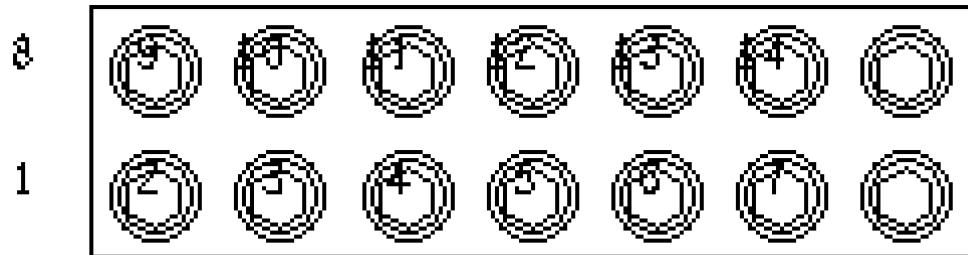


jumper8



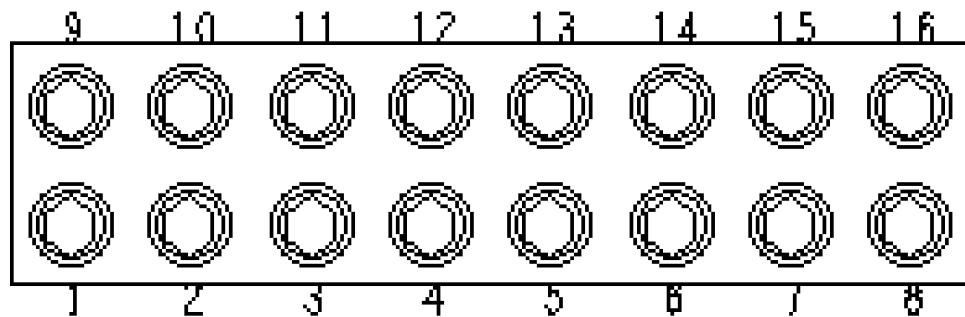
jumper14

J *



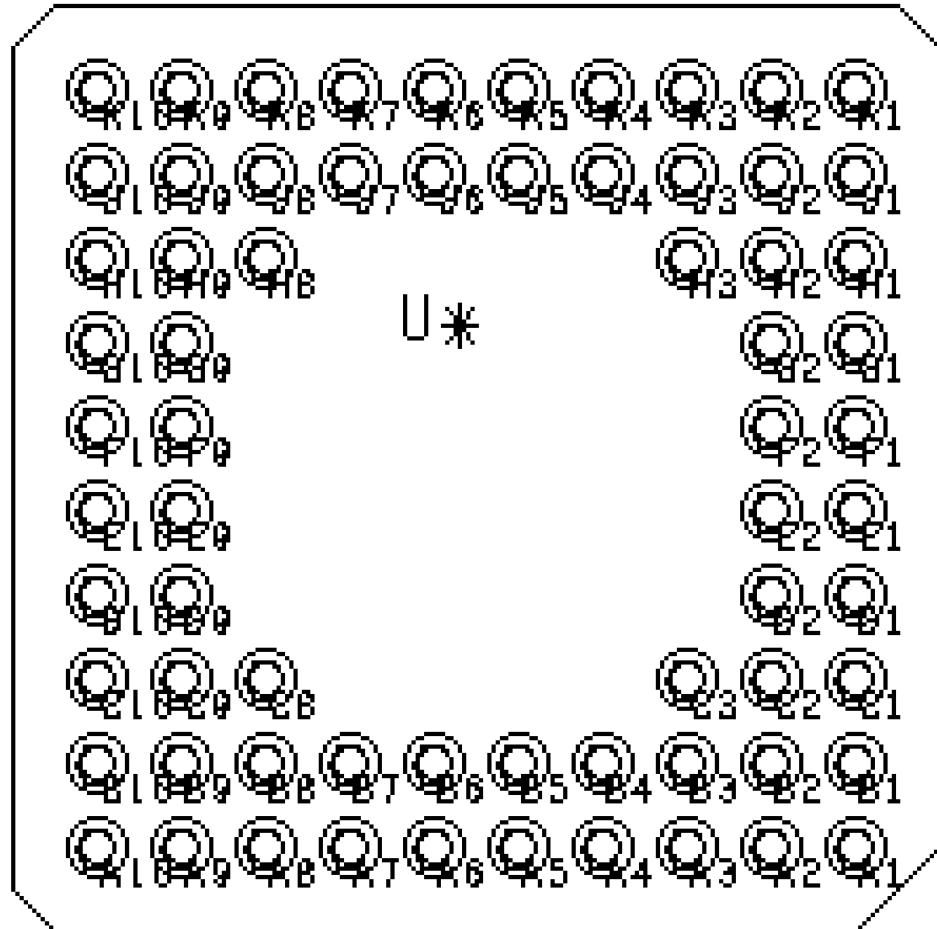
jumper16

J *

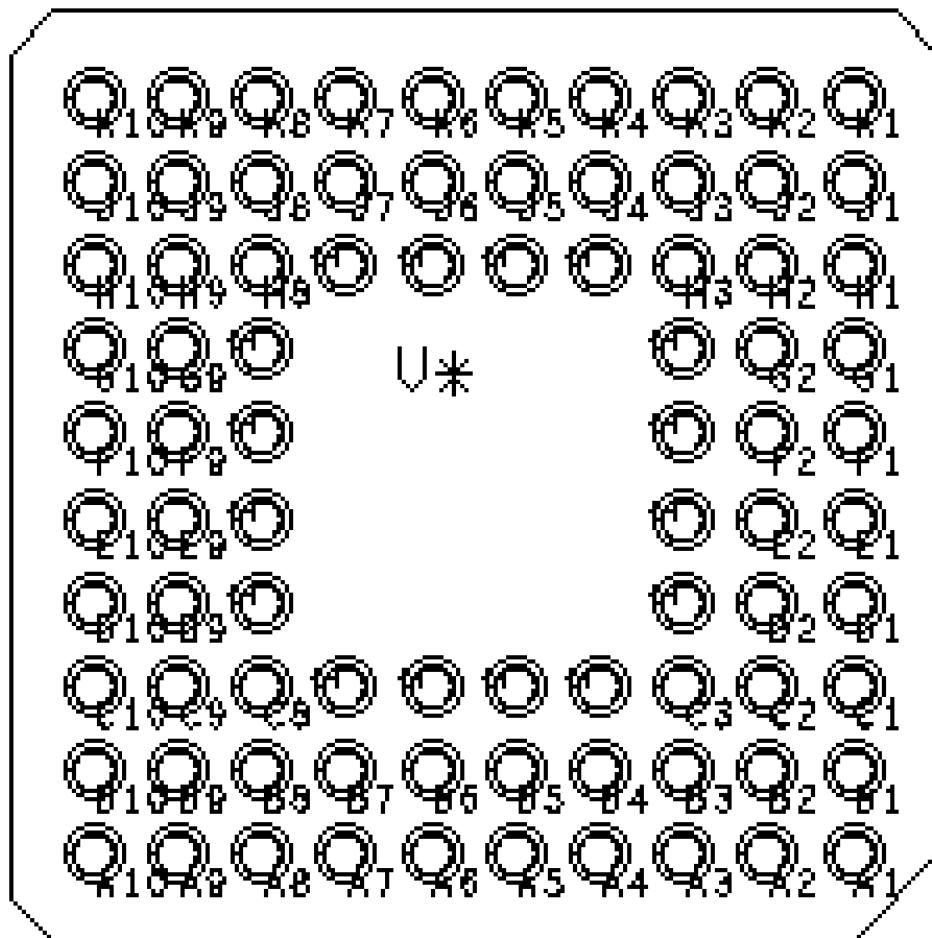


Pin Grid Arrays

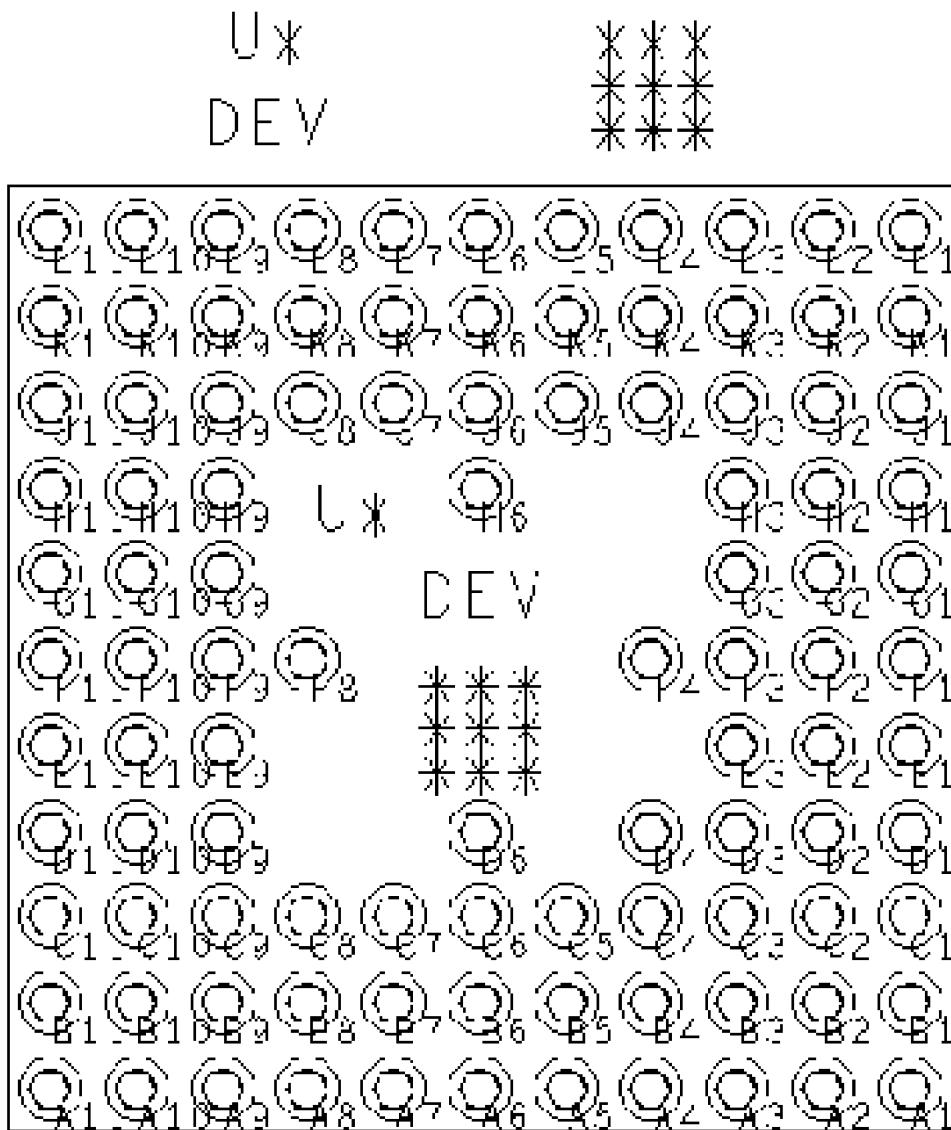
pga68



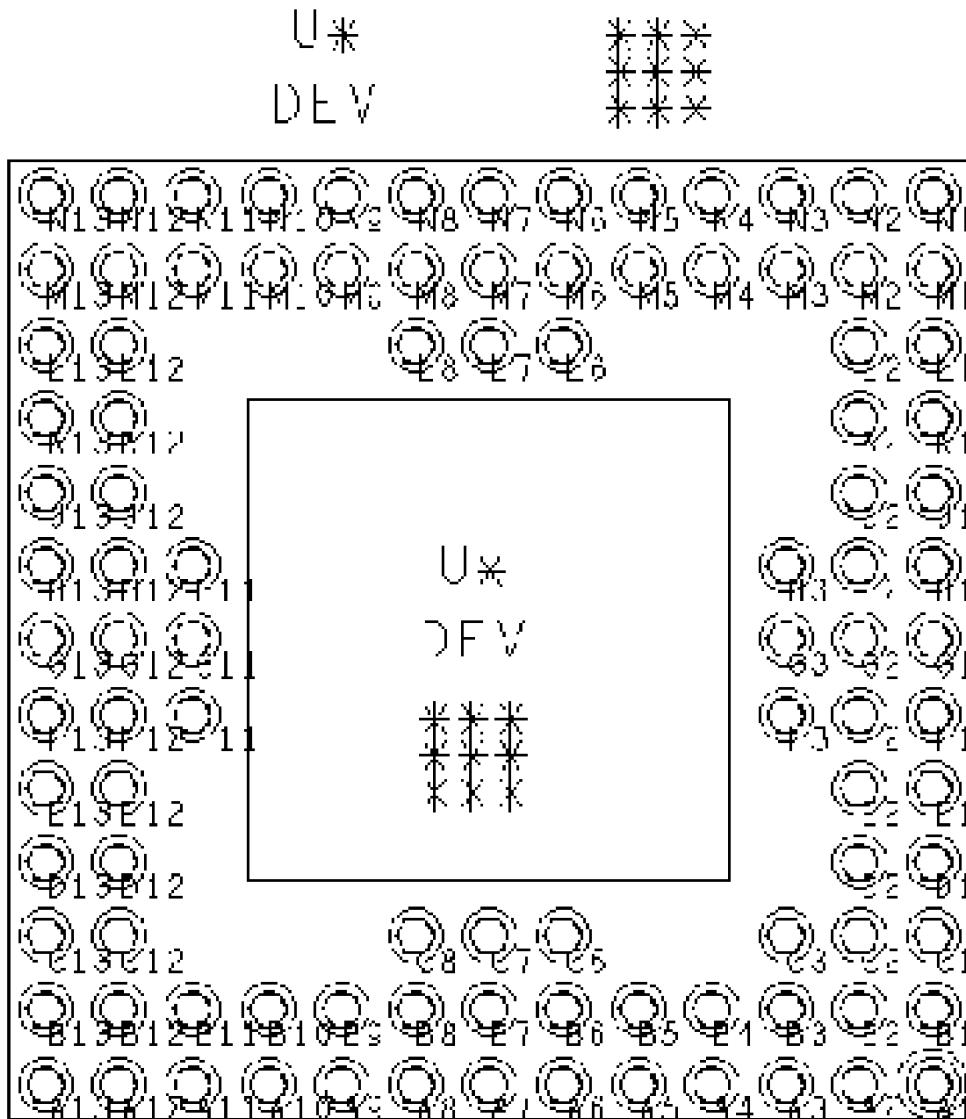
pga84



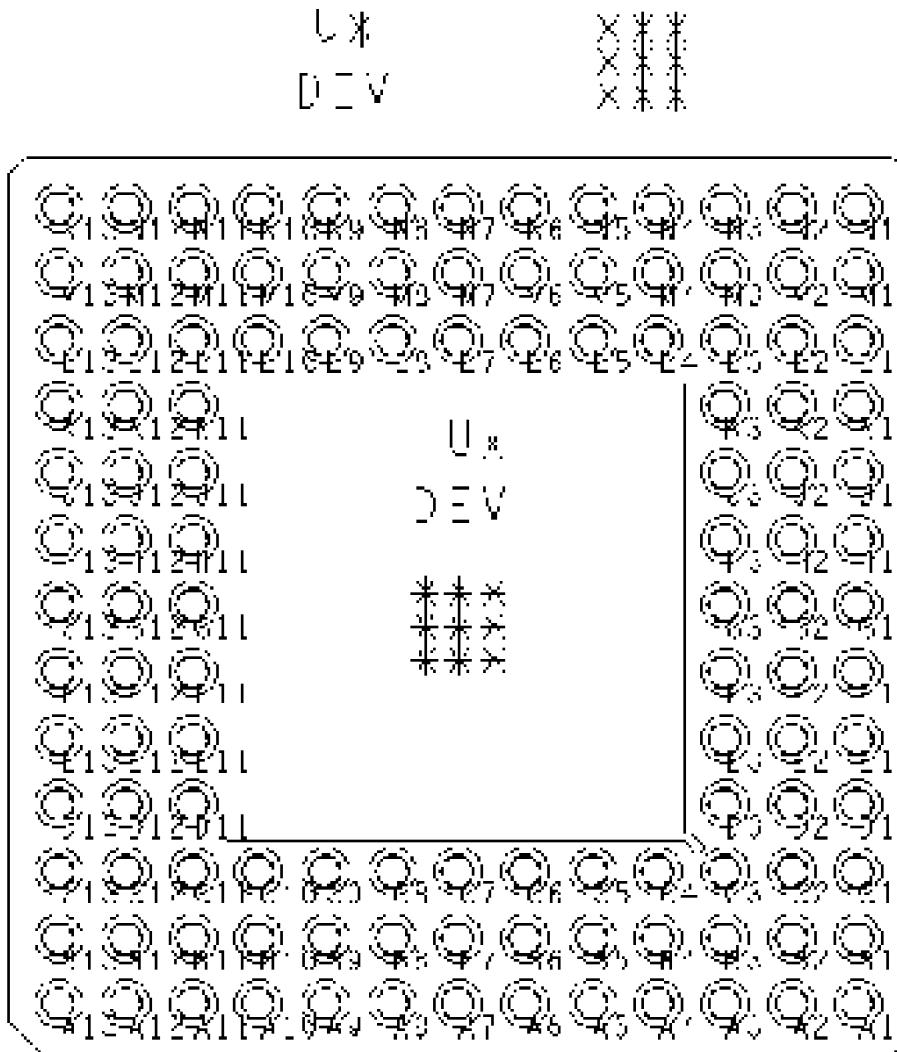
pga100



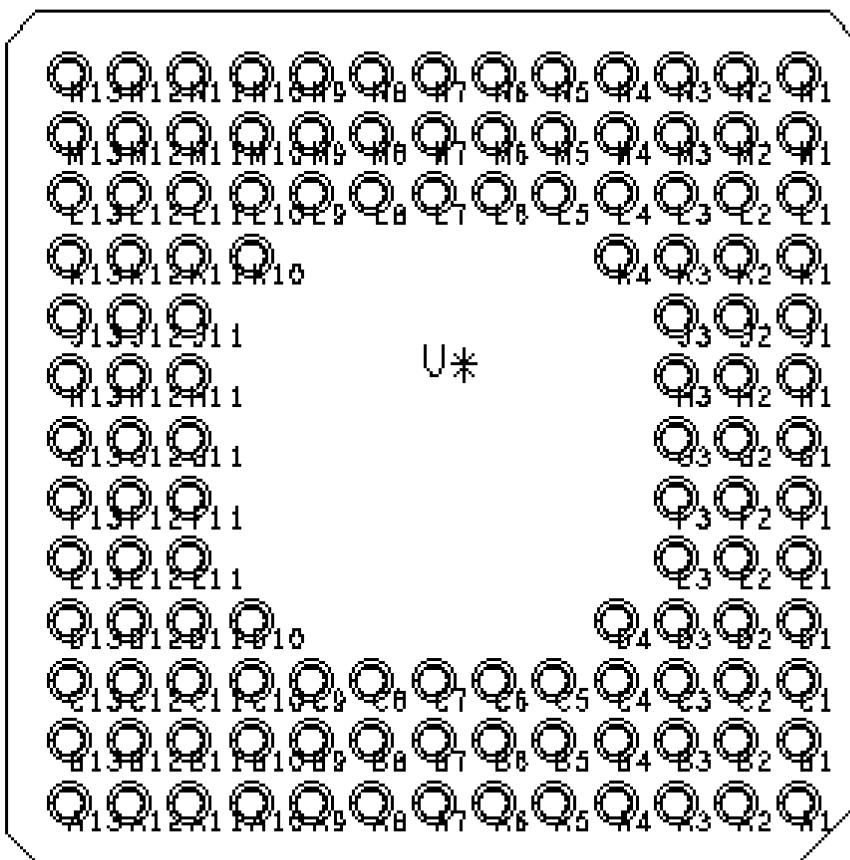
pga101



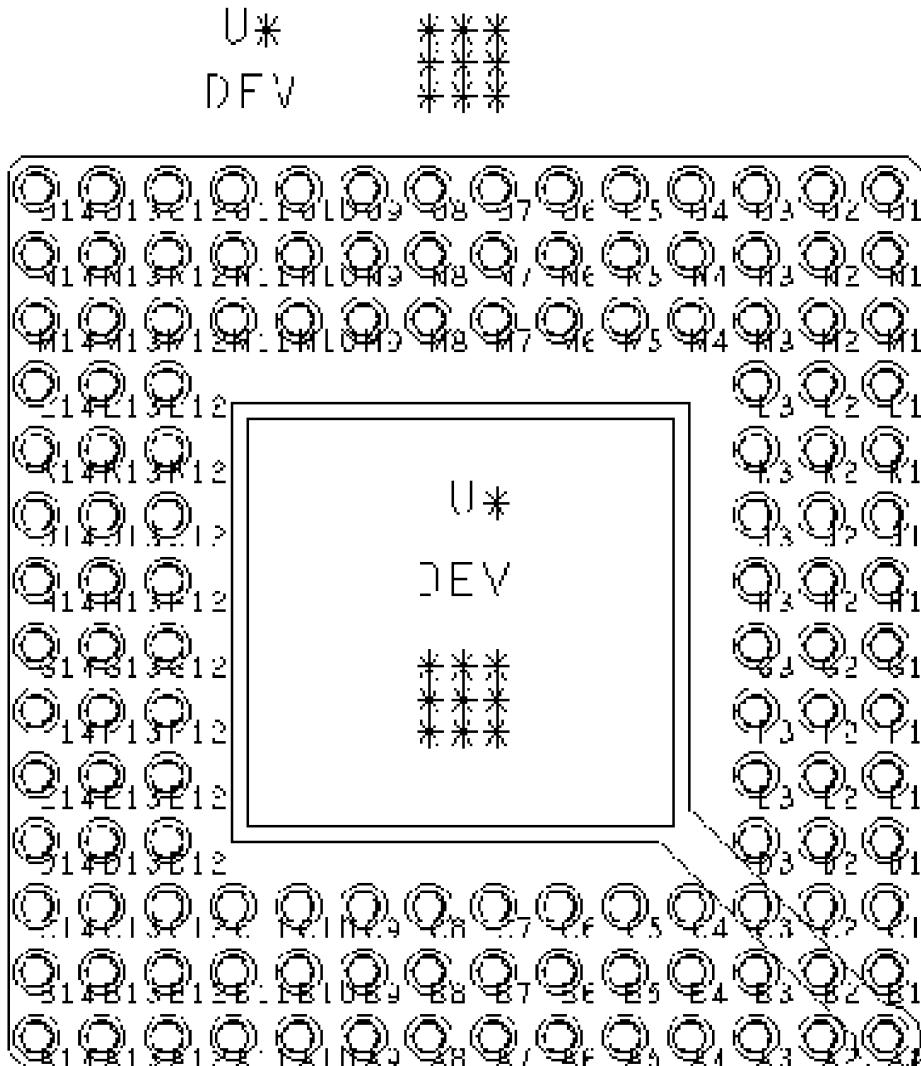
pga120



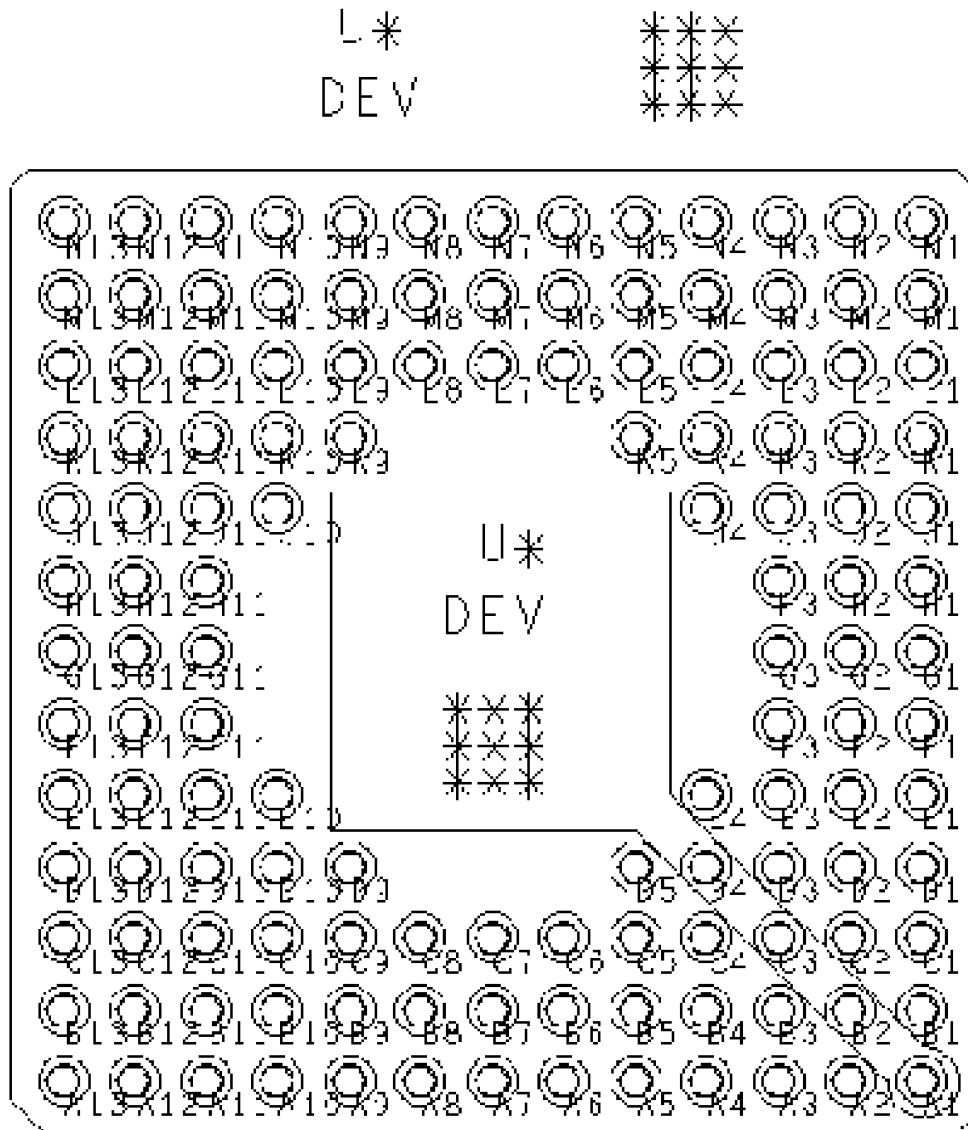
pga124



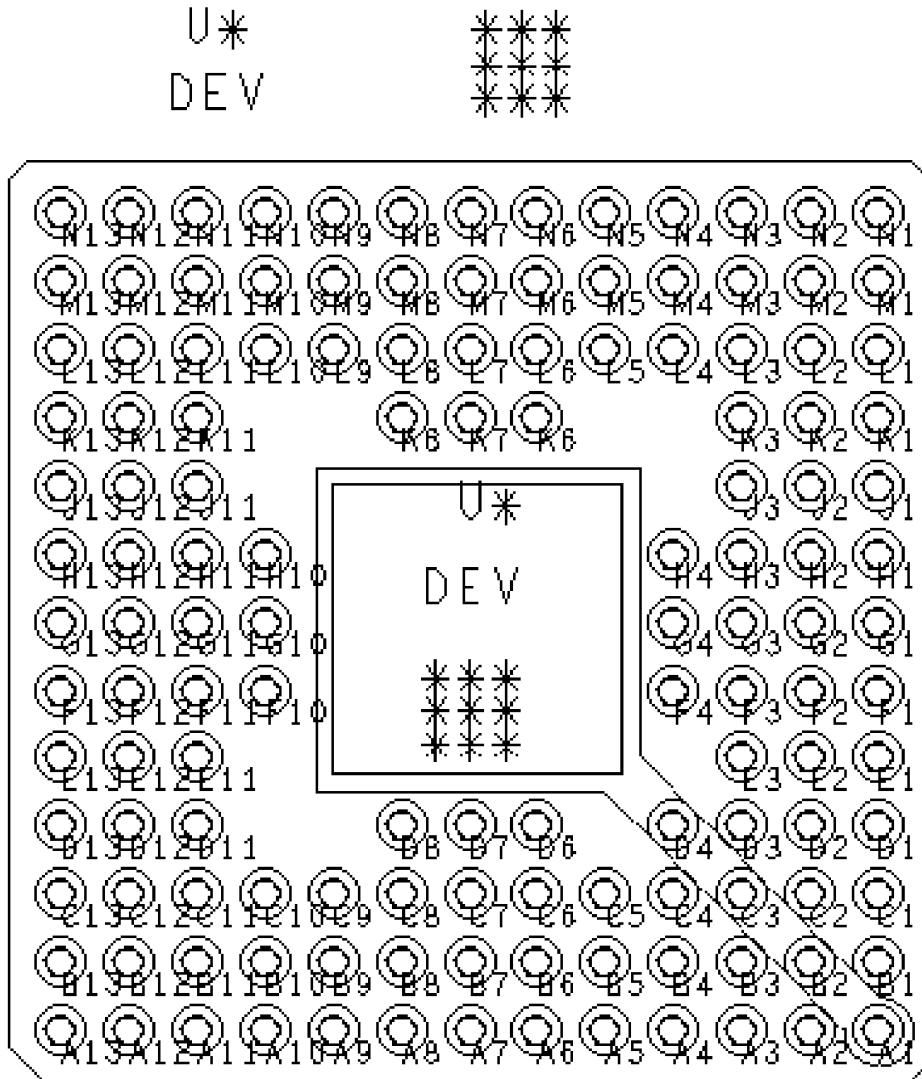
pga132



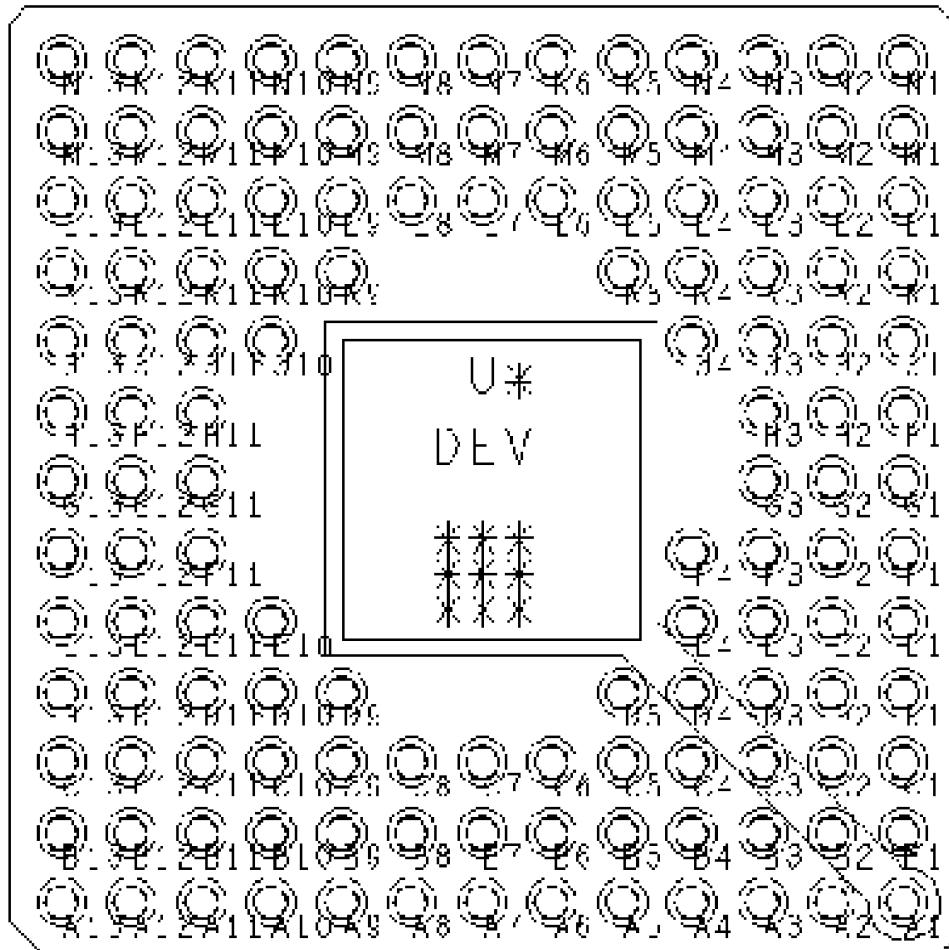
pga132_ci



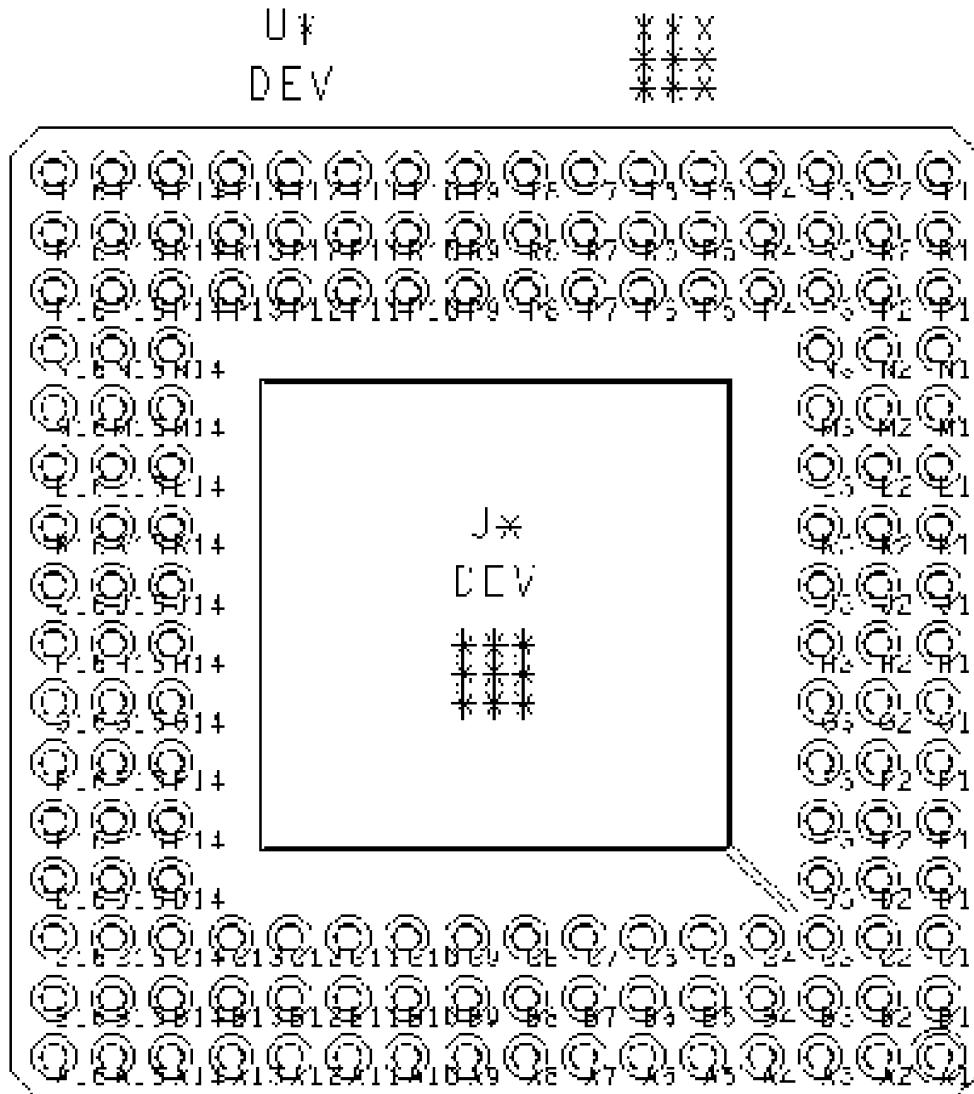
pga132-x



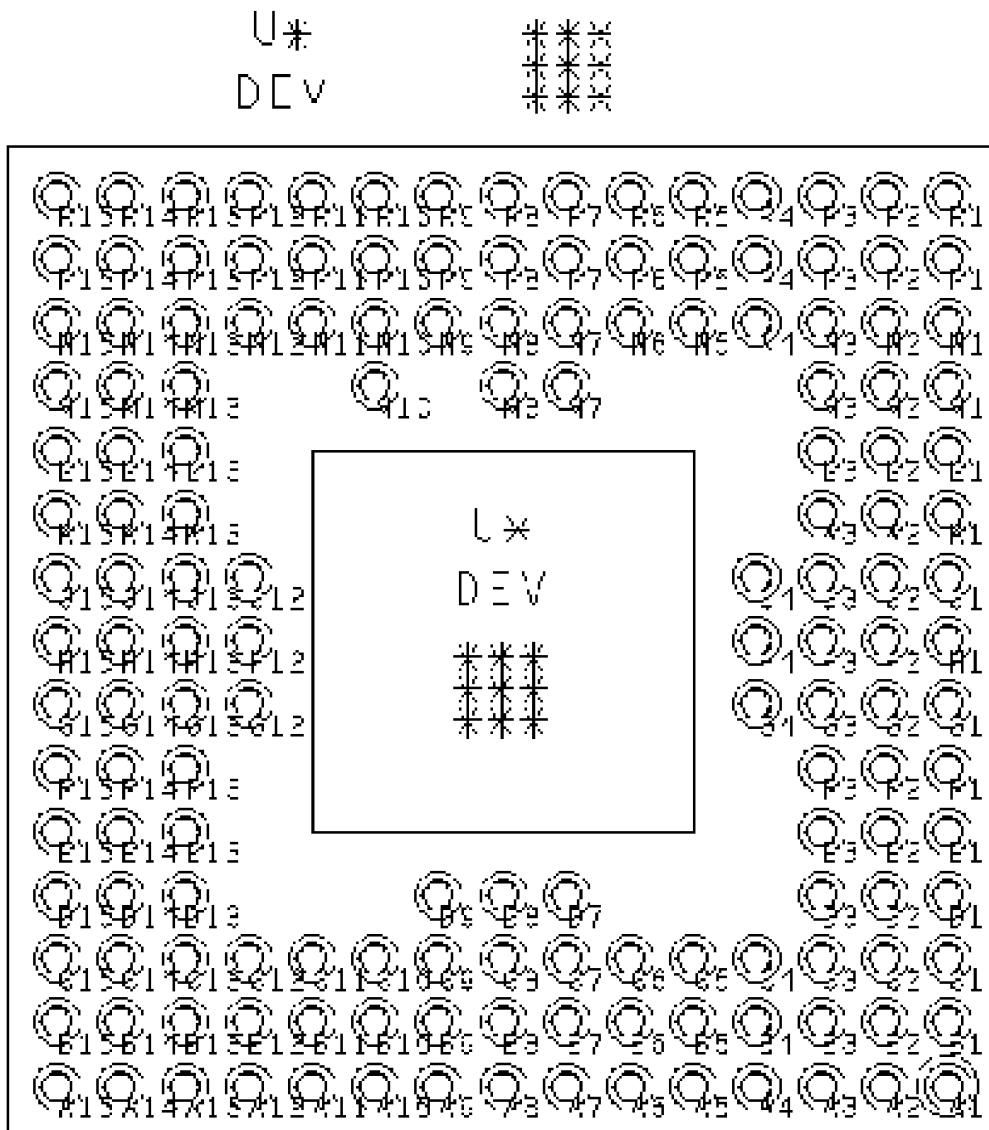
pga133



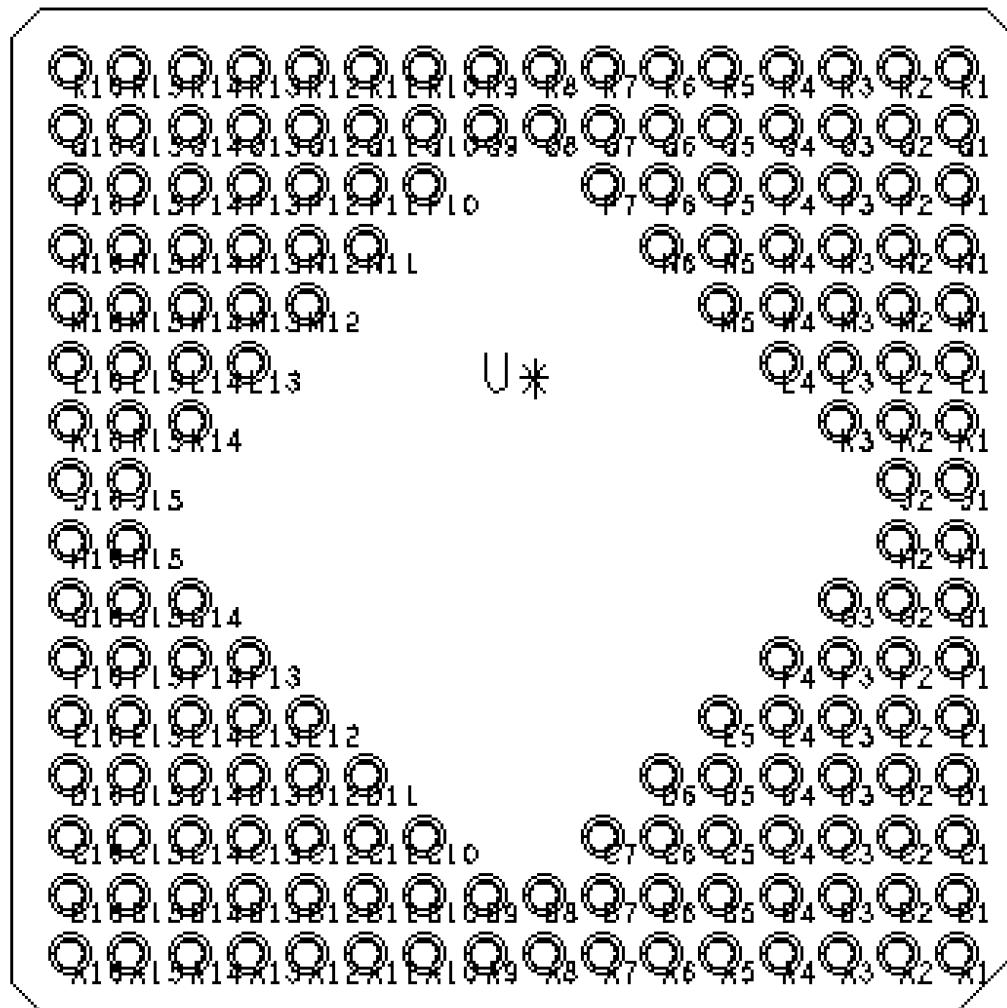
pga156



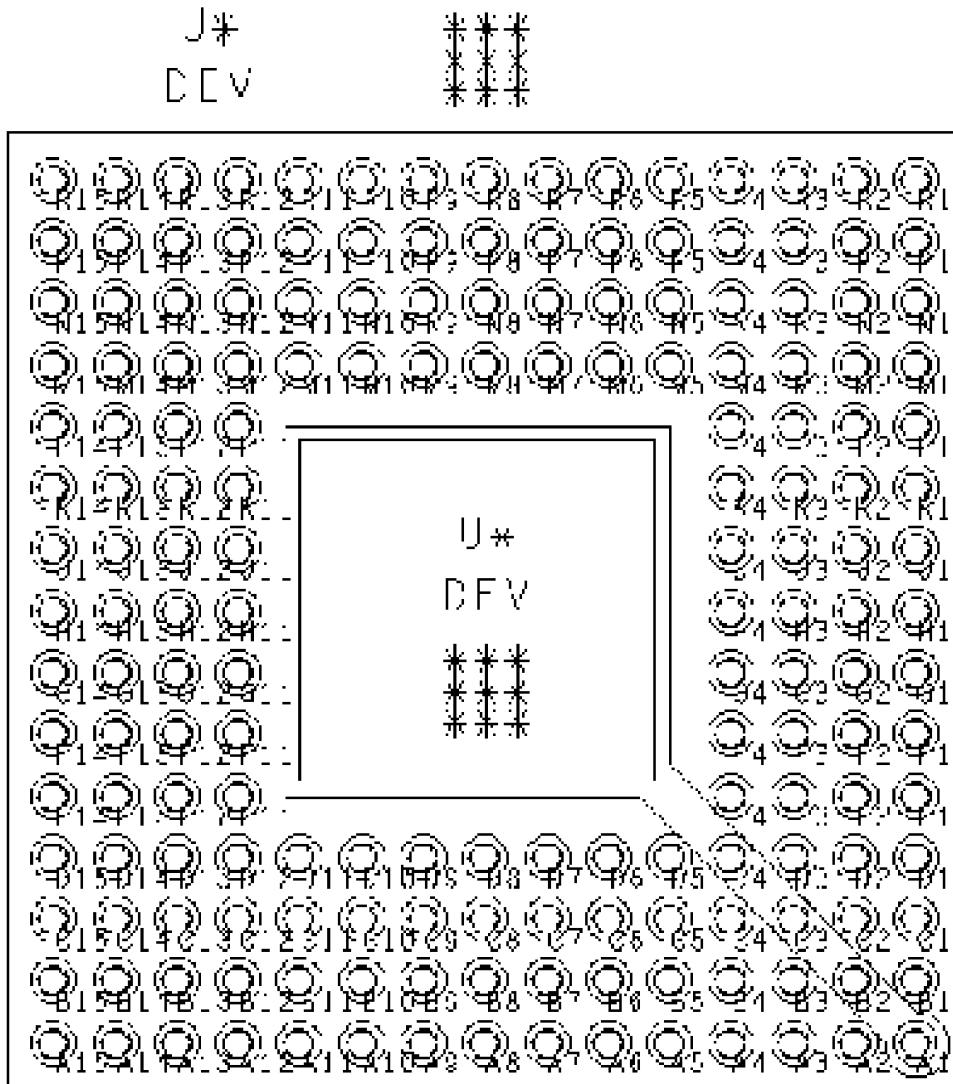
pga156_x



pga172

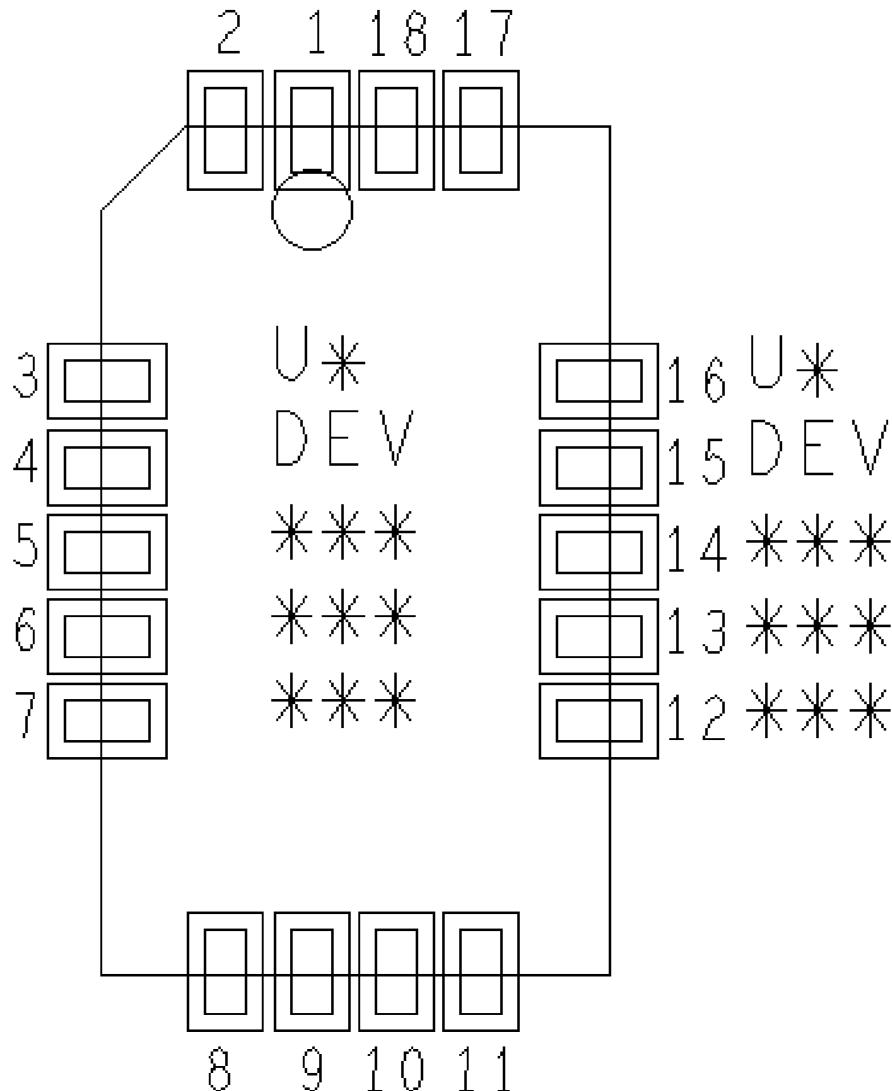


pga176

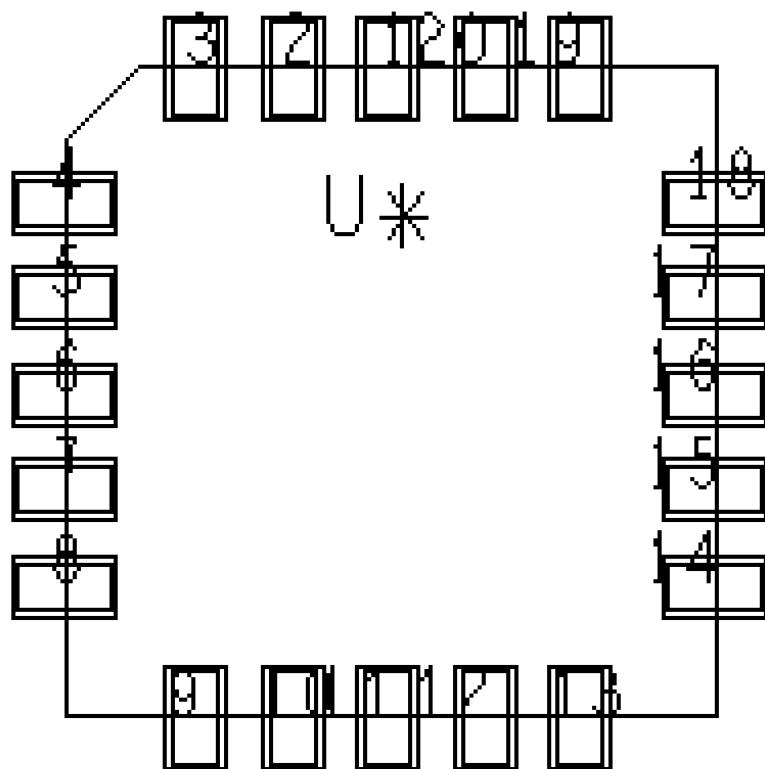


PLCCs

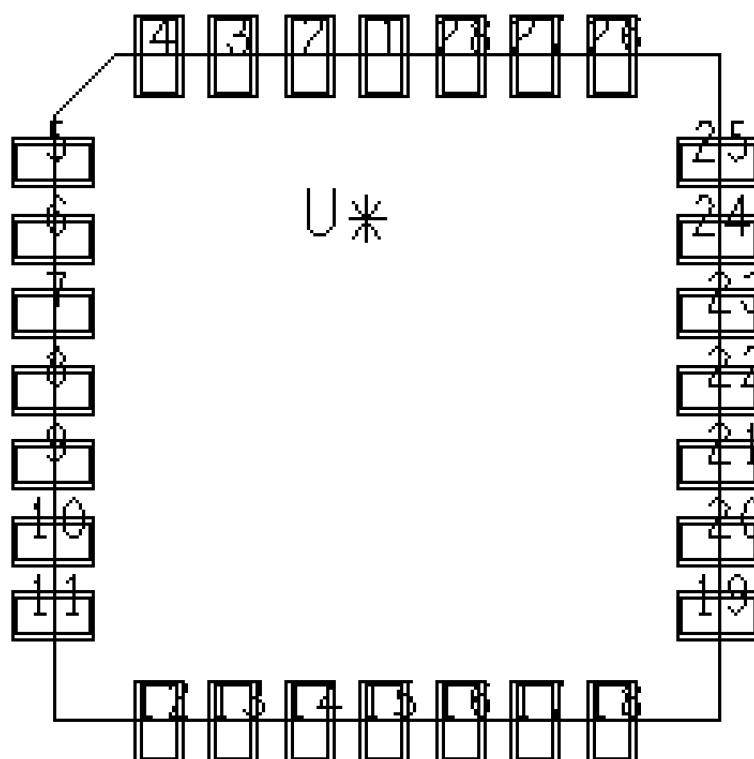
picc18



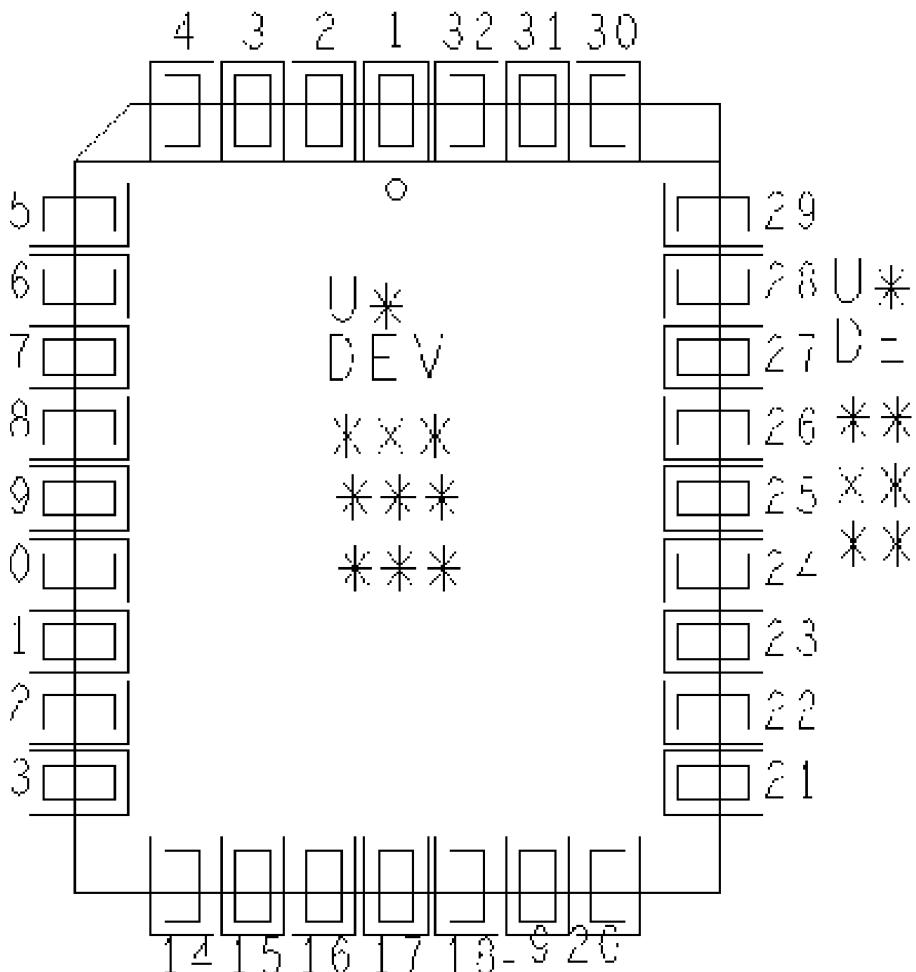
plcc20



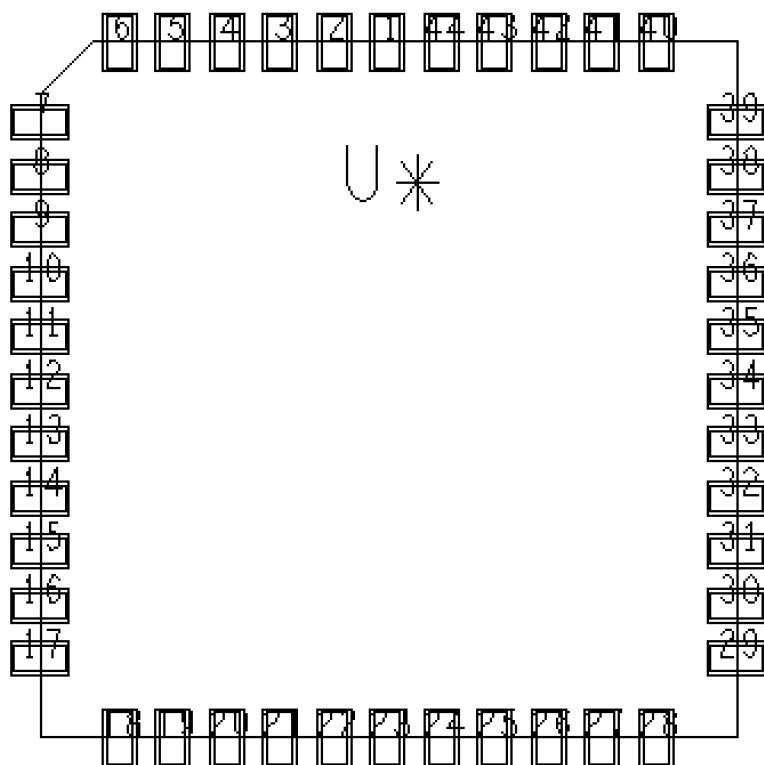
plcc28



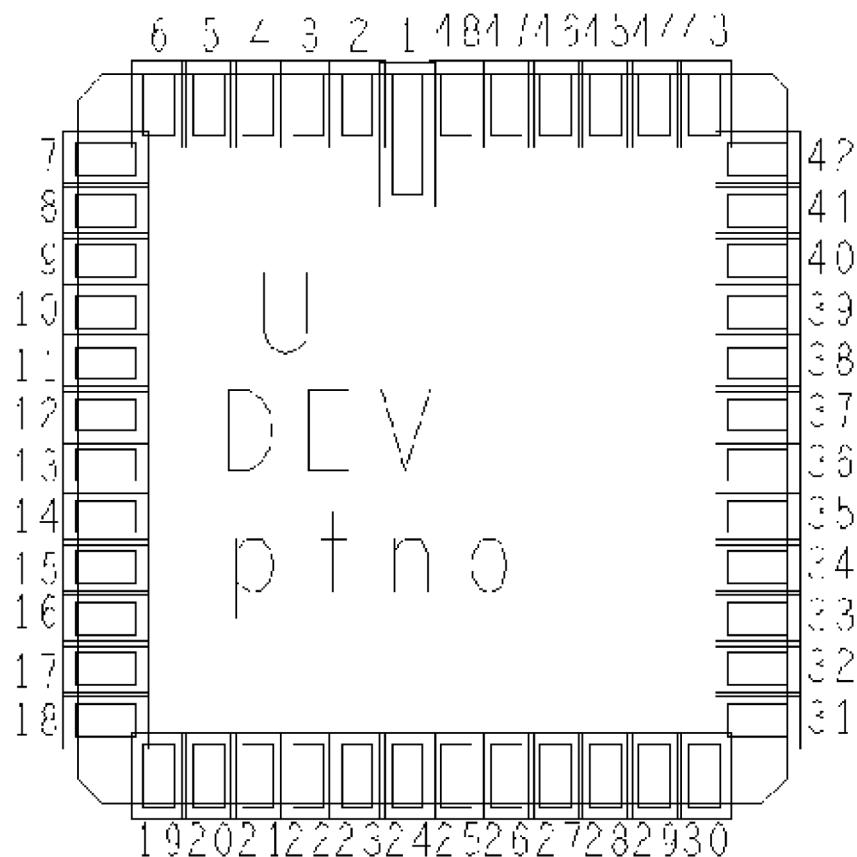
picc32



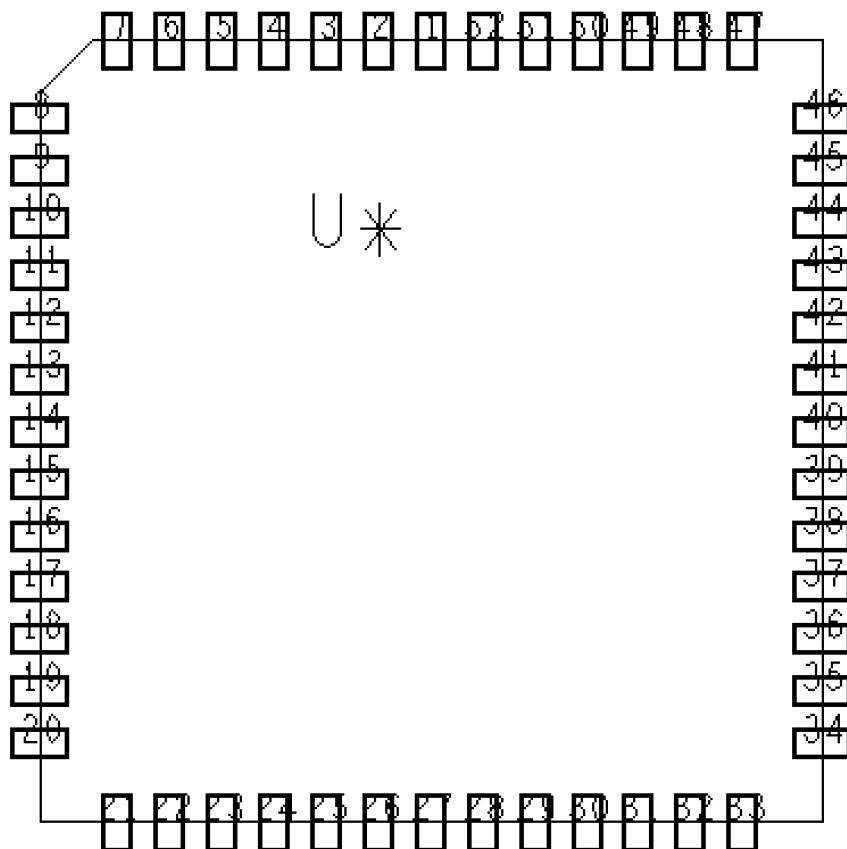
plcc44



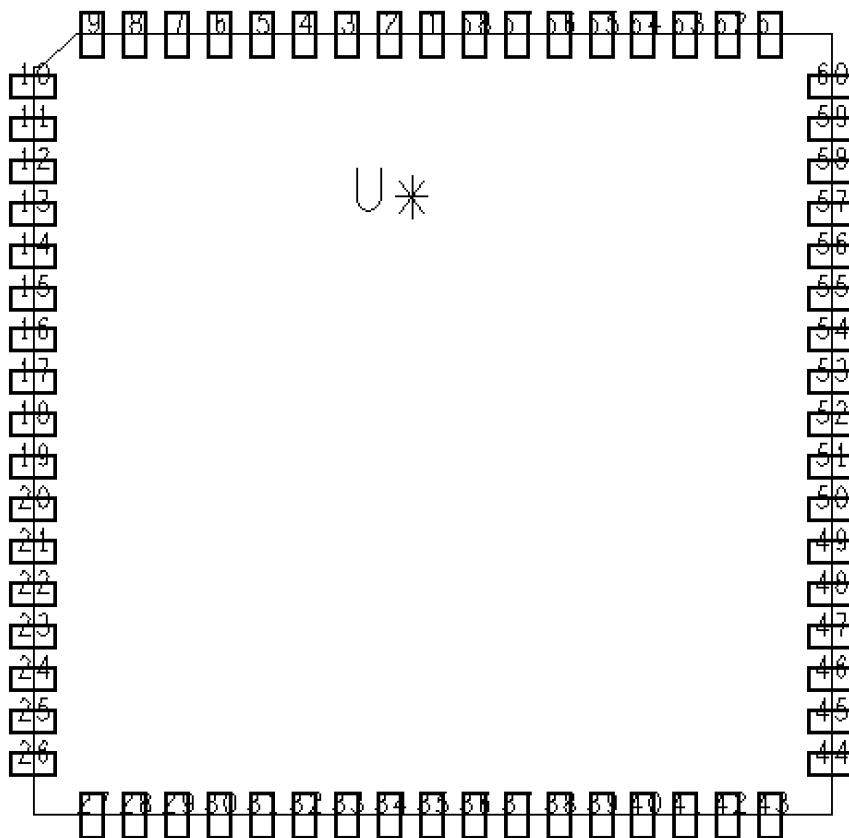
plcc48



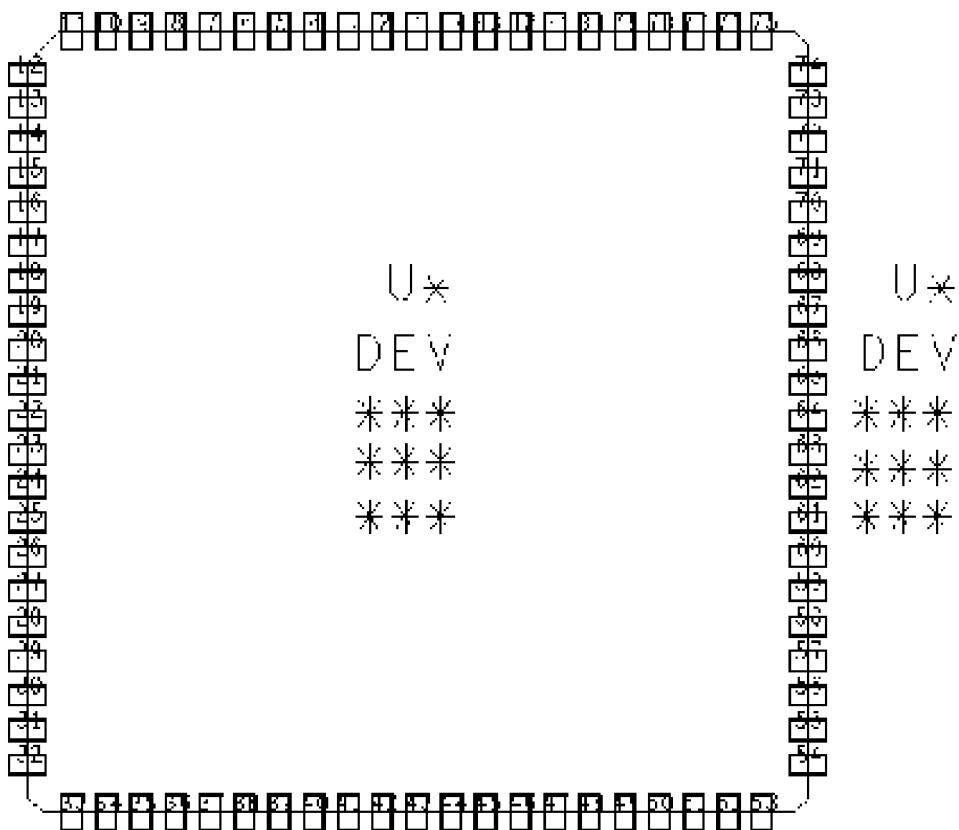
picc52



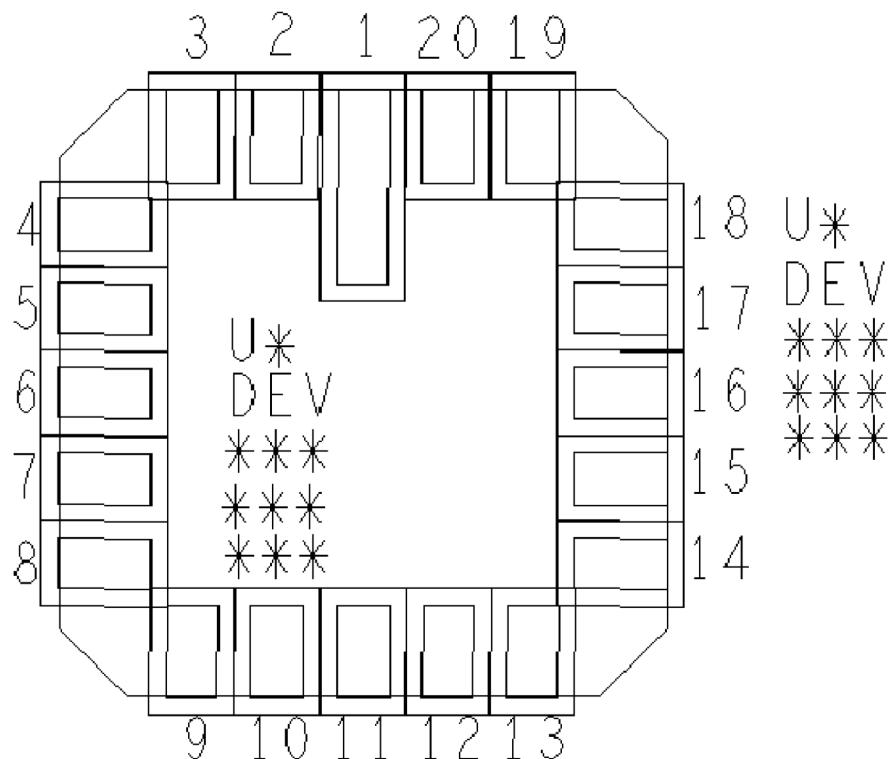
picc68



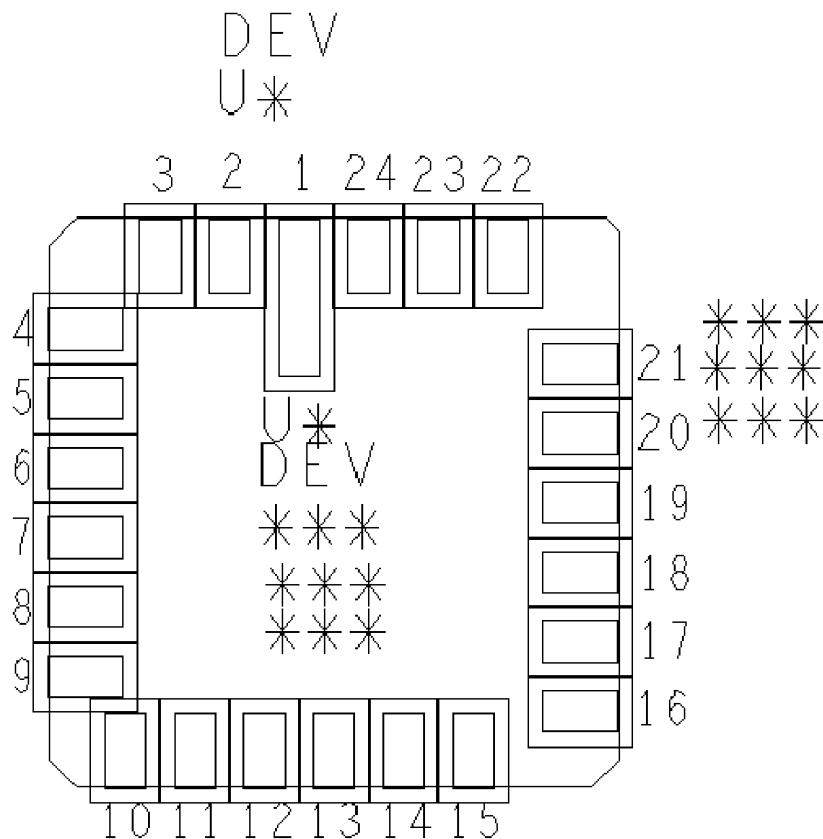
picc84



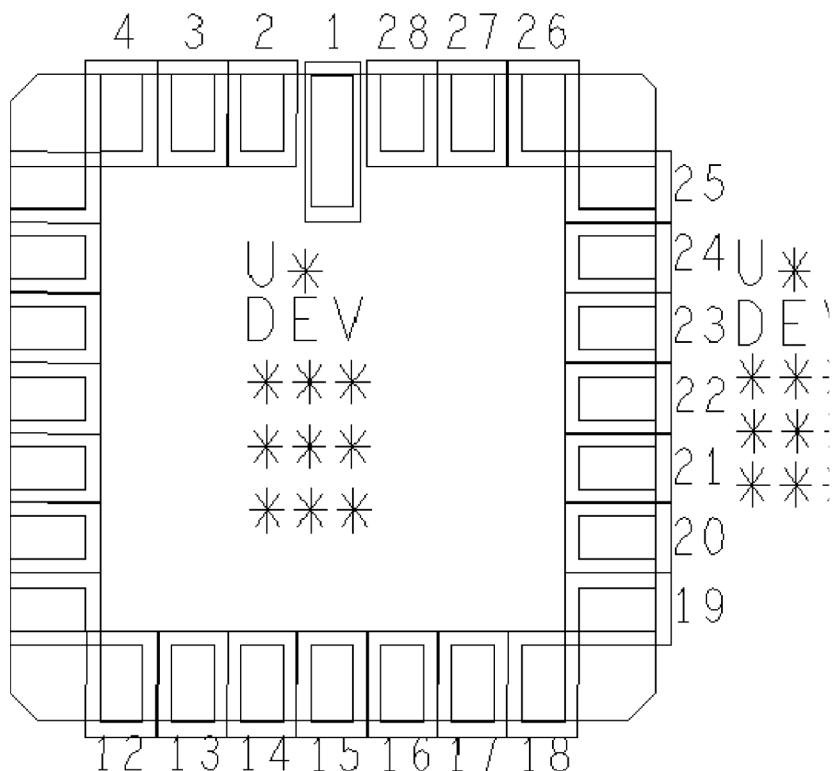
icc20



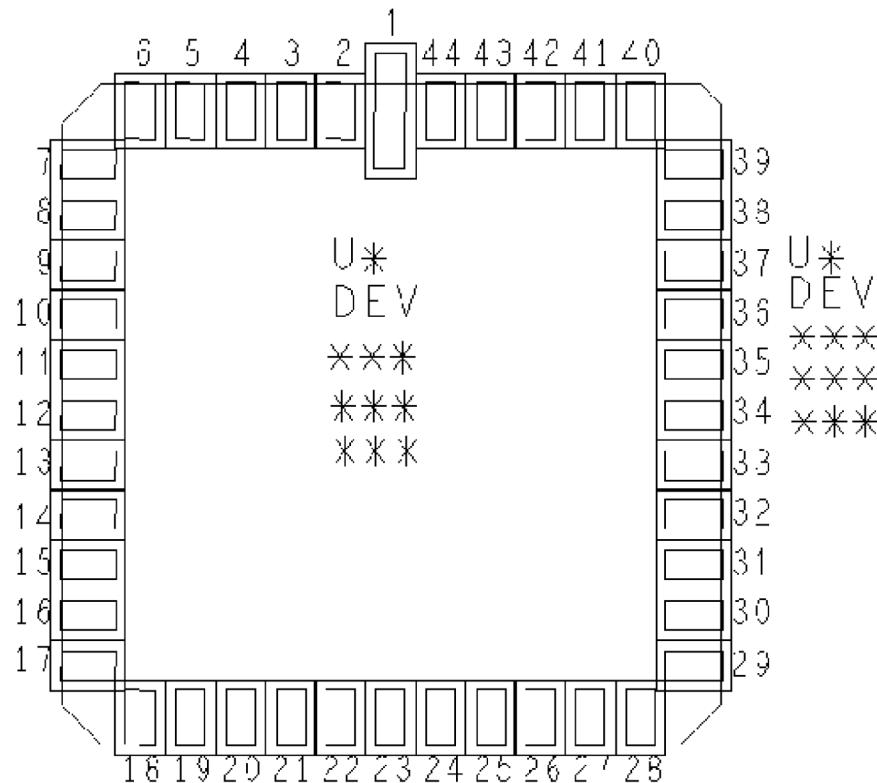
icc24



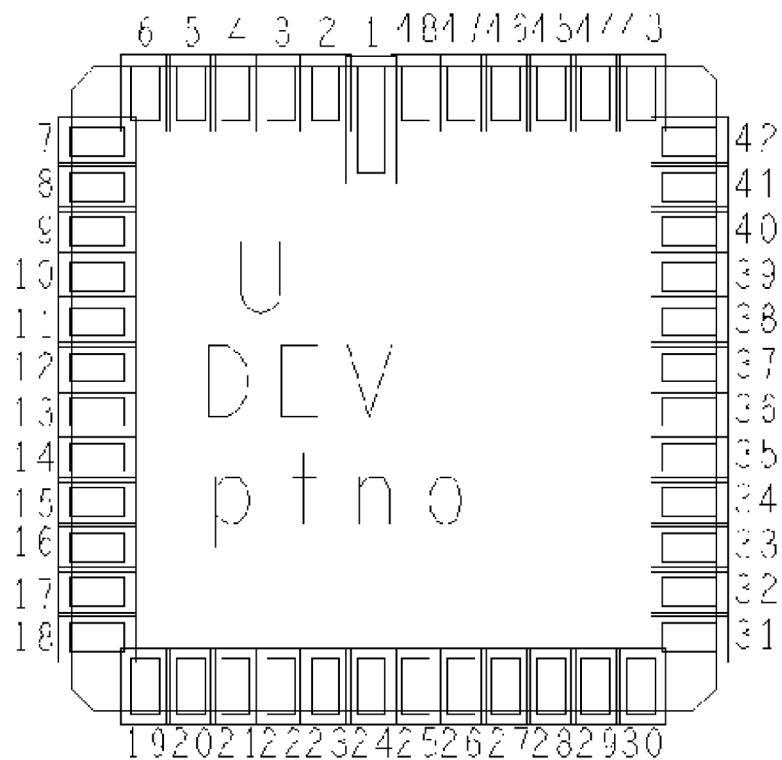
icc28



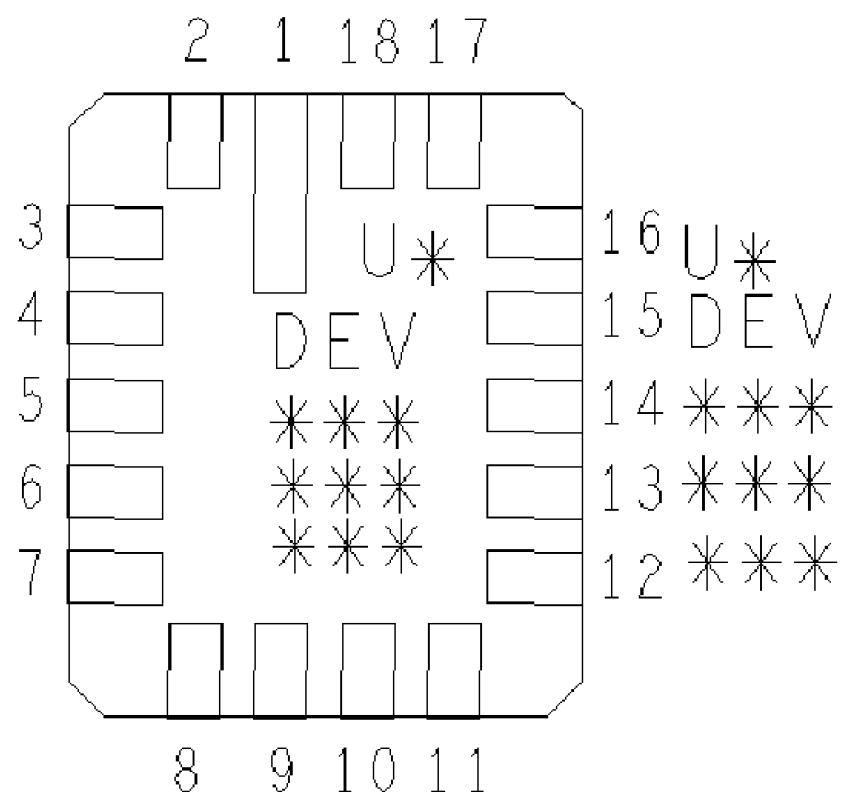
icc24



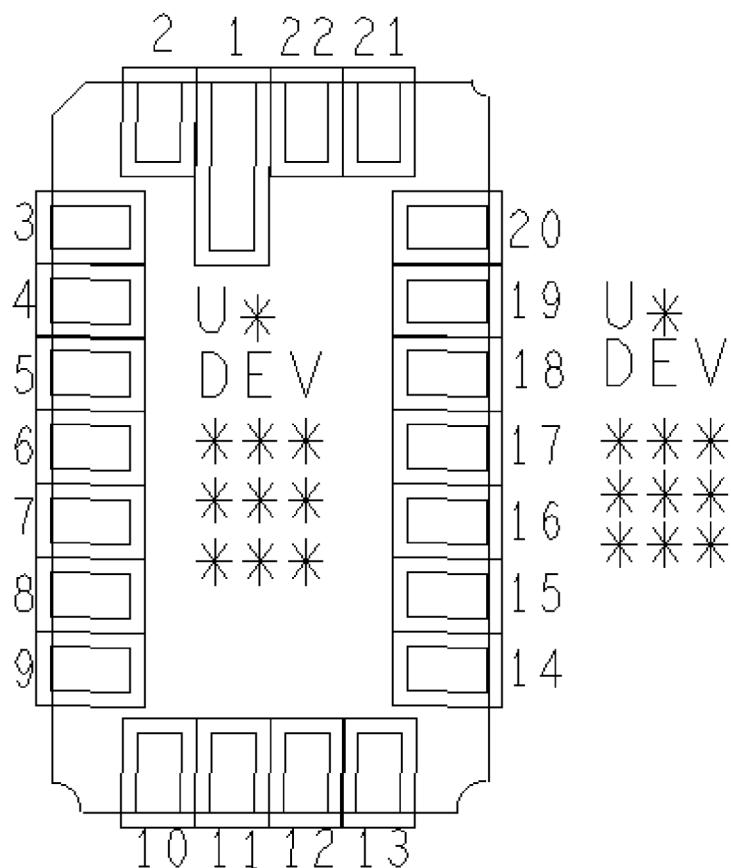
icc48



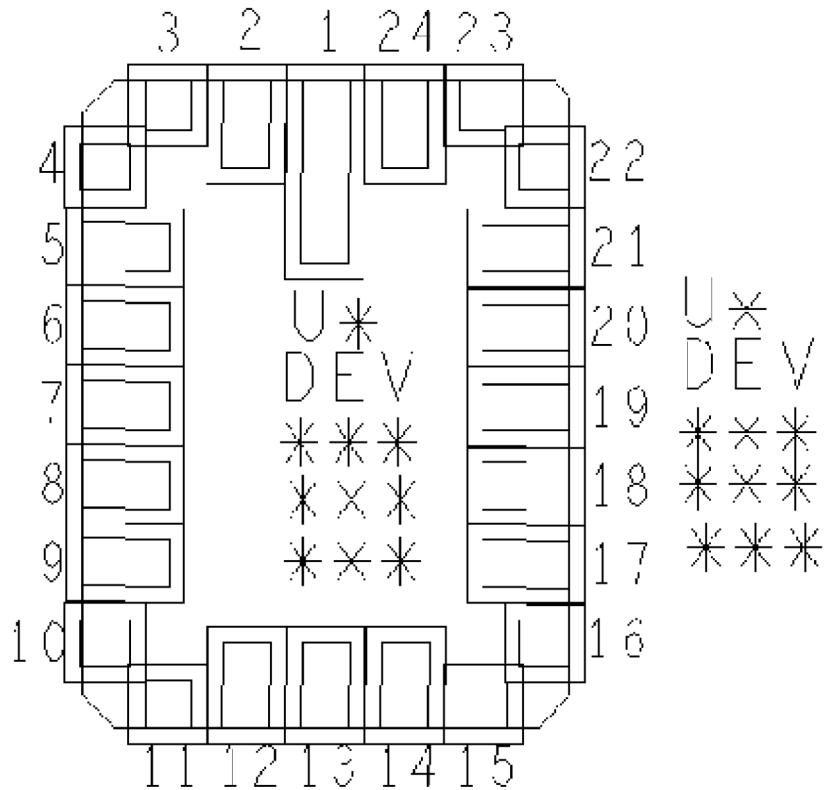
iccs18



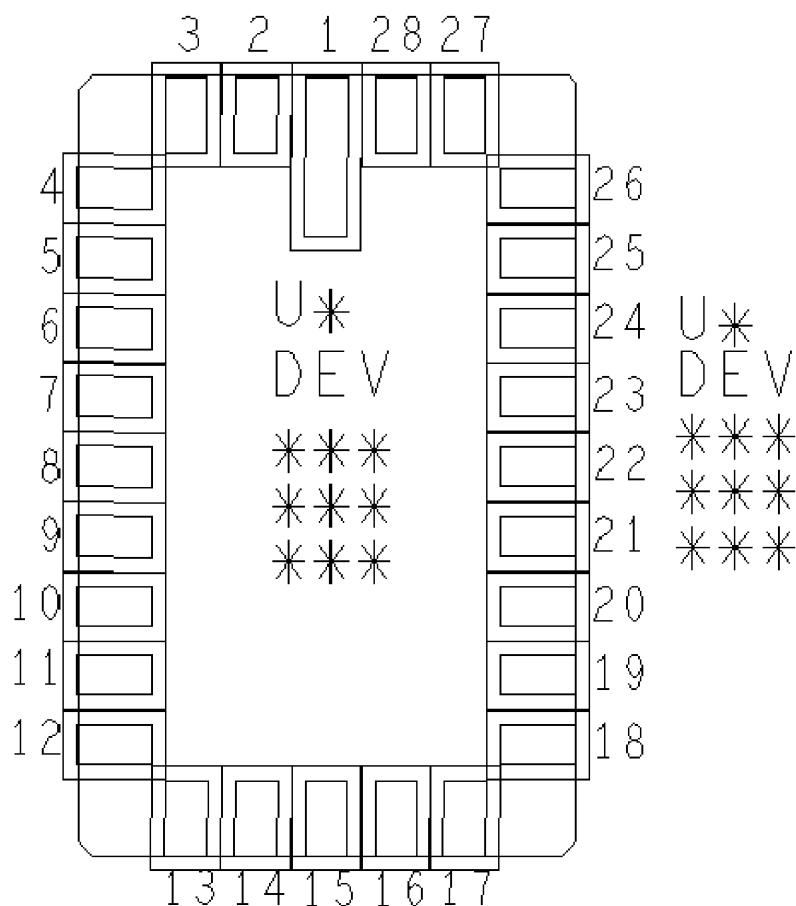
iccs22



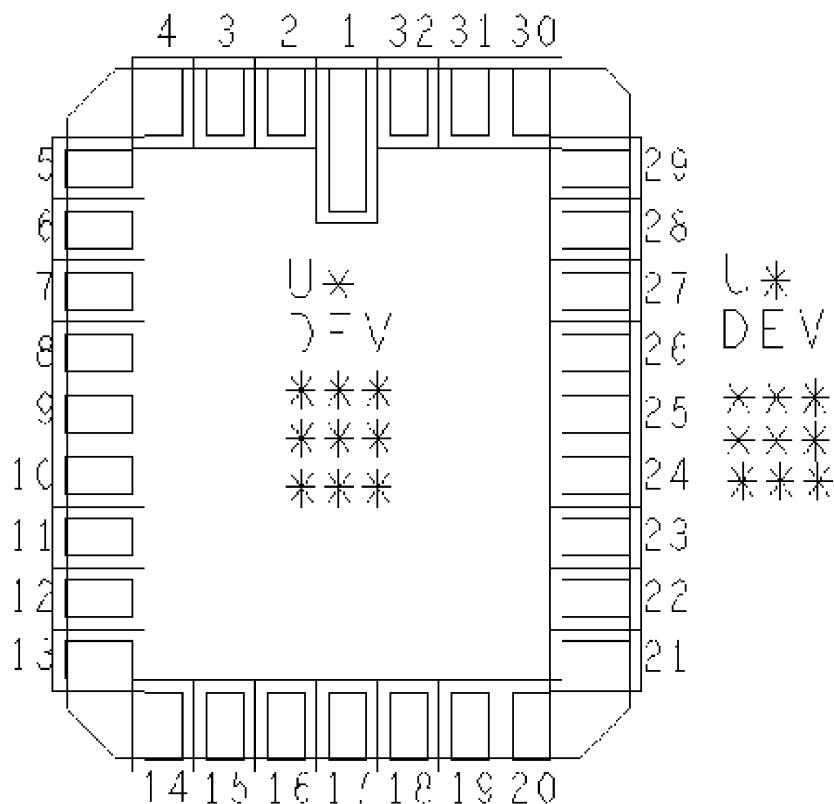
iccs24



iccs28

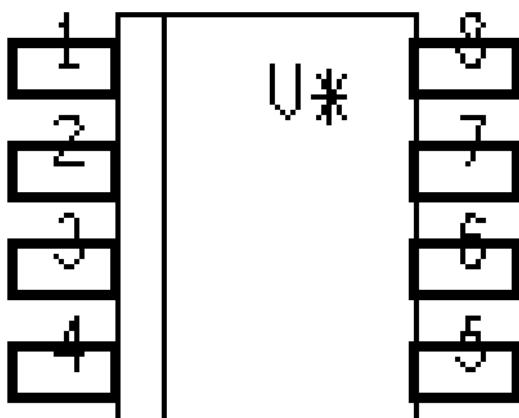


iccs32

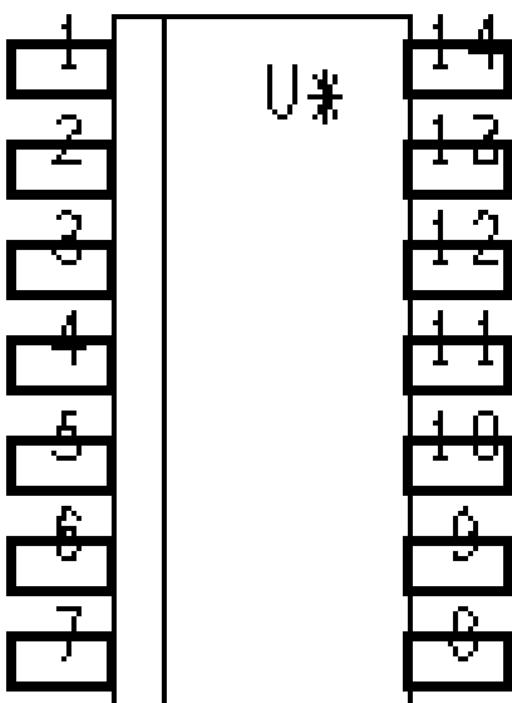


SOICs

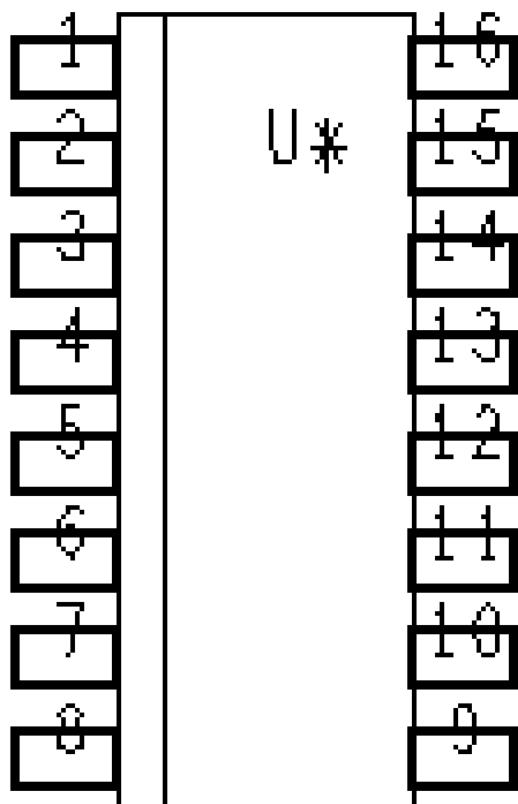
soic8



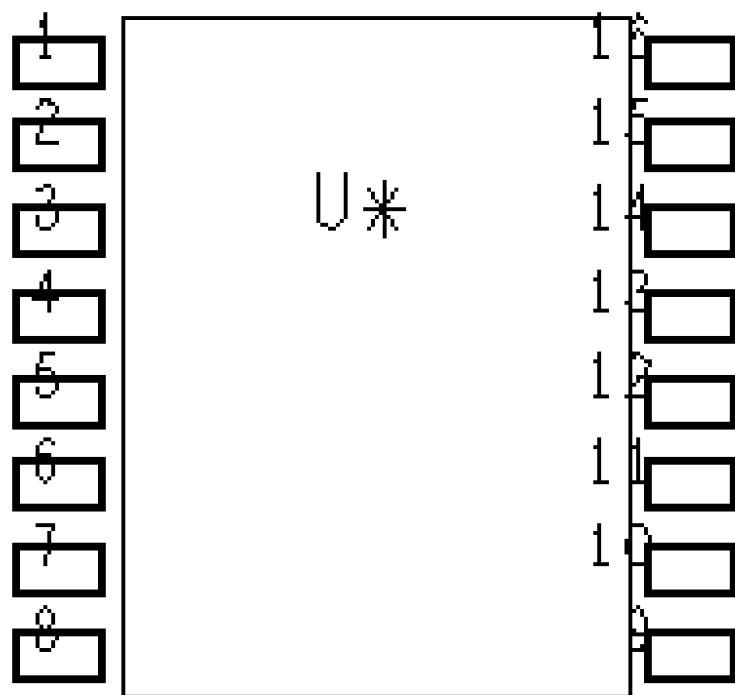
soic14



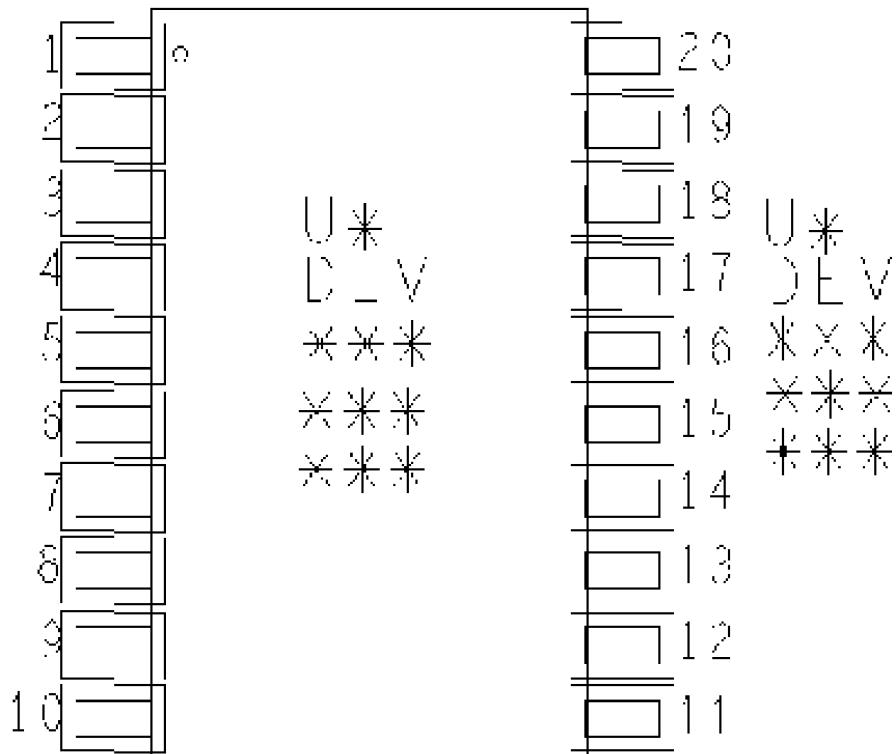
soic16



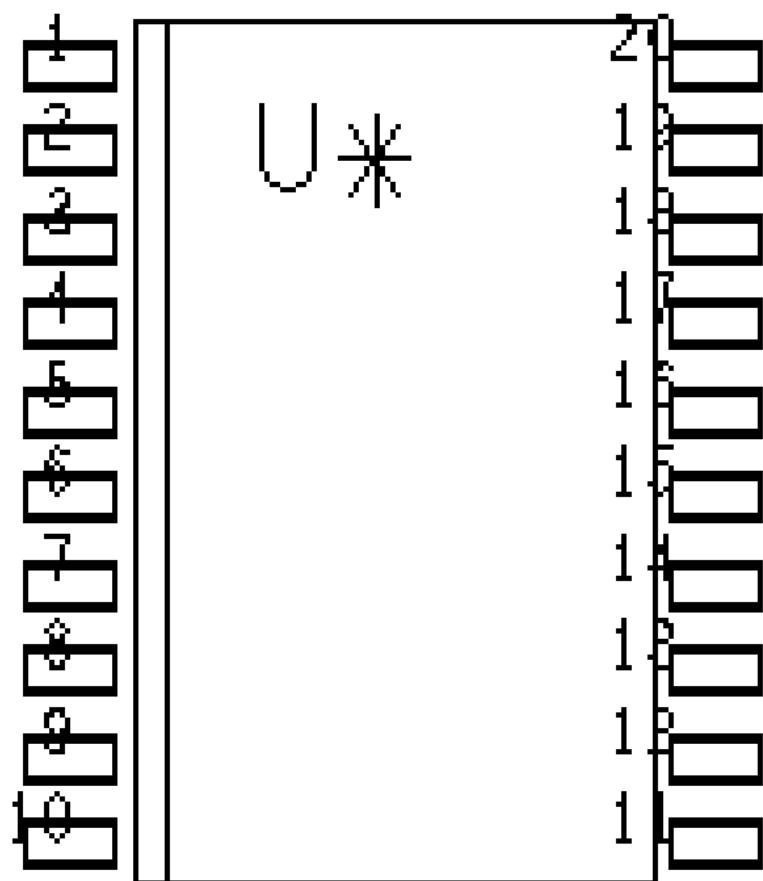
soic16w



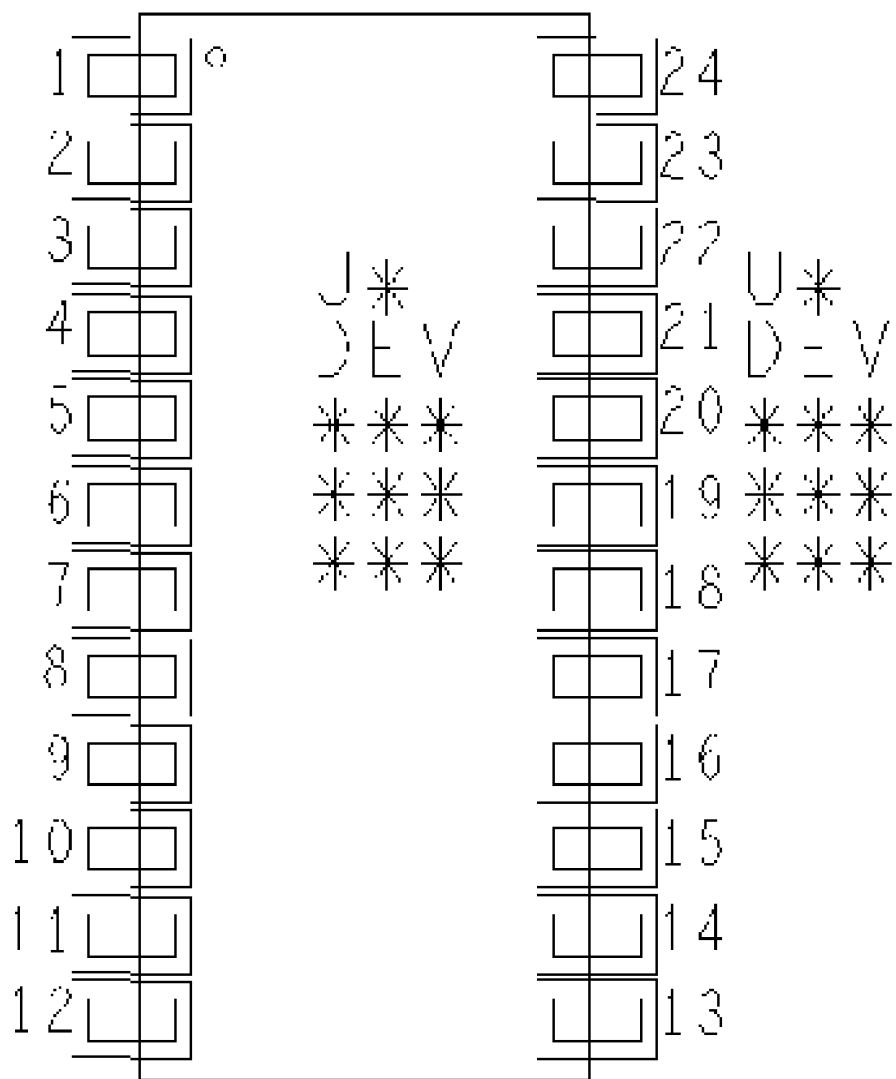
soic20



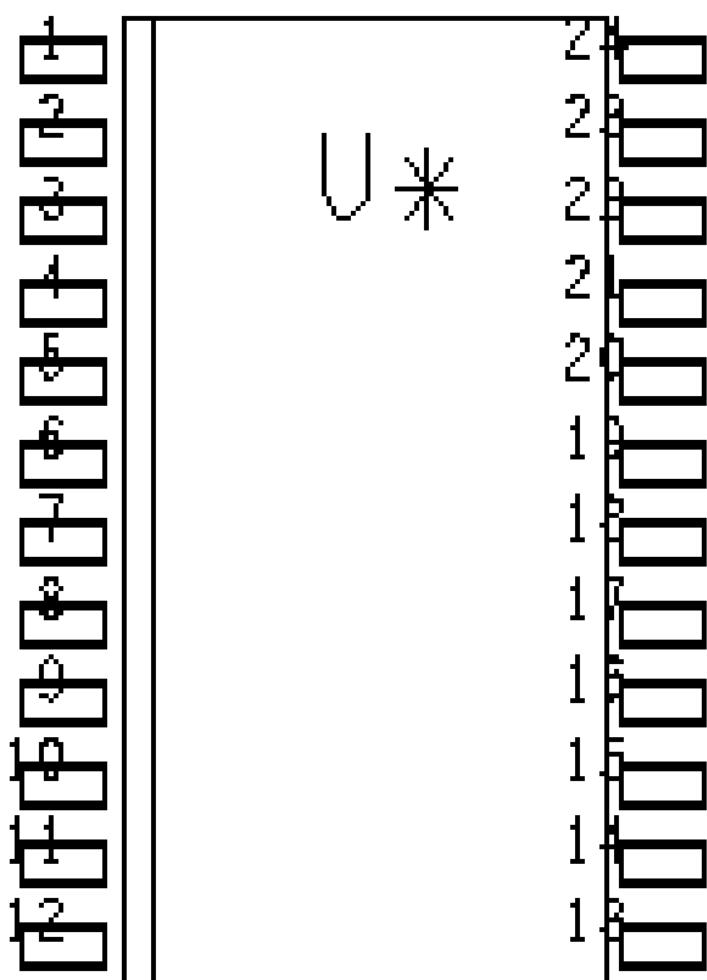
soic20w



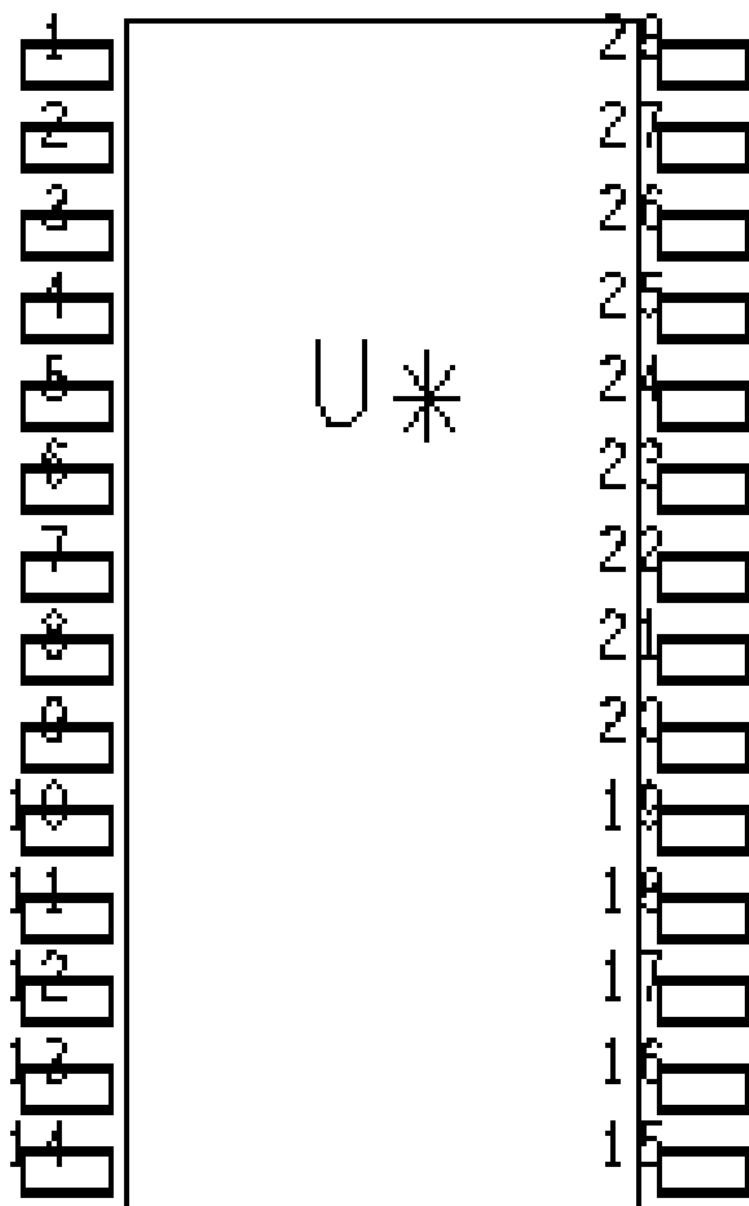
soic24



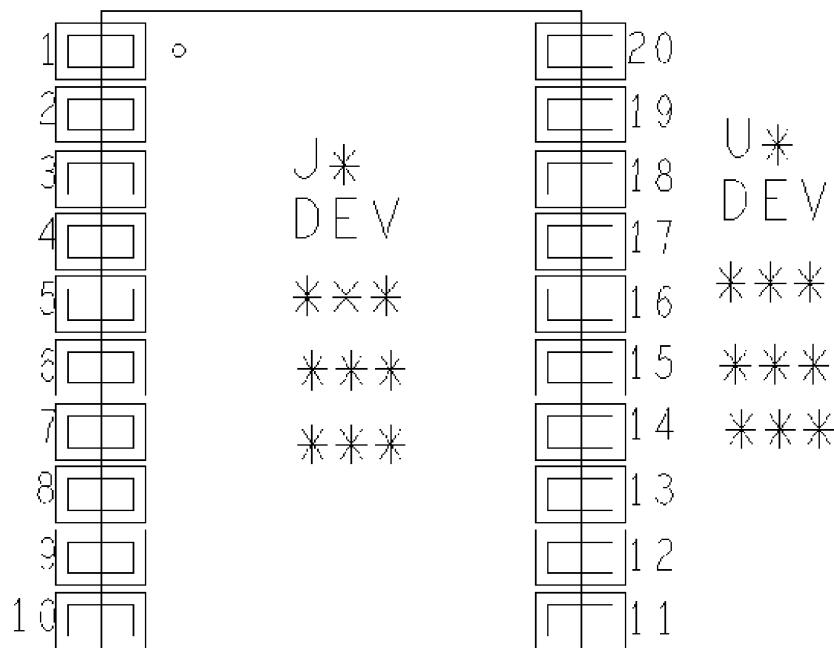
soic24w



soic28w

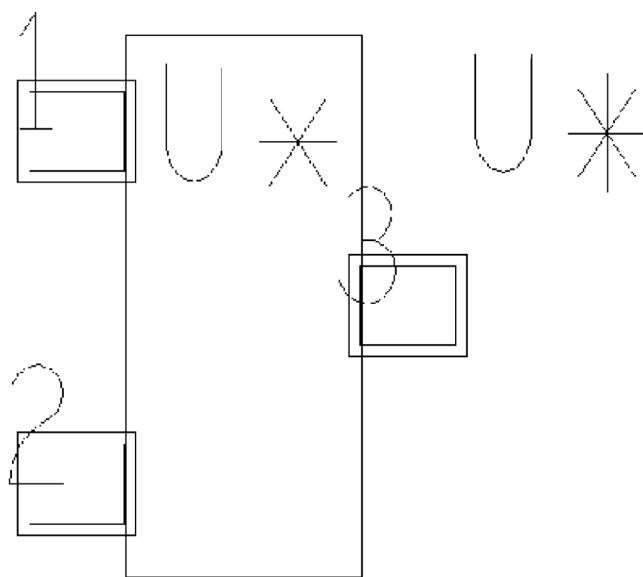


sol120

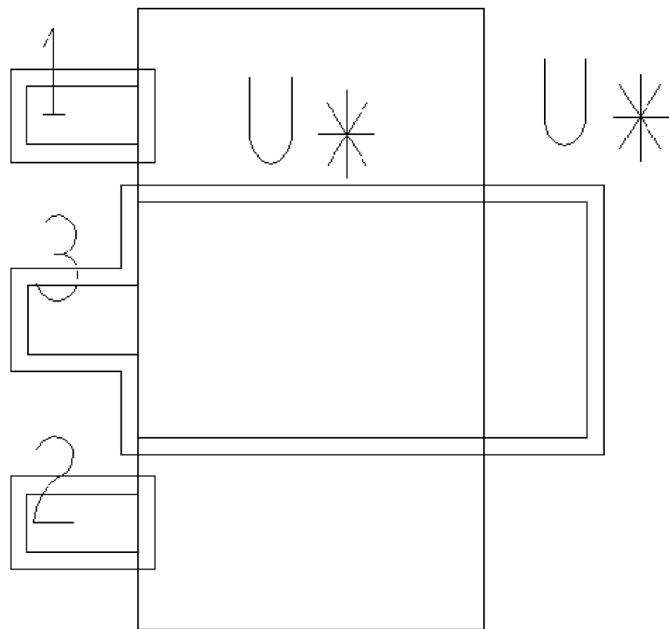


Transistors

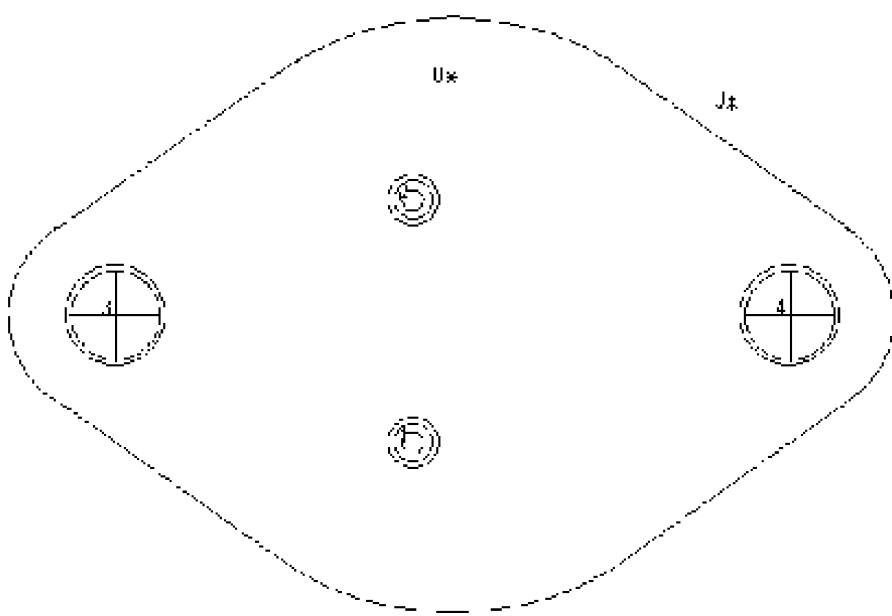
sot23



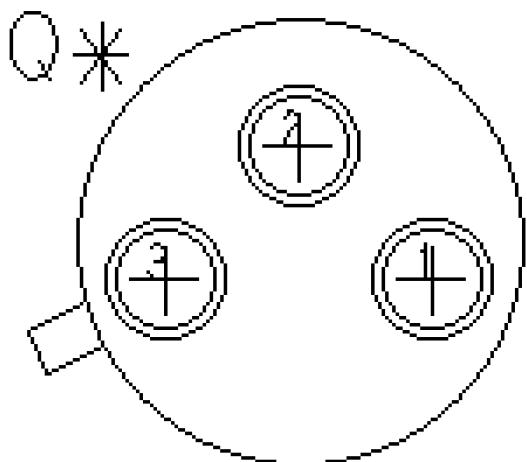
sot89



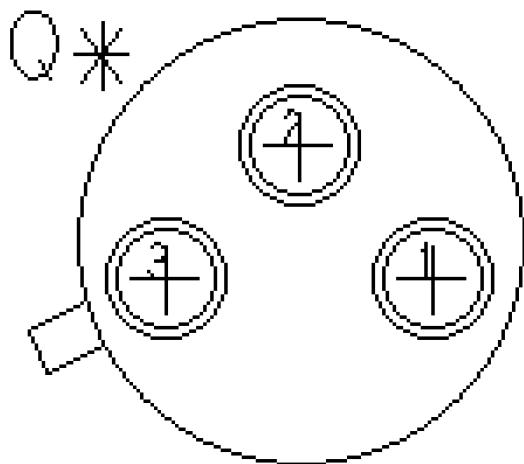
to3



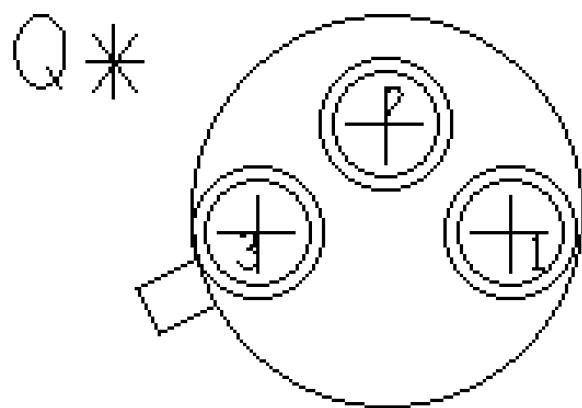
to5



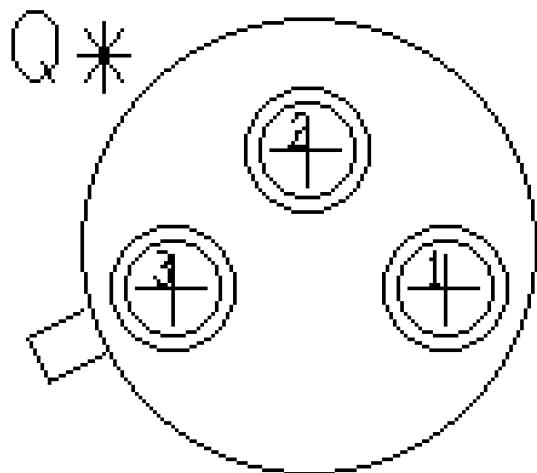
to12



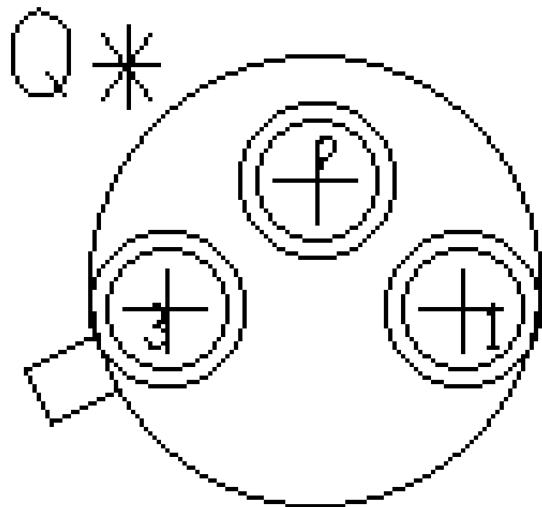
to18



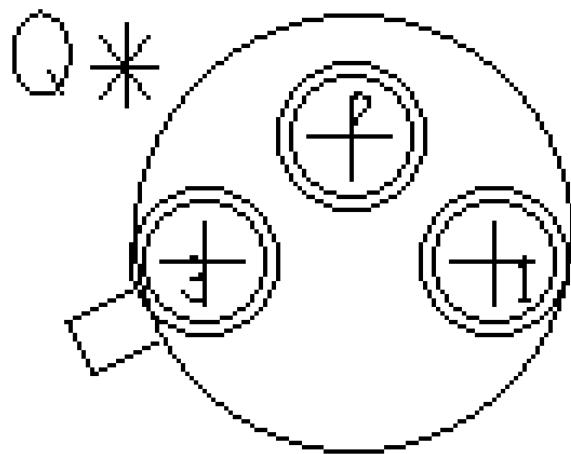
to39



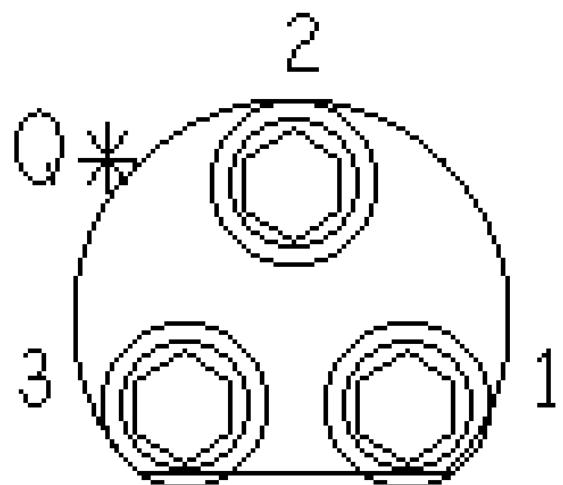
to46



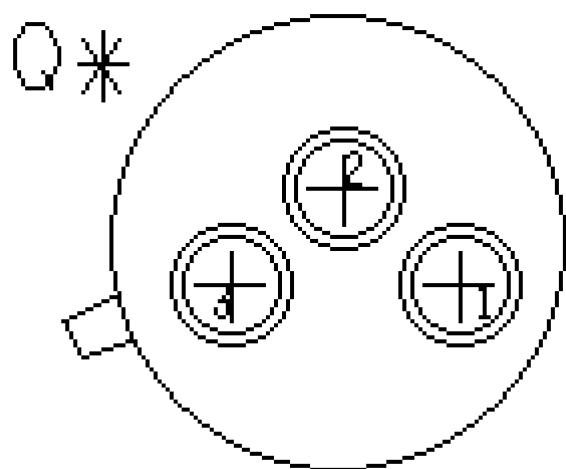
to52



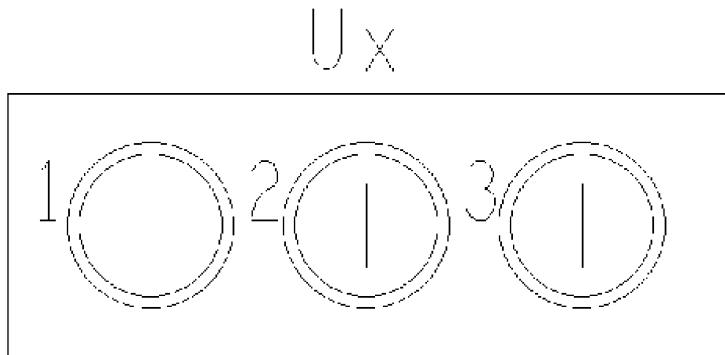
to92



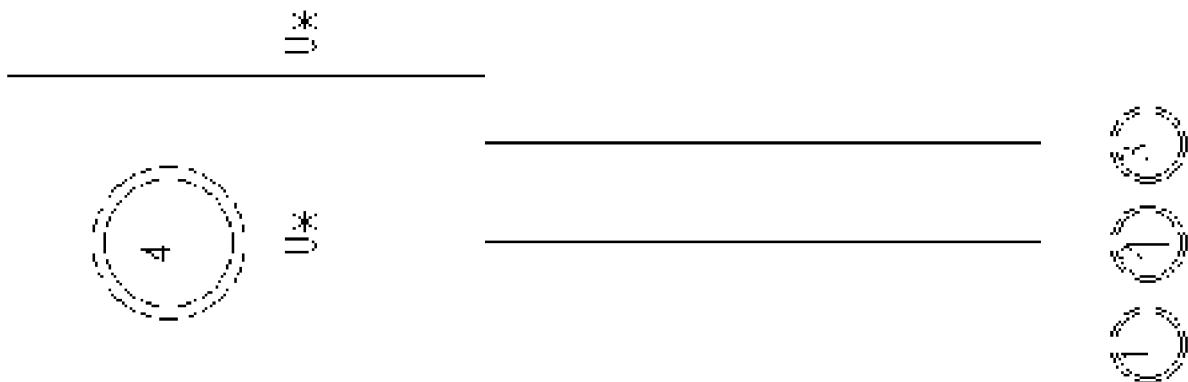
to107



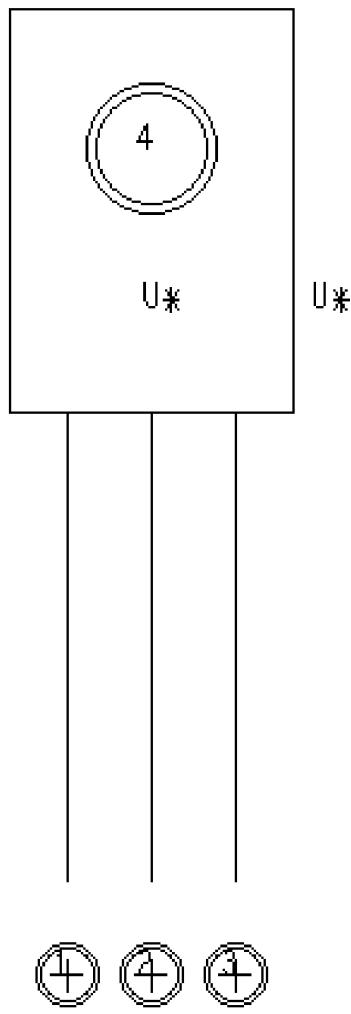
to126



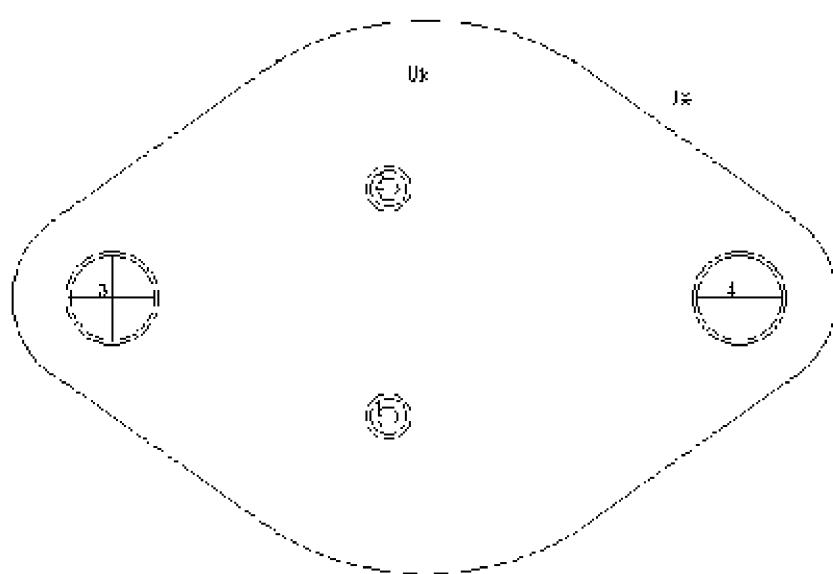
to126h



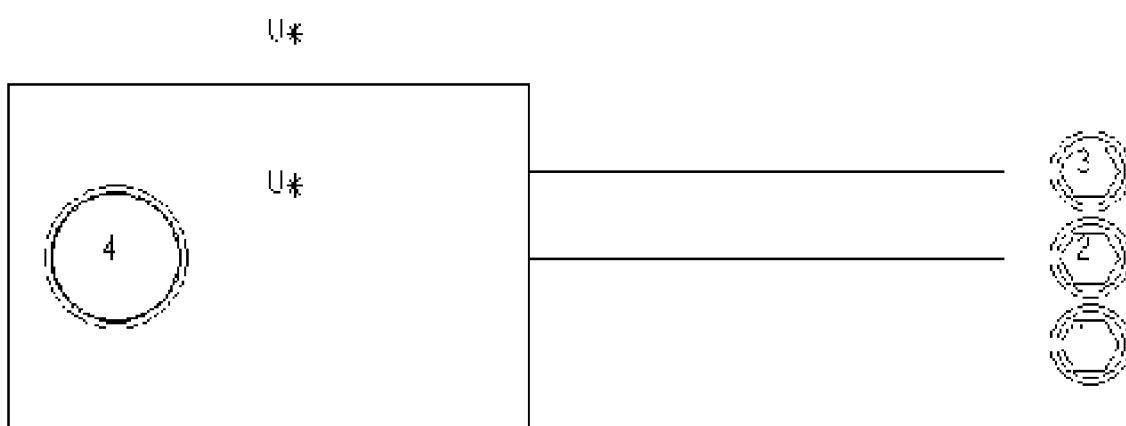
to126v



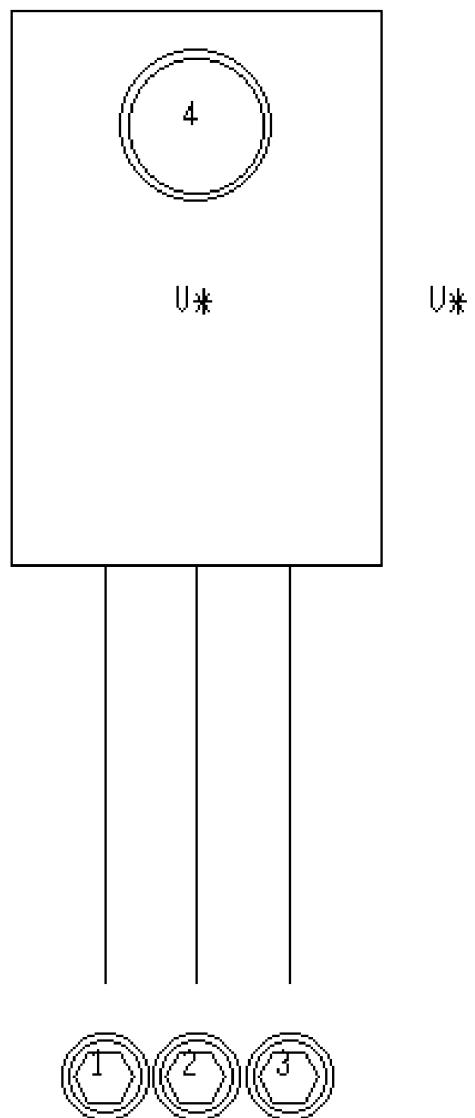
to204aa



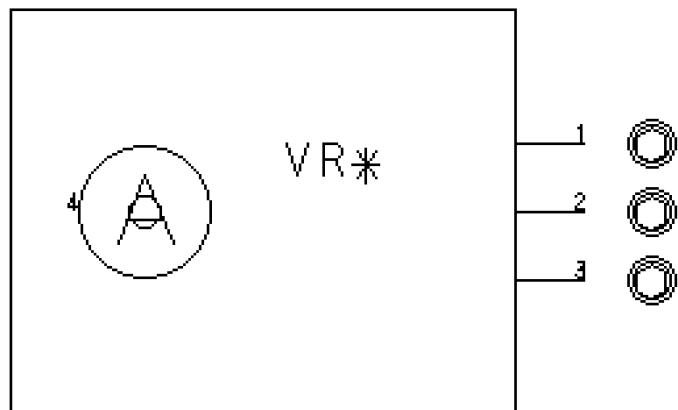
to220abh



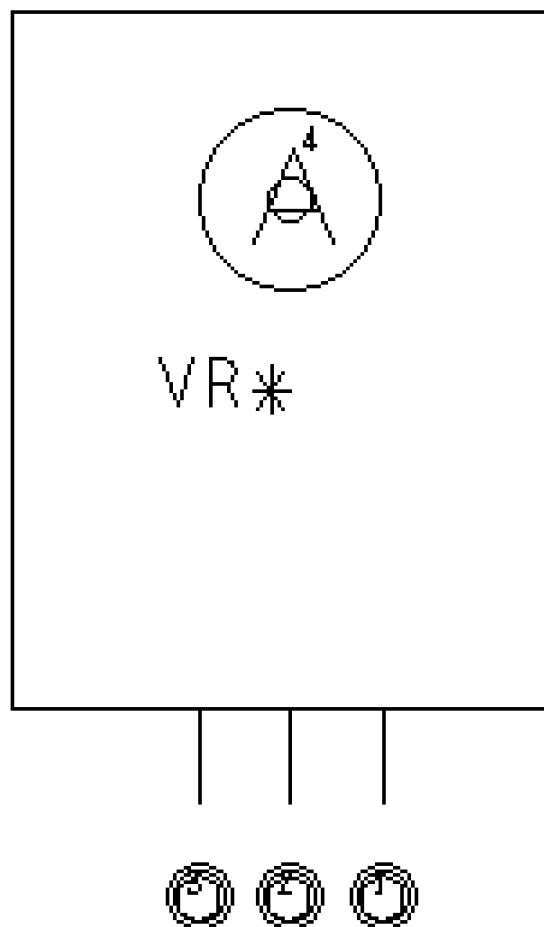
to220abv



to220h

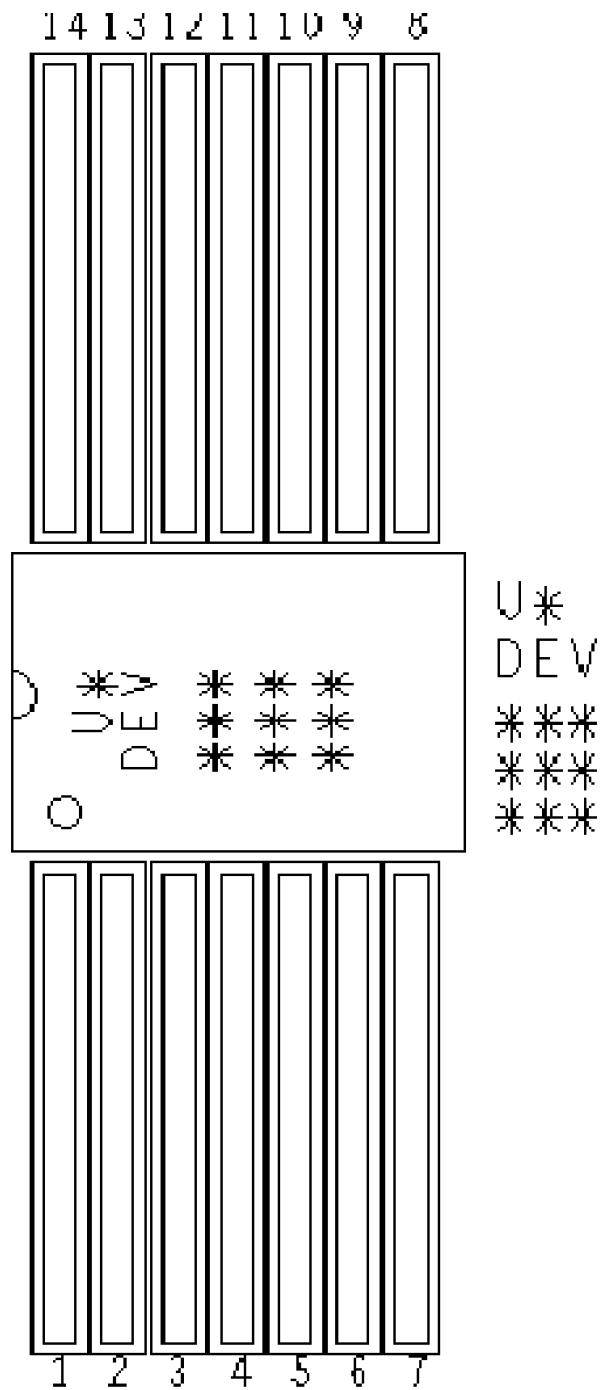


to220v

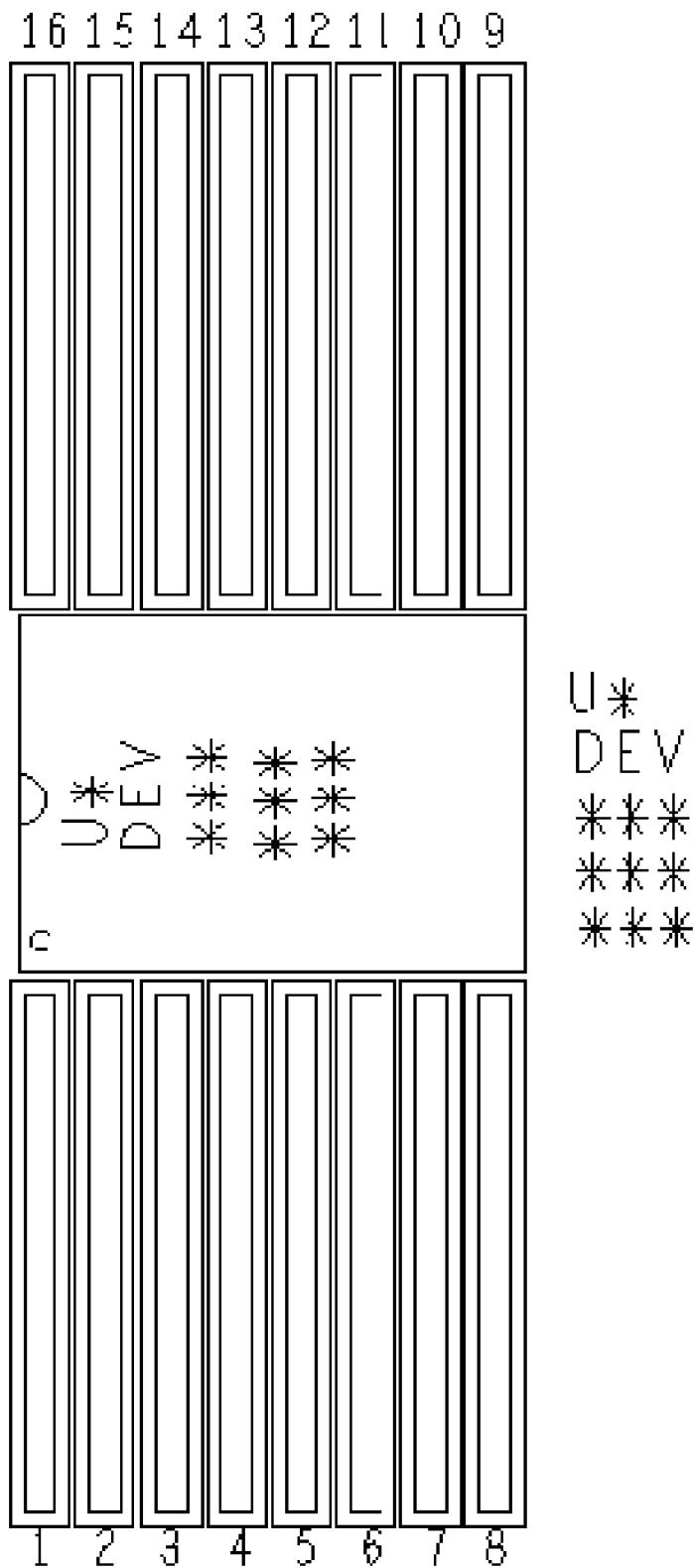


Flat Packs

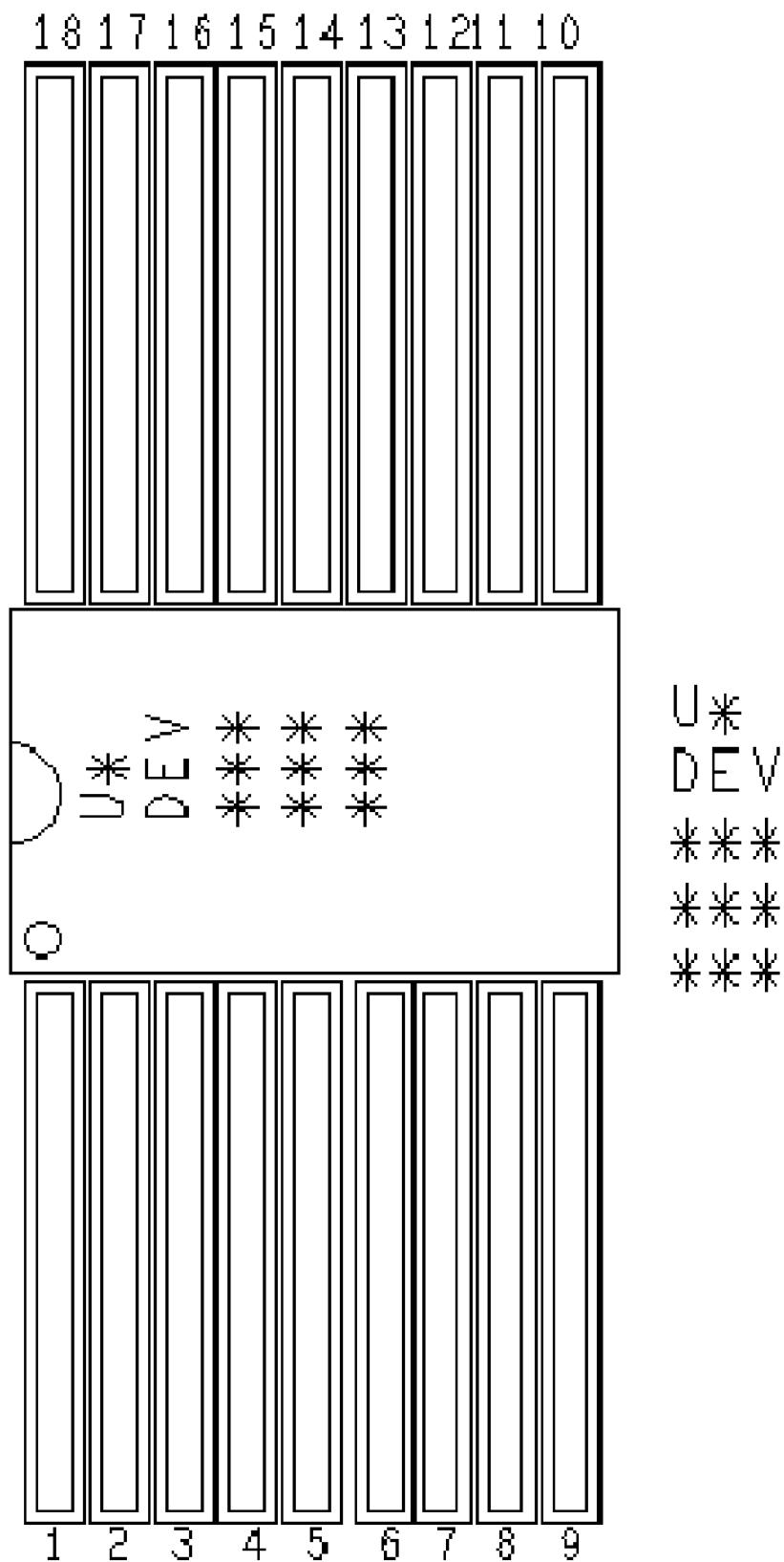
flat14



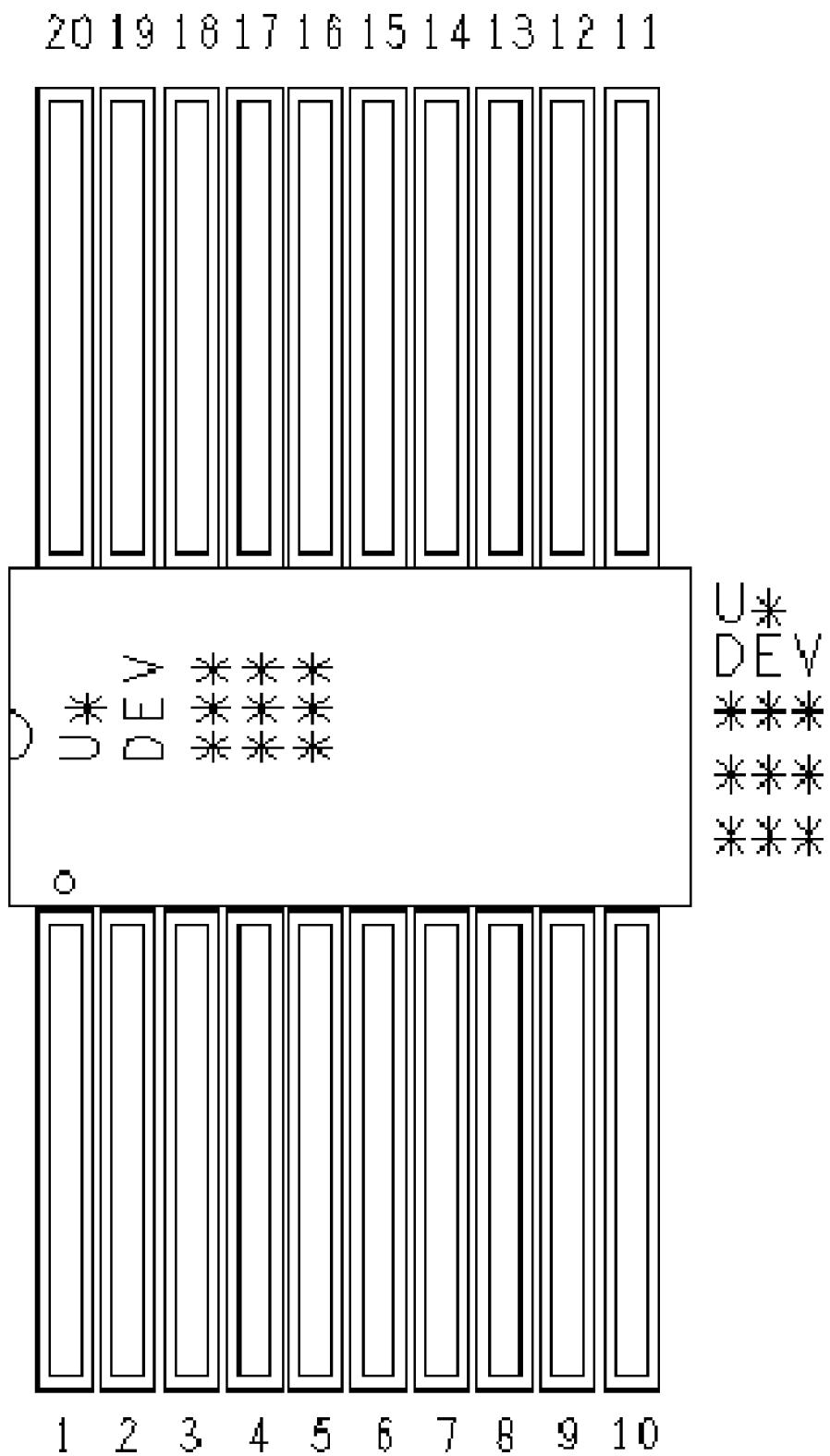
flat16



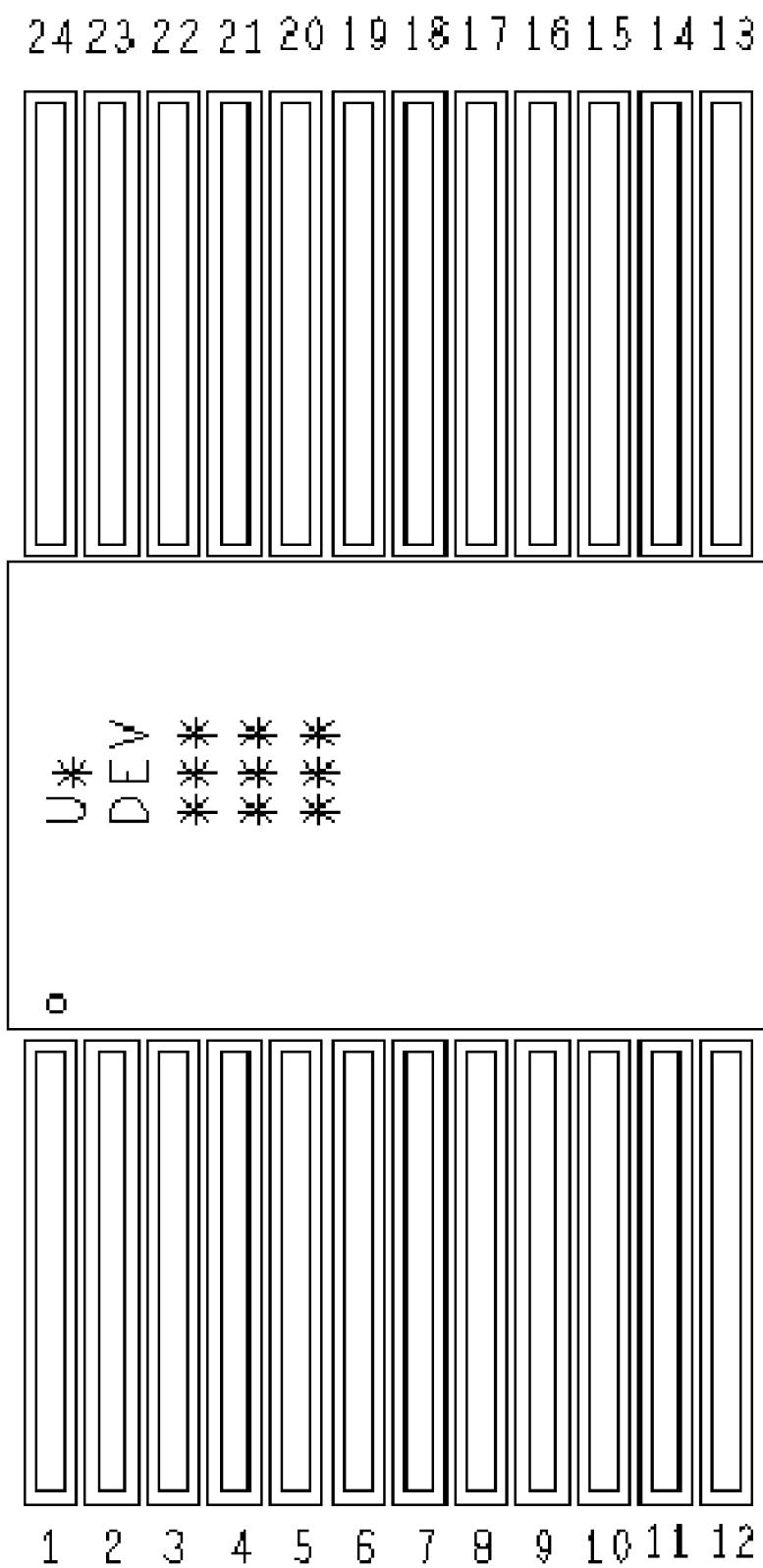
flat18



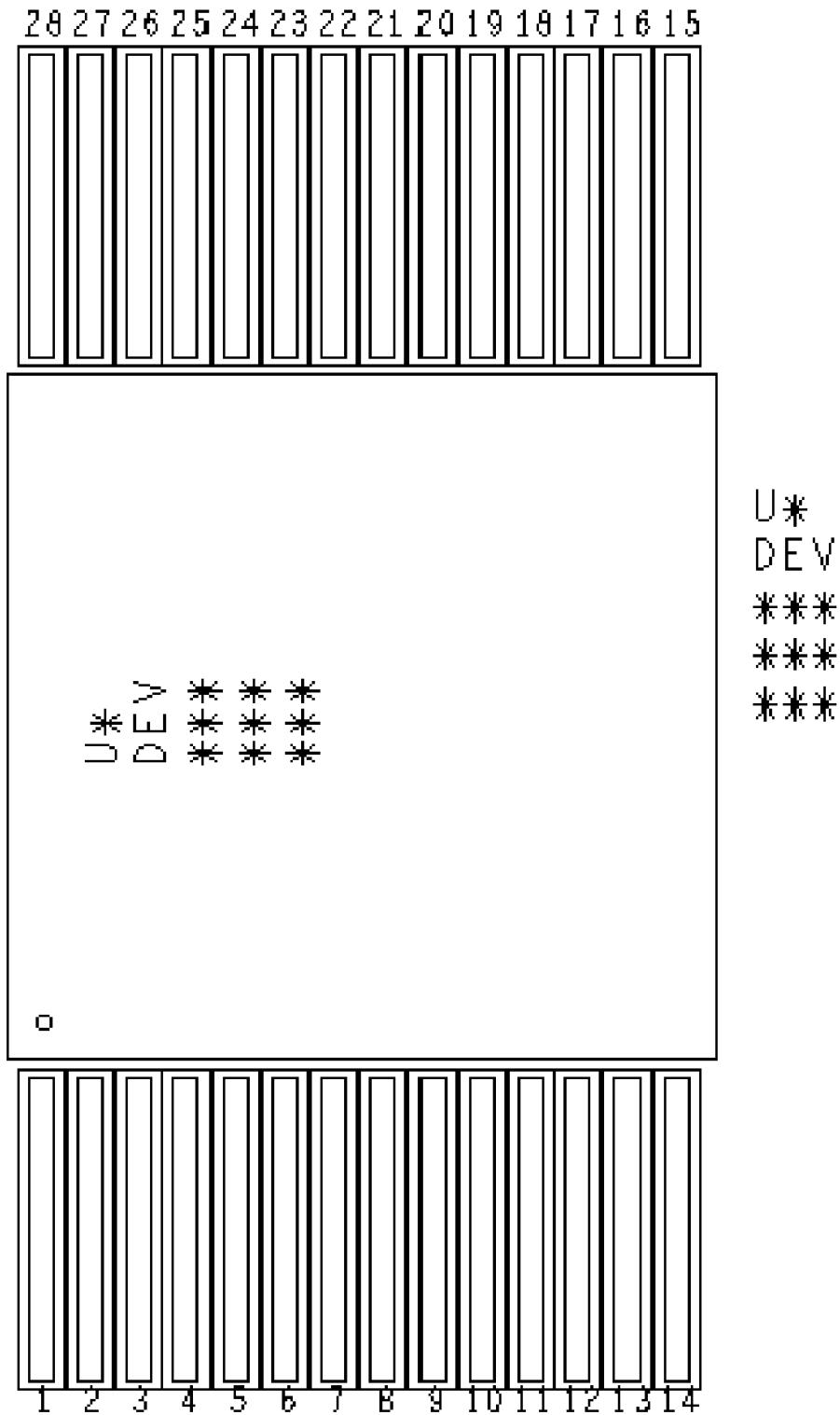
flat20



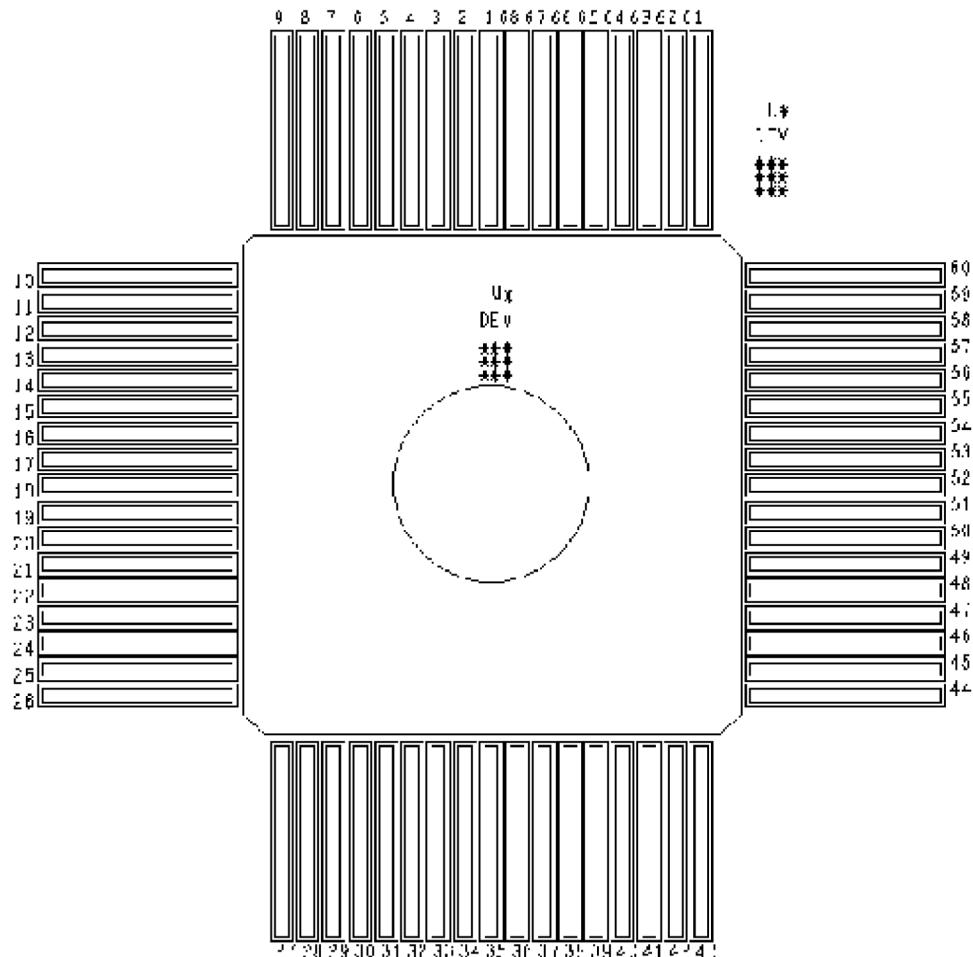
flat24



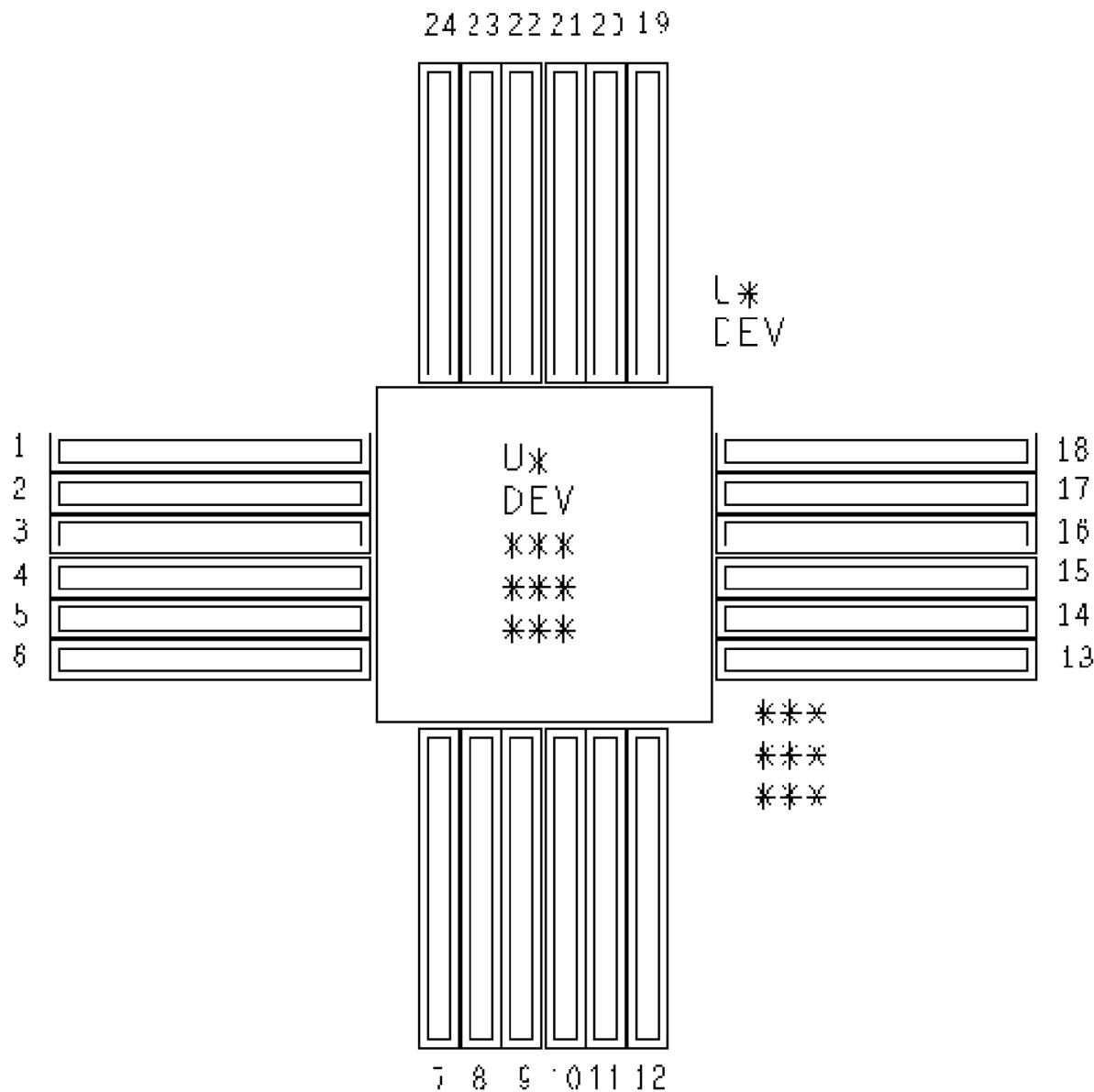
flat28



cpfp68

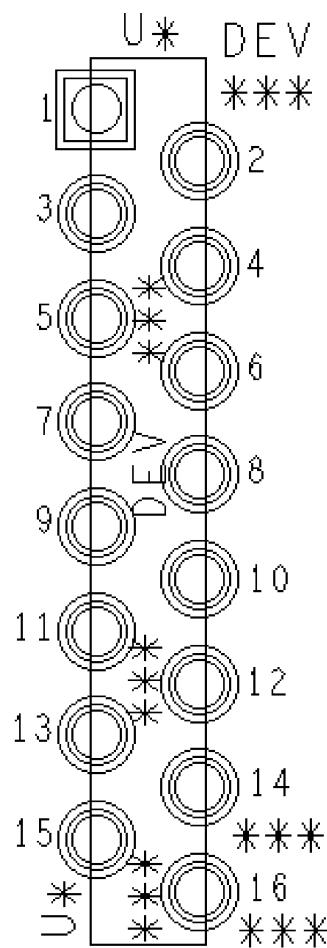


quadflat24

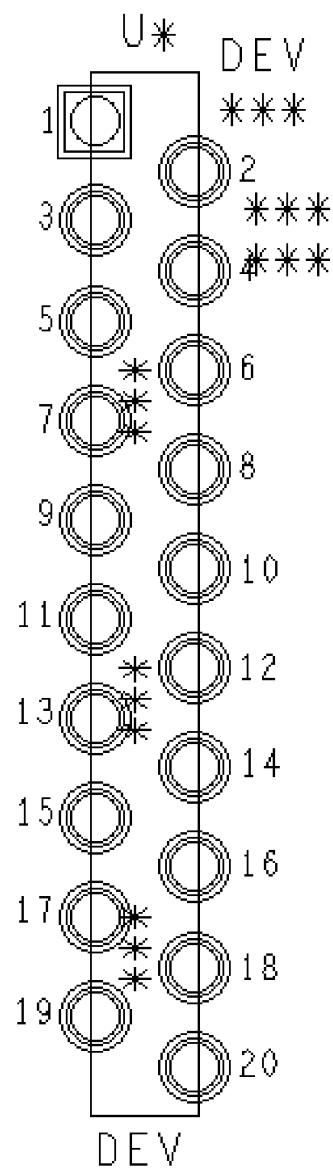


ZIPs

zip16



zip20

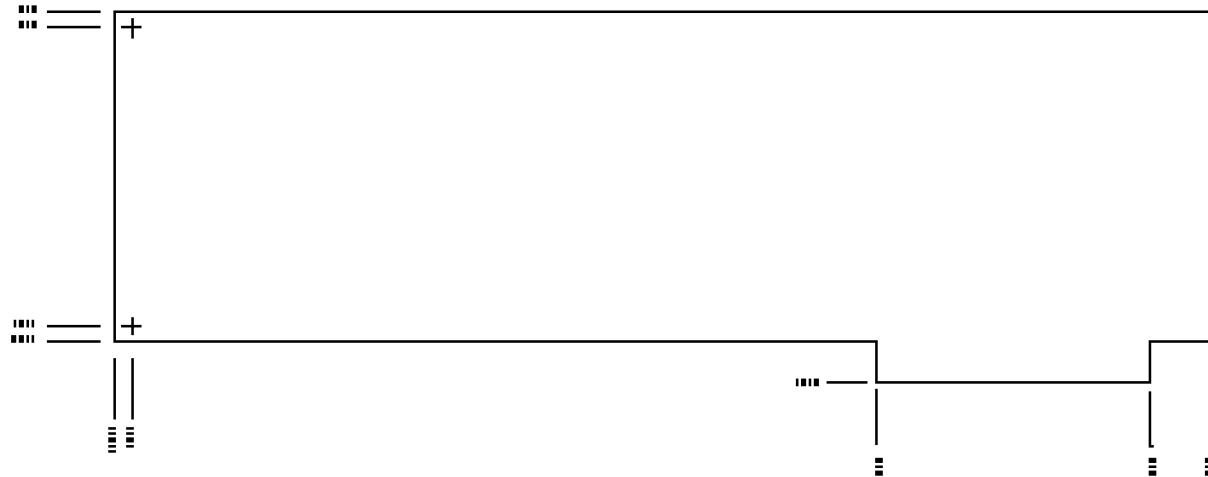


Appendix B: PCB Editor, Mechanical Symbol Library

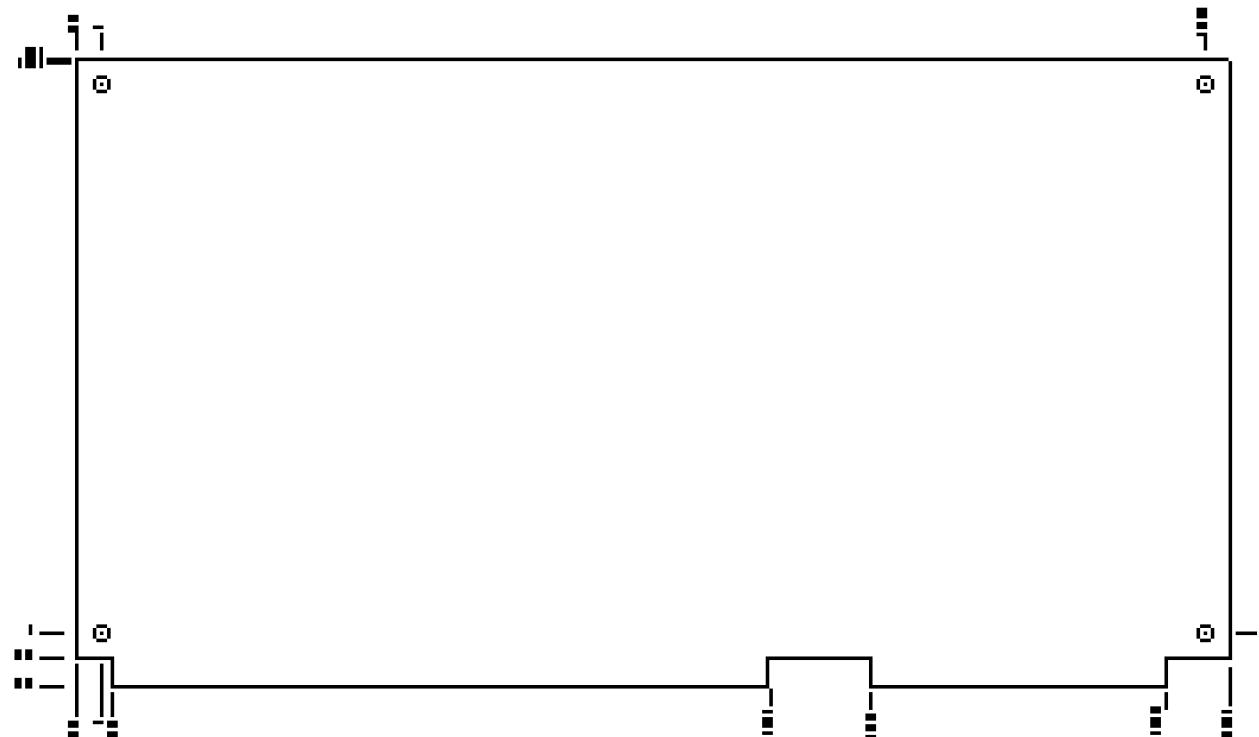
This section illustrates the mechanical symbols provided in the library.

Card Outlines

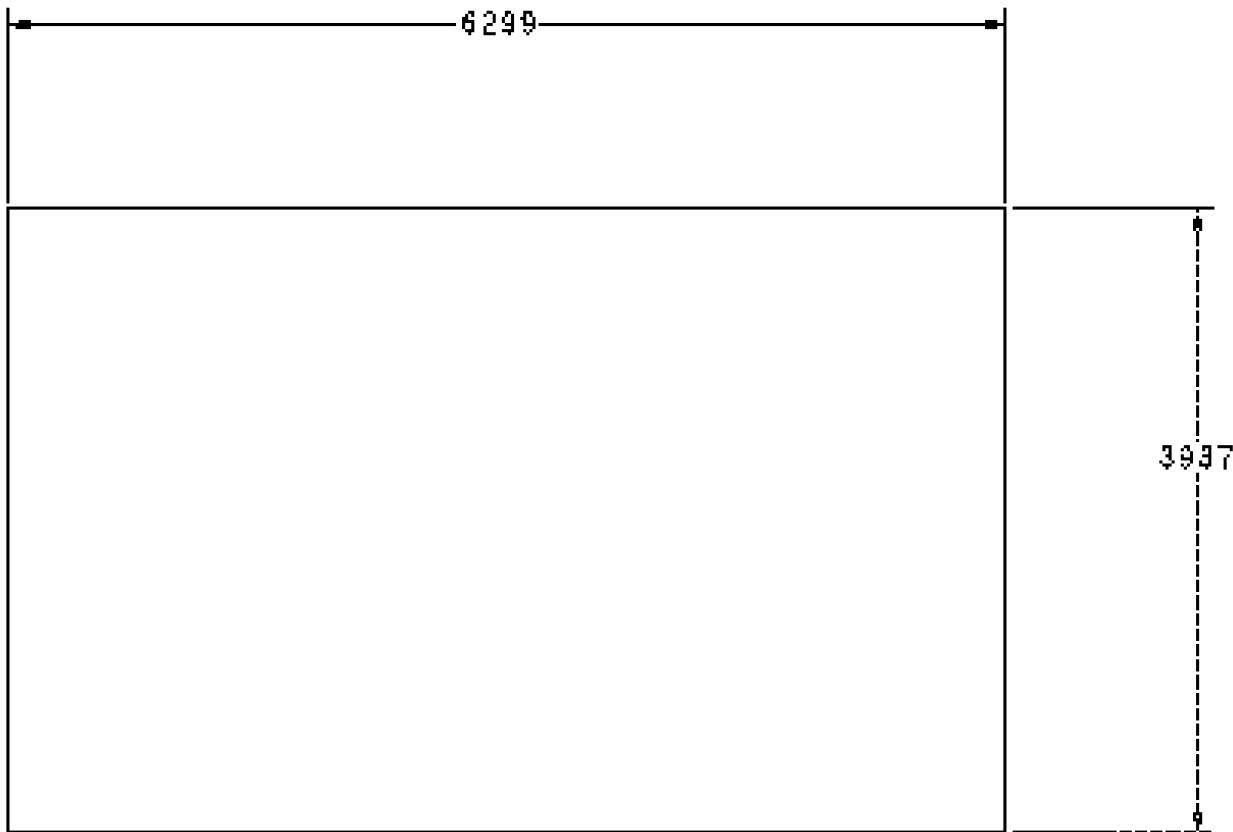
ibm



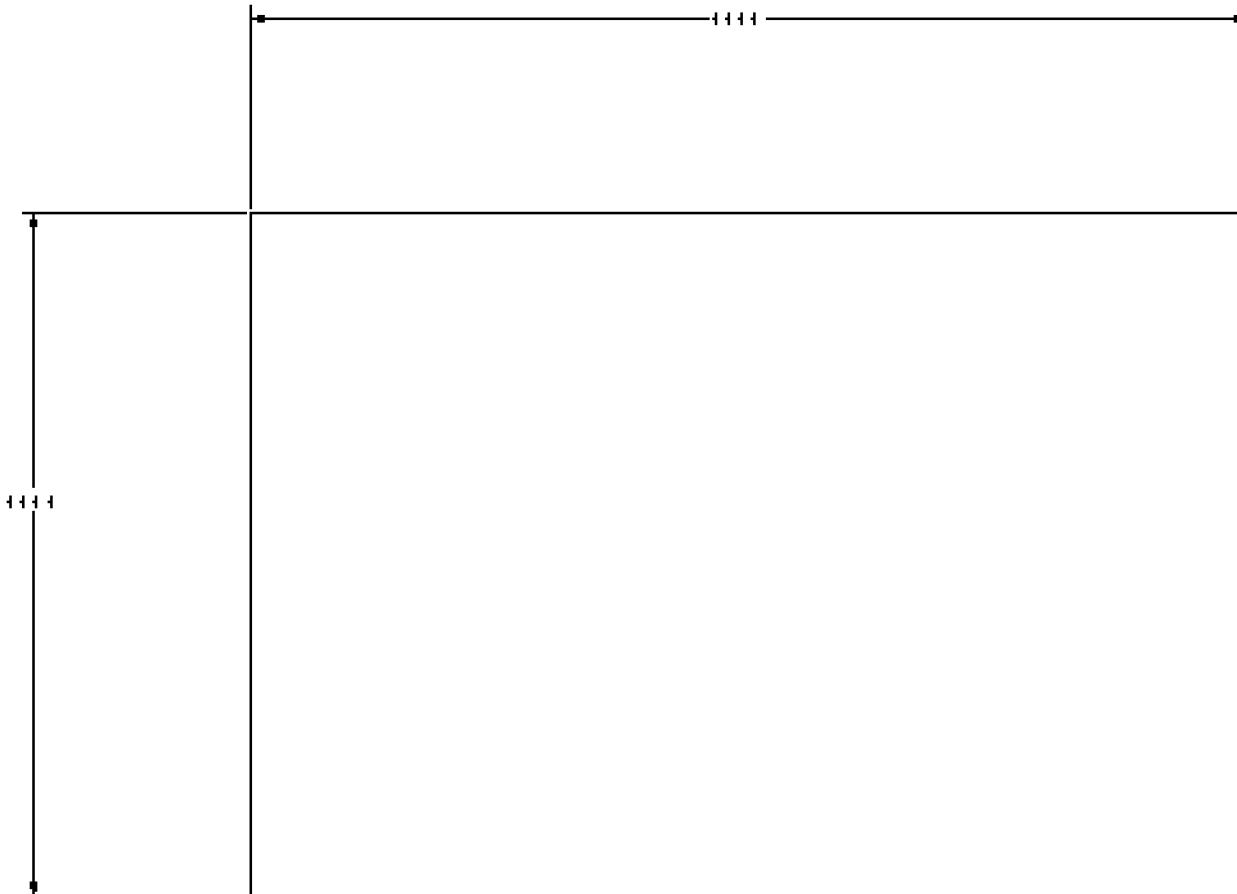
multibus



euros

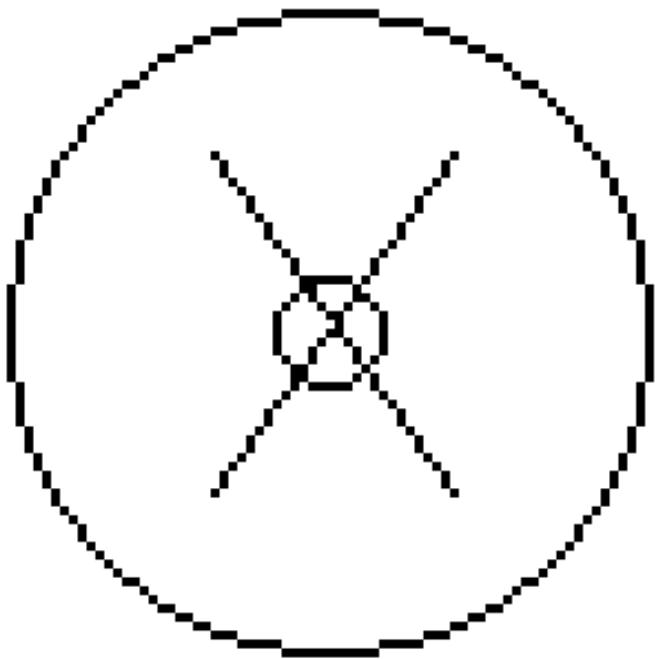


eurod

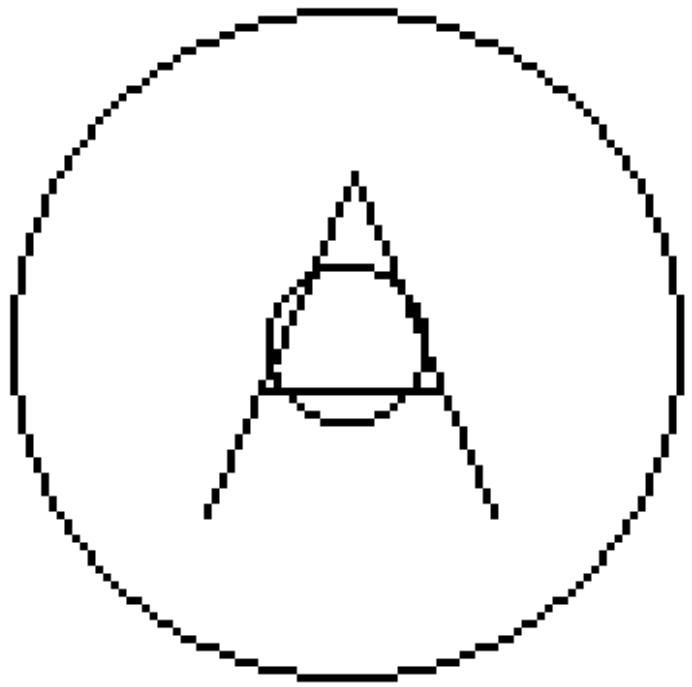


Mounting Holes

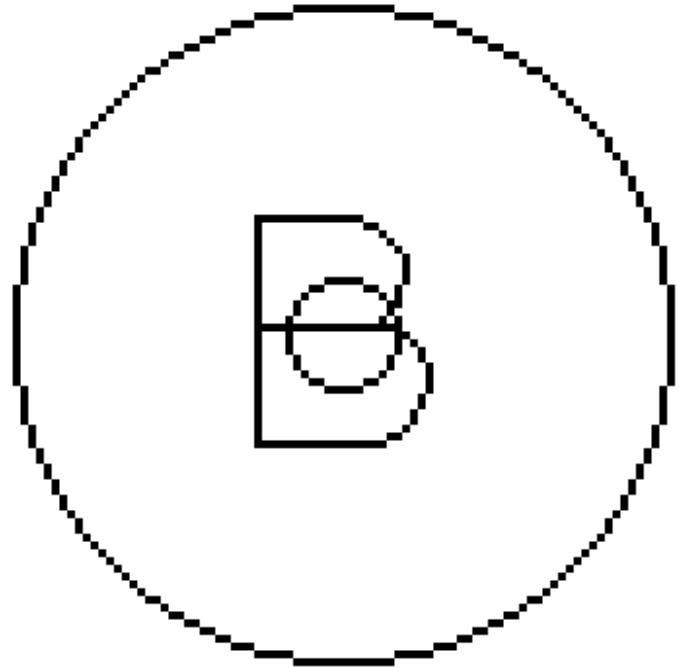
mtq125



mtq156



mtq250

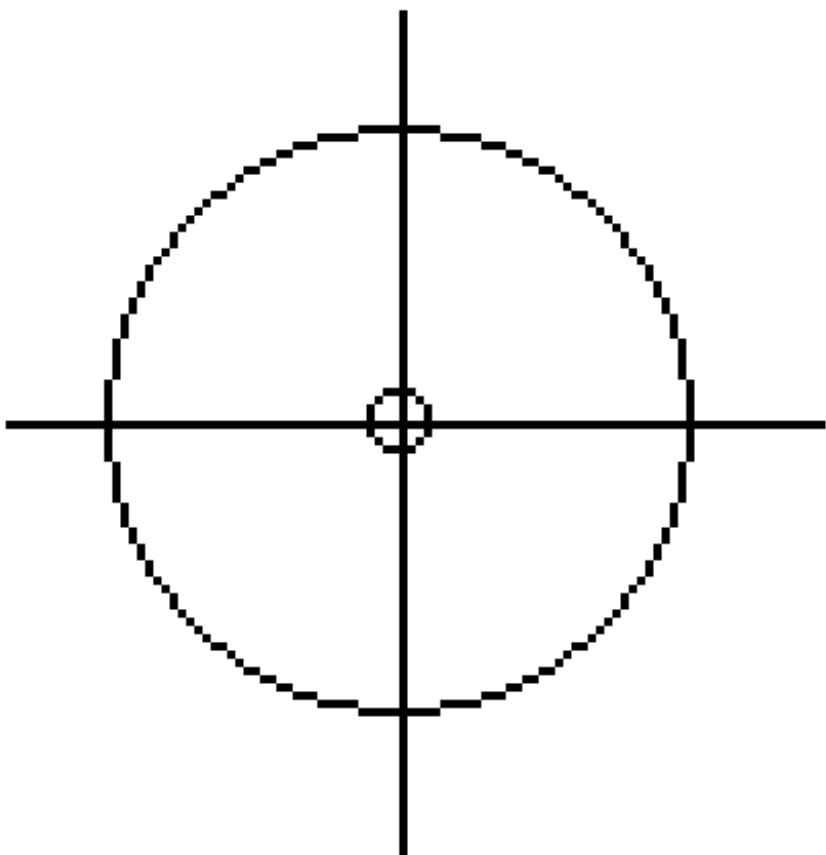


Appendix C: PCB Editor, Format Symbol Library

This section illustrates the format symbols provided in the library.

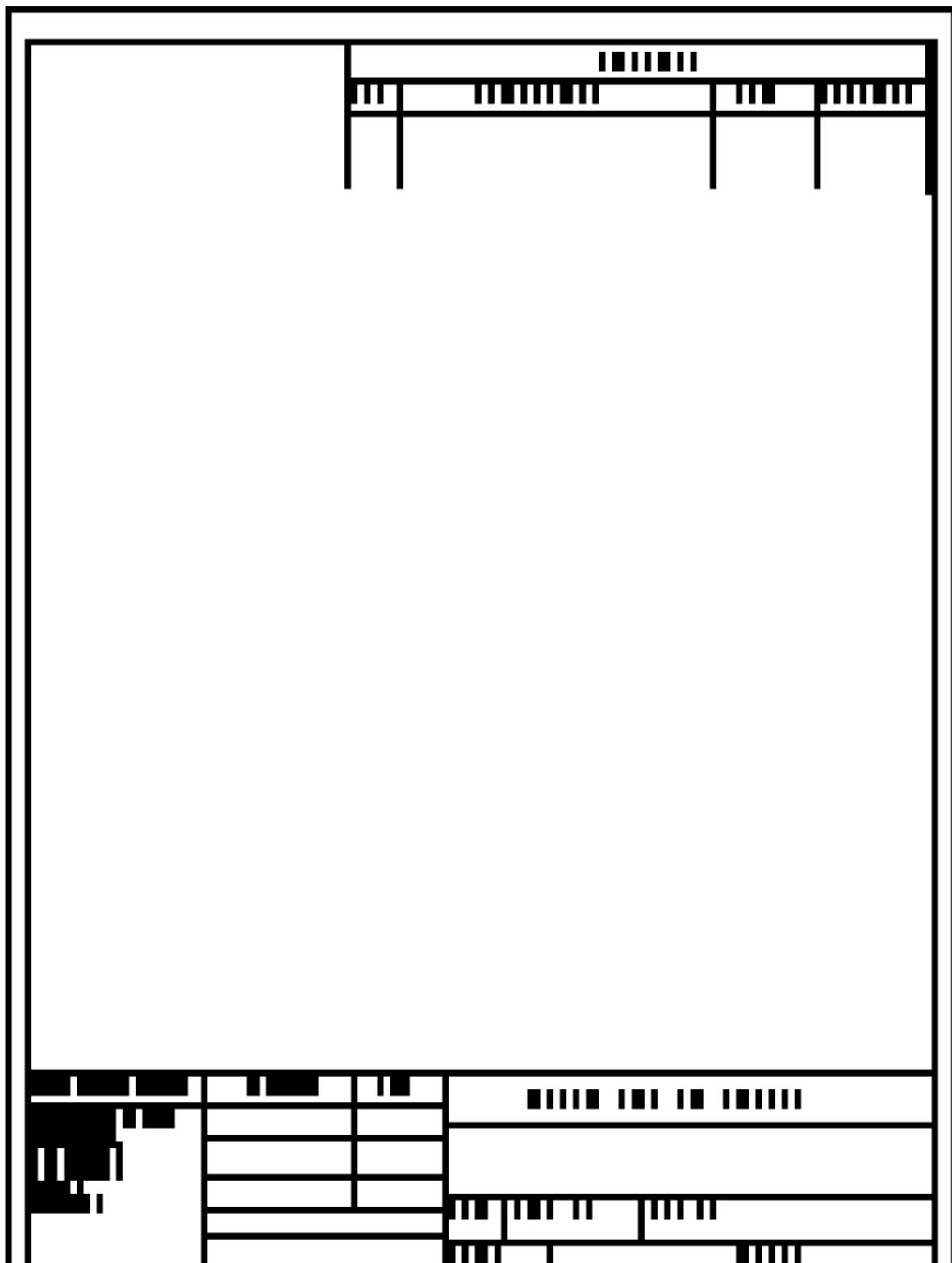
Target

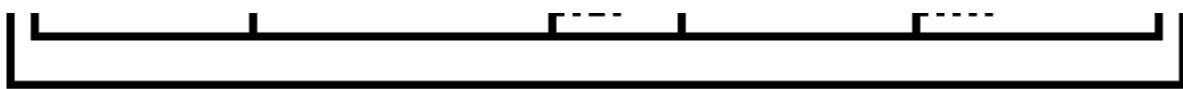
`target`



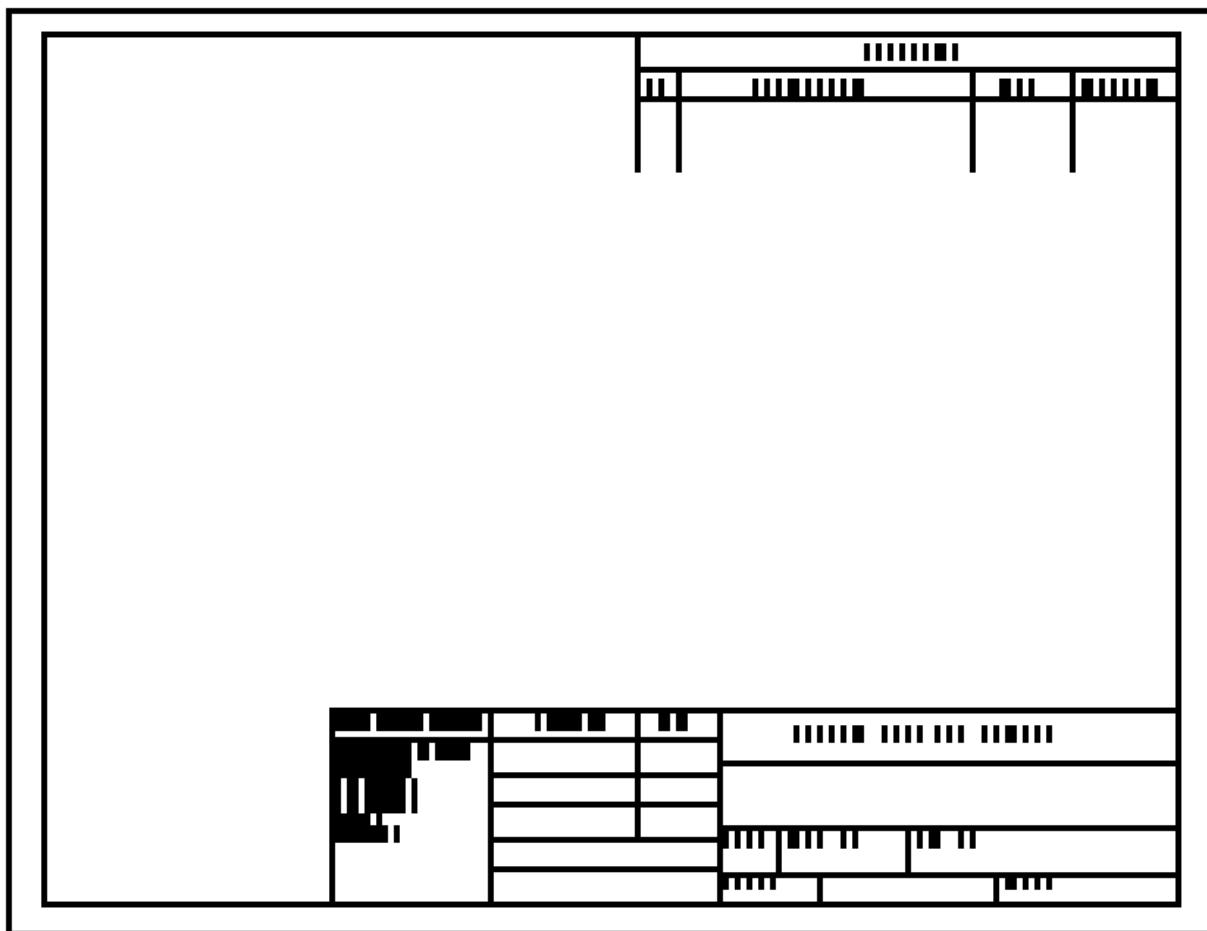
Drawing Formats

asizev

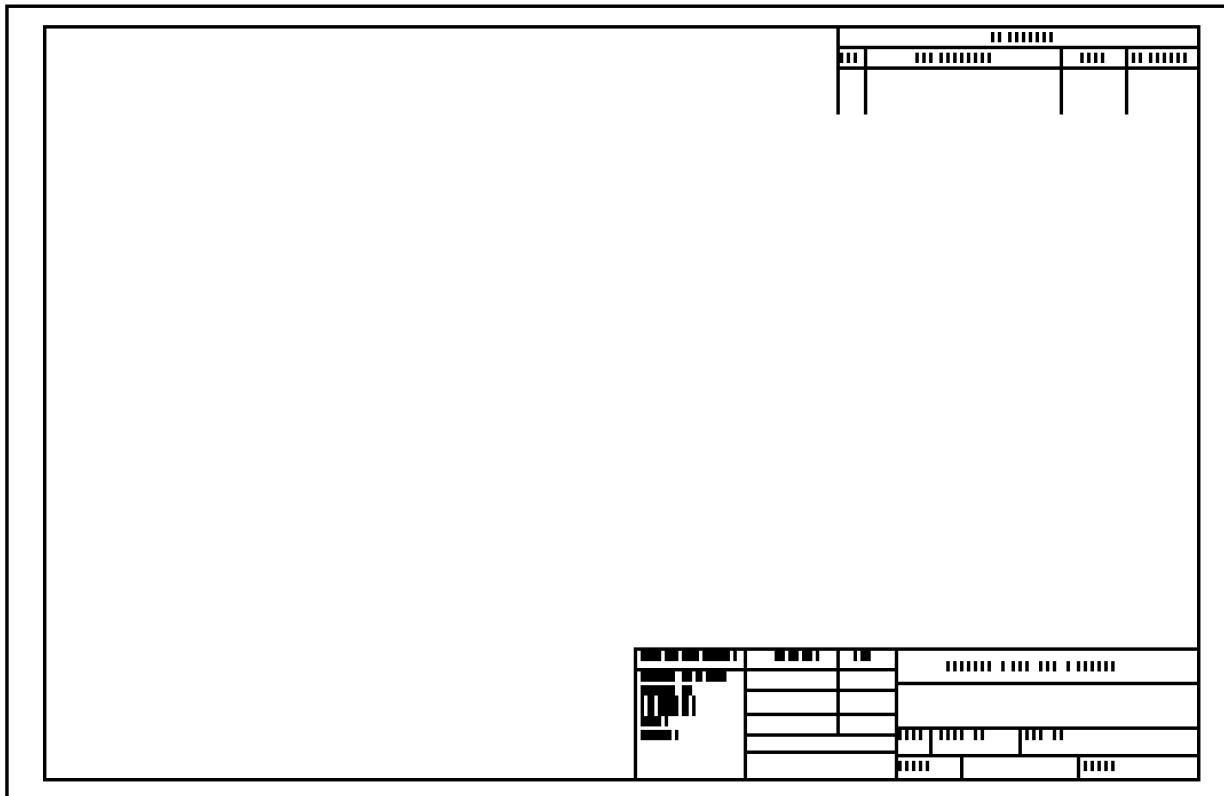




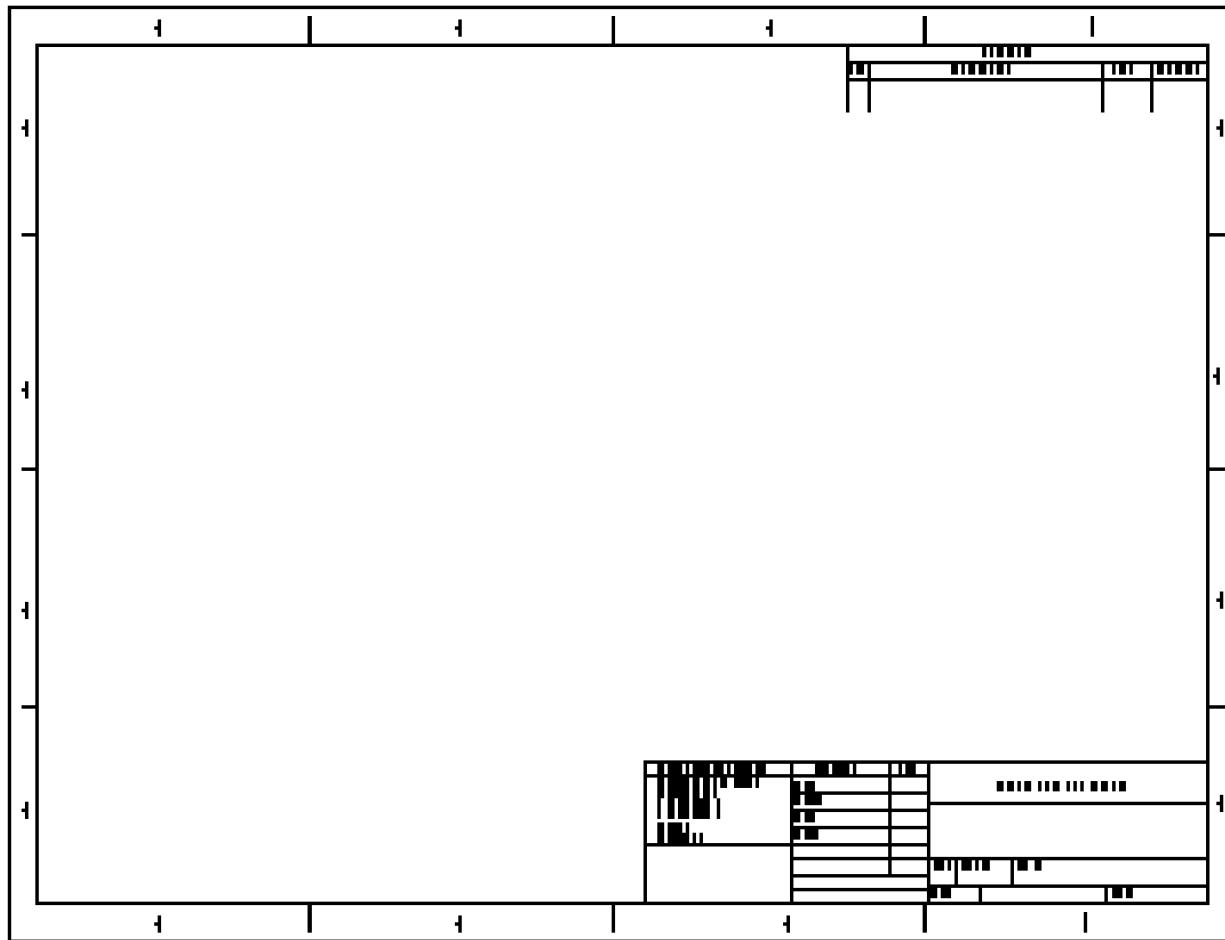
asizeh



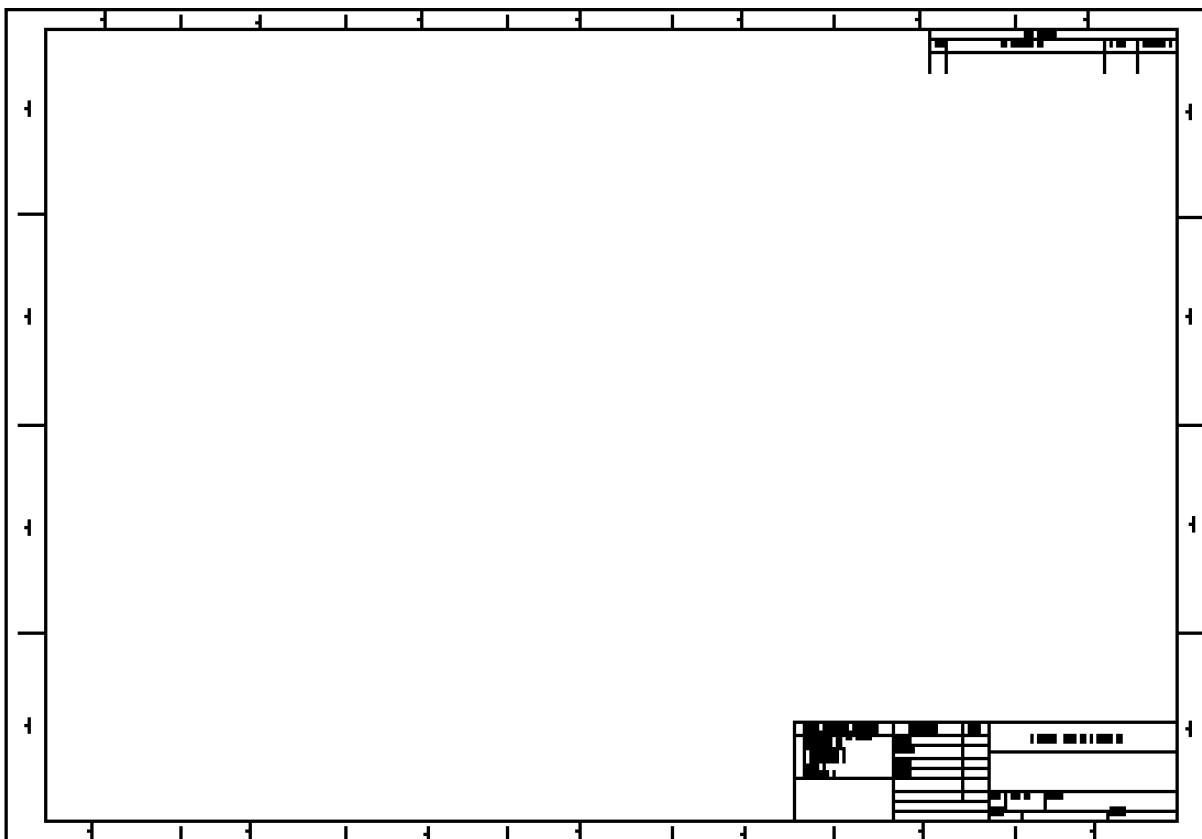
bsize



csize



dsize



esize

