System Designer Programmers Reference Manual

Database-Driven Hierarchical Design Tools

Cohesion Systems Inc. reserves the right to change any products described herein at any time and without notice. All data, circuits, and/or designs included in this publication (collectively called the "Program") are provided to you without warranty. Cohesion Systems makes no warranties whatsoever, express or implied, relating to the Design, and expressly excludes any warranty of merchantability, fitness for a particular purpose, or non-infringement of any proprietary rights of third parties.

You shall be solely responsible for obtaining all necessary licenses required to use the Program and shall indemnify, defend, and hold Cohesion Systems harmless from any and all liability, loss, costs, damage, judgment, or expense (including attorneys' fees and costs) resulting from or arising out of your manufacture, use, and sale or your customer's use or resale of products using any part of the Program that results in infringement of any third party patents, copyrights, or other proprietary rights.

Any use of the Program shall constitute your acceptance of all the above terms and conditions.

Trademark Acknowledgment:
Cohesion Systems and the corporate logo are registered trademarks of Cohesion Systems, Inc. All other trademarks are the property of their respective companies.

Contact Information
Contact us at:

Cohesion Systems Inc
1 Montelena Court
Woodside, California
USA 94062

TEL: (1) 650-994-4554
FAX: (1) 208-361-5627

sales@cohesionsystems.com

Product / New User Information
michaels@cohesionsystems.com

Sales

sales@cohesionsystems.com
General Company Information

info@cohesionsystems.com

User Support
support@cohesionsystems.com

Preface

This document is the primary source of technical information about the Cohesion Systems Systems Designer Program.

Audience

This manual is intended for:

- System designers and managers who are evaluating the Cohesion Systems Systems Designer Program for possible use.

Organization

This manual is divided into these chapters:

- Chapter 1, API Kit Programmers Reference
- Chapter 2, Hierarchy Data
- Chapter 3, Symbol and Schematic Data Extraction
- Chapter 4, Process Reference

Conventions

Please note the following notation examples and conventions that are used in this manual:

Examples/ Conventions	Explanation
0x1C3	"0x" prefix indicates a hexadecimal number.
11011_2	"2"subscript indicates a binary number.
RESERVED or *Res*	Indicates bit fields within registers that are not defined. These bits are used by this product's internal microcode, and modifying these bits should be done cautiously. If the register is modified during operation, the host should perform a Read-Modify-Write operation to preserve the state of the reserved bits. Writing the incorrect value to a RESERVED register bit causes indeterminate behavior. In addition, all registers and configuration parameters in DRAM that are not explicitly given names are also RESERVED, and accessing these registers may cause indeterminate results on current or future implementations of this product.
return();	C-style syntax presented in Courier typeface represents program pseudocode and equations.
vbv_delay	Names presented in Courier typeface represent names of items within the MPEG bitstream taken from the MPEG standard.
Top, bottom field	This manual uses the MPEG standard terms "top" and "bottom" to refer to the video fields. The top field can also be referred to as the first, or the even field. The bottom field can also be referred to as the second, or the odd field.
MSB, LSB	Most significant byte, least significant byte.

Related Publications

Cohesion Systems publishes additional documents related to this product. Request the latest information from your Cohesion Systems field applications engineer.

Table of Contents

Chapter 1
API Kit Programmer's Reference

This document is intended as a reference manual for EDA support programmers only. This module is not provided for typical Designer users. The API kit is a separate module available with a site license.

The license agreement for the API Kit permits you to copy any of the source code into your own processors, and circulate an unlimited number of copies of these processors within your company or organization. You are not allowed to circulate processors created with the API Kit outside of your company or organization without written permission from *Cohesion Systems, Inc.*

This document describes the features and operation of the API Kit used in the Hierarchical Design System (Designer). The API Kit allows you to access the Designer database using 'C' language calls. You can create your own custom interfaces using the API Kit.

Contents of the API Kit

The distribution contains three directories, outlined below:

- \LIB Contains the library files
- \INCLUDE Contains header files needed to be included by the processors to compile correctly.
- \SAMPLES Contain four examples of processors built with the API Kit. Each processor has one or more source files.

The API Kit provides the software functions you need to create programs that interface with the Hierarchical Design System. The functions are written in the 'C' language and compiled into libraries that may be linked with your application code.

Installation Instructions

Use the following procedure to ensure successful builds of the sample processors that use the Designer API Kit.

1. Copy the files from the distribution media into one directory on your system. These files will be linked with your compiled code.
2. Include this directory in the build settings for linking.
3. Once an executable is built, move it to the Designer install directory. Include a call to the executable from your Schematic Editor or Hierarchy Navigator menus by configuring the active ecs.ini or other .ini file.

The example processor can now be run directly from within the Designer environment.

Data Extraction

One of the primary advantages of the Designer Schematic Capture Package is that is is database-driven rather than connection-driven as typical schematic capture packages usually are. The advantage to this approach is that you can write custom programs using the utility functions that we have provided to link the schematic database into your appication

Hierarchy Data Extraction

This library contains the functions that interact with the Hierarchy Navigator's Hierarchical Data Structures. Applications that use this library can be launched from the any of the “Customizable” menus in the Hierarchy Navigator (i.e., Template, DRC, Tools, PCB). These applications run as child processes of the Hierarchy Navigator and depend on its presence. If the Hierarchy Navigator is not currently running when invoking one of these applications, Designer will automatically invoke the Hierarchy Navigator first (minimized) before beginning to run.

Applications treat the design as a single structure and are able to traverse the entire logical database. The logical database flattens buses and iterated symbol instances so that the resolution to individual elements has already been performed. Access to the graphical information of the schematic data files is not available in this mode.

This interface kit provides access to the logical structure of the design. It has been used to build the netlist extraction programs in the Designer.

Symbol & Schematic Data Extraction

This library of functions provides access to the data structures contained in the Symbol (.sym) and Schematic (.sch) files. Applications that use this library are self-contained. They can either be run as stand-alone processes or be launched by the Hierarchy Navigator, or from within the Schematic Editor or Symbol Editor.

The data is presented as it appears in the schematic or symbol. Buses and iterated symbol instances are not flattened. The application functions must

process these structures as appropriate. Access to the graphical as well as logical data is available in this interface.

The intended applications for this kit include graphical database export and netlist extraction where the bussing structure is to be preserved. The Designer programs for VHDL netlist extraction as well as the ASCOUT programs are built with this kit.

Utility Functions

Several utility functions are included with the data extraction functions for convenience.

void **AddExt**(char **name*, const char **ext*)

Removes any existing file extension from the *name* and replaces it with the extension specified in the *ext* parameter. A null string ("") removes the extension including the '.'. The *ext* should not exceed three characters after the period.

char ***FileInPath**(const char **path_name*)

This function returns a pointer to the file portion of *path_name*. It skips over any part of the *path_name* which represents the directory name.

char ***GetIntlDateTimeString**(char **buff*)

This function creates a formatted string in *buff* with the current date and time expressed in the correct local format.

int **MajorError**(const char **string*)

Displays an alert prompt showing the given *string.* The function waits until the user clicks the OK button before proceeding.

int **SpawnTask**(const char **program*, const char **command_line*, int *wait*)

Launches the selected *program* with the *command_line* as arguments. The *wait* variable is not used but is kept as a parameter for backward-compatibility, so this can be set to zero.

int **SysError**(char **message*)

Reports the *message* using an alert box technique. When the user acknowledges the message, the function causes an exit. This is intended for severe errors which prohibit further processing.

Licensing and Version Control

Beginning with Designer versions v3.0 (UNIX) and v4.4 (PC), new capabilities have been added to the API Kit to allow developers to control Licensing and Version compatibility. When an application is developed with the API Kit, settings can be made to force the application to only operate with certain run-time licenses or certain versions of Designer. Alternatively, the application may ignore these settings.

These features allow tool authors to control legacy or compatibility issues, especially when new features are added to future versions of Designer.

License Control

In the API Kit binaries, a reference is made to:

extern unsigned long **ProcRequiredLicBit**

This is a 32-bit word that indicates which Designer license level will be allowed to run the API Kit application. Normally, Designer runs with a single license level, checked-out from the license manager just once when Designer is invoked. All subsequent children processes, such as API Kit applications, inherit the same license word. For a API Kit application to run, at least one of the run-time license bits must match the **ProcRequiredLicBit** definition.

In the API Kit application, the definition of this word must be made:

unsigned long **ProcRequiredLicBit** = <32-BitMaskWord>;

If you want to bind your API Kit application to one or more specific Designer run-time licenses, please contact Cohesion Systems for the current license definitions. Otherwise, you can enable your API Kit application to run with all Designer license types with the **ProcRequiredLicBit** set to 0xFFFFFFFF.

Version Control

In the API Kit binaries, a reference is made to:

extern unsigned long **ProcRequiredVersion**

This is a 32-bit word that indicates which Designer software version will be allowed to run the API Kit application. With Designer version numbers, the lowest digit is ignored. This normally allows up to 10 "patch" releases before a version is considered to be obsolete. Thus, when a API Kit application is invoked, the version is checked against all digits except the lowest. The version is an integer, with two decimal places implied.

In the API Kit application, the definition of this word must be made:

unsigned long **ProcRequiredVersion** = <32-BitVersionWord>;

If you want to bind your API Kit application to a specific Designer version, you would use that version number here. Example:

unsigned long **ProcRequiredVersion** = 300; // Allow to run with version 3.0x

If instead, you want the API Kit application to run with all versions of Designer, you would set **ProcRequiredVersion** to 0.

Chapter 2
Hierarchy Data

This library contains the functions that interact with the Hierarchy Navigator's Hierarchical Data Structures. Applications that use this library can be launched from the any of the “Customizable” menus in the Hierarchy Navigator (i.e., Template, DRC, Tools, PCB). These applications run as child processes of the Hierarchy Navigator and depend on its presence. If the Hierarchy Navigator is not currently running when invoking one of these applications, it will automatically invoke the Hierarchy Navigator first (minimized) before beginning to run.

Applications treat the design as a single structure and are able to traverse the entire logical database. The logical database flattens buses and iterated symbol instances so that the resolution to individual elements has already been performed. Access to the graphical information of the schematic data files is not available in this mode.

This interface kit provides access to the logical structure of the design. It has been used to build the netlist extraction programs in the Designer.

Prodecural Interface - Hierarchy Data Extraction

The procedural interface to the Hierarchy Navigator's hierarchy data structure provides the means to build tools and processors that work with the logical data description of the design. These tools are linked to the Hierarchy Navigator and appear on the Hierarchy Navigator's menus. They are

launched by the Hierarchy Navigator and they run as child processes of the Hierarchy Navigator.

This interface is used to build the netlist extraction processors that are included in the Hierarchical Design System. Flattened netlist processors, such as spicenet, as well as hierarchical netlist processors, such as hspicent and silosnet are built with this interface.

This interface is also used to build analytic tools such as the checkckt program.

The routines comprising this interface are contained in the API Kit object library that is located in the appropriate API Kit directory or folder of the API Kit. This directory contains the platform-dependent files that are needed to build a API Kit application.

Several key concepts used in implementing this interface are:

1. Each element in the database is referenced by a handle. The handle is an unsigned long data element that does not have any numeric significance. The handles are the means by which the various routines interact with the database.

2. The data traversal routines are written in a call-forward style. When the traversal routine is called, it is provided with a function that it should call for each item encountered in the traversal. The traversal routine will scan the database and call the specified routine for each qualified item. After the last item is encountered, the traversal routine will return to the calling routine.

   As the database is traversed, the User_Functions are called and passed handles to the elements being accessed. These handles are used to extract data as well as to provide the starting point for a lower-level traversal.

   The called function returns a Boolean value to indicate whether or not the traversal should stop. Normally, the return value is FALSE. The TRUE return might be used to stop a traversal that was searching for a

particular item after the item was found. The returned value will be passed back to the function that initiated the traversal.

3. The data extraction routines are typically called to extract the data about the item presently encountered in a traversal. The routines return with a handle, data value, pointer to a data string, or pointer to a data structure. When a pointer is returned, the calling routine should consider the structure or string to be read-only. The returned data structure or string is a static structure that will be replaced when the next call to the data extraction routine is made.

4. The routines for adding attribute information to the symbols and schematics use the handles to identify the item. Attributes added to an item over-write the previous value for that item.

Global Variables

There are several global variables provided in the interface:

unsigned int **feature_code**

This variable must be declared in the Application. It is used by the API Kit main functions to restrict access to the API Kit application based on the user's license. Normally, this variable should be set to 0, allowing the application to run with any license.

unsigned int **feature_version**

This variable must be declared in the Application. It is used by the API Kit main functions to restrict access to the API Kit application based on the user's license. Normally, this variable should be set to 0, allowing the application to run with any license.

unsigned long **command_flags**

This variable is set by the API Kit front end to represent the control flags passed on the command line. The command line flags are chosen by the user when adding the program to one of the customizable menus. Each bit in the variable represents the first letter of the control flag. The low bit of the low word would represent a flag of -A.

Typical code to test these bits could be:

```
J_Flag = (int)( command_flags >> ( 'J' - 'A' ) ) & 1;
```

char **szRootName**[]

This variable contains the name of the root schematic.

TD_PTR **Root_TD**

This variable contains the Descriptor Handle of the root schematic (See below for a description of TD_PTR).

The Process

Applications built with this interface are divided into three functions. The first and third functions are optional and are intended to provide hooks onto which various user interface and analytic functions can be attached. If either or both of these functions are not provided, dummy routines will be linked from the object library.

The three functions that can be written as part of the application are:

int **PreProcess**(int *argc*, char **argv[]*)

This function is typically is used for setup of the process. The **command_flags** have already been processed and can be accessed. The **command_flags** represent only the flags that were specified as a single character following a forward slash or dash. The remaining arguments are available in the *argv* array. The first argument is not meaningful and the last argument is usually the name of the design. Data extraction and processing is not possible since the hierarchy data structures are not available to this function.

The dialog box interface used in the spicenet processors are typical examples of the functions that might be performed under the **PreProcess** function. The **command_flags** are used to pre-select the control parameters and the dialog box that is opened permits the user to override the normal defaults.

The OK and Cancel buttons give the user the means of continuing or stopping the process. A non-zero return value is an indication that the process is to be stopped.

void **Process**(int *argc*, char **argv[]*)

This is the main processing function in the application. Prior to calling this function the interface attaches the Hierarchy database and prepares to access the data. The data extraction and analysis work is performed as part of this function. The **command_flags** have already been processed and can be accessed. The **command_flags** represent only the flags that were speci-

fied as a single character following a forward slash or dash. The remaining arguments are available in the *argv* array. The first argument is not meaningful and the last argument is usually the name of the design.

When this function returns, the interface releases the data structures prior to calling the last application function (**PostProcess**).

void **PostProcess**(void)

This function can be used to report the process results to the user by opening an dialog box. It can also use the data extracted in the process step to do further analysis or simulation of the circuit. However, when this function is called the design data and traversal functions are no longer available.

Data Types

Several data types are used to pass data between the interface and the user's application. Most of the data types are represented by handles. The handles are defined as unsigned long integers. A NULL handle is not a valid handle. The following handle data types are defined:

Table 1: Data Handle Types

Typedef	Use	Represents
TD_PTR	Descriptor Handle	The basic schematic and symbol information
TI_PTR	Instance Handle	The instantiation of a symbol in a schematic
TN_PTR	Net Handle	The net within the schematic
TP_PTR	Pin Handle	The instance of a pin in the schematic
TG_PTR	Generic Pin Handle	The description of the original pin
TA_PTR	Attribute Handle	An attribute of a symbol, pin or net

Context

There are two different methods for traversing the design database. The flattening traversal functions visit each instance of the design in its full hierarchical context. This method is useful for producing flat netlists that require a complete and exact description of the design including all of the instance-specific attributes. The non-flattening traversal functions visit each schematic used in the design in its local context. This models each schematic as if it were the root of the design. Since the non-flattening traversal functions visit each block once, they are used to create hierarchical netlists. The context, or path, to the element is maintained by the traversal routines.

Attributes

Each element of the design has a set of attributes associated with it. Default values for symbol and pin attributes can be assigned in the symbol editor.

When the symbols are placed in the schematic, the Schematic Editor provides commands for adding and/or overriding the default attribute values. The Schematic Editor can also assign attributes to the nets that interconnect the symbols.

A schematic can appear multiple times in the design hierarchy. Initially, every instance of a given schematic will have the same values for the attributes as were defined in the Schematic Editor. The Hierarchy Navigator provides commands to add or override attributes on an instance-specific basis.

An element in the design can be viewed in either its local context or its hierarchical context.

Local context - The view is of the original symbol and schematic as defined in the respective editors. Any instance-specific attribute overrides that were added in the Hierarchy Navigator are not visible. The Hierarchy Navigator assigned values are ignored (except when processing the root schematic) and the Schematic or Symbol Editor assigned values are returned.

Hierarchical context - The symbol or schematic is viewed as an instance in the design. All attribute values are on an instance-specific basis. Attributes that depend on values from other levels in the hierarchy, including the net names on external signals and symbol pins, are mapped into the schematic. The order of precedence is to return the Hierarchy Navigator assigned values if any, then the schematic editor assigned values, and finally if no other values are present, the default values (for symbol and pin attributes) that were assigned to the symbol.

Attribute Numbers

Each of the attribute routines access attributes using an attribute number. The attribute numbers that are pre-assigned in the Designer system are

listed in the header file **attr.h**. Each attribute is "#defined" to a mnemonic that is used in the code. User-defined attributes should have numbers in the range of 100-199. Note that attribute numbers are used internally but the user sees the names of the attributes as they were assigned by the Setup program.

Attribute Routines

The following functions access the attributes in the design according to the context in which the view is being viewed. If no value is assigned, these routines all return a pointer to a NULL string (""). Note that sometimes the value returned is only a pointer to a temporary buffer that can be overridden by the next call to one of the attribute functions.

char ***Get_TDA**(TD_PTR *descriptor*, int *attr_num*)

Retrieves the default value (assigned by the Symbol Editor) for the symbol attribute *attr_num* of the given *descriptor*. Note, when *attr_num* represents a derived attribute this function will return the format string (as it was assigned by the Symbol Editor) in its raw form.

char ***Get_TIA**(TI_PTR *instance*, int *attr_num*)

Retrieves the symbol attribute *attr_num* for the given symbol *instance*. The value returned is a pointer to the character string value of the attribute.

For instance names or derived attributes, the pointer returned is the address of a temporary buffer that contains the attribute value. To save this value, copy it from this temporary buffer to a permanent storage area.

char ***Get_TNA**(TN_PTR *net*, int *attr_num*)

Retrieves the net attribute *attr_num* for the given *net*. Net attributes entered in the Hierarchy Navigator are assigned to the segment of the net that appears at the highest level of the hierarchy. This corresponds

to the segment in the schematic that is a local net. Attributes assigned to nets in the schematic editor should not be assigned to global or external nets.

If working in the hierarchical context, use the function **FindNetRoot** to find the highest level net segment. The pointer returned for net name attributes is the address of a temporary buffer that contains the attribute value. To save this value, copy it from this temporary buffer to a permanent storage area.

char ***Get_TPA**(TP_PTR *pin*, int *attr_num*)

Retrieves the pin attribute *attr_num* for the given *pin*.

For derived attributes, the pointer returned is the address of a temporary buffer that contains the attribute value. To save this value, copy it from this temporary buffer to a permanent storage area. For PCB Pin Numbers, see the function GetPcbPinNumber().

char *GetPcbPinNumber(TP_PTR pin)

Retrieves the PINNUM attribute for a PCB device. Intended for PCB applications, this is a wrapper around Get_TPA() that filters the space-delimited pin numbers that can appear on Gate devices.

char ***Get_TGA**(TG_PTR *generic_pin*, int *attr_num*)

Retrieves the default value (assigned by the Symbol Editor) for the pin attribute *attr_num* of the given generic *pin*. The handle to the generic pin for a pin can be obtained from the function **GenericPinOfPin**.

char ***Get_TIA_Override**(TI_PTR *instance*, int *attr_num*)

Retrieves the override value for symbol attribute *attr_num* for the given symbol *instance*. If the value of this attribute has not been overridden, the function returns a pointer to a NULL string ("").

The pointer returned for derived attributes is the address of a temporary buffer that contains the attribute value. To save this value, copy it from this temporary buffer to a permanent storage area.

char ***Get_TNA_Override**(TN_PTR *net*, int *attr_num*)

Retrieves the override value for net attribute *attr_num* for the given *net*. If the value of this attribute has not been overridden, the function returns a pointer to a NULL string ("").

char ***Get_TPA_Override**(TP_PTR *pin*, int *attr_num*)

Retrieves the override value for pin attribute *attr_num* for the given *pin*. If the value of this attribute has not been overridden, the function returns a pointer to a NULL string ("").

The pointer returned for derived attributes is the address of a temporary buffer that contains the attribute value. To save this value, copy it from this temporary buffer to a permanent storage area.

char ***LocalInstanceName**(TI_PTR *instance*)

Retrieves the local instance name for the given symbol *instance*. The name will contain only the name that was assigned to the instance on the schematic. That is, there will be no leading instance path. This is similar to calling **Get_TIA** when viewing the hierarchy in local mode except that there will be no leading period.

char ***LocalNetName**(TN_PTR *net*)

Retrieves the local net name for the given *net* segment. The name will contain only the name that was assigned to the net on the schematic. That is, there will be no leading instance path. This is similar to calling **Get_TNA** when viewing the hierarchy in local mode.

char ***InstanceName**(TI_PTR *instance*)

Retrieves the instance name for the given symbol *instance*. This is similar to calling **Get_TIA** except that there will be a leading period.

char ***NetName**(TN_PTR *net*)

Retrieves the net name for the given *net* segment. This is identical to calling **Get_TNA**.

Attribute Range Routines

The following functions can be used to access a range of attributes and to find the name of a particular attribute.

int **ForEachTIA**(TI_PTR *instance*, int *attr_first*, int *attr_last*, int *mode*,
int (**User_Function*)(int the_num, const char *the_name, char *the_val))

Scans the attributes on the given symbol *instance* between numbers *attr_first* and *attr_last*.

The *mode* determines which level of attribute overrides to view:

Mode View Similar to:

1 Default values only **Get_TDA**.

2 Local value only **Get_TIA** in local view

4 Overrides only **Get_TIA_Override**

6 all **Get_TIA**

Each time an attribute in the range has a value, the *User_Function* is called with the number, name and value of the attribute.

int **ForEachTPA**(TP_PTR *pin*, int *attr_first*, int *attr_last*, int *mode*,
int (**User_Function*)(int the_num, const char *the_name, char *the_val))

Scans the attributes on the symbol-pin between the numbers *attr_first* and *attr_last.*

The *mode* determines which level of attribute overrides to view:

Mode View Similar to:

1 Default values only **Get_TGA**.

2 Local value only **Get_TPA** in local view

4 Overrides only **Get_TPA_Override**

6 All **Get_TPA**

Each time an attribute in the range has a value, the *User_Function* is called with the number, name and value of the attribute.

int **ForEachTNA**(TN_PTR *net*, int *attr_first*, int *attr_last*, int *mode*,
int (**User_Function*)(int the_num, const char *the_name, char *the_val))

Scans the attributes on the *net* between the numbers *attr_first* and *attr_last.*

The *mode* determines which level of attribute overrides to view:

Mode View Similar to:

2 Local value only **Get_TNA** in local view
4 Overrides only **Get_TNA_Override**
6 All **Get_TNA**

Each time an attribute in the range has a value, the *User_Function* is called with the number, name and value of the attribute.

Adding or Modifying Attributes

The API Kit application can need to add or modify the attribute values in the design. The following functions are used to update a local copy of the design database. In order for these attribute values to become permanently added to the design, a call to **UpdateTree** is required at the end of the API Kit Process.

int **Add_TDA**(TD_PTR *descriptor,* int *attr_num*, const char **value*)

int **Add_TIA**(TI_PTR *instance*, int *attr_num,* const char **value*)

int **Add_TNA**(TN_PTR *net,* int *attr_num*, const char **value*)

int **Add_TPA**(TP_PTR *pin*, int *attr_num*, const char **value*)

int **Add_TGA**(TG_PTR *generic_pin,* int *attr_num*, const char **value*)

Attribute Names

The user assigned names of the various attributes are defined in the ecs.ini file. The following routines translate between names and numbers.

const char *GetNetAttrName(int *Attrib_Number*)

Returns names for attributes that are associated with nets.

const char *GetPinAttrName(int *Attrib_Number*)

Returns names for attributes that are associated with symbol pins.

const char *GetSymAttrName(int *Attrib_Number*)

Returns names for attributes that are associated with symbol definitions and instances.

int **GetNetAttrNumber**(const char **attrib_name*)

Returns the number of an attribute that is associated with nets.

int **GetPinAttrNumber**(const char **attrib_name*)

Returns the number of an attribute that is associated with symbol pins.

int **GetSymAttrNumber**(const char **attrib_name*)

Returns the number of an attribute that is associated with symbols.

Accessing Relationships

The hierarchy data structure is a specialized relational database that contains the design. Within the database are several types of records that interrelate to form the structure. These records reflect the relationships that existed in the Schematic and Symbol descriptions that were used to build the database. There are several relationships that must be discovered in order to obtain the needed information about an element. The following routines provide that access:

TD_PTR **DescriptorContainingNet**(TN_PTR *net*)

Returns the handle to the Descriptor that represents the schematic on which the given *net* appears.

TD_PTR **DescriptorOfInstance**(TI_PTR *instance*)

Returns the handle to the Descriptor that represents the given symbol *instance*.

TN_PTR **FindNetRoot**(TN_PTR *net*)

Returns the handle to the master Net to which the given *net* segment is connected. This is used in the hierarchical context.

TI_PTR **FirstInstanceOf**(TD_PTR *descriptor*)

Returns the handle to the first Instance of the symbol represented by the given *descriptor.* This is useful for obtaining a typical instance of a symbol for extracting its attributes. Use the traversal function **ForEachInstance** to access all of the instances of a symbol.

TI_PTR **FirstPinOf**(TI_PTR *instance*)

Returns the handle to the first Pin of the given symbol *instance*. This is useful for obtaining a typical pin of a symbol for extracting its pin

attributes. Use the traversal function **ForEachInstancePin** to access all of the pins of a symbol instance.

TG_PTR **GenericPinOfPin**(TP_PTR *pin*)

Returns the handle to the Generic Pin that corresponds to the given *pin*. This is used to access the default attribute values of the *pin*.

TI_PTR **InstanceContainingPin**(TP_PTR *pin*)

Returns the handle to the Instance in which the given *pin* is contained.

TD_PTR **OwnerOfInstance**(TI_PTR *instance*)

Returns the handle to the Descriptor that represents the schematic in which the given symbol *instance* appears.

TN_PTR **NetContainingPin**(TP_PTR *pin*)

Returns the handle to the Net to which the given *pin* is connected. This is the net segment appearing in the same schematic that contains the *pin*. Use the **FindNetRoot** function to find the master net in which the *pin* appears.

TN_PTR **NetDefinedByPin**(TP_PTR *pin*)

Returns the handle to the Net in the schematic represented by the given *pin*.

TI_PTR **ParentInstanceOf**(TD_PTR *descriptor*)

Returns the handle to the parent Instance of the symbol represented by the given *descriptor*. The handle will be NULL if there is no parent instance. This can be used in the hierarchical context to obtain the parent instance of a symbol. Use the function **Get_TIA** to access the attributes of the parent instance.

TP_PTR **PinDefiningNet**(TN_PTR *net*)

Returns the handle to the Pin in the parent symbol that represents the given *net*. This function only works for nets that are external to the schematic. Use the function **NetLocExtGbl** to determine if a net is external to the schematic.

Database Element Numbers

The hierarchy database assigns unique numbers to each instance, net, and driving (output or bi-directional) pin in the design. No assumptions should be made about the sequencing of these numbers. These numbers can be obtained or used to find the element with the following functions:

unsigned long **InstanceNumber**(TI_PTR *instance*)

Returns the number of the given *instance*.

unsigned long **NetNumber**(TN_PTR *net*)

Returns the number of the given *net*.

unsigned long **PinNumber**(TP_PTR *pin*)

Returns the number of the given instance *pin*.

Searching For a Particular Element

Several functions are provided that find elements by their name or by one of their other attributes.

TD_PTR **FindDescriptorNamed**(const char **name*)

Returns the handle to the descriptor with the given *name.* Does not change hierarchical context.

TI_PTR **FindInstanceNamed**(const char **name*)

Returns the handle to the instance with the given *name.* The name is the full hierarchical name with dots '.' used as the delimiter between the level names. This function sets the context of the hierarchy view and should not be called within a hierarchical traversal unless the context is saved with **SavePath** and then restored later with **RestorePath**.

TI_PTR **FindInstanceNumbered**(unsigned long *number*)

Returns the handle to the instance that is numbered with the given *number.* This function sets the context of the hierarchy view and should not be called within a hierarchical traversal unless the context is saved with **SavePath** and then restored later with **RestorePath**.

TI_PTR FindInstanceRefNamed(const char **name*)

Returns the handle to the instance with the REFNAME attribute value equal to the given *name*. If the REFNAME attribute is followed by the gate name, the function will search for the instance that has both the given REFNAME value and which has the correct set of PINNUM attributes to correspond to the given gate. For example, U1/B will search for an instance whose REFNAME is U1 and whose PINNUM attributes correspond to the second gate in the package. This function sets the context of the hierarchy view and should not be called within a hierarchical traversal unless the context is saved with **SavePath** and then restored later with **RestorePath**.

TN_PTR **FindNetNamed**(const char **name*)

Returns the handle to the net segment with the given *name.* The function **FindNetRoot** should be used to find the master net handle. The name is the full hierarchical name with dots '.' used as the delimiter between the level names. This function sets the context of the hierarchy view and should not be called within a hierarchical traversal unless the context is saved with **SavePath** and then restored later with **RestorePath**.

TN_PTR **FindNetNumbered**(unsigned long *number*)

Returns the handle to the net that is numbered with the given *number.* This function sets the context of the hierarchy view and should not be called within a hierarchical traversal unless the context is saved with **SavePath** and then restored later with **RestorePath**.

TP_PTR **FindPinNamed**(const char **name*)

Returns the handle to the pin with the given *name.* The name is the full hierarchical name with dots '.' used as the delimiter between the level names. The last portion of the name is the name of the pin on the specified instance. Either a dot '.' or a minus sign '-' can be used as the separator between the instance portion of the name and the pin portion of the name. This function sets the context of the hierarchy view and should not be called within a hierarchical traversal unless the context is saved with **SavePath** and then restored later with **RestorePath**.

TP_PTR **FindPinWithAttribute**(TI_PTR *instance*, int *attr_num*, const char **value*)

Scans the pins of the given symbol *instance* until a pin is found having attribute *attr_num* with value equal to the given *value*. Note that case is ignored while searching for the match. The handle to the symbol instance pin is returned.

Accessing Parameters

int **DescriptorType**(TD_PTR *descriptor*)

Returns the type code of the given *descriptor* indicating the type of symbol defined. See the header file for the definition of the 'SY_' codes.

int **GateNumberOfInstance**(TI_PTR *instance*)

Used for Printed Circuit Board applications. The symbol *instance* must be of type "gate." This function returns the number of the gate within the package that contains the pin numbers that are assigned to the pins of the given instance. The gate numbers start at '0' for the first gate in the package.

int **GlobalPin**(TP_PTR *pin*)

Returns TRUE if the given *pin* is an artificial pin that was added to connect the global nets in the hierarchy. Otherwise it returns false.

int **NetInOutBid**(TN_PTR *net*)

Returns the classification of the given (external) *net* as being one of the following:

The classification is indicated by a value of:

- Input 1
- Output 2
- BiDir 3

This function returns 0 if the net is not external.

int **NetLocExtGbl**(TN_PTR *net*)

Returns the classification of the given *net* segment as being one of the following:

- Classification Indicated by Value of local to schematic 0
- external (represented by a pin) 1
- global in the design 2

int **PrimitiveCell**(TD_PTR *descriptor*)

Returns TRUE if there is not a schematic represented by the *descriptor* cell in the hierarchy. Otherwise it returns false.

Traversing the Data Structures - Local Context

There are several mechanisms for traversing the data structures. Each mechanism results in a different view of the data. The routines are designed to call the User_Function as it visits each element in the traversal. When the User_Function is called, it is passed a handle to the element that is being accessed.

The first group of functions traverse the data structures in a local context. As each element is visited, the traversal views that element as being at the root level of the hierarchy. This method is generally used when creating hierarchical netlists. The routines are:

int **ForEachDescriptor**(int (***User_Function*)(TD_PTR *the_desc*))

This routine visits each of the descriptors in the hierarchical database. When it calls the *User_Function*, it passes the handle to the descriptor being visited.

This routine could be used to extract a list of the different symbols that are in the design. Each symbol would be listed once even though it can have been used more than once in a schematic or in more that one schematic of the design. The code fragment for extracting this list would be:

```
int List_Type( TD_PTR td )
{
fprintf( file, "%s\n", Get_TDA( td, NAME ) );
return( FALSE );
}
void Process()
{
fprintf( file, "Types Used:\n" );
ForEachDescriptor( List_Type );
return;
}
```

This traversal will provide a listing that was ordered with the higher blocks first. It is often necessary to extract a listing in a bottom-up order. In hierarchical netlists, this corresponds to a "declare before used" order. The following routines are intended for that purpose.

void **SetupBlockScan**(void)

This function is called prior to performing the traversal. It clears all of the "done" marks in the data structure (see **MarkBlockDone**).

int **ForEachBlockOrCell**(int (***User_Function*)(TD_PTR *the_desc*))

This function traverses the data structure in a bottom-up order. As it visits each descriptor for the first time, it calls the *User_Function* passing the handle to the descriptor. The following example creates the "Types Used:" listing with each block listed after all of the blocks that it uses:

```
int List_Type( TD_PTR td )
{
fprintf( file, "%s\n", Get_TDA( td, NAME ) );
return( FALSE );
}
void Process()
{
fprintf( file, "Types Used:\n" );
SetupBlockScan();
ForEachBlockOrCell( List_Type );
return;
}
```

int **ForEachBlock**(int (***User_Function*)(TD_PTR the_desc))

This function performs the same task as the one above except that it does not call the *User_Function* when it visits a descriptor that does not have any subcircuits. This is the version that is typically used for creating hierarchical netlists such as silosnet and hspicent.

It is common that a branch of the design is represented in the simulator by a model. In this case, the symbol representing the model would have its "xxxModel" attribute set to the model name. The underlying circuitry should not be visited in the traversal since the model already represents that portion of the design.

void **MarkBlockDone**(TD_PTR *descriptor*)

This routine is used with the **ForEachBlock** to mark those blocks that are not to be visited. A code fragment that uses this function to avoid listing those sections represented by a SilosModel is:

```
int List_Type( TD_PTR td )
{
fprintf( file, "%s\n", Get_TDA( td, NAME ) );
return( FALSE );
}
int Check_For_Model( TD_PTR td )
{
if ( *Get_TDA( td, SILOSMODEL ) )
MarkBlockDone( td );
return( FALSE );
}
void Process()
{
fprintf( file, "Types Used:\n" );
SetupBlockScan();
ForEachDescriptor( Check_For_Model );
ForEachBlock( List_Type );
return;
}
int TestMark( TD_PTR descriptor )
```

This function returns TRUE if the given *descriptor* has been marked by the **MarkBlockDone** function.

A descriptor represents a schematic. In creating a netlist, the symbols on the schematic are listed along with the connections to each of their pins. This process involves nesting levels of traversal functions under the **ForEachBlockScan** or **ForEachDescriptor** traversal.

int **ForEachSubBlock**(TD_PTR *descriptor*, int (**User_Function*)(TI_PTR the_inst))

This function traverses each of the sub-blocks that are instantiated in the block represented by the *descriptor* parameter. The order in which the sub-blocks are visited is indeterminate. As each sub-block is visited, the Instance handle is passed to the *User_Function*.

int **ForEachInstancePin**(TI_PTR *instance*, int (**User_Function*)(TP_PTR the_pin))

Traverses each of the pins in the symbol *instance* and calls the *User_Function* with the handle of the pin. The order is the order of the pins in the symbol definition and will remain constant for each instance of a given symbol. See **SortPinsByOrder**.

Expanding on the 'List_Type' function of the example above to list the type and instance name and the nets connected to each of the pins of each sub-block in each descriptor:

```
int List_Instance_Pin( TP_PTR tp )
{
TN_PTR tn;
tn = NetContainingPin( tp );
fprintf( file, " %s,", NetName( tn ) );
return( FALSE );
}
int List_Sub( TI_PTR ti )
{
TD_PTR td;
```

```
td = DescriptorOfInstance( ti );
/* print symbol type then instance name */
fprintf( file, "%s ,", Get_TDA( td, NAME ) );
fprintf( file, "%s (", LocalInstanceName( ti ) );
ForEachInstancePin( ti, List_Instance_Pin );
fprintf( file, " )\n" );
return( FALSE );
}
int List_Type( TD_PTR td )
{
fprintf( file, "SubCkt %s\n", Get_TDA( td, NAME ) );
ForEachSubBlock( td, List_Sub );
return( FALSE );
}
```

Each subcircuit in a hierarchical description should have a list of the ports of the subcircuit listed with the subcircuit definition. This list should be in the same order as the list of the connections to the ports made in the symbol instances. This is extracted by using one of the instances of the circuits to look up the pins and their names. In the case of the top-level circuit, this cannot be possible if a symbol for the circuit does not exist. In this case, any order is acceptable, so a list of all external signals would work.

int ForEachBlockNet(TD_PTR *descriptor,* int (**User_Function*)(TN_PTR the_net))

Traverses the nets that are in the given *descriptor*'s schematic and calls the *User_Function* with the handle of each net.

This would expand the 'List_Type' function to:

```
int List_Block_Pin( TP_PTR tp )
{
fprintf( file, " %s,", Get_TPA( tp, NAME ) );
return( FALSE );
}
int Check_Block_Net( TN_PTR tn )
{
if ( NetLocExtGbl( tn ) == EXTERNAL_NET )
fprintf( file, "%s,", NetName( tn ) );
return( FALSE );
}
int List_Type( TD_PTR td )
{
TI_PTR ti;
fprintf( file, "SubCkt %s (", Get_TDA( td, NAME ) );
ti = FirstInstanceOf( td );
if ( ti ) ForEachInstancePin( ti, List_Block_Pin );
else ForEachBlockNet( ti, Check_Block_Net );
fprintf( file,")\n" );
ForEachSubBlock( td, List_Sub );
return( FALSE );
}
```

int **ForEachNetLocalPin**(TN_PTR *net*, int (**User_Function*)(TP_PTR the_pin))

This function traverses the given *net* in the local context and calls the *User_Function* for each pin connected in the net. This function can be called in either a local or hierarchical traversal.

The complete netlist program is:

```
FILE *file;
int List_Instance_Pin( tp )
{
TN_PTR tn;
tn = NetContainingPin( tp );
fprintf( file, " %s,", NetName( tn ) );
return( FALSE );
}
int List_Sub( TI_PTR ti )
{
TD_PTR td;
td = DescriptorOfInstance( ti );
/* print symbol type then instance name */
fprintf( file, " %s,", Get_TDA( td, NAME ) );
fprintf( file, " %s (", LocalInstanceName( ti ) );
ForEachInstancePin( ti, List_Instance_Pin );
fprintf( file, " )\n" );
return( FALSE );
}
int List_Block_Pin( TP_PTR tp )
{
TG_PTR tg;
tg = GenericPinOfPin( tp );
fprintf( file, " %s,", Get_TGA( tg, NAME ) );
return( FALSE );
}
int Check_Block_Net( TN_PTR tn )
{
if ( NetLocExtGbl( tn ) == EXTERNAL_NET )
fprintf( file, "%s,", NetName( tn ) );
```

```
return( FALSE );
}
int List_Type( TD_PTR td )
{
TI_PTR ti;
fprintf( file, "SubCkt %s (", Get_TDA( td, NAME ) );
ti = FirstInstanceOf( td );
if ( ti ) ForEachInstancePin( ti, List_Block_Pin );
else ForEachBlockNet( td, Check_Block_Net );
fprintf( file,")\n" );
ForEachSubBlock( td, List_Sub );
fprintf( file, "EndSub\n" );
return( FALSE );
}
int Check_For_Model( TD_PTR td )
{
if ( *Get_TDA( td, xxxMODEL ) )
MarkBlockDone( td );
return( FALSE );
}
void Process()
{
char filename[40];
sprintf( filename, "%s.xxx,", szRootName );
file = fopen( filename, "w" );
if ( file )
{
SetupBlockScan();
ForEachDescriptor( Check_For_Model );
ForEachBlock( List_Type );
fclose( file );
}
return;
}
```

The preceding functions visit each element in the hierarchy once. They are used to create netlists that describe the circuit as a hierarchy of circuits con-

taining subcircuits. This is a more compact description of a design where the same elements are repeated in the hierarchy. The main disadvantage is the lack of definition for attaching the instance-specific attribute values.

Traversing the Data Structures - Hierarchical Context

The next functions traverse the data structures in the hierarchical context visiting each instance in the hierarchy. The traversal function calls the User_Function for each instance visited in the hierarchy. This results in a flattened representation of the design. This is often useful when attributes are to be attached to the specific instances in the design.

During these traversals, the concept of a context is maintained. A current path, which depicts the path to the schematic that contains the instance, is maintained. If it is necessary to jump to a different instance in the hierarchy, this path must be saved by the User_Function and restored before the User_Function returns to the traversal function.

void **SavePath**(void)

Saves the current path. This function maintains a single save buffer. Repeated calls will destroy the previously saved path.

void **RestorePath**(void)

Restores the last saved path.

The application can choose between two traversal orders. The Net order traversal visits each net in the design. This traversal order is typically used when the design is to be viewed by net as in a typical point-to-point netlist.

Each net is visited only once at its highest point. This is either in the root schematic or in the schematic in which the net is local.

int **ForEachNet**(int (***User_Function*)(TN_PTR the_net))

As each net is visited, the *User_Function* is called and passed the handle to the net. The routine only visits the highest level of each net so the **FindNetRoot** function is not needed to find the master.

The following code fragment will produce a list of all the net names in a design:

```
int List_Net( TN_PTR tn )
{
fprintf( file, "%s\n", NetName( tn ) );
return;
}
void Process()
{
ForEachNet( List_Net );
return;
}
```

int ForEachNetPin(TN_PTR *net*, int (**User_Function*)(TP_PTR the_pin))

This function traverses the given *net* and calls the *User_Function* for each pin connected in the net.

The previous example can be expanded to provide a full netlist. This is essentially the same program supplied as source code in the API Kit.

```
int List_Net_Pin( TP_PTR tp )
{
TI_PTR ti;
ti = InstanceContainingPin( tp );
fprintf( file, " %s", InstanceName( ti ) );
fprintf( file, "-%s,", Get_TPA( tp, NAME ) );
return(0);
}
int List_Net( TN_PTR tn )
{
fprintf( file, "%s,", NetName( tn ) );
ForEachNetPin( tn, List_Net_Pin );
fprintf( file, "\n" );
return(0);
```

```
}
void Process()
{
ForEachNet( List_Net );
return;
}
```

A similar set of functions are available for traversing the data by symbol and pin.

int ForEachInstance(int (*_User_Function_)(TI_PTR the_inst))

This function traverses the entire design visiting each block and cell in the hierarchy. It calls the *User_Function* at each element it visits.

int **ForEachPrimitiveInstance**(int (*_User_Function_)(TI_PTR the_inst))

This function traverses the entire design visiting each lower-level cell in the hierarchy. It calls the *User_Function* each time it encounters a leaf cell.

The pinorder netlist program is an example of this type of traversal. A simple version of that program would be:

```
FILE *file;
int List_Inst_Pin( TP_PTR tp )
{
TN_PTR tn;
tn = FindNetRoot( NetContainingPin( tp ) );
fprintf( file, "%s ,", Get_TPA( tp, NAME ) );
fprintf( file, "%s\n", NetName( tn ) );
return(0);
}
int List_Instance( TI_PTR ti )
{
TD_PTR td;
```

```
td = DescriptorOfInstance( ti );
fprintf( file, "%s,", Get_TDA( td, NAME ) );
fprintf( file, " %s\n", InstanceName( ti ) );
ForEachInstancePin( ti, List_Inst_Pin );
fprintf( file, "\n" );
return(0);
}
void Process()
{
char filename[40];
sprintf( filename, "%s.xxx,", szRootName );
file = fopen( filename, "w" );
if ( file )
{
ForEachPrimitiveInstance( List_Instance );
}
return;
}
```

Hierarchy Recursion Functions

Several hierarchy recursion functions are included for processing an entire design.

int **hierSetRootBlock** (char **name* , int *data_len*)

This function initializes the process and records the root schematic by *name*. The function also establishes a buffer of 'data_len' bytes that will be attached to each of the symbol/schematic nodes processed. This buffer can be used to retain information about the individual schematics processed. This function always returns FALSE.

int hierAddSubBlocks(void)

This function traverses the symbol list for the currently loaded schematic and creates the instantiation records for each symbol of gate, block, cell or component type. This function always returns FALSE.

char *hierGetNextBlock(void)

This function returns the name of the next block to process. It is up to the calling process to load the actual schematic and symbol data as needed. This function returns (char *)NULL when there are no more blocks to process.

int **hierForEachBlock** (int (**User_Function*)(const char *name))

This function traverses each of the blocks in a "define before use" order. As each block is encountered, the *User_Function* is called and passed the name of the block. The process continues until the list of blocks is exhausted or the *User_Function* returns a non-zero value. The function returns 0 if the list of blocks was completed and returns the *User_Function* return value if it was stopped.

int **hierForEachInstance**(int (**User_Function*)(char *path))

This function traverses each instance in the hierarchy and calls the specified *User_Function*. When the call is made, the function is passed the full hierarchical name of the instance. The process continues until each instance in the hierarchy has been visited or the *User_Function* returns a non-zero value. The function returns 0 if the list was completed and returns the *User_Function* return value if it was stopped.

char **hierPathSeparator**[2] = "."

This global variable is used to separate the instance names in the hierarchical path. The calling program can change this character as needed.

Several hierarchy recursion functions are included for processing an entire design. The traversal functions **hierGetNextBlock**, **hierForEachBlock** and **hierForEachInstance** maintain a notion of a current block. This block represents a symbol/schematic pair and has a data buffer associated with the data record. The contents of this buffer can be set and retrieved with the following functions;

int **hierSetData**(char **data*)

This function copies the contents of *data* into the buffer reserved for each block type. The *data_len* argument of **hierSetRootBlock** establishes the length of the *data* copied. With appropriate casting on the calling end, data can be any list or structure. This function always returns FALSE.

int **hierGetData**(char **data*)

This function copies the saved data for the current block into *data*. The *data_len* argument of **hierSetRootBlock** establishes the length of the *data* copied. This function always returns FALSE.

Utility Functions

int **UpdateHierarchy**(const char ***filename*, int *save*)

This command tells the Hierarchy Navigator to read a file containing attribute overrides. The Hierarchy Navigator will add the attribute values to the Hierarchy Database. The *save* parameter, if TRUE, instructs the Hierarchy Navigator to make the values a permanent part of the database.

int **UpdateTree**(int *repaint*)

This function is typically called at the end of the **Process** to cause the Hierarchy Navigator to accept any changes that have been made to the data structures as a result of the process. It is mainly used in back-annotation types of processes where attributes were added or modified in the data structures.

The *repaint* parameter if TRUE, tells the Hierarchy Navigator to repaint the schematic views to reflect the new attribute values.

int **SortPinsByOrder**(int *attrib_number*)

This function may be called before doing any traversals to sort the pins of all symbols based on the value of the given *attrib_number*. This function will affect the order that the pins are traversed in **ForEachInstancePin.** The attribute values are treated as integers.

The following three functions are used to control and interrogate a flag that is attached to each descriptor.

int **ClearDescriptorFlag**(TD_PTR *descriptor*)

int **GetDescriptorFlag**(TD_PTR *descriptor*)

int **SetDescriptorFlag**(TD_PTR *descriptor*)

The **ClearDescriptorFlag** and **SetDescriptorFlag** functions are designed so that they can be the User_Function in a traversal of the descriptors, that is:

ForEachDescriptor(ClearDescriptorFlag);

int **ForEachGlobalNetName**(int (**User_Function*)(const char *the_name))

Parses the list of global net names and calls the *User_Function* with the name of the global net.

int **MajorError**(const char **string*)

Displays an alert prompt showing the given *string.* The function waits until the user clicks the OK button before proceeding.

Chapter 3
Symbol and Schematic Data Extraction

This library of functions provides access to the data structures contained in the Symbol (.sym) and Schematic (.sch) files. Applications that use this library are self-contained. They can either be run as stand-alone processes or be launched by the Hierarchy Navigator, or from within the Schematic Editor or Symbol Editor.

The data is presented as it appears in the schematic or symbol. Buses and iterated symbol instances are not flattened. The application functions must process these structures as appropriate. Access to the graphical as well as logical data is available in this interface.

The intended applications for this kit include graphical database export and netlist extraction where the bussing structure is to be preserved. The Designer programs for VHDL netlist extraction as well as the ASCOUT programs are built with this kit.

Procedural Interface - Symbol & Schematic Data Extraction

The procedural interface to the symbol and schematic databases provides a means extracting the graphical and structural data. The interface consists of several database traversal routines combined with a set of data extraction routines.

A set of routines is also included in this interface which permits attribute values to be assigned in the symbol and schematic data files. These can be

used for building library maintenance utilities and for creating a schematic level (as opposed to hierarchy level) back annotation interface.

The routines comprising the interface are contained in the scpik object library which is located in the appropriate API KIT directory or folder of the API Kit.

The key concepts used in implementing this interface are:

1. Each element in the database is referenced by a handle. The handle is an unsigned short data element which does not have any numeric significance. The handles are the means by which the various routines interact with the database.

2. The data traversal routines are written in a call-forward style. When the traversal routine is called, it is provided with the function which it should call for each item encountered in the traversal. The traversal routine will scan the database and call the specified routine for each qualified item. After the last item is encountered, the traversal routine will return to the calling routine.

   As the database is traversed, the User_Functions are called and passed handles to the elements being accessed. These handles are used to extract data as well as to provide the starting point for a lower level traversal.

   The called function returns a Boolean value to indicate if the traversal should stop. Normally, the return value is FALSE. The TRUE return might be used in a traversal which was searching for a particular item and the item was found. The returned value will be passed back to the function which initiated the traversal.

3. The data extraction routines are typically called to extract the data about the item presently encountered in a traversal. The routines return with a handle, data value, pointer to a data string, or pointer to a data structure. When a pointer is returned, the calling routine should consider the structure or string to be read-only. The returned data structure or string is a static structure which can be modified when the next call to the data extraction routine is made.

4. The routines for adding attribute information to the symbols and schematics use handles to identify the item. Attributes added to an item over-write the previous value for that item.

Coordinate System

The schematic and symbol data is based on a gridded coordinate system. The coordinate system has its origin in the upper-left corner of the schematic sheet and at the origin of the symbol. Positive X is to the right and positive Y is down. All coordinate values are expressed in terms of grid units.

Graphic elements can be on any grid unit. Circuit elements, including symbol pins and origins, symbol instances and wire elements are constrained to grid units that are multiples of 16.

Data Types

There are several data types which are used to pass data between the interface and the user's application. The first class of data type is the item handles. All handles are defined as unsigned short integers and will be in the range of 1 through 65,535. A NULL handle is not a valid handle. The following handle data types are defined:

typedef use
NT_PTR net_handle
BR_PTR branch_handle
ST_PTR symbol_type_handle
SI_PTR symbol_instance_handle
SP_PTR symbol_pin_handle
TB_PTR table_handle
PN_PTR pin_handle (from symbol definition)

There are also a few structures that are defined for passing data to the call function.

```
struct _gr_item { Describes Graphic Line Items
short type; Type of the item ( see GR_ in include file )
short width; 0 = Normal, 1 = Heavy line weight
short style; 0= Normal, 1= dash, 2= dot, 3= dashdot, 4= dashdotdot
int x[4], y[4]; coordinates
};

struct _gr_text { Defines Graphic Text Items
int x, y; Origin of the text string
short font; 0 = Small, 1 = Medium, 2 = Large
short rot; 0 = Horizontal, 1 = Rotate 90 degrees
short just; Justification - 1 through 9 - see below
const char *string; The text string
}

struct _bounding_box{
int l, t, r, b; The bounding box of a symbol
};
```

Justification of text indicates the location of the text origin relative to the string. Justification is performed prior to rotation and rotation is about the origin. The following diagram indicates the location of the origin for each text point.

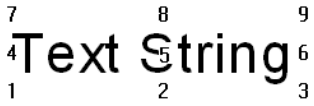

Figure 1: Justification Codes

struct _pin { Symbol Data - Pin Record
int xo, yo; Origin of the pin
short name_offset; Offset of pin name from pin (0 - 31)
short name_dir; Direction of name from pin
short name_font; Text size (0 - 7) to use for name
}

name_dir field values are:
0 Don't show name
2,4,6,8 Justify as in text (above)
0x12, 0x14... Same but Rotated 90 degrees

struct _twin { Attribute Text Windows
short number; Number of the Text Window
int xo, yo; Origin of text in the window
short font; Text size (0 - 7) to use
short just; Justification of the text
short rot; Rotation of the text
}

struct _inst { Symbol Instances

short page; Page number of the instance
int xo, yo; Origin of Instance
short rot_mir; Rotate/Mirror - Value 0 - 7
int l, t, r, b; Bounding Box of the Symbol
}

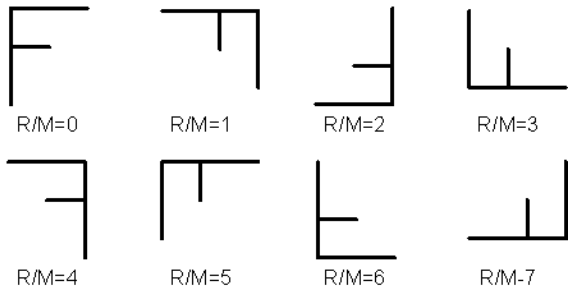

Figure 2: Table of Rotation / Mirror Codes

struct _wire { Connectivity Elements
short type; Type of Element
int xo, yo; Start of Element
int x1, y1; End of Element
short name_flag; Justification if Name Flag - Same as Text above
short io_flag; I/O Indicator if Name Flag
SP_PTR sp; Pointer to Symbol Pin
}

element type fields used
0 - Wire xo, yo, x1, y1 - constrained to 8-directions
1 - Name_Flag xo, yo, name_flag, io_flag
2 - Tap xo, yo (tap end), x1, y1 - orthogonal only
3 - Pin xo, yo, sp
4 - Attr_Flag xo, yo, sp

struct _table { Data Tables
int xo, yo; origin of table
short rows, cols; number of rows and columns
short row_0_height; height of first row
short col_0_width; width of first column
short row_height; height of remaining rows
short col_width; width of remaining columns
int font[6]; Text size (0 - 7) to use
int just[6]; Justification of the text
int rot[6]; Rotation of the text
}

font, jus and rot for tables:
0 name
1 title
2 row 0, column 0
3 row 0 (column > 0)
4 column 0 (row > 0)
5 everything else (row > 0 and column > 0)

struct _date_time { Date and Time
int year; for example, 1991
int mon; 1 = Jan, 2 = Feb, etc.
int day; 1 to 31
int hour; 0 to 23
int min; 0 to 59
int second; 0 to 59
}

Global Variables

Global variables are used in the schematic API Kit. The following variables are defined in the application program.

unsigned int **feature_code**

This variable must be declared in the Application. It is used by the schematic API Kit **Initialize** function to restrict access to the schematic API Kit application based on the user's license. Normally, this variable should be set to 0, allowing the application to run with any license.

unsigned int **feature_version**

This variable must be declared in the Application. It is used by the schematic API Kit **Initialize** function to restrict access to the schematic API Kit application based on the user's license. Normally, this variable should be set to 0, allowing the application to run with any license.

char **FullFileName**[256];

This character string has the full path name of the schematic file which is currently being processed.

Loading and Saving Data

int **Initialize**(void)

This function must be called before any other routines are called. Its main purpose is to load the Designer ini files which establish the working environment.

Functions are provided to access the schematic and symbol data files. Each routine returns TRUE if successful and FALSE if an error occurred.

int **LoadSymbol**(const char ***name*)

Loads the symbol with the specified *name* and the .sym suffix. The symbol file is checked for validity. The directories on the project and symbol library paths are searched until a symbol is found.

int **CheckTopSymbol**(char ***buffer*)

This function checks the pins of the current symbol against the I/O markers on the current schematic. Prior to calling this function the schematic should be loaded using **LoadSchematic** and the symbol should be loaded using **LoadSymbol**. The *buffer* is used by the function to formulate error messages. The buffer should be 256 bytes long. If the function detects errors, they will be posted by calling the **ErrorMsg** function. If there are errors, the return value will be FALSE (not ok) and *buffer* will contain a final error message that the calling program should pass to **ErrorMsg**.

int **BuildTopSymbol**(void)

This function will build a symbol to represent the current schematic. A schematic must have been loaded using **LoadSchematic** before calling this function. The I/O markers on the schematic are used to determine the pins for the symbol. The symbol is created in memory, not on disk.

The symbol can be accessed using the same functions that would apply if the symbol had been loaded using **LoadSymbol**.

int **LoadSchematic** (const char **name*)

Loads the schematic with the specified *name* and the .sch suffix. The schematic file is checked for validity.

int **SaveSchematic** (const char **name*)

Saves the updated version of the schematic file with the specified *name*. Uses the FullFileName as the path and root of the filename in which to save the symbol. Returns FALSE if unable to save the file.

int **LoadSymbolsUsed**(void)

Can only be called after the schematic is loaded. This routine loads all of the symbol files that are used in the schematic. The directories on the project and symbol library paths are searched until a symbol is found. The return code is 0 if some symbols were not found, otherwise the return code is 2 if some symbols were out of date. The return code is 1 if all symbols were ok. If the return code is 2, it is possible that there may be a mismatch between pins on the symbol definition and pins in the schematic instance. Most net list programs should refuse to run if the return code is not 1.

char * **GetSchematicPath** (const char **name*, char **buffer*)

This function searches the Model and Project Library paths to find the full path name of the schematic with the given *name*. The full path name is placed in *buffer* and a pointer to *buffer* is returned.

char ****GetSymbolPath**(const char **name*)

This function searches the Symbol and Project Library paths to find the full path name of the symbol with the given *name*. The full path name is copied into a static buffer and a pointer to that buffer is returned.

void **FreeMemory**(void)

Discards all memory used by symbol and schematic files. Should be used at the end of the process and between processing individual files.

Active Symbol

The concept of an Active Symbol is used by the routines to determine the symbol which is being examined. Initially, no symbol is active. When the **LoadSymbol** function loads the main symbol, it becomes the active one. During traversals of the symbol types and instances in the schematic, the appropriate symbol is set to active (assuming the **LoadSymbolsUsed** function has been called). The main symbol can be reactivated by calling the **MainSymbol** function.

int **MainSymbol**(void)

Causes the main symbol to become the Active Symbol. If **LoadSymbol** did not successfully load the main symbol, this function returns FALSE.

int **GetTypeOfSymbol**(void)

Returns the type code of the Active Symbol. See the spikproc.h file for the type names and values. Returns -1 if there is no Active Symbol.

int **SymbolType**(void)

Returns the type code of the Active Symbol. See the spikproc.h file for the type names and values. There must be an Active Symbol.

struct _bounding_box ***GetSymbolBoundingBox**(void)

Returns a pointer to the bounding box which encloses all pins and graphic line elements of the Active Symbol.

struct _date_time ***GetSymbolDateTime**(void)

Returns a pointer to the structure which contains the date and time information about the Active Symbol.

Traversing The Symbol Data Structures

There are several routines that are used to traverse the symbol data structures. This set of routines expects that a symbol is currently selected as the Active Symbol. If no symbol is active, it is considered a programming error and the function will issue a System level error and exit.

Symbol data files consist of lists of each element type. Traversing the data structures involves scanning the appropriate list. As each element in the list is viewed, the user supplied function is called with the handle to the element.

The called function is expected to return FALSE if the traversal is to continue and TRUE if the traversal is to be stopped. The traversal routine will return TRUE if it was not permitted to complete the traversal.

int **ForEachSymbolPin**(int (**User_Function*)(PN_PTR the_pn, struct _pin *pin))

Traverses the pins in the currently active symbol. The User_Function is passed a pointer to the structure describing the pin. The order is the order of the pins in the symbol definition. See **SortPinsByOrder**.

int **ForEachSymbolTextWindow**(int (**User_Function*)(struct _twin *twin))

Traverses each of the attribute display windows in the currently active symbol. The User_Function is passed a pointer to the structure describing the text window.

int **ForEachSymbolGraphicItem**(int (**User_Function*)(struct _gr_item *gr_item))

Traverses the graphic elements in the symbol. The User_Function is passed a pointer to the structure describing the element.

int **ForEachSymbolGraphicText**(int (**User_Function*)(struct _gr_text *gr_text))

Traverses the graphic text elements in the symbol. The User_Function is passed a pointer to the structure describing the text item.

Accessing Symbol Data - Attributes

Attributes can be attached to the symbol definition and given fixed or default values in the Symbol Editor. The values can be obtained with the following functions. If no value is preassigned, the functions return a NULL string ("").

char _far ***Get_SYA**(int *Attrib_Number*)

Accesses the attributes assigned to the currently active symbol. Returns a pointer to the value of the specified attribute.

char _far ***Get_PNA**(PN_PTR *pin*, int *Attrib_Number*)

Accesses the attributes assigned to the specified *pin* in the currently active symbol. Returns a pointer to the value of the specified attribute.

int **ForEachSymbolAttribute**(int *min*, int *max*, int (**User_Function*)(int the_num, const char *the_value))

Traverses each of the symbol attributes that have been defined. If the attribute number is in the range of *min* <= attrib_number <= *max*, the *User_Function* is run and passed the attribute number and the value of the attribute. The name of the attribute can be obtained with the function **GetSymAttrName**.

int **ForEachSymbolPinAttribute**(PN_PTR *pin*, int *min*, int *max*, int (**User_Function*)(int the_num, const char *the_value))

Traverses each of the attributes that have been defined for the specified *pin*. If the attribute number is in the range of *min* <= attrib_number <= *max*, the *User_Function* is run and passed the attribute number and the value of the attribute. The name of the attribute can be obtained with the function **GetPinAttrName**

Traversing The Schematic Data - Sheets

int **ForEachSheet**(int (***User_Function*)(int the_sheet_num))

Traverses the sheets of the schematic. Passes the number of the sheet to the *User_Function.*

int **GetSheetWidth**(int *sheet_number*)

Returns the width of the given sheet expressed in grid units.

int **GetSheetHeight**(int *sheet_number*)

Returns the height of the given sheet expressed in grid units.

Traversing The Schematic Data - Symbol Data

int **ForEachSymbolType**(int (***User_Function*)(ST_PTR the_sym_type))

Traverses each of the symbol types used in the schematic. Passes the handle of the symbol type to the *User_Function*

int **TypeOfType**(ST_PTR *descriptor*)

Returns the type code of the given *descriptor* indicating the type of symbol defined. See the spikproc.h file for the type names and values.

int **ForEachSymbolInstance**(int *sheet_number,* ST_PTR *symbol_type*, int (*User_Function)(SI_PTR the_sym_inst, struct _inst *inst))

Traverses each of the symbol instances of the specified type and on a selected sheet. If *sheet_number* is NULL, instances on all sheets are visited. If the *symbol_type* handle is NULL, symbols of all types are visited. The *User_Function* is passed the handle of the instance and a pointer to the structure describing the instance.

int **ForEachInstancePin**(SI_PTR *symbol_instance*, int (***User_Function*) (SP_PTR the_sym_pin, PN_PTR pn, struct _pin *pin))

Traverses the pins in the given *symbol_instance*. The User_Function is passed the handle of the symbol pin, the handle of the pin in the symbol definition, and a pointer to a structure describing the pin. If the symbols for the schematic are not loaded, the pn handle is NULL.

int **ForEachInstanceTextWindow**(SI_PTR *symbol_instance*, int (***User_Function*) (struct _twin *twin))

Traverses each of the attribute display windows in the given *symbol_instance* and passes a pointer to the structure describing the text window.

Traversing The Schematic Data - Net Data

A net can be either a scalar signal or a bus. The relationship between scalar nets and buses is "many to many." A scalar can be contained in one or more buses and a bus can include one or more scalars. A variety of traversal routines are supplied to permit flexibility in selecting the nets to traverse.

int **ForEachNet**(int (**User_Function*)(NT_PTR the_net))

Traverses each of the nets. Passes the handle of the scalar or bus net to the *User_Function.*

int **ForEachBus**(int (**User_Function*)(NT_PTR the_net))

Traverses each of the nets which are buses (contain scalar nets). Passes the handle of the bus to the *User_Function*.

int **ForEachScalar**(int (**User_Function*)(NT_PTR the_net))

Traverses each of the scalar nets. Passes the handle of the scalar net to the *User_Function.*

int **ForEachNetNotBus**(int (**User_Function*)(NT_PTR the_net))

Traverses each of the scalar nets which is not a member of any bus. Passes the handle of the scalar net to the *User_Function.*

int **ForEachNetInBus**(NT_PTR *net*, int (**User_Function*)(NT_PTR the_net))

Traverses each of the scalar nets in the bus specified in the *net* parameter. Passes the handle of the scalar net to the *User_Function.*

int **ForEachBusContainingNet**(NT_PTR *net*, int (**User_Function*)(NT_PTR the_net))

Traverses each of the buses that contain the scalar *net*. Passes the handle of the bus to the *User_Function.*

Nets electrically connect a group of symbol pins. The following routine traverses all of the branches of a net and visits each pin. Pins that are connected by a bus are not considered part of the same net as pins that are connected by the scalars within the bus.

int **ForEachNetPin**(NT_PTR *net*,
int (**User_Function*)(SP_PTR the_sym_pin, PN_PTR pn))

Traverses the pins in the given *net*. Passes the handle of the symbol pin and the handle of the pin in the symbol definition to the User_Function. When **ForEachNetPin** is called from inside of any traversal except **ForEachNetFlattened**, it only recognizes the pins connected by *net* and not any of the buses containing *net*. When **ForEachNetPin** is called from inside **ForEachNetFlattened**, any pins connected by buses containing *net* will be visited. Also pins of iterated instances will be visited for each instance of the symbol.

Nets, scalar and bus, are made up of one or more branches. A branch is a contiguous set of wire elements. Multiple branches of a net are connected by the appearance of a name_flag on a wire of each branch. The branches of a net can appear on one or more sheets of the schematic.

int **ForEachBranch**(NT_PTR *net*, int *sheet_number*,
int (**User_Function*)(BR_PTR the_branch, the_sheet_num))

Traverses each of the branches of the selected *net* on the specified sheet. If *sheet_number* is NULL, branches on all sheets are visited. The handle of the branch and the number of the sheet containing the branch are passed to the *User_Function*.

int **ForEachWire**(BR_PTR *branch*, int (**User_Function*)(struct _wire *wire))

Traverses each of the wire elements of the given *branch*. The User_Function is passed a pointer to the structure describing the wire. The traversal will not start on a name_flag unless it is an isolated element. Bus taps will be reported separately from the connecting wire segment and will always be 4 units long.

Traversing The Schematic Data - Flattening Buses and Instances

In contrast to the Hierarchy Data Structures, the Schematic Data Structures are not flattened. Most of the functions for Schematic data extraction visit the elements just as they appear in the schematic. This is convenient for converting the data to another format. The following functions are intended to simplify the process of extracting flat netlists from the Schematic Data Structures.

int **ForEachNetFlattened**(int (***User_Function*)(NT_PTR the_net))

Traverses each of the nets. Passes the handle of the scalar or bus net to the *User_Function.* This function behaves the same as **ForEachNet** except that **ForEachNetFlattened** sets an internal flag which will cause **ForEachNetPin** to flatten each bus, bus pin, and iterated instance in the schematic.

int **ForEachInstance**(ST_PTR *symbol_type*, int *flatten*,
int (*User_Function)(SI_PTR the_sym_inst))

Traverses each of the symbol instances of the specified type. If the *symbol_type* handle is NULL, symbols of all types are visited. If *flatten* is 0, the instances are not flattened. If *flatten* is 1, the iterated instances are flattened. The *User_Function* is passed the handle of the instance.

int **ForEachInstPin**(SI_PTR *symbol_instance*, int *mode*, int (***User_Function*)
(SP_PTR the_sym_pin, PN_PTR pn))

Traverses the pins in the given *symbol_instance.* This function is similar to the **ForEachInstancePin** except that bus pins can be flattened during traversal depending on the *mode* parameter. If this function is called during a traversal of **ForEachInstance**, buses and pins will be flattened according to the setting of the *flatten* parameter from **ForEachInstance** and the *mode* parameter. If the *flatten* parameter to **ForEachInstance** is 0 or if this function is not called from within **ForEachInstance**, no flattening will occur. If the *flatten* parameter to

ForEachInstance is not zero and *mode* is zero, the connections between buses and scalar pins of iterated instances will be flattened. If the *flatten* parameter to **ForEachInstance** is not zero and *mode* is not zero, the connections between buses and pins of all instances will be flattened. The User_Function is passed the handle of the symbol pin and the handle of the pin in the symbol definition. If the symbols for the schematic are not loaded, the pn handle is NULL.

The following functions are designed to be called from within a call to **ForEachNetPin** inside of a **ForEachNetFlattened** traversal or from within a call to **ForEachInstPin** inside of a **ForEachInstance** traversal. Each of these functions contain a static string which holds the value which is preserved until a subsequent call to the function modifies it.

char ***GetInstanceName**(SI_PTR *symbol_instance*)

Returns a pointer to the static string containing the instance name of the *symbol_instance* connecting to the signal being visited in a **ForEach-NetFlattened** traversal.

char ***GetRefDesignator**(SI_PTR *symbol_instance*)

Returns a pointer to the static string containing the reference designator of the *symbol_instance* connecting to the signal being visited in a **ForEachNetFlattened** traversal. In the case of an iterated symbol the reference designator attribute will contain a list of reference designators. This function returns NULL if the list of reference designators was too short for the number of iterated instances.

char ***GetPinName**(PN_PTR *pin*)

Returns a pointer to the static string containing the name of the *pin* connected to the signal being visited in a **ForEachNetFlattened** traversal. If the net is contained in a bus that is visiting a bus pin, the pin name will reflect the position of the net within the bus. This function returns NULL if there is no pin that matches the position of the net in the bus.

char ***GetPinNumber**(SP_PTR *the_symbol_pin*, PN_PTR *pin*)

Returns a pointer to the static string containing the pin number of the *pin* connected to the signal being visited in a **ForEachNetFlattened** traversal. If the pin is a bus pin on a Component type symbol the pin number will be the n'th pin number from the bus pin attributes BUSPIN1 through BUSPIN8 (#90 - #97) where n is the position of the net within the bus. This function returns NULL if there is no pin number which matches the position of the net in the bus.

NT_PTR **GetNetOnPin**(SP_PTR *the_symbol_pin*)

Returns a NT_PTR to the net connected to *the_symbol_pin* being visited in a **ForEachInstPin** during a **ForEachInstance** traversal. If the pin is connected to a bus, the setting of the *flatten* parameter to **ForEachInstance** and the *mode* parameter to **ForEachInstPin** will determine whether the NT_PTR to the bus or the scalar signal is returned.

Traversing The Schematic Data - Graphic Data

int **ForEachGraphicItem**(int *sheet_number*, int (**User_Function*)
(struct _gr_item *gr_item))

Traverses the graphic elements on the given sheet. The *User_Function* is passed a pointer to the structure describing the graphic item.

int **ForEachGraphicText**(int *sheet_number*, int (**User_Function*)
(struct _gr_text *gr_text))

Traverses the graphic text elements on the sheet. The *User_Function* is passed a pointer to the structure describing the text item.

Traversing Schematic Data - Miscellaneous Data

int **ForEachGlobalNetName**(int (***User_Function*)(const char *the_name))

Traverses each of the global signals and passes the signal name to *User_Function*.

int **ForEachTable**(int *sheet_number*, int (***User_Function*)
(TB_PTR the_table, struct _table *table))

Traverses each of the tables on the given sheet. If *sheet_number* is NULL, tables on all sheets are visited. The *User_Function* is passed the handle of the table and a pointer to the structure describing the table.

Accessing Schematic Data - Attributes

Attributes can be attached to symbol types, instances, instance pins, and nets. Individual attribute values can be accessed with the following functions. If no attribute is defined, a NULL string ("") is returned. Attribute numbers that are reserved are listed in the header file attr.h.

char ***Get_NA**(NT_PTR *net*, int *Attrib_Number*)

Accesses attributes that are associated with the given *net*. Returns a pointer to the value string of the requested attribute.

char ***Get_DA**(ST_PTR *symbol_type*, int *Attrib_Number*)

Accesses attributes associated with the given *symbol_type*. Typically, the name of the symbol is the only attribute in this class. Returns a pointer to the value string of the requested attribute.

The following functions access attribute values that were assigned or overridden on an instance basis in the schematic editor. Attribute values which were originally assigned to the symbol definition can be obtained with the functions **Get_SYA** and **Get_PNA**.

char ***Get_IA**(SI_PTR *symbol_instance*, int *Attrib_Number*)

Accesses attributes that are associated with the given *symbol_instance.* Returns a pointer to the value string of the requested attribute.

char ***Get_PA**(SP_PTR *symbol_pin*, int *Attrib_Number*)

Accesses attributes that are associated with the given *symbol_pin*. Returns a pointer to the value string of the requested attribute.

The following functions access attribute values regardless of the attribute origin. These functions do all of the work necessary to get the correct value of an attribute. If the attribute has been overridden, the override will be returned. If there is no override, the default value will be returned. If the attribute number represents a derived attribute, the attribute will be evaluated.

char ***Get_SIA**(SI_PTR *symbol_instance* , int *Attrib_Number*)

Accesses attributes that are associated with the given *symbol_instance.* The default value from the symbol definition will only be accessed if the symbol has been loaded. Returns a pointer to the value string of the requested attribute.

char ***Get_SPA**(SP_PTR *symbol_pin*, PN_PTR *pn*, int *Attrib_Number*)

Accesses attributes that are associated with the given *symbol_pin*. The default value from the symbol definition will only be accessed if the symbol has been loaded. The PN_PTR can be passed to speed up the function. If the *pn* handle is NULL the function will find the matching PN_PTR from the symbol definition (If it needs it). Returns a pointer to the value string of the requested attribute.

The following functions access attribute values and data from Data Tables.

char ***Get_Table_Attr**(TB_PTR *table*, int *Attrib_Number*)

Accesses attributes that are associated with the given *table*. The name of the table is attribute number 0 and the title of the table is attribute number 1. Returns a pointer to the value string of the requested attribute.

char ***Get_Table_Data**(TB_PTR *table*, int *row*, int *column*)

Accesses data from the specified *row* and *column* of the given *table*. Returns a pointer to the value string of the requested data.

Routines are provided to scan the attributes. Each of the routines scans the attribute list of the specified item and calls the User_Function for each attribute encountered in the specified range.

int **ForEachNetAttrib**(NT_PTR net, int first, int last, int (**User_Function*)(int the_attr_num, const char *the_value))

Scans the list of attributes attached to the given *net*. If there is an attribute whose number is between *first* and *last*, the *User_Function* is called.

As above, these routines return the attribute values which were assigned in the schematic editor. To obtain the values that were originally assigned in the symbol editor, use the functions **Get_SYA** and **Get_PNA**.

int **ForEachInstanceAttrib**(SI_PTR *symbol_instance*, int first, int last, int (**User_Function*)(int the_attr_num, const char *the_value))

Scans the list of attributes attached to the given *symbol_instance*. If there is an attribute whose number is between *first* and *last*, the *User_Function* is called.

int **ForEachInstancePinAttrib**(SP_PTR *symbol_pin*, int *first*, int *last*, int (**User_Function*)(int the_attr_num, const char *the_value))

Scans the list of attributes attached to the given *symbol_pin*. If there is an attribute whose number is between *first* and *last*, the *User_Function* is called.

int **ForEachTypeAttrib**(ST_PTR *symbol_type*, int first, int last, int (**User_Function*)(int the_attr_num, const char *the_value))

Scans the list of attributes attached to the given *symbol_type*. If there is an attribute whose number is between *first* and *last*, the *User_Function* is called.

Adding Schematic Data - Attribute Overrides

Attributes on symbol instances and symbol pins have the values that were assigned in the symbol definition. Assignment of an attribute in the schematic overrides the default value on an instance-specific basis.

int **Add_IA**(SI_PTR *symbol_instance,* int *Attrib_number,* const char **Value*)

Sets the value of the specified attribute for the given *symbol_instance*. Deletes any previous value. Specifying a null string as a value will cause the default value to be used.

int **Add_PA**(SP_PTR *symbol_pin,* int *Attrib_number*, const char **Value*)

Sets the value of the specified attribute for the given *symbol_pin.* Deletes any previous value. Specifying a null string as a value will cause the default value to be used.

Attributes on nets are not given any default values. When extracting flattened netlists from hierarchical designs, the net attributes attached to net segments are ignored for all but the segment appearing as a local net in a schematic or in the root level schematic.

int **Add_NA**(NT_PTR *net,* int *Attrib_number*, const char **Value*)

Sets the value of the specified attribute for the given *net*. Deletes any previous value. Specifying a null string as a value will cause the attribute to be deleted.

The following functions add attribute values and data to Data Tables.

int **Add_Table_Attr**(TB_PTR *table*, int *Attrib_Number*, const char **Value*)

Adds attributes to the given *table*. The name of the table is attribute number 0 and the title of the table is attribute number 1. Deletes any previous value.

int **Add_Table_Data**(TB_PTR *table*, int *row*, int *column*, const char **Value*)

Adds data to the specified *row* and *column* of the given *table.* Deletes any previous value.

Schematic Data - Attribute Names

The user assigned names of the various attributes are defined in the ecs.ini file. The following routines translate between names and numbers.

const char ***GetNetAttrName**(int *Attrib_Number*)

Returns names for attributes that are associated with nets.

const char ***GetPinAttrName**(int *Attrib_Number*)

Returns names for attributes that are associated with symbol pins.

const char ***GetSymAttrName**(int *Attrib_Number*)

Returns names for attributes that are associated with symbol definitions and instances.

int **GetNetAttrNumber**(const char **attrib_name*)

Returns the number of an attribute that is associated with nets.

int **GetPinAttrNumber**(const char **attrib_name*)

Returns the number of an attribute that is associated with symbol pins.

int **GetSymAttrNumber**(const char **attrib_name*)

Returns the number of an attribute that is associated with symbols.

int GetAttrOfWindow(int *window*)

Returns the attribute which is currently displayed in the specified *window*. Returns -1 if no attribute is displayed in the window.

Schematic Data - Miscellaneous

NT_PTR **FindNetNamed**(const char **name*)

Returns the handle of the net with the given *name*.

char ***GetCoordinateUnits**(void)

Returns the units in which the coordinate space is defined, that is, Inches, Centimeters or Millimeters

int **GetGridSize**(void)

Returns the size of the grid in 1/100ths of the physical unit.

SI_PTR **InstanceContainingPin**(SP_PTR *symbol_pin*)

Returns handle of instance containing the given symbol_pin.

int **IsBusName**(const char **name*)

Returns TRUE if the *name* is not a scalar signal.

int **InOutBidir**(NT_PTR *net*)

Returns a value indicating the type of the *net* as follows:

Net Type Indicated by Value of:

input 1
output 2
bi-directional 3

This function only works for nets which are external to the schematic, and returns 0 if the net is not external.

int **LocExtGbl**(NT_PTR *net*)

Returns a value indicating the type of the *net* as follows:

Net Type Indicated by Value of:

local to the schematic 0
external to the schematic (an I/O port) 1
global net 2

PN_PTR **MatchingSymbolPin**(SP_PTR *symbol_pin*)

Returns the handle of the pin on the symbol definition which corresponds to the given *symbol_pin*.

NT_PTR **NetContainingPin**(SP_PTR *symbol_pin*)

Returns handle of net containing the given *symbol_pin*.

NT_PTR **NetContainingPoint**(int *x*, int *y*)

Returns the handle of the net containing the specified point on the currently active sheet. Returns NULL if nothing is at the point.

int **ParseInstanceName**(SI_PTR *symbol_instance*, const char **instance_name*, int (**User_Function*)(SI_PTR the_sym_inst, char *the_name))

Parses the *instance_name* into the individual names. This is used to flatten iterated symbols used in schematics.

int **ParseNetName**(const char **net_name*, int (**User_Function*)(char *the_name))

Parses the *net_name* into scalar signal names. This is used to expand buses for netlisting.

int **ParseList**(const char **list*, int (**User_Function*)(char *the_name))

Parses the *list* into the individual names. This is used to parse pin numbers on bus pins and lists of reference designators for iterated symbols used in schematics.

ST_PTR **TypeOfInstance**(SI_PTR *symbol_instance*)

Returns handle of symbol type describing the given *symbol_instance*.

char ***GetInstanceCoordinates**(SI_PTR *symbol_instance*)

Creates a string which will describe the given *symbol_instance*. The error viewer in the Schematic Editor will interpret the string correctly and will place an 'X' at the indicated location. The string will be of the form: <pp,xx,yy> with the page, x and y locations of the *symbol_instance*. The string is created in a static buffer which will only remain valid until the next call to **GetInstanceCoordinates**.

char ***GetPinCoordinates**(SP_PTR *symbol_pin*)

Creates a string which will describe the given *symbol_pin*. The error viewer in the Schematic Editor will interpret the string correctly and will place an 'X' at the indicated location. The string will be of the form: <pp,xx,yy> with the page, x and y locations of the *symbol_pin.* The string is created in a static buffer which will only remain valid until the next call to **GetPinCoordinates**.

int **SortPinsByOrder**(int *attrib_number*)

This function may be called before traversing the symbol pins to sort them based on the value of the given *attrib_number*. This function will affect the order that the pins are traversed in **ForEachSymbolPin.** The attribute values are treated as integers.

Utility Functions

int **MajorError**(const char **string*)

Displays an alert prompt showing the given *string*. The function waits until the user clicks the OK button before proceeding.

int **SysError**(const char **message*)

Reports the *message* using an alert box technique. When the user acknowledges the message, the function causes an exit. This is intended for severe errors which prohibit further processing.

Chapter 4
Process Reference

This section of the manual provides reference information for the various Processes that produce a destination or end-result (as opposed to commands that perform a certain function.) It is used to describe processes such as netlisters, report generators, and back-annotators.

Archive Design

Run From - Hierarchy Navigator

Synopsis - Creates an archive of one whole design, including symbol, schematic, hierarchy map files, and supporting configuration files. Files are copied to a specific directory, apart from any active libraries. Once archived in this manner, a design may be re-loaded without dependency on external factors.

Requirements - Must have appropriate file system permissions to create and write into the specified archive directory.

Details - Use the Hierarchy Navigator to load the highest level of the design to be archived. Select the "Archive" command. The following files will be copied into the destination directory, as specified:

- Symbols (.sym), to <destination>
- Schematics (.sch), to <destination>

- Hierarchy Maps (.tre), to <destination>
- Project Settings (.ini), to <destination>
- Other Settings (other .ini), to <destination>/config

Caveats - Design files are copied into a flat destination directory, loosing any hierarchy they might have had in the original libraries. This is safe for archival purposes, as Designer references symbol and schematic dependencies by filename only (path to file is not part of the database).

In a version-controlled environment, files are not copied with their respective RCS/ repository structure. Only the bare files are archived, directly from the active nodes of the repository. If any such files are in transit (contemporaneously being saved or moved by other users), the archive process will fail and an appropriate message will be generated. In such case, the archive command should be re-run.

Executable Name - 'archive'

Dialog Options - Following options can be set if the pop-up dialog is invoked:

`Destination Path`

Specify the destination directory for archive copies.

Command Options - Following command parameters can be passed from a custom menu entry:

-?

Open the pop-up dialog before running archive. Using the pop-up dialog will override any other parameters.

-save=destination

Write archive to a specific destination. Destination path can be relative or absolute, depending on conventional leading-dot or leading-slash notation.

See Also

- Custom Menu Entries

- Version Control (as built-in interface)

Verilog Netlist

VHDL Netlist

Hierarchical Spice Netlist (hspicent)

Flat Spice Netlist (spicenet)

Allegro Netlist

(I think exists in older man page format)

Allegro Back-annotate

older format exists?)

Check Circuit (checkckt)

Generic Lister / Checker (lister)

hierplay

Check Schematic (checksch)

Plain Netlist (asciinet)

Generic Pcb Netlist (pcbnet)

Generic PCB Back-annotate

PCB BOM Generator (packlist)

Flat EDIF Netlist

Hierarchical EDIF Netlist