Programmers Interface Kit

This section describes the features and operation of the Programmers Interface Kit (PIK) used in the Engineering Capture System (ECS). The PIK allows you to access the ECS database using 'C' language calls. You can create your own custom interfaces using the PIK.

This section of the manual is intended as a reference manual for CAD support programmers only. This module is not provided for users. It is not expected that average ECS users will make use of the PIK routines.

The PIK is a separate module available on a site license basis.

The major topics covered in this section are:

- Contents of PIK
- Memory allocation
- Hierarchy data extraction
- Flat schematic data extraction
- Function definitions

Contents

Programmer's Interface Kit

The Programmer's Interface Kit provides the software functions needed to programmatically interface with the Engineering Capture System. The functions are written in the 'C' language and compiled into libraries which may be linked with the User's application code.

The Kit is divided into two sections which interface to different portions of the Engineering Capture System. The sections are:

1. Hierarchy Data Extraction
   This library contains the functions which interact with the Navigator's Hierarchical Data Structures. Applications that utilize this library may be launched from the Process or Tools menu of the Navigator. These applications run as child processes of the Navigator and are dependent on its presence.

   Applications treat the design as a single structure and are able to traverse the entire logical data base. The logical data base "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 net list extraction programs in the ECS. It also provides the basis for the the ECS - Design Analysis Tools.

2. 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 utilize this library are self contained. They may either be run as stand-alone processes or be launched by the Navigator.

   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 data base exportation and net list extraction where the busing structure is to be preserved. The ECS programs for VHDL net list extraction as well as the ASCOUT programs are built with this kit.

Files Supplied

The Programmer's Interface Kit contains three object files and several header (.h) files which will be included in the application programs. The header files are placed in the "h" directory and include:

win.h	The platform specific definitions and macros
pikproc.h	The typedefs and definitions for the Hierarchy Data.
pikfuncs.h	The function prototypes for the Hierarchy Data Extraction.

spikproc.h The typedefs, definitions and prototypes for the Schematic Data.
attr.h The attribute definitions.

The object file libraries that are supplied are platform dependent and include the following files:

For the UNIX Workstation:

pik_u.o The main function for Hierarchical Data Extraction
spik.o The Symbol & Schematic Data Extraction Library

For the PC:

pik.lib The Hierarchical Data Extraction Library
pik.res The resource file for Hierarchical Data Extraction
scpik.lib The Symbol & Schematic Data Extraction Library
scpik.res The resource file for Symbol & Schematic Data Extraction

For the Macintosh:

Pik1.π The Hierarchical Data Extraction Library (Part 1)
Pik2.π The Hierarchical Data Extraction Library (Part 2)
PreProc.π The PreProcess function for Hierarchical Data Extraction
PostProc.π The PostProcess function for Hierarchical Data Extraction
Scpik1.π The Symbol & Schematic Data Extraction Library (Part 1)
Scpik2.π The Symbol & Schematic Data Extraction Library (Part 2)

New For Release 2.4 in Hierarchical Data Extraction

Several routines have been changed for the 2.4 release. The **Process** and **PreProcess** functions now have the *argc* and *argv* parameters similar to the parameters to main. These parameters can be used to supply options in addition to the flags /A through /Z which are always available to PIK programs.

The following new routines have been added. **Get_TNA_Override**, **Get_TPA_Override**, **ForEachTNA**, **ParentInstanceOf**, **FindPinNamed**, **PinDefiningNet**, **AddExt**, **FileInPath** and **GetIntlDateTimeString**.

New For Release 2.4 in Symbol and Schematic Data Extraction

There have been some changes to data structures. The style parameter was added to the _gr_item structure. The _date_time and _table structures are new.

Several routines have been changed for the 2.4 release. The **SysError** function no longer has a number as its first parameter. The **MajorError** function should be used instead of **SysError**.The **ForEachNetPin** function has been changed slightly. The *User_Function* is passed both the schematic pin and symbol pin handle. The function behavior is the same except when called during a **ForEachNetFlattened** traversal.

The following new routines have been added. **GetInstanceCoordinates**, **GetInstanceName**, **GetPinCoordinates**, **GetPinName**, **GetPinNumber**, **GetRefDesignator**, **Get_SIA**, **Get_SPA**, **Get_Table_Attr**, **Get_Table_Data**, **Add_Table_Attr**, **Add_Table_Data**,

ForEachInstanceTextWindow, **ForEachNetFlattened**, **ForEachTable**, **GetSymbolDateTime**, **MajorError** and **GetIntlDateTimeString**.

Memory Allocation

Each of the platforms have different memory management and pointer requirements. The PC platform requires that memory be allocated with the GlobalAlloc function, locked with GlobalLock, unlocked with GlobalUnlock and freed with GlobalFree. The Macintosh uses NewHandle, HLock, HUnlock and DisposeHandle. The Unix platform uses malloc and free. In order to produce applications which run on all platforms, a set of memory management routines are provided to insulate the application from this platform dependant problem.

These functions allocate _far memory on the PC. This means that the memory must be referenced by a 32 bit pointer. If it is necessary to allocate more than 64K of data the memory pointers should be cast as _huge. The PC platform has a restriction that "huge" data must be represented as an array of objects each of which is a power of 2 in size. The reason for this restriction is that a single structure must not cross a 64K segment boundary.

The _huge keyword is meaningful on the PC platform and is discarded on the other platforms. A function for changing a "huge" character pointer to a "near" one is provided on the PC platform. This function call, **Fn** is defined as null on the Macintosh and Unix platforms. This function must be used with care as all it does is copy the huge string to a local static string buffer and return the address of that buffer. Subsequent calls to **Fn** will return the same address.

```
typedef char _huge *        MEMPTR
typedef unsigned long       MEMBLOCK
typedef long                MEMSIZE
```

MEMBLOCK **env_AllocMem**(MEMSIZE *size*)

This routine allocates a block of memory of the given *size* and returns a handle (MEMBLOCK) to it. The returned value of should be used when locking, unlocking, reallocating or freeing this block of memory. The function **env_GraspMem** must be used to access the data in this block. The memory is initialized to all zeros.

MEMBLOCK **env_ReAllocMem**(MEMBLOCK *handle*, MEMSIZE *size*)

This routine attempts to reallocate the given block to the new *size*. The handle returned may not be the same as the original *handle* parameter The memory must not be locked (see **env_UnGraspMem**) when this function is called. The additional memory is not initialized to zero.

MEMPTR **env_GraspMem**(MEMBLOCK *handle*)

This routine locks the given block of memory and returns a _huge pointer (MEMPTR) to it. The block must be unlocked with **env_UnGraspMem**before being freed with **env_FreeMem**.

void **env_UnGraspMem**(MEMBLOCK *handle*)

This routine unlocks the given block of memory. This routine must be called to unlock any memory block that was locked with **env_GraspMem** before calling **env_ReAllocMem** or **env_FreeMem**.

void **env_FreeMem**(MEMBLOCK *handle*)

This routine frees the given block of memory. The memory must not be locked when this routine is called.

MEMSIZE **env_CompactMem**(MEMSIZE *size_wanted*)

This routine attempts to perform a memory compaction to see if it would be possible to allocate a block of memory of the given size. The return value indicates the amount of memory available.

Character Strings

In order to maintain compatibility with the PC platform, the functions that work with character string pointers are written to use huge pointers. It is recommended that character strings that are passed to these functions always have the cast (char _huge *) included. Likewise, the return values from these functions must be processed as huge pointers. These pointers may not be passed as arguments to any of the standard C library functions. The strings may be copied to a temporary local buffer with the function function **Fn**.

char ***Fn**(char huge **string*)

On Unix and Macintosh platforms this function is "#defined" to do nothing. On the PC platform, this function copies the given string into a local static string of 256 bytes and returns the address of that static string.

This function is intended to be used to convert one or two arguments in calls to sprintf. Because there are only two static buffers used for this function, every second call to this function (on the PC) will replace the results of the previous call. This limits the multiple use of this function to two arguments in a single sprintf function call. Saving the returned pointer from this function will not give the proper results.

In general, when a huge pointer needs to be processed as a near string, the **hstrcpy** function should be used to copy the string to a local buffer.

int **hstrlen**(char _huge **str*)

A version of strlen which uses the huge form of pointers on the PC. This routine is "#defined" as strlen on the Unix and Macintosh platforms.

int **hstrcmp**(char _huge **str1*, char _huge **str2*)

A version of strcmp which uses the huge form of pointers on the PC. This routine is "#defined" as strcmp on the other platforms.

int **hstricmp**(char _huge *_str1_, char _huge *_str2_)

A version of stricmp which uses the huge form of pointers on the PC. This routine is "#defined" as stricmp on the other platforms.

int **hstrncmp**(char _huge *_str1_, char _huge *_str2_, size_t _len_)

A version of strncmp which uses the huge form of pointers on the PC. This routine is "#defined" as strncmp on the other platforms.

char _huge ***hstrncpy**(char _huge *_dest_, char _huge *_source_, size_t _len_)

A version of strncpy which uses the huge form of pointers on the PC. This routine is "#defined" as strncpy on the Unix and Macintosh platforms.

char _huge ***hstrcpy**(char _huge *_dest_, char _huge *_source_)

A version of strcpy which uses the huge form of pointers on the PC. This routine is "#defined" as strcpy on the Unix and Macintosh platforms.

char _huge ***hstrcat**(char _huge *_dest_, char _huge *_source_)

A version of strcat which uses the huge form of pointers on the PC. This routine is "#defined" as strcat on the Unix and Macintosh platforms.

char _huge ***hmemcpy**(char _huge *_dest_, char _huge *_source_, unsigned long _len_)

A version of memcpy which uses the huge form of pointers on the PC. This routine is "#defined" as memcpy on the Unix and Macintosh platforms.

Procedural Interface: Hierarchy Data Extraction

Overview

The procedural interface to the 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 Navigator and appear on the Navigator's menus. They are launched by the Navigator and they run as child processes of the Navigator.

This interface is used to build the net list extraction processors that are included in the Engineering Capture System. Flattened net list processors, such as "spicenet", as well as hierarchical net list 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 included in the ECS-Design Analysis Tools.

The routines comprising this interface are contained in the "pik" object library which is located in the appropriate PIK directory or folder of the Programmer's Interface Kit. This directory contains the platform dependent files which are needed to build a "pik" application.

Several key concepts used in implementing this interface are:

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

2. The data traversal routines are written in a call-forward style. When the traversal routine is called, it is provided with a function which it should call for each item encountered in the traversal. The traversal routine will scan the data base 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 data base 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 to indicate whether or not the traversal should abort. Normally, the return value is FALSE. The TRUE return might be used to stop a traversal which was searching for a particular item after 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 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 utilize 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 long **permission_mask**
This variable is declared in the Application and used by the PIK main functions. It is used to restrict access to the PIK application on the basis of the "OEM" version of the program. Normally, this variable should be set to all 1's (0xffffffff).

unsigned long **command_flags**
This variable is set by the PIK front end to represent the control flags passed on the command line. The command line flags are chosen by the user during the Setup of the Process or Tools menu. 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 may be attached. If either or both of these functions are not provided, dummy routines will be linked from the object library.

The three functions which may be written as part of the application are:

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

This function which is typically is used for setup of the process. The **command_flags** have already been processed and may be accessed. The **command_flags** represent only the flags which were specified as a single character following a forward slash. 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 "silosnet" and "spicenet" processors are typical examples of the functions which 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 aborting the process. A non-zero return value is an indication that the process is to be aborted.

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 data base 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 may be accessed. The **command_flags** represent only the flags which were specified as a single character following a forward slash. 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 may be used to report the process results to the user by opening an dialog box. It may 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

There are several data types which 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:

typedef	use	
TD_PTR	Descriptor Handle	- Represents the basic schematic and symbol information
TI_PTR	Instance Handle	- Represents the instantiation of a symbol in a schematic
TN_PTR	Net Handle	- Represents the net within the schematic

TP_PTR	Pin Handle	- Represents the instance of a pin in the schematic
TG_PTR	Generic Pin Handle	- Represents the description of the original pin
TA_PTR	Attribute Handle	- Represents an attribute of a symbol, pin or net

Context

There are two different methods for traversing the design data base. The "flattening" traversal functions visit each instance of the design in its full "hierarchical" context. This method is useful for producing flat Spice net lists which 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 which 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 net lists. 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 may 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 may also assign attributes to the nets that interconnect the symbols.

A schematic may 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 Navigator provides commands to add or override attributes on an instance specific basis.

An element in the design may be viewed in either its "local context" or its "hierarchical context". In the "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 navigator are not visible.

In the "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 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.

In the "local context" the Navigator assigned values are ignored (except when processing the root schematic) and the Schematic or Symbol Editor assigned values are returned.

Each of the attribute accessing routines use an attribute number to access the attribute. The attribute numbers that are pre-assigned in the ECS system are listed in the header **attr.h**.

Each attribute is "#defined" to a mnemonic which 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.

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 routine all return a pointer to a NULL string (""). Note that sometimes the value returned is only a pointer to a temporary buffer which may be overridden by the next call to one of the attribute functions.

char _huge ***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 _huge ***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 which contains the attribute value. If it is necessary to save this value, the value must be copied from this temporary buffer to some permanent storage area.

char _huge ***Get_TNA**(TN_PTR *net*, int *attr_num*)
Retrieves the net attribute *attr_num* for the given *net* .

Net attributes which are entered in the Navigator are assigned to the segment of the net which appears at the highest level of the hierarchy. This corresponds to the segment in the schematic which 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 which contains the attribute value. If it is necessary to save this value, the value must be copied from this temporary buffer to some permanent storage area.

char _huge ***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 which contains the attribute value. If it is necessary to save this value, the value must be copied from this temporary buffer to some permanent storage area.

char _huge ***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 may be obtained from the function **GenericPinOfPin**.

char _huge ***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 which contains the attribute value. If it is necessary to save this value, the value must be copied from this temporary buffer to some permanent storage area.

char _huge ***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 _huge ***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 which contains the attribute value. If it is necessary to save this value, the value must be copied from this temporary buffer to some permanent storage area.

char _huge ***Get_Inst_Name**(TI_PTR *instance*)
Retrieves the "local" instance name for the given symbol *instance*. This is equivalent to calling **Get_TIA** when viewing the hierarchy in local mode.

char _huge ***Get_Net_Name**(TN_PTR *net*)
Retrieves the "local" net name for the given *net* segment. This is equivalent to calling **Get_TNA** when viewing the hierarchy in local mode.

The following functions may 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, char *the_name, char _huge *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 = 1	View default values only	Similar to **Get_TDA**.
mode = 2	View "local" value only	Similar to **Get_TIA** in local view
mode = 4	View "overrides" only	Similar to **Get_TIA_Override**
mode = 6	View all	Similar to **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, char *the_name, char _huge *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 = 1	View default values only	Similar to **Get_TGA**.
mode = 2	View "local" value only	Similar to **Get_TPA** in local view
mode = 4	View "overrides" only	Similar to **Get_TPA_Override**
mode = 6	View all	Similar to **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, char *the_name, char _huge *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 = 2	View "local" value only	Similar to **Get_TNA** in local view
mode = 4	View "overrides" only	Similar to **Get_TNA_Override**
mode = 6	View all	Similar to **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.

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

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

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

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

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

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

Relationships

The hierarchy data structure is a specialized relational data base which contains the design. Within the data base 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 data base. 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 which represents the schematic on which the given *net* appears.

TD_PTR **DescriptorOfInstance**(TI_PTR *instance*)
Returns the handle to the Descriptor which represents the given symbol *instance*.

TN_PTR **FindNetRoot**(TN_PTR *net*)
Returns the handle to the "master" Net in 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 pin attributes. Use the traversal function **ForEachInstance** to access all of the instances of a symbol.

TG_PTR **GenericPinOfPin**(TP_PTR *pin*)
Returns the handle to the Generic Pin which 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 which 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 which 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 may 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 which represents the given *net*. This function only works for nets which are external to the schematic. Use the function **NetLocExtGbl** to determine if a net is external to the schematic.

Database Element Numbers

The hierarchy data base assigns unique numbers to each instance, net, and driving (output or bidirectional) pin in the design. No assumptions should be made about the sequencing of these numbers. These numbers may 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*. The corresponding "Find" function is not supplied.

Searching For a Particular Element

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

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

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

TI_PTR **FindInstanceNamed**(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**(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 which has both the given REFNAME value and which has the correct set of PINNUM attributes to correspond to the given gate. e.g. 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**(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**(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 '-' may 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*, 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.

int **NetInOutBid**(TN_PTR *net*)
Returns the classification of the given (external) *net* as being "Input" - 1, "Output" - 2, or "BiDir" - 3.

int **NetLocExtGbl**(TN_PTR *net*)
Returns the classification of the given *net* segment as being "local to schematic" - 0, "external" (represented by a pin) - 1 or "global in the design" - 3.

int **PrimitiveCell**(TD_PTR *descriptor*)
Returns TRUE if there is not a schematic represented by the *descriptor* cell in the hierarchy.

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 which 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" net lists. The routines are:

int **ForEachDescriptor**(int (**User_Function*)(TD_PTR the_desc))
This routine visits each of the descriptors in the hierarchical data base. 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 which are in the design. Each symbol would be listed once even though it may 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", Fn( 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 net lists, 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", Fn( Get_TDA( td, NAME ) ) );
   return( FALSE );
 }

void Process()
 { fprintf( file, "Types Used:\n" );
   SetupBlockScan();
   ForEachBlock( 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 which does not have any sub-circuits. This is the version that is typically used for creating hierarchical net lists 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 which are not to be visited. A code fragment that utilizes this function to avoid listing those sections represented by a SilosModel is:

```
int List_Type( TD_PTR td )
 { fprintf( file, "%s\n", Fn( 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 net list, 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.

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", Fn( Get_TNA( tn, NAME ) ) );
   return( FALSE );
 }

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

int List_Type( TD_PTR td )
 { fprintf( file, "SubCkt %s\n",
                  Fn( Get_TDA( td, NAME ) ) );
   ForEachSubBlock( td, List_Sub );
   return( FALSE );
 }
```

Each sub-circuit in a hierarchical description should have a list of the ports of the sub-circuit listed with the sub-circuit 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 may not 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", Fn( Get_TPA( tp, NAME ) ) );
   return( FALSE );
 }

int Check_Block_Net( TN_PTR tn )
 { if ( NetLocExtGbl( tn ) == EXTERNAL_NET )
```

```
      fprintf( file, "%s", Fn( Get_TNA( tn, NAME ) )
    );
    return( FALSE );
  }

int List_Type( TD_PTR td )
  { TI_PTR ti;
    fprintf( file, "SubCkt %s (",
                   Fn( 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 may be called in either a “local” or “hierarchical” traversal.

The complete net list program is:

```
FILE *file;

int List_Instance_Pin( tp )
  { TN_PTR tn;
    tn = NetContainingPin( tp );
    fprintf( file, " %s", Fn( Get_TNA( tn, NAME ) ) );
    return( FALSE );
  }

int List_Sub( TI_PTR ti )
  { TD_PTR td;
    td = DescriptorOfInstance( ti );
             /* print symbol type then instance name */
    fprintf( file, "   %s", Fn( Get_TDA( td, NAME ) )
    );
    fprintf( file, " %s (", Fn( Get_TIA( ti, NAME ) )
    );
    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", Fn( Get_TGA( tg, NAME ) ) );
   return( FALSE );
 }

int Check_Block_Net( TN_PTR tn )
 { if ( NetLocExtGbl( tn ) == EXTERNAL_NET )
      fprintf( file, "%s", Fn( Get_TNA( tn, NAME ) )
  );
   return( FALSE );
 }

int List_Type( TD_PTR td )
 { TI_PTR ti;
   fprintf( file, "SubCkt %s (",
            Fn( 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 net lists which describe the circuit as a hierarchy of circuits containing sub-circuits. 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 which 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 may 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 net list.

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", Fn( Get_TNA( tn, NAME ) ) );
   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 net list. This is essentially the same program supplied as source code in the Programmers Interface Kit.

```
int List_Net_Pin( TP_PTR tp )
 { TI_PTR ti;
   ti = InstanceContainingPin( tp );
   fprintf( file, " %s", Fn( Get_TIA( ti, NAME ) ) );
   fprintf( file, "-%s", Fn( Get_TPA( tp, NAME ) ) );
   return(0);
 }

int List_Net( TN_PTR tn )
 { fprintf( file, "%s", Fn( Get_TNA( tn, NAME ) ) );
   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 leaf cell in the hierarchy. It calls the *User_Function* each time it encounters a leaf cell.

The "pinorder" net list program is an example of this type of traversal. A simplified 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 ", Fn( Get_TPA( tp, NAME ) ) );
   fprintf( file, "%s\n", Fn( Get_TNA( tn, NAME ) ) );
```

```
    return(0);
  }

int List_Instance( TI_PTR ti )
  { TD_PTR td;
    td = DescriptorOfInstance( ti );
    fprintf( file, "%s", Fn( Get_TDA( td, NAME ) ) );
    fprintf( file, " %s\n", Fn( Get_TIA( ti, NAME ) )
    );
    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;
  }
```

Utility Functions

int **AddExt**(char *_name_, 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 '.'. For compatibility among all platforms the *ext* should not exceed three characters after the period.

char ***FileInPath**(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 **UpdateHierarchy**(char *_filename_, int *save*)
This command tells the Navigator to read a file containing attribute overrides. The Navigator will add the attribute values to the Hierarchy Data Base. The *save* parameter, if TRUE, instructs the Navigator to make the values a permanent part of the data base.

int **UpdateTree**(int *repaint*)
This function is typically called at the end of the **Process** to cause the 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 Navigator to repaint the schematic views to reflect the new attribute values.

The following three functions are used to control and interrogate a flag which 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 may be the User_Function in a traversal of the descriptors, i.e.:

```
ForEachDescriptor( ClearDescriptorFlag );
```

int **ForEachGlobalNetName**(int (**User_Function*)(char *the_name))
Parses the list of global net names and calls the *User_Function* with the name of the global net.

int **MajorError**(char ***string*)
Displays an alert prompt showing the given *string*. The function waits until the user clicks the "OK" button before proceeding.

int **SpawnTask**(char ***program*, char ***command_line*, int *wait*)
Launches the selected *program* with the *command_line* as arguments. The *wait* variable determines if the Application may proceed or must wait until the launched *program* has completed. The PC and Macintosh platforms do not support the *wait* option.

Procedural Interface: Symbol & Schematic Data Extraction

Overview

The procedural interface to the symbol and schematic data bases provides a means extracting the graphical and structural data. The interface consists of a several data base 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 are 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 PIK directory or folder of the Programmer's Interface Kit.

The key concepts used in implementing this interface are:

1. Each element in the data base 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 data base.

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 data base 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 data base 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 to indicate that the traversal should abort. 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 may 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 utilize the 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 grid 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 may 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 4.

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
    int type;              Type of the item ( see GR_ in include file )
    int width;             0 = Normal, 1 = Heavy line weight
    int 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
    int font;              0 = Small, 1 = Medium, 2 = Large
    int rot;               0 = Horizontal, 1 = Rotate 90 degrees
    int just;              Justification - 1 through 9 - see below
    char *string;          The text string
    }

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

Justification of text is 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.

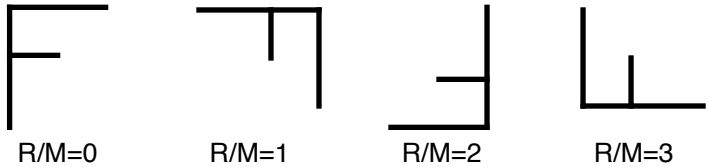
Justification Codes

```
struct _pin {                   Symbol Data - Pin Record
    short 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, 1, 2) 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
    int number;                 Number of the Text Window
    int xo, yo;                 Origin of text in the window
    int font;                   Text size (0, 1, 2) to use
    int just;                   Justification of the text
    int rot;                    Rotation of the text
    }

struct _inst {                  Symbol Instances
    short xo, yo;               Origin of Instance
    short rot_mir;              Rotate/Mirror - Value 0 - 7
    short l, t, r, b;           Bounding Box of the Symbol
    }
```



R/M=0 R/M=1 R/M=2 R/M=3

R/M=4 R/M=5 R/M=6 R/M-7

Table of Rotation / Mirror Codes

```
struct _wire {              Connectivity Elements
    short type;             Type of Element
    short xo, yo;           Start of Element
    short 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

struct _table {             Data Tables
    short 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 height;           height of remaining rows
    short width;            width of remaining columns
    int font[6];            Text size (0, 1, 2) 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;               e.g. 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 PIK. The following variable is defined in the application program.

unsigned long **permission_mask**
This 32 bit mask has bits set for each of the OEM versions of ECS which is permitted to use the application. The values for this mask is defined in the header file "spikproc.h". Normally, this variable should be set to all 1's (0xffffffff).

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 "ecs.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**(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 **LoadSchematic**(char **name*)
Loads the schematic with the specified *name* and the ".sch" suffix. The schematic file is checked for validity.

int **SaveSchematic**(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)
May 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.

int **FreeMemory**(void)
Discards all memory used by symbol and schematic files. Always returns TRUE. Should be used at the end of the process and between processing individual files.

char ***GetSymbolPath**(char **name*)
Returns a character string with the path to the symbol. The full path name is copied into the *name* buffer.

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 aborted. 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.

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 may be attached to the symbol definition and given fixed or default values in the Symbol Editor. The values may 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, char _huge *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 invoked and passed the attribute number and the value of the attribute. The name of the attribute may be obtained with the function **GetSymAttrName**.

int **ForEachSymbolPinAttribute**(PN_PTR *pin*, int *min*, int *max*,
int (**User_Function*)(int the_num, char _huge *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 invoked and passed the attribute number and the value of the attribute. The name of the attribute may 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 **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 may be either a scalar signal or a bus. The relationship between scalar nets and buses is "many:many". A scalar may be contained in one or more buses and a bus may 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 may 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 net lists 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.

The following functions are designed to be called from within a call to **ForEachNetPin** inside of a **ForEachNetFlattened** 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 **ForEachNetFlattened** 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 which 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.

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*)(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 may be attached to symbol types, instances, instance pins, and nets. Individual attribute values may 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 _huge ***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 _huge ***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 which were assigned or overridden on an instance basis in the schematic editor. Attribute values which were originally assigned to the symbol definition may be obtained with the functions **Get_SYA** and **GET_PNA**.

char _huge ***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 _huge ***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 _huge ***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 _huge ***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 may 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 _huge ***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 _huge ***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 **ForEachNetAttribute**(NT_PTR *net*, int *first*, int *last*,
int (**User_Function*)(int the_attr_num, char _huge *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 **ForEachInstanceAttribute**(SI_PTR *symbol_instance*, int first, int last,
int (***User_Function*)(int the_attr_num, char _huge *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 **ForEachPinAttribute**(SP_PTR *symbol_pin*, int *first*, int *last*,
int (***User_Function*)(int the_attr_num, char _huge *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 **ForEachTypeAttribute**(ST_PTR *symbol_type*, int first, int last,
int (***User_Function*)(int the_attr_num, char _huge *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*, char _huge **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*, char _huge **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 net lists 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*, char _huge **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*, char _huge **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*, char _huge **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.

char ***GetNetAttrName**(int *Attrib_Number*)
Returns names for attributes that are associated with nets.

char ***GetPinAttrName**(int *Attrib_Number*)
Returns names for attributes that are associated with symbol pins.

char ***GetSymAttrName**(int *Attrib_Number*)
Returns names for attributes that are associated with symbol definitions and instances.

WORD **GetNetAttrNumber**(char **attrib_name*)
Returns the number of an attribute that is associated with nets.

WORD **GetPinAttrNumber**(char **attrib_name*)
Returns the number of an attribute that is associated with symbol pins.

WORD **GetSymAttrNumber**(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 selected.

Schematic Data - Miscellaneous

NT_PTR **FindNetNamed**(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, i.e. Inches, Centimeters or Millimeters

int **GridSize**(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**(char **name*)
Returns TRUE if the *name* is not a scalar signal.

int **InOutBidir**(NT_PTR *net*)
Returns a value indicating whether the *net* is local to the schematic (0), external to the schematic, i.e., an i/o port (1) or a global net (2).

int **LocExtGbl**(NT_PTR *net*)
Returns a value indicating whether the *net* is an input net (0), an output net (1) or a bidirectional net (2). This function only works for nets which are external to the schematic.

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*, char _huge **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**(char **net_name*, int (**User_Function*)(char *the_name))
Parses the *net_name* into scalar signal names. This is used to expand buses for net listing.

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**.

Utility Functions

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

int **AddExt**(char **name*, 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 '.'. For compatibility among all platforms the *ext* should not exceed three characters after the period.

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

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

Launches the selected *program* with the *command_line* as arguments. The *wait* variable determines if the Application may proceed or must wait until the launched *program* has completed. The PC and Macintosh platforms do not support the *wait* option.

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.

Index