CHAPTER 9

DIAL

DIAL User's Manual

9.1 INTRODUCTION

The Design Interface and Access Library (DIAL) has been created to give users of the SCALDsystem access to the various components of the design data base. It consists of a collection of routines, subprograms, and support utilities that can be assembled into a powerful user-defined interface program written in Pascal. Pascal was chosen as the implementation language for several reasons:

1. Pascal is a well known programming language; the user is likely to have people with Pascal experience available.
2. Pascal, as a general purpose programming language, places no restrictions on the user in terms of the kinds of operations that can be performed.
3. As the SCALDsystem evolves, the interface language remains the same. New library functions may be added, but old interfaces will always work. There is no danger they will become obsolete or have to be changed with new releases.
4. Interfaces written in Pascal, like all of the SCALDsystem software, can be executed on Valid's hardware or the user's host.

DIAL provides the user with a flexible and powerful interface into the SCALDsystem data base. Numerous common operations are provided as utility procedures. Routines that read and process each of the data base files are provided to minimize the amount of work required to implement a custom interface. Using these procedures, the user can format desired reports, output lists, make queries, and perform design verification.

While DIAL can be used for many things, this document is primarily concerned with the implementation of interfaces between the SCALDsystem and other CAD/CAE systems. The use of DIAL for generating reports, performing design rule verification, or making general queries follows naturally from the capabilities described below.

It is assumed that the reader is familiar with the SCALDsystem, its programs and data bases, and the programming language Pascal.

9.2 TYPES OF INTERFACES

DIAL can be used to implement two kinds of interfaces: logical and physical. These interfaces differ in the kinds of information they access, what type of system they are intended to interface to, and how they fit into the SCALDsystem design process. A logical interface is used to access the logical description of the design. It is used, for example to interface to a logic simulator. A physical interface is used to access the physical description of the design. It is used, for example, to interface to a physical layout system. These interfaces are described in more detail in the following sections.

-- SCALDsystem Interface Process Flow --

```
+----------+                +-----------+
|          |                |           |
|  SCALD   |                |   SCALD   |      expanded
| Compiler |--> expansion -->| Packager  |--> part and
|          |       file     |           |      net list
+----------+                +-----------+
                   |              ^               |
                   |              |               |
                   V              |               V
             +-----------+        |         +-----------+
             |           |        |         |           |
             |  Logical  |<-- primitive -->| Physical  |
             | Interface |     libraries    | Interface |
             |           |                  |           |
             +-----------+                  +-----------+
                   |                              |
                   V                              V

                Logical                       Physical
              CAD system                     CAD system
```

LOGICAL INTERFACE

A DIAL logical interface processes the output of the SCALD Compiler (the expansion file). No physical information (pin numbers, location designators, etc.) is accessable by a logical interface since this information is added to the design by the Packager. Information about the design hierarchy, however, is available and, with appropriate restrictions in the use of the SCALD III language, hierarchical descriptions of the design can be generated. SIZE replication can be performed for components that use the SIZE property, so that the user can take advantage of vectored components. TIMES expansion, WIRE GATE elimination, and signal versioning are not performed since these are strictly Packager functions. Designs that use these features MUST be processed with a physical interface or they must be handled by whatever system the interface is being used to communicate with. A logical interface can also access primitive libraries if such are needed by the particular interface.

There is no provision for feedback when using a logical interface since the feedback is handled by the Packager. A logical interface bypasses the Packager making it impossible to generate feedback information that the Packager can understand. If feedback from the destination system is needed, a physical interface MUST be used.

Logical interfaces are used when interfacing to logical CAD tools such as a logic simulator. These tools require a logical description of the design and do not need physical information. Such tools also commonly handle hierarchical designs which are capable of being described with a logical interface.

PHYSICAL INTERFACE

A DIAL physical interface processes the output of the Packager (the expanded part and net lists). The interface has access to all of the physical information in the design, but little of the hierarchical structure - the design is treated as being flat. The interface can also access part libraries if needed. Since the design is processed by the Packager before reaching the physical interface, there are no restrictions in the use of the SCALD III design language.

Feedback from the destination system is possible since the interface processes the Packager's output files.

Physical interfaces are used when interfacing to physical CAD tools such as a PC layout system. The tools require, in general, a flat description of the physical design complete with pin numbers, location designators, and the like. Hierarchical interfaces to physical CAD systems are not directly supported.

9.3 STRUCTURE OF AN INTERFACE

A DIAL interface, whether logical or physical, is implemented in Pascal. The user writes procedures that perform the desired processing of the data base and outputs information in the desired format. DIAL includes several utilities to make the task of implementing an interface easier. Included are procedures to parse input files, handle character strings, create and manipulate data structures, sort items, output information. These are further described below.

DIAL INTERFACE PROGRAM FLOW

A DIAL interface program consists of several phases. DIAL includes routines for each of the phases except output generation, which must be written by the user. The program phases are:

1. Initialization

2. Read the data base

3. Perform processing

4. Design analysis

5. Output the results

The first three steps are common to all DIAL interfaces. The output phase is customized for the particular interface of interest. A template is supplied with DIAL which a user's program should be designed from. This template will define the intended order in which the DIAL procedures should be called. The user may choose to restructure the interface program. The phases listed above are intended only as a guide. The method in which a program can be set up and run on a specific operating system is described in the appendices.

INITIALIZATION

Various tables and runtime "constants" are initialized in this phase. The user can add his own initialization code as well. The initialization also includes the reading of a directive file to allow user-settable control of the execution of the interface program. The following routines do the initialization:

1. INIT_DIAL -- Initialize the DIAL constants.

2. READ_DIRECTIVES_FILE -- DIAL is supplied with a set of standard directives which makes it possible to change the environment for an interface every time it is run. It is also possible for the user to add new directives to the interface by using the routine PROCESS_DIRECTIVES.

DATA BASE INPUT PHASE

Files must be read in to set up the data base to represent a design. The data base to be set up can be either a logical or physical data base. There are three routines which are supplied to set up the appropriate data base. The routines which are supplied are:

1. READ_LOGICAL_DATA_BASE -- Read in the compiler expansion file and set up a data base which can be used to be related to by a logical simulation device. This routine will read in the expansion file, do expansion of replicated parts, assign physical net names, and give a logical part a physical designator.

2. READ_PHYSICAL_DATA_BASE -- Read in the Packager produced expanded net and part list files. These files are used to set up a data base which represent the design in a way which can be used for a physical device.

3. READ_DATA_BASE -- This routine makes it possible for an interface to be either logical or physical. The standard directive INTERFACE_TYPE is supplied which makes it possible for the interface to tell whether the data base should be set up as logical or physical. (see Read_directives_file for information about directives)

DESIGN ANALYSIS

The user may wish to apply some site-specific design rule verification to a design. This is accomplished in this optional phase. A user-written routine is used to walk the data structures and apply certain tests to detect specific problems. Utilities are provided to make accessing the data structures simple and direct.

OUTPUT PHASE

During the output phase, the design is output to text files with user-supplied routines. These routines decide which data are to be output and what the format should be. Utilities are provided to make I/O as straight forward as possible. The user has complete control of the format of the output. An output procedure designed as the last phase of a logical interface can be added as the last phase of a physical interface without modification. This allows the user to get maximum mileage from development efforts. See the section on utilities for a description of routines for implementing the output phase.

9.4 DATA STRUCTURES REPRESENTING THE DESIGN

The design is represented in memory in a number of data structures supported by DIAL. The data structures will be described in this section, but the source declarations for all of the data structures can be found in the type declaration section of the types.pas file. These data structures are designed to be able to represent both the logical and the physical design. For this reason, much of the structure formation revolves around the relation of the logical data to the physical data. When designing a logical interface, some of the physical information may not be needed.

The word "physical", in the case of logical DIAL, means an element which is not in SCALD format. The SCALD language allows considerable latitude in naming parts, signals and pins. Most systems being interfaced with have considerably more restricted notions of what a legal name may consist of. A "physical" name is, in general, an abbreviation of the SCALD name to a simpler form. In the case of a physical interface, these abbreviations take the form of location designators (U31), pin numbers and net names. The notion of physical names is used in a logical interface to make the notions consistent for all DIAL interfaces. In logical DIAL, a physical name is a name not in SCALD format, but is in the user system format. In logical DIAL a physical part is a one section part which has a one to one correspondence with each Valid logical part.

DRAWING NAMES

A Drawing_Name is the occurrance of a drawing in a design. A Drawing_name is defined as:

```
  drawing_name_structure =
     record
       next_in_generic_bucket: drawing_name_ptr;
```

```
    next_in_instance_bucket: drawing_name_ptr;
    first_generic_bucket: drawing_name_ptr;
    generic_name: string;
    instance_name: string;
    is_unique: boolean;
    parts: logical_part_ptr;
  end;
```

A drawing name is entered into two tables: the generic_drawing_table and the instance_drawing_table. The instance table has an entry in it for each time a drawing is used in a design. The instance drawings are connected by the thread NEXT_IN_INSTANCE_BUCKET. The name (INSTANCE_NAME) of the entry is the name of the drawing, with the path of the drawing it is in proceeding it. The generic drawing is the occurance of the actual drawing itself. The name (GENERIC_NAME) of the generic drawing is the actual name of the drawing. If the generic drawing is used more than once in a design IS_UNIQUE is FALSE, and FIRST_GENERIC_BUCKET points to the first entry in the generic table for the drawing. All of the parts which are in the drawing are contained on the list of parts (PARTS).

LOGICAL PARTS

A logical part represents an instance of a part in a drawing. It's name is the path name of the instance which is created and assigned by the SCALD Compiler. The Logical part structure is defined as:

```
 logical_part_structure = record

   next_in_bucket:  logical_part_ptr;        { part table thread }
   next_sorted:     logical_part_ptr;        { list of all parts }
   drawing:         drawing_name_ptr;        { part's drawing }
   part_name:       string;                  { name of logical part }
   part_type:       part_type_ptr;           { physical part type }
   body_properties: property_ptr;            { part propertiess }
   pins_on_part:    logical_pin_ptr;         { pins of the part }
   TIMES_property:  times_property_range;    { TIMES property }
   expanded_parts:  SIZE_expanded_part_ptr;  { SIZE-expanded parts }
 end;
```

The logical parts are kept in a table and linked together with the NEXT_IN_BUCKET field. The NEXT_SORTED field links together all of the logical parts in a particular drawing. The DRAWING field points to the drawing table entry describing the drawing in which the part appears. The logical part name is stored in PART_NAME. The part type is specified by PART_TYPE which points to the library part type description. Properties which are specific to the part are kept in BODY_PROPERTIES. The pins attached to the part are kept in a list pointed to by PINS_ON_PART. The TIMES property attached to the part is specified by TIMES_PROPERTY (This field is not used in Logical DIAL, it will always be zero). The SIZE_expanded parts are kept in a list pointed to by EXPANDED_PARTS.

SIZE EXPANDED PARTS

A SIZE expanded part describes a SIZE replicated logical part. Every logical part has at least one SIZE expanded part (since SIZE > 0). The Size expanded parts represent a two dimensional array of SIZE by TIMES properties (in logical DIAL this will always be a one dimensional array since there is not a TIMES property). A SIZE expanded part is defined as:

```
 SIZE_expand_part_structure = record

   next_part_of_parent: SIZE_expanded_part_ptr;  { SIZE parts }
   next_version:        SIZE_expanded_part_ptr;  { version of part
   parent_part:         logical_part_ptr;        { parent part }
   nodes_on_part:       node_ptr;                { nodes }
   physical_sections:   physical_section_ptr;    { sections }
   SIZE_offset:         bit_range;               { value for SIZE
   version_number:      version_number_range;    { TIMES repl vers
 end;
```

The list of SIZE expanded parts is rooted on a logical part and threaded together with NEXT_PART_OF_PARENT. If there is a TIMES property associated with a part, the list of new versions is rooted in a SIZE_expanded_part and threaded with NEXT_VERSION. The parent logical part is specified by PARENT_PART. The nodes connected to the part are NODES_ON_PART and are sorted by physical pin number (after physical section assignment). The physical section allocated to the SIZE_expanded_part is listed by PHYSICAL_SECTIONS.

LOGICAL PINS

For each pin on the logical part, there is an associated logical pin structure which describes this pin. The Logical Pin structure is defined as:

```
 logical_pin_structure = record

   next_pin_on_part: logical_pin_ptr; { list of pins of part }
   pin_def:          pin_def_ptr;     { physical pin description }
   left_bit:         bit_range;       { the MSB of the bit subscri
   right_bit:         bit_range;         { the LSB of the bit subscript }
   was_scalar:        boolean;           { not used }
   pin_properties:    property_ptr;      { instance specific properties }
 end;
```

Logical pin list are rooted on a logical part and threaded with NEXT_PIN_ON_PART. The physical pin is described by PIN_DEF (where the pin name and pin numbers can be found). The bit subscript for the pin are kept in LEFT_BIT (the most significant bit) and RIGHT_BIT (the least significant bit). The field WAS_SCALAR will not be used by a DIAL program. The properties of the pin (instance specific properties) are listed in PIN_PROPERTIES.

NODES

A node describes a pin on a specific part. The node structure is defined as:

```
 node_structure = record

   next_node_on_net:      node_ptr;                   { nodes on a net }
   next_node_on_part:     node_ptr;                   { nodes on a part }
   next_version_of_node:  node_ptr;                   { not used }
   net:                   net_ptr;                    { net node is on }
   SIZE_expanded_part:    SIZE_expanded_part_ptr;     { logical part }
   physical_pin:          section_pin_ptr;            { physical pin desc }
   logical_pin:           logical_pin_ptr;            { logical pin desc
   bit_offset:            bit_range;                  { bit offset }
   node_type_or_version:  version_number_range;       { unused }
 end;
```

Lists of nodes are found on nets (using the NEXT_NODE_ON_PART thread) and SIZE_expanded logical parts (using the NEXT_NODE_ON_PART thread - ordered by pin and offset). The NEXT_VERSION_OF_NODE field is not used by a user's DIAL program. Each node points to the net to which it is connected (NET). The pin is described by PHYSICAL_PIN and LOGICAL_PIN (which also uses the logical bit offset (BIT_OFFSET). The logical part for the node is SIZE_EXPANDED_PART. The NODE_TYPE_OR_VERSION field is not used by a user's DIAL program.

NETS

A net describes a one bit wide logical/physical net. The net structure is defined as:

```
 net_structure = record
```

```
   next_logical_net_in_bucket:  net_ptr;     { logical net name thread
   next_physical_net_in_bucket: net_ptr;     { phys net name thread }
   logical_name:                string;      { logical net name }
   bit_offset:                  bit_range; {  bit subscript }
   physical_name:               string;      { physical net name }
   nodes_on_net:                node_ptr;    { nodes on the net }
   version_number:   version_number_range; { version due to TIMES }
   versions_used:    version_number_range; { not used }
   net_properties:           property_ptr; { properties of the net }
 end;
```

The logical name for the net (LOGICAL_NAME) is the logical signal name. The logical name also includes a bit offset (BIT_OFFSET). If the logical net name is a scalar (has no bit subscript), the bit offset is set to -1. Each net also has a physical name (PHYSICAL_NAME). The nets are kept in two tables: physical_net_table (using the NEXT_PHYSICAL_NET_IN_BUCKET thread) and the logical net table (using the NEXT_LOGICAL_NET_IN_BUCKET thread). Each net has a list of nodes to which it is attached (NODES_ON_NET). If the net was created because of TIMES property processing, it is given a version number (VERSION_NUMBER). It is not possible to have versions in Logical DIAL, so this field does not apply. The properties of the net are listed in NET_PROPERTIES. The field VERSIONS_USED is not used by DIAL.

There are three special nets which are passed from a design to DIAL. These nets are the NC, 0 and 1 nets. The NC net contains all of the nodes which are not explicitly connected to any net. If these nodes are to be printed out, each node should have an integer appended to it. Then, all of the nodes will not be tied together.

The 0 and 1 nets are constant signals. The 0 net has a logical net name of the numeric 0 and the physical net name of "ZERO". This net can be used in a logic simulator to show the net has a constant value of 0. The 1 net has a logical net name of the numeric 1 and a physical net name of "ONE". This net can be used in a logic simulator to show the net has a constant value of 1.

PART TYPES

A PART_TYPE describes a physical part type. The part types are defined in a chips file. The Part Type structure is defined as:

```
 part_type_structure = record

   next_in_bucket:             part_type_ptr;         { next part }
```

```
    part_type_name:              string;                 { name of part }
    sections_on_part:            section_def_ptr;        { sections }
    number_of_pins:              pin_count_range;        { # pins on part }
    pins:                        pin_list_ptr;           { pins of part }
    pin_defs:                    pin_def_ptr;            { pin names }
    has_common_pins:             boolean;                { if common pins }
    common_pins:                 pin_def_ptr;            { common pins }
    body_properties:             property_ptr;           { properties }
    partially_allocated_parts: physical_part_ptr;        { not used }
    fully_allocated_parts:       physical_part_ptr;      { fully alloc }
    number_of_sections:          section_number_range; { # of sections }
    power_pins:                  power_pin_list_ptr;     { power pins }
    is_wire_or:                  boolean;                { not used }
    is_wire_and:                 boolean;                { not used }
    is_flag_body:                boolean;                { flag body }
    found_in_Chips_file:         boolean;                { if found }
  end;
```

A table of part types is kept with entries threaded by NEXT_IN_BUCKET. The name of the part type is specified by PART_TYPE_NAME. A list of the sections and all of the pins connected to each type is stored in SECTIONS_ON_PART. The number of the pins on the part is stored in NUMBERS_OF_PINS. The pins of the part are listed in order (by ascending pin number) in a list specified by PINS. The names of the pins are specified in a list rooted at PIN_DEFS. If the part has any pins common to more than one section, HAS_COMMON_PINS will be TRUE. If HAS_COMMON_PINS is true, COMMON_PINS threads the PIN_DEFS of the pins. The properties of the part type are kept in BODY_PROPERTIES. The field PARTIALLY_ALLOCATED_PARTS is not used in DIAL. A list of all physical parts of this part type is kept in FULLY_ALLOCATED_PARTS. The number of sections contained in the part is specified in NUMBER_OF_SECTIONS. The power pins of the part are kept in a list (POWER_PINS). The fields IS_WIRE_AND and IS_WIRE_OR are not used in DIAL. When the part type is found in the Chips file and completed, the FOUND_IN_CHIPS_FILE flag is set TRUE.

There are some part types which are not part of the design, but they may appear in the logical design. These parts are the wire gate bodies which are used for versioning and flag bodies and a root part type which are used to find the interface signals. When a design is being written out, it should be taken into account that these bodies are not part of the actual design, but they are still there. There are routines which are supplied to handle these parts.

PIN DEFS

A PIN_DEF describes a physical pin as defined in a chips file. A pin_def is defined as:

```
  pin_def_structure = record

        next_pin_on_part:     pin_def_ptr;       { thread for pins }
        next_common_pin:      pin_def_ptr;       { common pins }
        part_type:            part_type_ptr;     { pin's part type }
        pin_name:             string;            { logical name }
        bit_offset:           bit_range;         { bit offset }
        left_bit:             bit_range;         { MSB }
        right_bit:            bit_range;         { LSB }
        pin_properties:       property_ptr;      { properties }
        is_output_pin:        boolean;           { pin is output }
        is_input_pin:         boolean;           { pin is input }
        drive_0_state:        real;              { 0-state drive }
        drive_1_state:        real;              { 1-state drive }
        load_0_state:         real;              { 0-state load }
        load_1_state:         real;              { 1-state load }
        is_a_common_pin:      boolean;           { pin in >1 section }
        is_a_total_common_pin: boolean;          { common to ALL sect }
        containing_sections: set_of_sections;  { sections with pin }
        section_pins:         section_pin_ptr;  { pin by section }
        is_wire_gate_output: boolean;           { not used }
        found_in_Chips_file: boolean;           { in Chips file }
     end;
```

A list of all the pins for a part is rooted on the part type and threaded with NEXT_PIN_ON_PART. The next pin in the list of common pins of the part type is NEXT_COMMON_PIN. The part type the pin is part of is specified by PART_TYPE. The name of the pin (logical_name) is specified by PIN_NAME. The bit offset for the pin is specified by BIT_OFFSET (which is -1 if the pin is a scalar). The union of the bit subscripts referred to by all of the logical parts is specified by LEFT_BIT (the Most Significant Bit) and RIGHT_BIT (the Least significant Bit). After the Chips file is read, LEFT_BIT and RIGHT_BIT specify the range for the pin on a physical part section.

The properties attached to the pin are specified by PIN_PROPERTIES. If the pin is an input pin, IS_INPUT_PIN is TRUE. If the pin is an output pin, IS_OUTPUT_PIN is TRUE. The DC current drive and loading for both the 0-state and 1-state are specified by DRIVE_0_STATE, DRIVE_1_STATE, LOAD_0_STATE and LOAD_1_STATE. If the pin appears in more than one section, the IS_A_COMMON_PIN flag is TRUE. If the pin is common to ALL sections of the part and has the same pin number on all sections the IS_A_TOTAL_COMMON_PIN flag is set to TRUE.

A set of the sections (by number) in which the pin appears is kept in CONTAINING_SECTIONS. The pins of the part are listed in order by section in SECTION_PIN. Each pin_def is initially constructed when a pin is referenced in the compiler expansion file. When the pin is found in the Chips file and completed, the FOUND_IN_CHIPS_FILE flag is set to TRUE. The field IS_WIRE_GATE_OUTPUT is not used in DIAL.

POWER PINS

A power pin describes a pin of the part connected to a power supply. A power pin on a part is defined as:

```
power_pin_list =
      record
        next_power_pin: power_pin_list_ptr; { next in the list }
        power_pin: power_pin_name_ptr;      { name of the pin }
        pin_number: name_ptr;               { power pin number }
      end;
```

The next pin in the list of the power pins is given by NEXT_POWER_PIN. The POWER_PIN points to a description of a generic power pin. The pin number corresponding to the power pin is given by PIN_NUMBER. The list is rooted in a part_type and is sorted by position contained in the power pin name description.

A power pin name describes a generic power pin. This list contains all of the power pins in the design and it is rooted in a global variable (POWER_PIN_NAMES) and is sorted by pin name. The power pin name is defined as:

```
power_pin_name =
       record
         next_power_pin: power_pin_name_ptr; { next in list }
         pin_name: name_ptr;                 { name of the pin }
         position: power_pin_position_range; { position in output }
       end;
```

The next in the list of power pin names is given by NEXT_POWER_PIN. The name of the power pin is PIN_NAME. Its position in the output list is specified by POSITION.

SECTION DEFS

A section def describes all the pins attached to a section of a part. The definition of a section def is:

```
 section_def_structure = record

   next_on_part:    section_def_ptr;          { next section }
   section_number:  section_number_range;     { section number }
   number_of_pins:  pin_count_range;          { number of pinss }
   pins:            section_pin_ptr;          { list of pins }
 end;
```

The list of sections is rooted in a part type and threaded by NEXT_ON_PART. The number of the section is SECTION_NUMBER. NUMBER_OF_PINS specifies the number of pins that appear in this section. An ordered list of all the pins on the section is kept in PINS.

SECTION PINS

A Section_pin describes a pin of a section. The section pin is defined as:

```
 section_pin_structure = record

   next_on_section:      section_pin_ptr;          { list of pins }
   next_section_of_pin:  section_pin_ptr;          { list of sections }
   next_common_pin:      section_pin_ptr;          { next common pin }
   first_common_pin:     section_pin_ptr;          { first common pin }
   section_number:       section_number_range;     { section number }
   pin_def:              pin_def_ptr;              { pin description }
   pin_number:           name_ptr;                 { pin's number }
 end;
```

The section pins are listed by pin name (in s list rooted at a pin def) and threaded by NEXT_SECTION_OF_PIN. All pins on a section are listed (in a list rooted on a section_def) and threaded by NEXT_ON_SECTION. A list of all pins that are common (share the same physical pin) are threaded by NEXT_COMMON_PIN. The number of the section in which the pin resides is SECTION_NUMBER. The pin definition is specified by PIN_DEF. The pin number is specified by PIN_NUMBER.

PHYSICAL PARTS

A physical part describes an instance of a physical part. A physical part may be a physical package, as in physical DIAL or a physical part may be a a part which has been transformed out of the VALID representation into another representation, as in logical DIAL. A physical part is defined as:

```
 physical_part_structure = record

   next_in_bucket:            physical_part_ptr;       { next part }
   next_part_of_same_type:    physical_part_ptr;       { next same type }
   sections_of_part:          physical_section_ptr;    { sections }
   part_name:                 string;                  { LOCATION }
   part_type:                 part_type_ptr;           { part's type }
   has_been_removed:          boolean;                 { not used }
   group:                     group_ptr;               { not used }
 end;
```

All physical parts are kept in a table whose entries are threaded by NEXT_IN_BUCKET. A list of all the physical parts that have the same part type are threaded by NEXT_PART_OF_SAME_TYPE which is rooted in the appropriate part type. A list of all of the sections of the part is rooted in SECTIONS_OF_PART. The physical part's name is stored in PART_NAME. The part type of the physical part is indicated by PART_TYPE. The fields HAS_BEEN_REMOVED and GROUP are not used by DIAL.

PHYSICAL SECTIONS

A physical_section describes a section of a physical part. A physical section is defined as:

```
  physical_section_structure = record

      next_section_of_part:    physical_section_ptr;    { next sect }
      physical_part:           physical_part_ptr;       { phys part }
      section_number:          section_number_range;    { which sect }
      SIZE_expanded_part:      SIZE_expanded_part_ptr;  { which part }
    end;
```

All sections of a particular physical part are linked by the NEXT_SECTION_OF_PART thread (the entries in the list are ordered by ascending section number) rooted in the physical part that these sections are a part of (specified by PHYSICAL_PART). The section number for the section is specified by SECTION_NUMBER (the same section number as specified in the part type). The logical part that the section is allocated to is specified by SIZE_EXPANDED_PART. If SIZE_EXPANDED_PART is NIL, the physical section has yet to be allocated.

PROPERTIES

Properties can appear on almost every object in the design: logical parts, logical pins, part types, part type pins, and nets. A property is defined as:

```
 property_list = record
                   next: property_ptr;     { next property }
                    name: name_ptr;        { property name }
                    text: string;          { property value }
                  end;
```

The list of properties are threaded by NEXT. The property has a property name associated with it (NAME), as well as a value for the property (TEXT).

9.5 GRAPH OF STRUCTURES

The following graph is a simple pictorial description of the interrelation of the data structures.

```
+------------------+     +------------+    +------------------+
|                  |     |            |    |                  |
|logical_part_table|---->|logical_part|--->|SIZE_expanded_part|->...
|                  |     |            |    |                  |
|                  |     +------------+    +------------------+
|                  |      |     |                    |       |
|                  |   ----     |                    |       |
+------------------+   |        V                    V       |
                       |  +-----------+           +------+   |
                       |  |           |           |      |   |
                       |  |logical_pin|---------->| node |-- |
                       |  |           |--         |      | | |
                       |  +-----------+ |         +------+ | |
                       |        |       |            |     | |
                       |        V       |            V     | |
                       |        .       |            .     | |
                       V        .       V            .     | |
+----------+      +---------+      +-------+         .     | |
|          |      |         |      |       |               | |
|part_table|----->|part_type|----->|pin_def|-->...         | |
|          |   -->|         |      |       |               | |
|          |   |  +---------+      +-------+               | |
|          |   |       |               |     --------------- |
|          |   |       |               |     |               |
+----------+   |       V               V     V               |
               |  +-----------+    +-----------+             |
               |  |           |    |           |             |
               |  |section_def|--->|section_pin|--->...      |
               |  |           |    |           |             |
               |  +-----------+    +-----------+             |
               |                                             |
               |-----------------                            |
                                |                            |
                                V                            V
+-------------------+     +-------------+     +----------------+
|                   |     |             |     |                |
|physical_part_table|---->|physical_part|---->|physical_section|
|                   |     |             |<----|                |
|                   |     +-------------+     +----------------+
|                   |
+-------------------+
```

9.6 TABLE STRUCTURE ROUTINES

The main data structures of DIAL, which are described in the structure section, are kept in their respective tables. Each of these tables have routines which are designed to manipulate them. The routines are described below.

LOGICAL PARTS

The Logical Part Structure, which was described in the structure chapter, is kept in the Logical Part Table. The following routine is supplied to find entries in this table:

FIND_LOGICAL_PART

```
function find_logical_part(name: string;
                  var part: logical_part_ptr): boolean;
```

Search for the given logical part (specified by the part's path name only) in the table. If found, return TRUE and a pointer to the part. If not found, return FALSE.

NETS

The two Net structures, which are described in the structures section, are kept in two tables. The Physical Nets are kept in the Physical Net Table and the Logical Nets are kept in the Logical Net Table. Both tables have a routine which makes it possible to find entries in the tables. The routines are:

FIND_LOGICAL_NET

```
function find_logical_net(name:string; subscript:
      integer; version_number: version_number_range;
      var net: net_ptr): boolean;
```

Search for the given logical net in the logical net name table and return FALSE if not found. If found, return TRUE and a pointer to the net.

FIND_PHYSICAL_NET

```
function find_physical_net(name: string; var net:
      net_ptr): boolean;
```

Search for the given physical net in the physical net table. If not found return FALSE, otherwise return TRUE and a pointer to the net.

PART TYPES

The part type structures, which are described in the strucure chapter, are kept in the part type table. The routine which finds entries in this table is:

FIND_PART_TYPE

```
function find_part_type(part_name: string;
    var part_type: part_type_ptr): boolean;
```

Search for the given part type in the part type table. If not found return FALSE, otherwise return TRUE and a pointer to the part type.

PHYSICAL PARTS

The physical part structures, which was described in the structure chapter, are kept in the physical part table. The routine to find entries in this table is:

FIND_PHYSICAL_PART

```
function find_physical_part(part_name: string;
    var physical_part: physical_part_ptr): boolean;
```

Search for the given physical part in the physical part table. If not found return FALSE, otherwise return TRUE and a pointer to the physical part.

9.7 OTHER DIAL DATA STRUCTURES

There are several utility data structures provided that are intended to make it easier to implement a DIAL interface. These are described below.

NAMES

The standard form for character manipulation is with the ALPHA type defined as follows:

```
ALPHA = PACKED ARRAY [1..16] OF CHAR;
```

This construct is useful for much work with text but is severely restricted (see the section on strings below for a more flexible character string manipulation mechanism). Alpha names are used most frequently to represent property names. Since property names (and other name as well) are often reused - the PATH property, for example, appears on every logical part - all alpha names are placed in a table. This allows names to be compared by comparing pointers: a definite efficiency improvement. The table is maintained with the following routine:

ENTER_NAME

```
   function enter_name(name: alpha): name_ptr;
```

Search the alpha name table for the given alpha name. If not found, enter it into the table. Return a pointer to the name.

Names are represented by the pointers from this table and have the form:

```
     NAME_PTR = ^NAME_TYPE;
     NAME_TYPE = RECORD
                   NEXT_NAME: NAME_PTR;
                   NAME: ALPHA;
                 END;
```

STRINGS

Pascal provides no support for a dynamic string type. A heap allocated string package is therefore provided. The basic data type is the STRING with the following definitions:

```
     MAX_STRING_LENGTH = 255;
     STRING = ^STRING_TYPE;
     STRING_TYPE = PACKED ARRAY [0..MAX_STRING_LENGTH] OF
     CHAR;
```

The first element of the string is interpreted as being the string's length. IT IS FIXED AT THE TIME OF CREATION AND CANNOT BE CHANGED! A dynamic string is available by creating a string of MAX_STRING_LENGTH. The length of such strings can be manipulated. The manipulation routines assume that the string's length can be extended to MAX_STRING_LENGTH with the first character defining the string's current length. Strings can be released and the space returned to a free list from which future strings will be allocated. The routines that form the string package are:

CREATE_A_STRING

```
   procedure create_a_string(var str: string; length:
   string_range);
```

Create a new string of the given length. If there is a string of the desired length in the free string list, use it, otherwise create a new string from the heap.

RELEASE_STRING

```
procedure release_string(var str: string);
```

Release the given string returning the string to the free string list.

COPY_STRING

```
procedure copy_string(source: string; var dest: string);
```

Copy the source string to the destination string. If the destination string is not the correct length, release it, create a new string of the correct length, and copy the source string into it.

COPY_FROM_STRING

```
procedure copy_from_string(str: string; var name: alpha);
```

Copy from the given string into the given ALPHA. Truncate if the string is longer than 16 characters and blank pad if shorter.

COPY_TO_STRING

```
procedure copy_to_string(name: alpha; var str: string);
```

Copy from the given ALPHA to the given string. Do not copy trailing blanks. If the string is not of the correct length, release it, create a new string, and copy to it.

CMPSTRLEQ

```
function CmpStrLEQ(s1, s2: string): boolean;
```

Return TRUE if the first string is <= the second string.

CMPSTRLT

```
function CmpStrLT(s1, s2: string): boolean;
```

Return TRUE if the first string is < the second string.

CMPSTRGT

```
function CmpStrGT(s1, s2: string): boolean;
```

Return TRUE if the first string is > the second string.

CMPSTREQ

```
function CmpStrEQ(s1, s2: string): boolean;
```

Return TRUE if the two strings are equal.

COMPARE_STRINGS

```
function compare_strings(s1, s2: string): compare_type;
```

Return an enumerated type (LT, EQ, GT) after comparing the two given strings.

ADD_CHAR_TO_STRING

```
function add_char_to_string(str: string; ch: char):
boolean;
```

Add the given character to the end of the given string. It is assumed that the string can be extended to MAX_STRING_LENGTH. Return FALSE if the string is overflowed.

ADD_STRING_TO_STRING

```
function add_string_to_string(dest, source: string):
boolean;
```

Add the second string to the end of the first string. It is assumed that the destination string has been created with MAX_STRING_LENGTH. This is very important since the routine assumes this is the case and if this is not the case some other string on the heap will be changed when the two strings are added together. The best way to use this procedure is to have the destination string be a temporary storage string. The reason for this is that if a lot of string adding is to be done, a lot of strings of MAX_STRING_LENGTH must be created and this will eat up all of the storage allocated for the string heap very quickly. The temp string should be created with MAX_STRING_LENGTH, then the first byte of the string should be set to zero length by the following Pascal instructions:

```
create__a__string(foo, MAX__STRING__LENGTH):
foo^[0] := chr(0);
```

These two instructions cause a string named 'foo' to be created with MAX_STRING_LENGTH. The string is then changed to make it of length zero but there are still MAX_STRING_LENGTH bytes allocated to it. Now when another string is added to 'foo' there will be enough bytes allocated to make it possible to make the addition. If the total number of bytes when the two strings are added together are greater than MAX_STRING_LENGTH, the funcion will return FALSE.

ADD_ALPHA_TO_STRING

```
function add_alpha_to_string(dest: string; ident: alpha):
boolean;
```

Add the given alpha to the end of the given string. Do not copy trailing blanks from the alpha. It is assumed that the string can be extended to MAX_STRING_LENGTH. Return FALSE if the string is overflowed.

ADD_NUMBER_TO_STRING

```
function add_number_to_string(str: string; number:
                    integer): boolean;
```

Add the given integer (signed) to the end of the given string. It is assumed that the string can be extended to MAX_STRING_LENGTH. Return FALSE if the string is overflowed.

CONCAT_STRING_TO_STRING

```
function concat_string_to_string(var dest: string;
                    s1, s2: string): boolean;
```

Add one string (S2) to the end of another string (S1).

Another string will be created on the string heap which will hold the result of the addition of the strings. If the lengths of the two strings add up to more than MAX_STRING_LENGTH characters, the resulting string will be truncated to MAX_STRING_LENGTH characters and the function will return FALSE.

CONCAT_ALPHA_TO_STRING

```
function concat_alpha_to_string(var dest: string;
                    str: string;
                    ident: alpha): boolean;
```

Add the alpha (IDENT) to the end of the string (STR). A string (DEST) will be made which contains the result of the addition. If the resulting string is greater than MAX_STRING_LENGTH characters, the string will be truncated and the function will return FALSE.

CONCAT_CHAR_TO_STRING

```
function concat_char_to_string(var dest: string;
                    str: string;
                    ch: char): boolean;
```

Add the character (CH) to the end of the string (STR). The result will be put into a newly created string (DEST). If adding the character makes the string greater than MAX_STRING_LENGTH the character will not be added and the function will return FALSE.

CONCAT_NUMBER_TO_STRING

```
function concat_number_to_string(var dest: string;
                    str: string;
                    number: integer): boolean;
```

Add the number (NUMBER) to the end of the string (STR). The result will be placed into a newly created string (DEST). If the addition results in a string which is greater than MAX_STRING_LENGTH, the result wil be truncated and the function will return FALSE.

PROPERTIES

The property structure, which was described in the structures section of this manual, is greatly used in DIAL. Therefore the following routine is needed to find an entry in a list of properties.

ADD_TO_PROP_LIST

```
procedure add_to_prop_list(var prop_list: property_ptr;
                           property_name: name_ptr;
                           property_value: string);
```

Add the given property to the property list. PROPERTY_NAME is the name which the property will be known as, and PROPERTY_VALUE is the value of the new property.

FIND_PROPERTY

```
function find_property(prop_list: property_ptr;
              name: name_ptr;
              var property: property_ptr): boolean;
```

Search for the property pointed to by the name pointer in the property list pointed to by the property pointer. If not found return FALSE, otherwise return a pointer to the property and return TRUE.

9.8 DIAL UTILITIES

DIAL includes several utilities for use in implementing interfaces. It should be noted that it is Valid's goal to develop and support programs that run on the VAX, and 370 as well as the VALID machines. For this reason, ALL of Valid's analysis tools are written in standard Pascal (Jensen and Wirth). This is true for DIAL as well. Utilities are provided that extend the capabilities of Pascal and make it easier to use. None of the DIAL utilities use non-standard extensions to Pascal that may be present in any particular pascal compiler.

INPUT/OUTPUT

File I/O Routines

The following routines are designed to be used to control the user's files, so they can be written and read from in a correct manner.

RESET_FILE

```
function reset_file(filename: string;
     which: parse_file_type): boolean;
            unknown_file
```

Open the specified file for read using the given file name (FILENAME). If the file cannot be opened, return FALSE otherwise return TRUE.

OPEN_A_FILE

```
function open_a_file(filename: string;
     which: parse_file_type): boolean;
```

Open the specified file (F) for read. Calls INSYMBOL to read the first token from the file. If the file cannot be opened, return FALSE, otherwise return TRUE.

REWRITE_FILE

```
function rewrite_file(var f: textfile;
        which: output_file_names): boolean;
```

Open the specified file (F) for write. If the file cannot be opened, return false, otherwise return TRUE.

CLOSE_OUTPUT_FILE

```
procedure close_output_file(var f: textfile;
                            which: output_file_names);
```

Close the specified output file (F). If the file cannot be closed, output an error message.

CLOSE_PARSE_FILE

```
procedure close_parse_file(which: parse_file_type);
```

Close the specified input file (WHICH). If the file cannot be closed, output an error message.

REWRITE_NAMED_FILE

```
function rewrite_named_file(var f: textfile;
                            file_name: alpha): boolean;
```

Open the specified file (F) for writing. The file will be opened using the given name (FILE_NAME).

CLOSE_NAMED_FILE

```
function close_named_file(var f: textfile;
                          file_name: alpha): boolean;
```

Close the specified file (F). The file will be closed using the given name (FILE_NAME).

Simple Print Routines

Simple print routines are given to make it easier for a user to print specified data types out to a file. These routines do not have any formatting capabilities.

PRINT_CRLF

```
procedure print_crlf(var f: textfile);
```

Print an EOL (end_of_line) to the given file (F).

PRINT_INDENT

```
procedure print_indent(var f: textfile; num:
natural_number);
```

Print the given number (NUM) of blanks to the given file (F).

PRINT_STRING

```
procedure print_string(var f: textfile; str: string);
```

Print the given string (STR) to the given file (F).

PRINT_STRING_WITH_QUOTES

```
procedure print_string_with_quotes(var f: textfile; str:
string);
```

Print the given string (STR) to the given file (F) with quotes delimiting.

PRINT_ALPHA

```
procedure print_alpha(var f: textfile; name: alpha);
```

Print the given alpha (NAME) to the given file (F).

PRINT_NAME

```
procedure print_name(var f: textfile; name: name_ptr);
```

Print the name which is pointed to by a pointer (NAME_PTR) to the given file (F).

PRINT_INTEGER

```
procedure print_integer(var f: textfile; num: integer);
```

Print the given integer (NUM) to the given file (F).

Print With Continue

These routines enhance the user's capability to print out specified data structures. With these routines it is possible for a user to specify a line length. If the given data structure is too long to fit on a given line, a user specified continuation character will be printed out on the current line and the remainder of the data structure will be printed on the next line.

The following variables can be specified by the user:

continuation_char:

This variable specifies the continuation character which will be printed out when a line is overflowed. The default is a tilde "~".

max_output_file_length:

Specifies the maximum line length. The default is 80 characters per line.

continue_at_end:

Specifies whether continuation character (CONTINUATION_CHAR) should be printed at the beginning or the end of a line when it overflows.

column:

Keeps track of the column of the current line.

Since column is an integer, it is not possible for the print continue routines to be used by more than one file. If more than one file is to be written to using the print continue routines, the current value of column must be saved and restored whenever changing from printing to one file to another file.

INIT_OUTPUT_CONTINUE

```
procedure init_output_continue;
```

Initialize the column global variable for continuation output. This routine must be called before any Print_Continue routine is to be used on a file.

PRINT_CRLF_CONTINUE

```
procedure print_crlf_continue(var f: textfile);
```

Print an EOL (end-of-line) to the given file (F). Column is set to zero.

PRINT_INDENT_CONTINUE

```
procedure print_indent_continue(var f: textfile;
                                    num: natural_number);
```

print out the given number of blanks (NUM) to the given file (F). The global variable COLUMN is incremented for each blank which is output.

PRINT_STRING_CONTINUE

```
procedure print_string_continue(var f: textfile; str:
string);
```

Print the given string (STR) to the given file (F). If the end of the current line is reached, a continuation character will be printed, and the string will be continued on the next line.

PRINT_STRING_QUOTED_CONTINUE

```
procedure print_string_quoted_continue(var f: textfile;
str: string);
```

Print the given string (STR) to the given file (F) with quotes surrounding the string. If the end of the current line is reached, a continuation character will be printed at either the end of the current line or the beginning of the new line depending on the value of continue_at_end. The rest of the string and the ending quote will be printed on the new line.

PRINT_ALPHA_CONTINUE

```
procedure print_alpha_continue(var f: textfile; name:
alpha);
```

Print the given alpha (NAME) to the given file (F). If the end of the current line is reached, a continuation character will be printed, and the alpha will be continued on the next line.

PRINT_NAME_CONTINUE

```
procedure print_name_continue(var f: textfile; name:
name_ptr);
```

Print the name table entry pointed to be NAME_PTR to the given file (F).

PRINT_CHAR_CONTINUE

```
procedure print_char_continue(var f: textfile; ch: char);
```

Print the given character (CH) to the given file (F). If the end of the current line is reached, a continuation character will be printed, and the char will be continued on the next line.

PRINT_INTEGER_CONTINUE

```
procedure print_integer_continue(var f: textfile; num:
integer);
```

Print the given integer (NUM) to the given file (F). If the end of the current line is reached, a continuation character will be printed, and the integer will be continued on the next line.

Print Token Routines

These routines enhance the user's capability to print out specified data structures. With these routines it is possible for a user to specify a line length. If the given data structure is too long to fit on a given line, a user specified continuation character will be printed out on the current line and the data structure will be printed on the next line.

The following variables can be specified by the user:

continuation_char:

This variable specifies the continuation character which will be printed out when a line is over flowed. The default is a tilde "~".

max_output_file_length:

Specifies the maximum line length. The default is 80 characters per line.

continue_at_end:

Specifies whether continuation character (CONTINUATION_CHAR) should be printed at the beginning or the end of a line when it overflows. If continue_at_end is TRUE the continuation character will be printed at the end of the line.

column:

Keeps track of the column of the current line.

Since column is an integer, it is not possible for the print continue routines to be used by more than one file. If more than one file is to be written to using the print continue routines, the current value of column must be saved and restored whenever changing from printing to one file to printing to another file.

PRINT_STRING_TOKEN_CONTINUE

```
procedure print_string_token_continue(var f: textfile;
str: string);
```

Print the string (STR) to the given file (F). If the current value of column plus the number of printable characters in the string will make the line greater than MAX_OUTPUT_FILE_LENGTH then a contiuation character is printed. The continuation character is either printed at the end of the current line or the beginning of the next line depending on whether continue_at_end is TRUE or FALSE. After a continuation character is printed and a new line is started, the printable characters in the string are printed.

PRINT_ALPHA_TOKEN_CONTINUE

```
procedure print_alpha_token_continue(var f: textfile;
name: alpha);
```

Print the alpha (NAME) to the given file (F). If the current value of column plus the number of printable characters in the alpha will make the line greater than MAX_OUTPUT_FILE_LENGTH then a continuation character is printed. The continuation character is either printed at the end of the current line or the beginning of the next line depending on whether continue_at_end is TRUE or FALSE. After a continuation character is printed and a new line is started, the printable characters in the alpha are printed.

PRINT_NAME_TOKEN_CONTINUE

```
procedure print_name_token_continue(var f: textfile;
name: name_ptr);
```

Print the name pointed to by the pointer (NAME_PTR) to the given file (F). If the current value of column plus the number of printable characters in the name will make the line greater than MAX_OUTPUT_FILE_LENGTH then a contiuation character is printed. The continuation character is either printed at the end of the current line or the beginning of the next line depending on whether continue_at_end is TRUE or FALSE. After a continuation character is printed and a new line is started, the printable characters in the name are printed.

PRINT_INTEGER_TOKEN_CONTINUE

```
procedure print_integer_token_continue(var f: textfile;
num: integer);
```

Print the integer (NUM) to the given file (F). If the current value of column plus the number of digits in the number will make the line greater than MAX_OUTPUT_FILE_LENGTH, then a contiuation character is printed. The continuation character is either printed at the end of the current line or the beginning of the next line depending on whether continue_at_end is TRUE or FALSE. After a continuation character is printed and a new line is started, the integer is printed.

PRINT_CHAR_TOKEN_CONTINUE

```
procedure print_char_token_continue(var f: textfile; ch:
char);
```

Print the character (CH) to the given file (F). If the current value of column plus one will make the line greater than MAX_OUTPUT_FILE_LENGTH, then a continuation character is printed. The continuation character is either printed at the end of the current line or the beginning of the next line depending on whether continue_at_end is TRUE or FALSE. After a continuation character is printed and a new line is started, the character is printed.

PRINT_CRLF_TOKEN_CONTINUE

```
procedure print_crlf_token_continue(var f: textfile);
```

Write a carriage return-linefeed into the specified file (F). Column is set to zero.

PRINT_INDENT_TOKEN_CONTINUE

```
print_indent_token_continue(var f: textfile);
```

Write out as many blanks as specified by the variable continue_indent. If the spaces cannot fit on the current line, a new line will be started and the new line will be indented those spaces.

Print Left Justified Routines

These routines enhance Pascal to print out a given data type left justified in a field with a given length.

PRINT_STRING_LEFT_JUST

```
procedure print_string_left_just(var f: textfile;
   str: string; length: natural_number);
```

Print the given string (STR) to the given file (F) left justified in a field of the given length (LENGTH). If the string is less than the specified length, print the string and pad with blanks. If the string is greater than the specified length, truncate the string.

PRINT_ALPHA_LEFT_JUST

```
procedure print_alpha_left_just(var f: textfile;
   name: alpha; length: natural_number);
```

Print the given alpha (NAME) to the given file (F) left justified in a field of the given length (LENGTH). If the alpha is less than the specified length, print the alpha and pad with blanks. If the alpha is greater than the specified length, truncate the alpha.

PRINT_NAME_LEFT_JUST

```
procedure print_name_left_just(var f: textfile;
   name: name_ptr; length: natural_number);
```

Print the entry in the name table which is pointed to by NAME_PTR to the given file (F). If the name is less than the specified length (LENGTH), print the name and pad with blanks. If the name is greater than the specified length, truncate the name.

PRINT_INTEGER_LEFT_JUST

```
procedure print_integer_left_just(var f: textfile;
   num: integer; length: natural_number);
```

Print the given integer (NUM) to the given file (F) left justified in a field of the given length (LENGTH). If the integer is less than the specified length, print the integer and pad with blanks. If the integer is greater than the specified length, truncate the integer.

Print Right Justified

Print out the given data structure right justified in a field of the given length.

PRINT_STRING_RIGHT_JUST

```
procedure print_string_right_just(var f: textfile;
   str: string; length: natural_number);
```

Print the given string (STR) to the given file (F) right justified in a field of the given length (LENGTH). If the string is less than the specified length, print the string and pad with blanks. If the string is greater than the specified length, truncate the string.

PRINT_ALPHA_RIGHT_JUST

```
procedure print_alpha_right_just(var f: textfile;
   name: alpha; length: natural_number);
```

Print the given alpha (NAME) to the given file (F) right justified in a field of the given length (LENGTH). If the alpha is less than the specified length, print the alpha and pad with blanks. If the alpha is greater than the specified length, truncate the alpha.

PRINT_NAME_RIGHT_JUST

```
procedure print_name_right_just(var f: textfile;
   name: name_ptr; length: natural_number);
```

Print the name which is pointed to by NAME_PTR to the given file (F). If the name is less than the specified length, print the name and pad with blanks. If the name is greater than the specified length, truncate the alpha.

PRINT_INTEGER_RIGHT_JUST

```
procedure print_integer_right_just(var f: textfile;
   num: integer; length: natural_number);
```

Print the given integer (NUM) to the given file (F) right justified in a field of the given length (LENGTH). If the integer is less than the specified length, print the integer and pad with blanks. If the alpha is greater than the specified length, truncate the integer.

PARSING

A lexical analyzer is provided that returns tokens from an input stream. This can be used to construct, along with other routines, a recursive descent parser. The routine is:

INSYMBOL

```
procedure insymbol;
```

Read a token from the input stream and return it as an enumerated type element. There are four global variables of interest:

- SY: current token returned from INSYMBOL.
- ID: value of the identifier if SY = IDENT.
- CONST_VAL: value of the constant is SY = CONSTANT.
- LEX_STRING: value of string if SY = STRINGS.

Recursive descent parsers are provided to read the Compiler expansion file, the primitive library files, and the expanded part and net lists. Another important parsing routine is:

PARSE_STRING

```
procedure parse_string(str: string; way_to_parse:
parse_type);
```

Use the specified string (STR) as the input to INSYMBOL. WAY_TO_PARSE will be used to decide how the string is to be parsed. The two ways to parse are:

1. parse_transparently: The string is parsed as if it is just a line being read from an input file.

2. parse_separately: Parse the string as a stand alone. Keywords cannot be parsed using this method.

POP_PARSED_STRING

```
procedure pop_parsed_string(string_to_parse: string);
```

Pop the top of the parse string stack until a string found on the stack matches the string passed to the routine.

GET_NUMBER

```
procedure get_number(var number: name_ptr);
```

Parse the input string for an alphanumeric pin number. The routine will check to see if the token is a identifier or a constant. If the token is an identifier, it will be entered into the name table and the pointer will be passed back as the parameter (NUMBER). If the token is a constant, it will be converted into an alpha and padded with nulls so it will sort correctly. The alpha will then be entered into the name table and the pointer will be passed back in the parameter (NUMBER). This routine should be called by a parser in place of INSYMBOL if the next token to be checked for is to be alpha-numeric. For example, this routine would be called in place of INSYMBOL if a pin number is expected in the input stream to the parser.

SKIP

```
procedure skip(syms: setofsymbols);
```

Read in symbols until SY is equal to an element in SYMS.

ERROR REPORTING

An error reporting package is supported that allows the user to define error messages and to report information about the error condition. The routines supported are:

ERROR

```
procedure error(error_num: error_range);
```

Print the error message corresponding to the given error number. If this error occurs during scanning an input line, print the input line along with a pointer to the current position in the line.

ERROR_DUMP_LOGICAL_PART

```
procedure error_dump_logical_part(part:
logical_part_ptr);
```

Output the name of the given logical part as part of the error message.

ERROR_DUMP_SIZE_EXPANDED_PART

```
procedure error_dump_size_expanded_part(expanded_part:
           SIZE_expanded_part_ptr);
```

Output the name of the given SIZE expanded part as part of the error message.

ERROR_DUMP_LOGICAL_NODE

```
procedure error_dump_logical_node(node: node_ptr);
```

Output the name of the given node as part of the error message.

ERROR_DUMP_LOGICAL_NET

```
procedure error_dump_logical_net(net: net_ptr);
```

Output the name of the given net as part of the error message.

ERROR_DUMP_PART_TYPE

```
procedure error_dump_part_type(part: part_type_ptr);
```

Output the name of the given part type as part of the error message.

ERROR_DUMP_PHYSICAL_PART

```
procedure error_dump_physical_part(part:
physical_part_ptr);
```

Output the name of the given physical part as part of the error message.

ERROR_DUMP_PROPERTY

```
procedure error_dump_property(name: name_ptr; text:
string);
```

Output the given property as part of the error message.

ERROR_DUMP_FILE_NAME

```
procedure error_dump_file_name(file_name: string);
```

Output the name of the current input file as part of the error message.

ERROR_DUMP_FILE_TYPE

```
procedure error_dump_file_type;
```

Output the current file type as an error message.

ERROR_DUMP_PIN_DEF_NAME

```
procedure error_dump_pin_def_name(pin_def: pin_def_ptr);
```

Output the pin name of the given pin def as part of the error message.

ERROR_DUMP_STRING

```
procedure error_dump_string(str: string; print_CRLF:
boolean);
```

Output the given string as part of the error message. Print an EOL (end-of-line) after the string if print_CRLF is TRUE.

ERROR_DUMP_ALPHA

```
procedure error_dump_alpha(data: alpha; print_CRLF:
boolean);
```

Output the given ALPHA as part of the error message. Print an EOL (end-of-line) after the alpha if print_CRLF is TRUE.

ERROR_DUMP_INDENT

```
procedure error_dump_indent(indentation: natural_number);
```

Output the specified number of spaces to the error file.

ERROR_DUMP_INTEGER

```
procedure error_dump_integer(int: integer;
                             print_CRLF: boolean);
```

Output the given integer (INT) as part of the error message. Print an EOL (end-of-line) after the integer if print_CRLF is TRUE.

REPORT_EXCEPTION_ERROR

```
procedure report_exception_error;
```

Report information about the exception error which was last encountered. An exception error may occur when some file function could not be done. For example, a file could not be opened.

DISPLAY_ERROR_SUMMARIES

```
procedure display_error_summaries;
```

Display the error information for the entire run of DIAL.

DEBUGGING

A very important part of interface development is the debugging of the program. DIAL includes a debug mechanism along with several routines that dump the data structures. Debug statements take the form of a test of a global debug flag (20 are supported) and calling a debug routine, or writing to the debug file. The debug flags are settable with directives processed by the interface program. Debug output is directed to a special file. the major debug routines are:

ASSERT

```
procedure assert(assertion_number: assert_range);
```

Generate an assertion failure message corresponding to the given number. Used to check assumptions about the relationships between data items. This is very useful in detecting bugs within the program.

DUMP_PROPERTIES

```
procedure dump_properties(var f: textfile;
                              prop_list: property_ptr;
                              indent: naturalnumber);
```

Dump the specified property list to the debug file. Indent the line by the given amount.

DUMP_NETS

```
procedure dump_nets(var f: textfile);
```

Dump all the nets to the debug file. Output both the logical net name and the physical net name. Output all the nodes connected to the net.

DUMP_LOGICAL_PARTS

```
procedure dump_logical_parts(var f: textfile);
```

Output all of the logical parts to the debug file. For each part, output is: part type, name, body properties, logical pins, and size expanded parts. Output the node list for each SIZE expanded part.

DUMP_PART_TYPES

```
procedure dump_part_types(var f: textfile);
```

Output all of the part types to the debug file. For each part output: name, type, properties, number of pins and what they are. For each pin output: its name, pin number(s), and properties.

DUMP_PHYSICAL_SECTIONS

```
procedure dump_physical_sections(var f: textfile;
                                 section_list:
                                 physical_section_ptr);
```

Output the names of the physical sections in the given list to the debug file. Bindings are also printed out.

DUMP_PHYSICAL_PARTS

```
procedure dump_physical_parts(var f: textfile);
```

Output all of the physical parts to the debug file. For each part output its name, physical sections, and part type.

DUMP_ALL_NODES

```
procedure dump_all_nodes(var f: textfile);
```

Output all of the nodes to the specified file (F). The routine will walk through all of the logical parts and output the nodes which are connected to each SIZE expanded part. Each SIZE expanded part will be output with the nodes which are connected to that part. The information about each node will be output. This information will include the node's physical and logical name, along with the pin the node is connected to.

DUMP_CONFIGURATION

```
procedure dump_configuration(var f: textfile);
```

The configuration in which a signal is represented will be dumped.

DUMP_INSTANCE_DRAWING_TABLE

```
procedure dump_instance_drawing_table(var f: textfile);
```

Dump all of the entries in the instance drawing table to the file (F).

DUMP_GENERIC_DRAWING_TABLE

```
procedure dump_generic_drawing_table(var f: textfile);
```

Dump all of the entries in the generic drawing table to the file (F).

GENERAL

WIDTH_OF_INTEGER

```
function width_of_integer(i: integer): natural_number;
```

Return the number of characters needed to print the given integer.

WIDTH_OF_ALPHA

```
function width_of_alpha(name: alpha): natural_number;
```

Calculates the number of characters in the given alpha (NAME) and returns the total.

WIDTH_OF_STRING_WITH_NULLS

```
function width_of_string_with_nulls(str: string)
                                   : string_range;
```

Calculates the length of the given string (STR) ignoring the null character. The total number of printable characters is returned.

ADD_NULLS_TO_ALPHA

```
procedure add_nulls_to_alpha(var name: alpha);
```

Add leading nulls to an alpha to make it sort correctly.

CONVERT_ALPHA_TO_NUMBER

```
function convert_alpha_to_number(name: alpha;
               var number: integer): boolean;
```

Convert the given alpha (NAME) to an integer number (NUMBER).

CONVERT_NUMBER_TO_ALPHA

```
function convert_number_to_alpha(number: integer;
               var name: alpha): boolean;
```

Convert the given number(NUMBER) into an alpha(NAME).

FIND_PIN_FROM_SECTION

```
function find_pin_from_section
              (section_number: section_number_range;
               part_type: part_type_ptr;
               var pin_number: name_ptr): boolean;
```

Find a pin number which is unique to the given section number (SECTION_NUMBER). The part type which the section is on is also passed as PART_TYPE.

FIND_SECTION_FROM_PIN

```
function find_section_from_pin
             (pin_number: pin_number_range;
              part_type: part_type_ptr;
              var section: section_pin_ptr);
```

Find a section pin which corresponds to the given pin number (PIN_NUMBER). If the pin number corresponds to a common pin SECTION is passed back with a NIL value.

IS_BOGUS_PART

```
function is_bogus_part(part: part_type_ptr): boolean;
```

Check to see if the part (PART) is a real part. If it is, then TRUE is returned. Parts which are not considered TRUE are the root type part and flag bodies. Anytime the part_type_table is traversed, this routine should be called for each entry in the table.

FIND_ALL_FLAG_NODES

```
function find_all_flags_nodes(root_drawing: string):

node_and_pin_list_ptr;
```

A flag body is a body which is put on an interface signal to help determine whether the interface pin is an input, output, or bidirectional pin. If this body is not on an interface pin, a DIAL program will not be able to determine where the interface pins are. This routine will find all of the flag bodies in the logical parts table and return a sorted list of the nodes connected to each. For logical DIAL, if there is not a root part in the chips file, then the pin number for the interface pins will be automatically generated. For physical DIAL, the pin numbers will always be automatically generated.

OUTPUT_GENERIC_INTERFACE_PIN_LIST

```
procedure output_generic_interface_pin_list(var f:
textfile):
```

Output the list of interface pins for the current module to the given file. It is assumed that the file has been opened. If a header is to be output to the file, it should already have been written to the file.

NOT_SINGLE_NODE_NET

```
function not_single_node_net(net: net_ptr): boolean;
```

TRUE is returned if the given net has more than one node on it or if the global flag allowing single node nets is on. The global flag can be set by a standard directive (See below).

INIT_PARSE_ENVIRONMENT

```
procedure init_parse_environment;
```

Initialize the parse environment. This routine should be called any time a new file is to be parsed.

ADD_NULLS_TO_DESIGNATOR

```
procedure add_nulls_to_designator(source, dest: string);
```

Pad the physical designator with nulls for sorting purposes. The routine will search a physical name from the beginning until it finds a numeric character. It will then pad the leading characters with nulls until the name takes up the specified number of bytes. This will make it possible to correctly sort the physical names.

UPPER_CASE_CHAR

```
function upper_case_char(ch: char): char;
```

Take the given character (CH) and convert it into an upper case character. Return the result to the calling routine.

FIND_DIRECTIVE

```
function find_directive(name: name_ptr;
              var directive: directive_type): boolean;
```

See if the given name corresponds to a standard DIAL directive. If the name is a directive, then return the enumerated type in a parameter (DIRECTIVE) and return TRUE.

NEW_PHYS_DES_PREFIX

```
procedure new_phys_des_prefix(var list:
phys_des_prefix_ptr);
```

Create a new physical designator prefix and add it to the head of the given list (LIST).

IS_CHAR_IN_STRING

```
function is_char_in_string(str: string; ch: char):
boolean;
```

Check to see if the given character (CH) is in the given string (STR). If the character is in the string, then the procedure returns TRUE. Otherwise FALSE is returned.

PRINT_BEFORE_CHAR

```
procedure print_before_char(var f: textfile; str: string;
             delimiter: char);
```

Print the contents of the given string (STR) until the given delimiter is found. If the delimiter is not in the string, the entire string will be printed.

PRINT_AFTER_CHAR

```
procedure print_after_char(var f: textfile; str: string;
             delimiter: char);
```

Print the contents of the string which occurs after the delimiter is found. If the delimiter is not found, none of the string will be printed.

GENERATE_CROSS_REFERENCES

```
procedure generate_cross_references;
```

Generates the various cross references for the design. The structures which are to be output are: the local parts for each drawing, the global signals and the global parts.

HEAP MANAGEMENT

Management Routines

Two routines are needed to control the heap structures. One is called by programs which must use some of the heap, while the other routine reports how much storage the heap structures use.

INCREMENT_HEAP_COUNT

```
procedure increment_heap_count(structure:
             heap_structures; numbytes: integer);
```

Increment the space used for the specified structure by one instance.

REPORT_HEAP_USAGE

```
procedure report_heap_usage(var f: textfile);
```

Report the heap space usage by structure. The following structures have heaps associated with them: strings, name table, logical pins, logical parts, pin defs, nodes, nets, SIZE expanded parts, properties, part types, section pins, physical parts, physical sections, net list, section def, part type list, physical part list, pin list, power pin list, power pin name, drawing names and physical designator prefixes.

When this routine is called, the name of the structure whose emunerated type is passed will be printed out with the number of bytes which are used by that data structure.

Allocation Routines

There is a group of routines which can allocate or release storage for a specified data structure in a linked list. These lists are used by the merge sort which is described below. The following routines manipulate the structures:

NEW_NET_LIST

```
procedure new_net_list(var list: net_list_ptr);
```

Create a new net list element and assign it to the specified pointer (LIST).

RELEASE_NET_LIST

```
procedure release_net_list(var NLP: net_list_ptr);
```

Release the specified net list element (NLP) to the free list.

NEW_NODE_LIST

```
procedure new_node_list(var list: node_list_ptr);
```

Create a new node list element and assign it to the specified pointer (LIST).

RELEASE_NODE_LIST

```
procedure release_node_list(var NLP: node_list_ptr);
```

Release the specified node list element (NLP) to the free list.

NEW_PART_TYPE_LIST

```
procedure new_part_type_list(var list:
part_type_list_ptr);
```

Create a new part type list element and assign it to the specified pointer (LIST).

RELEASE_PART_TYPE_LIST

```
procedure release_part_type_list(var PTLP:
part_type_list_ptr);
```

Release the specified part type list element (PTLP) to the free list.

NEW_PHYSICAL_PART_LIST

```
procedure new_physical_part_list
          (var list: physical_part_list_ptr);
```

Create a new physical part list element and assign it to the specified pointer (LIST).

RELEASE_PHYSICAL_PART_LIST

```
procedure release_physical_part_list
             (var PPLP: physical_part_list_ptr);
```

Release the specified physical part list element (PPLP) to the free list.

NEW_LOGICAL_PART_LIST

```
procedure new_logical_part_list(var list:
logical_part_list_ptr);
```

Create a new logical part list element and assign it to the specified pointer (LIST).

RELEASE_LOGICAL_PART_LIST

```
procedure release_logical_part_list
              (var LPLP: logical_part_list_ptr);
```

Release the specified logical part list element (LPLP) to the free list.

NEW_DRAWING_NAME_LIST

```
procedure new_drawing_name_list(var list:
drawing_name_list_ptr);
```

Create a new drawing name list element and assign it to the specified pointer (LIST).

RELEASE_DRAWING_NAME_LIST

```
procedure release_drawing_name_list
              (var DNLP: drawing_name_list_ptr);
```

Release the specified drawing name list element (DNLP) to the free list.

NEW_FILE_NAME_LIST

```
procedure new_file_name_list(var list:
file_name_list_ptr);
```

Create a new file name list element and assign it to the specified pointer (LIST).

SORTING

A number of routines are provided that search for items in the data structures and sort elements of the data structures. There are two routines which are supplied to accomplish the sorting. The first routine is needed to make up the list of elements to be sorted. The second routine will get the next element which comes in the list. The routines which are supplied are:

INIT_PHYSICAL_NET_SORT

```
function init_physical_net_sort: net_list_ptr;
```

Initializes the physical net sorting algorithm.

GET_PHYSICAL_NET

```
function get_physical_net(firstpart: net_list_ptr):
net_ptr;
```

Return the next physical net in the given list of physical nets. The next net will be the next net in alphabetical order. When a specific entry in the list of nets is empty, the entry is released to the heap as a free element. When all of the nodes are released, the routine will return NIL.

INIT_LOGICAL_NET_SORT

```
function init_logical_net_sort: net_list_ptr;
```

Initializes the logical net sorting algorithm.

GET_LOGICAL_NET

```
function get_logical_net(firstpart: net_list_ptr):
net_ptr;
```

Get the next sorted logical net from the net list. The entries in the table will be sorted by name. When the node in the net is pointing to NIL the node is released. When all of the nodes are released the routine will return NIL.

INIT_PHYSICAL_PART_SORT

```
function init_physical_part_sort: physical_part_list_ptr;
```

Set up a list of the physical parts for sorting. The list will contain all of the entries in the physical part table.

GET_PHYSICAL_PART

```
function get_physical_part(firstpart:
        physical_part_list_ptr): physical_part_ptr;
```

Get the next physical part from the specified list of physical parts. The physical parts will be sorted by name. When the last physical part is released from the list, the routine will return NIL.

INIT_LOGICAL_PART_SORT

```
function init_logical_part_sort: logical_part_list_ptr;
```

Set up a list of the logical parts for sorting. The list will contain all of the entries in the logical part table.

GET_LOGICAL_PART

```
function get_logical_part(firstpart:
logical_part_list_ptr):
                     logical_part_ptr;
```

Get the next logical part from the specified list of logical parts. The list will be sorted by name, then size number, then version number. When the last logical part is released from the list, the routine will return NIL.

INIT_PART_TYPE_SORT

```
function init_part_type_sort: part_type_list_ptr;
```

Set up the part types in a list for merge sorting. The list will be sorted by name. Each element in the list will be a copy of an entry in the part type table array. Therefore, each element in the list points to more than one entry in the part type table.

GET_PART_TYPE

```
function get_part_type(var firstpart: part_type_list_ptr):
                                      part_type_ptr;
```

Get the next sorted part type from the specified list of part types. The entries in the list will be returned in a order sorted by name. After all of the entries which are pointed to by an element in the list are used, that element is released.

INIT_GENERIC_DRAWING_SORT

```
function init_generic_drawing_sort: drawing_name_list_ptr;
```

Sort the table of generic drawings. The drawings are sorted by name.

GET_GENERIC_DRAWING_NAME

```
function get_generic_drawing_name
```

Get the next entry from the specified list of sorted generic drawing names. When all of the names have been received from the list, the routine will return NIL.

INIT_INSTANCE_DRAWING_SORT

```
function init_instance_drawing_sort:
drawing_name_list_ptr;
```

Sort the table of instance drawing names. The list is sorted by name.

GET_INSTANCE_DRAWING_NAME

```
function get_instance_drawing_name
    (var first_part: drawing_name_list_ptr):
     drawing_name_ptr;
```

Get the next entry from the specified list of sorted instance drawing names. When all of the entries have been received from the list, the routine will return NIL.

INIT_SIZE_PART_NODE_SORT

```
function init_SIZE_part_node_sort
    (SIZE: SIZE_expanded_part):
     node_list_ptr;
```

Make a list of the nodes attached to the specified SIZE expanded part. The nodes on the part will be sorted by name.

INIT_PHYSICAL_SECTION_NODE_SORT

```
function init_physical_section_node_sort
    (section: physical_section_ptr):
     node_list_ptr;
```

Make a sorted list of the nodes attached to the pins of a physical section of a part. The list will be sorted by alpha-numeric pin name.

INIT_PHYSICAL_PART_NODE_SORT

```
function init_physical_part_node_sort
     (part: physical_part_ptr):
      node_list_ptr;
```

Make a sorted list of the nodes associated with the pins which are attached to the specified physical part. The nodes will be sorted by the pin's alpha-numeric pin name.

INIT_NODE_ON_NET_SORT

```
procedure init_node_on_net_sort(net: net_ptr);
     node_list_ptr);
```

Make a sorted list of the nodes attached to the specified net. The nodes will be sorted by physical part name, then alphanumeric pin name.

GET_NODE

```
function get_node(var first_node: node_list_ptr):node_ptr;
```

Get the next node from the specified list of sorted nodes. As each node is picked off of the list, the list element will be released. When the list element is picked off of the list, NIL will be returned.

The init routines must be called before the get routines or else there will not be anything to sort. The init routines return a pointer to linked list of elements. The get routine gets the next element and returns a pointer to it. NIL is returned when there aren't any elements left in the list.

NODE LIST ROUTINES

There are a set of routines which are supplied to make a list of nodes contain only a certain type of node. There are three types of nodes: input, output and bidirectional. These routines will make it possible to make a list contain only the nodes which are desired. All of the routines are passed a list of nodes which is then changed to only contain the nodes which are desired. These routines are destructive routines in that they do not keep the node list as it originally is when it is passed to the routine. One of the node sort initialization routines must be called to get a list of nodes before any of these routines can be used (See above).

GET_INPUT_ONLY_PINS

```
procedure get_input_only_pins(var node_list:
node_list_ptr);
```

This routine will change the list of nodes so that only the nodes which describe input pins will be included in the new node list. These pins will be strictly input pins. Bidirectional pins will not be included in this new list.

GET_OUTPUT_ONLY_PINS

```
procedure get_output_only_pins(var node_list:
node_list_ptr);
```

This routine will change the list of nodes so that only the nodes which describe output pins will be included in the new node list. These pins will be strictly output pins. Bidirectional pins will not be included in this new list.

GET_BIDIRECTIONAL_PINS

```
procedure get_bidirectional_pins(var node_list:
node_list_ptr);
```

This routine will change the list of nodes so that only the nodes which describe bidirectional pins will be included in the new node list. These pins will be strictly birectional pins. Input and output nodes will not be included in this list.

GET_INPUT_PINS

```
procedure get_input_pins(var node_list: node_list_ptr);
```

This routine will change the list of nodes so that every node which describes a pin which has an input load will be included in the new list of nodes. This means that all the nodes which describe an input or bidirectional pin will be included in the new list.

GET_OUTPUT_PINS

```
procedure get_output_pins(var node_list: node_list_ptr);
```

This routine will change the list of nodes so that every node which describes a pin which has an output load will be included in the new list of nodes. This means that all the nodes which describe an output or bidirectional pin will be included in the new list.

TIME KEEPING ROUTINES

There are two routines supplied which make it possible to find out how much time is spent in DIAL. The routines are:

EXEC_TIME

```
procedure exec_time(var last_elapsed_time: integer;
                    var last_CPU_time: integer;
                    just_delta: boolean);
```

Display the execution time, both in CPU time and elapsed time. If JUST_DELTA, then only the delta time from last_CPU_time to the current CPU time is displayed. The last_CPU_time and last_elapsed_time are reset.

PRINT_TIME

```
procedure print_time(var f: textfile; current_time:
integer);
```

Print the time to the given file (F). Zeroes will be output for all of the time allocation spaces which are not filled. For example, if the time to be represented only has seconds in it, the minute spaces will be filled with zeroes.

9.9 MODIFICATION ROUTINES

DIAL is designed for the user to be able to customize procedures, so the job of modifying the output of DIAL can be easily accomplished. Templates for several routines have been supplied to meet this goal. The templates supplied make it possible for the physical net names, physical part names and a directives processor to be easily changed to fit the user's needs.

PHYSICAL NET ROUTINES

Two routines are supplied to name the physical nets. One of the routines produces a net name, while the other routine makes sure the physical net name is unique. Both of these routines can be modified to produce any type of net name which the user might need. The routines are:

CREATE_NET_ABBREVIATION

```
   function create_net_abbreviation(net: net_ptr): string;
```

Create an abbreviation for the logical net name to be used as the physical net name. The template supplied will produce an alpha-numeric name. The name created will be the returned value.

FIX_NAME

```
   procedure fix_name(var name: string);
```

Fix the name given. This is done if the net name to be entered is found to already be entered in the physical net list. The template supplied will increment the last alphabetic character in the name. This is so the bit offset (appended to the end of the abbreviation) is preserved.

A routine is also provided which makes it possible to change the physical net names from their existing assignment within the data base. This routine is:

REASSIGN_PHYSICAL_NET_NAMES

```
   procedure reassign_physical_net_names;
```

This routine will delete all of the current physical net names. It will then call the user routine create_net_abbreviation (see above) to rename the nets in a way which is specified by the user.

PHYSICAL PART ROUTINES

Templates for two routines are supplied which make it possible to create a physical part name. One of the routines is designed to create the name, while the other routine will make the name unique.

FIND_PREFIX_AND_UNIQUE_NUMBER

```
   procedure find_prefix_and_unique_number(part:
      logical_part_ptr;
         var prefix:
            phys_des_prefix_ptr;
         var prefix_name: string;
         var unique_number:
            natural_number);
```

A physical part name is created. The template routine will check to see if a physical designator prefix property is associated with the part. If there is, then that prefix is used and a unique number is attached to it to make a physical part name. If the property is not found the letter "U" is used and a unique number is attached to it.

INCREMENT_UNIQUE_NUMBER

```
  function increment_unique_number(prefix:
      phys_des_prefix_ptr): natural_number;
```

Increment the unique number and attach it to the prefix. Return the new unique number, so it can be saved for the next call to this routine.

DIRECTIVE ROUTINE

A template for a routine is supplied which makes it possible to add new directives to a particular interface. This routine is called from READ_DIRECTIVES_FILE which is described below.

PROCESS_DIRECTIVES

```
 procedure process_directives;
```

Process_directives gives the user the ability to create directives to make it possible to create a variable environment for an interface. Process_directives is called by READ_DIRECTIVES_FILE if there is a directive in the directives file which is not a standard directive. In order to create a new directive, the following must be done:

1. The directive which is to be used must be initialized before the procedure READ_DIRECTIVES_FILE is called. This is done by making a variable of name_ptr type and initialize it to point to an entry in the name table which has an alpha of the directive name. An example is:

   ```
   loc_directive_name := enter_name('LOC');
   ```

   This statement creates an entry in the name table which has the alpha value of LOC. The entry is pointed to by loc_directive_name.

2. The LOC directive can now be used by the interface, but process directives must be made to recognize the directive. If the LOC directive is a directive which sets a string variable, so process directives would be changed to appear as:

```
procedure process_directives;

begin
  if id^.name = LOC_directive_name then
    begin
      insymbol;
      if sy = strings then
        copy_string(lex_string, LOC_string);
      else error(33 { string expected });
      insymbol;
    end
  else
    error(51 { unknown directive });
  skip([semi,endofdatasy]);
end;
```

The READ_DIRECTIVES_FILE routine will pass an identifier to PROCESS_directives, so the example checks to see if the name pointer for the identifier is the same value as the pointer to the LOC directive. If the values are not the same an error message is written out which states that the current identifier is not a directive. This is done since all of the standard directives have been checked and all of the user's directives have been checked.

If the pointers are found to be the same, INSYMBOL is called. The next symbol should be a string. If it is, then the value of LEX_STRING is copied into LOC_STRING. The copy must be done because the value of LEX_STRING will be constantly changing and LOC_STRING should not change. After the string is copied, INSYMBOL must be called again. This will make the parser pass up the string which is the value of the directive. (SKIP is then called to skip all of the symbols until a semicolon or the end of the file is found.) This is done so the parser will not be messed up. The process will then return to READ_DIRECTIVES_FILE and continue parsing the directives file.

9.10 DIAL INITIALIZATION ROUTINES

There are a few routines which are included in DIAL which make it easy to set up the data base for a specific interface. These routines will set initialize DIAL and read in the correct files. A routine is also available to make it possible to read in directives to make the running of DIAL easier.

INIT_DIAL

procedure init_DIAL;

Initialize DIAL to set up the data base and run the interface. Init_dial will initialize all of the global variables and tables which are used in the running of DIAL. This routine must be the first DIAL routine called by an interface. Any other DIAL routine will not run until this routine has been called.

READ_DIRECTIVES_FILE

procedure read_directives_file;

This routine reads in a file of directives to make it possible to make an interface be more flexible. For example, an interface can be made to read different libraries for different data bases by using the library file directive. There are several directives which are built into the read directives routine and can be used by any interface. These directives are:

1. LIBRARY_FILE: This directive makes it possible to specify a specific library for a design. An example of the library directive is:

   ```
   LIBRARY_FILE 'lsttl.prt';
   ```

   This example tells DIAL that the LSTTL library is to be used with the design run at this time.

2. DEBUG: This directive turns on a debug flag in DIAL. An example of this directive is:

   ```
   DEBUG 1;
   ```

   This example causes the flag debug_1 to be turned on.

3. INTERFACE_TYPE: This directive tells DIAL whether the data base is to be set up as a logical interface or a physical interface. This directive must be used if the routine READ_DATA_BASE (See below) is to be used. This directive causes the global variable LOGICAL_INTERFACE_FLAG to be set to TRUE if the interface is to be logical, or FALSE if the interface is physical. For example,

    ```
    INTERFACE_TYPE LOGICAL;
    ```

    This tells DIAL that the interface is a logical interface.

4. HEADER_FILE: This directive makes it possible to read in a file which is used as a header in the interface's output file.The HEADER_FILE directive puts the string in the directive into the header_file variable. This file will be read and output when the routine OUTPUT_HEADER_FILE is called. An example of this directive is:

    ```
    HEADER_FILE 'header.dat';
    ```

    This example cause will the file header.dat to be read and output when the routine OUTPUT_HEADER_FILE is called.

5. INCLUDE_IO_LIST: This directive tells DIAL to set the global variable INCLUDE_IO_LIST to TRUE or FALSE. This variable tells the interface whether global IO is to be output and whether the interface pins of the design should be found. For example:

    ```
    INCLUDE_IO_LIST ON;
    ```

    This example causes the variable to be set to TRUE. The routine FIND_ALL_FLAG_NODES must be called, since this is the routine which sets up the interface pins.

6. SINGLE_NODE_NETS: This directive tells DIAL to set up the global variable OUTPUT_SINGLE_NODE_NETS to TRUE or FALSE. This variable determines if the interface outputs single node nets. For example:

    ```
    SINGLE_NODE_NET ON;
    ```

    This example causes the variable to be set to TRUE, and the interface can output single node nets.

7. MAX_ERRORS: Used to specify the maximum number of errors allowed before DIAL terminates. When the condition occurs, an error message is printed. For example:

   ```
   MAX_ERRORS 500;
   ```

   This sets the maximum number of errors to 500. If not specified, the run is terminated after 1000 errors.

8. SUPPRESS: Used to suppress specific warning and oversight messages. Warnings and oversights are used to grade the severity of error conditions. Warnings are considered to be the least severe followed by oversights, and then errors. Since neither warnings nor oversights are as severe as an error, and since there may be many of these messages in a good design, this directive is supplied to suppress the message that would be produced. A list of warning messages may be specified. For example:

   ```
   SUPPRESS 132,133;
   ```

   This supresses messages 132 and 133. All warning messages can be suppressed with the WARNING directive (see below). Error messages cannot be suppressed. If unspecified, no warnings or oversights are suppressed.

9. WARNINGS: Used to control whether warning messages should be printed. This directive can be used to suppress all warning messages (although it would be better to make the changes in the design). The total number of warning conditions encountered is reported at the end of the program regardless of whether warnings are displayed or not. For example:

   ```
   WARNINGS OFF;
   ```

   This causes the interface to suppress all warning messages. The default is that warning messages will be output.

10. OVERSIGHTS: Used to control whether oversight messages are to be displayed. An oversight should be corrected but the design will probably run without fixing it. The total number of oversights detected is always reported at the end of the program regardless of whether they were printed or not. This directive is used to turn off all oversight messages. For example:

```
OVERSIGHTS OFF;
```

This causes the interface to suppress all oversight messages. The default is that oversight messages will be output.

11. PART_NAME_LENGTH: This directive can be used in logical DIAL to set up the maximum length of physical part names. For example:

    ```
    PART_NAME_LENGTH 6;
    ```

    This causes DIAL to create physical part names of no more than six characters.

12. NET_NAME_LENGTH: This directive makes it possible to limit the length of a name assigned to a physical net. This directive can always be used in logical DIAL and it can be used in physical DIAL if the net names are to be reassigned. For example:

    ```
    NET_NAME_LENGTH 6;
    ```

    This causes The DIAL physical net naming routine to create names of no more than six characters.

If a directive is found which is not one of these standard directives, PROCESS_DIRECTIVES will be called to see if the directive is a user defined directive.

READ_DATA_BASE

```
procedure read_data_base;
```

Check the global variable LOGICAL_INTERFACE_FLAG to see if the data base is to set up for a logical or physical interface. If the variable is set to TRUE then the data base is to be set up for a logical interface. The directive INTERFACE_TYPE can be used to set up the variable. This routine makes it possible to make an interface able to run both as a logical and physical interface, just by using a standard directive. If this routine is used, the routines read_Logical_data_base and read_Physical_data_base must not be used.

READ_LOGICAL_DATA_BASE

```
procedure read_logical_data_base;
```

The data base is set up for a logical interface. This causes the interface to always be logical. If this routine is used the routines READ_PHYSICAL_DATA_BASE and READ_DATA_BASE must not be used.

READ_PHYSICAL_DATA_BASE

```
procedure read_physical_data_base;
```

The data base is set up for a physical interface. If this routine is used, the routines READ_LOGICAL_DATA_BASE and READ_DATA_BASE must not be used.

9.11 DIAL HINTS

These sections will describe some hints which will make it easier to develop a DIAL program. They are designed to make the development of a DIAL program simpler and quicker.

ADDING ERROR MESSAGES TO A PROGRAM

A designer of a program may decide that there are certain problems that a user of his program should be aware of, and an error message may be used to do this. DIAL makes it easy for a program designer to add error messages to a program. The error numbers 200-250 are available to the user for errors.

The error message must be added to the error table. This is done by deciding what the error number is to be, and initializing the entry in the error table for that error number. This is done by having a line like:

```
error_strings[200]    := 'Not legal type                 ';
```

in the program. Now when an error is called with the number 200, this error message will be printed out.

FILES IN DIAL

There are many files which are supplied with DIAL. These file variables are:

1. INFILE: This variable is initially used to read in the directives file. This file can also be used to read any other file. This can be done by passing this file variable and an alpha file name to the open_parse_file routine.

2. OUTFILE: If the debug flag is turned on, all of the debug information will be written out to the file which this variable is assigned to.

3. CmpExp: This variable is set to read in the compiler expansion file. By default, the file CMPEXP.DAT is read in when this variable is used.

4. Chips: This variable is used to read in the chips files which are specified in the directives file. When this variable is used, an alpha name is passed with it to open a file.

5. PstXNet: This file is set to read in the expanded net list. By default, the file PSTXNET.DAT will be read in when this variable is used.

6. PstXPrt: This variable is set to read in the expanded part list. By default, the file PSTXPRT.DAT will be read in when this variable is used.

7. monitor: This variable is used to write information out to the console.

8. DIALLst: The information which summarizes the run of the DIAL program will be written out to this variable.

9. DIALSPEC: This variable can be used to output the information which the user's program is designed to output.

9.12 DIAL ON THE VAX

The files which are to be used to create a DIAL interface on the VAX are stored in a logical directory which is named SYS$DIAL. The files which the directory contains are:

1. consts.pas - This file contains all of the global constant declarations for DIAL. This file is to be included in the design program and should be used as reference in writing a DIAL program.

2. types.pas - This file contains all of the global types declarations for DIAL. This file is to be included in the design program and should be used as reference in writing a DIAL program.

3. vars.pas - This file contains all of the global variable declarations for DIAL. This file is to be included in the design program and should be used as reference in

writing a DIAL program.

4. vaxuser.pas - This file contains all of the global procedure declarations for DIAL. This file is to be included in the design program and should be used as reference in writing a DIAL program.

5. dial.obj - This is the pre-compiled DIAL library. It is linked with the user's DIAL program to create an executable program. It contains all of the DIAL procedures and functions.

6. dialassgn.com - This file contains assignments of DIAL logical environment variables to standard DIAL VAX file names. It can be copied to the user's local directory if the run time bindings of logical file to VAX file name are to be changed.

7. template.pas - A Pascal source file template for a DIAL program is provided. This program is provided to be used as the starting point for a user's DIAL program. It should be copied to a local directory and edited. Comments in the file describe how to make a logical or a physical DIAL program.

8. utilities.pas - This is the source of the user modifiable routines. It contains definitions for several routines used by DIAL which the user may wish to modify. If it is to be modified, it should be copied to a local directory and edited.

9. dial.cmd - This is an example of a directives file for DIAL. This example will create a logical data base (if the routine READ_DATA_BASE is used to set up the data base). The command file also says the LSTTL library is to be used as a CHIPS file.

CREATING A DIAL PROGRAM

Before starting to write a DIAL program, the user should create a local directory in which to do the development. The skeleton should be copied to the local directory with the following command:

```
            copy sys_$dial:template.pas <user's program name>
```

The template program should look like the following:

```
(*$S-*) (* allow non-standard Pascal features  *)
(*$C+*) (* turn on range checking              *)
```

```
(*$X-*) (* save a few trees                         *)
(*$W-*) (* don't display warnings                   *)


program userprog(CmpExp, Chips, PstXNet, PstXPrt, PstXRef, monitor,
                 DIALLst, DIALSPEC, inprog, infile, outfile, DIALback,
                 DIALStat, DIALSigB, DIALPrtB);

  { this is a template source file for a DIAL program.  The user
    should copy this file to a local directory and edit to create
    a DIAL program. }


const

%INCLUDE 'sys$dial:CONSTS.PAS'

type

%INCLUDE 'sys$dial:TYPES.PAS'

var

  {  input files  }

  infile,                   { input file }
  inprog,                   { user input file }
  CmpExp,                   { compiler expansion file }
  Chips,                    { library chips file }
  PstXNet,                  { expanded net list }
  PstXPrt,                  { expanded part list }

  { output files }


  PstXRef,                  { cross reference file }
  monitor,                  { output execution running summary }
  DIALLst,                  { main list file - errors + summary }
  DIALSPEC,                 { special output format }
  DIALback,                 { back annotation file }
  DIALStat,                 { State file }
  DIALSigB,                 { Signal State file }
  DIALPrtB: textfile;       { Part State file }

%INCLUDE 'sys$dial:VARS.PAS'


%INCLUDE 'sys$dial:VAXUSER.PAS'


%INCLUDE 'sys$dial:UTILITIES.PAS'
```

```
 (*****************************************)
 (*                                       *)
 (*       USER'S PROCEDURES GO HERE       *)
 (*                                       *)
 (*****************************************)


 procedure close_all_files;
   { close all of the files which are still open }
 begin
   close_output_file(DIALLst, list_file);
 end; { close_all_files }


begin { main program body }

 init_DIAL;                     { initialize DIAL }

 read_directives_file;           { read the DIAL directives }


 { Set up the correct data base: LOGICAL or PHYSICAL.
   Only one of the three routines can be used in a program. }

 read_data_base;              { The directive INTERFACE_TYPE can be
                                used to define whether the design is
                                to be logical or physical. }

 read_logical_data_base;      { read the logical design and CHIP
                                files to set up data base structures

 read_physical_data_base;     { read the physical design and CHIP
                                files to set up data base structures

 { call to find_all_flag_nodes is not needed for physical DIAL or
   the I/O signals for the design are not available or needed.
   The directive INCLUDE_IO_LIST can be used to set the finding of
   flag nodes on. }

 if (errors_encountered * fatal_errors = []) and
    include_IO_list THEN
   flag_nodes := find_all_flag_nodes(root_drawing_name);
```

```
  (*****************************************)
  (*                                       *)
  (*  CALLS TO USER'S PROCEDURES GO HERE   *)
  (*                                       *)
  (*****************************************)


  if errors_encountered * fatal_errors <> [] then
    error(112 { run stopped });

  display_error_summaries;        { report errors encountered }

  exec_time(start_elapsed_time, start_CPU_time, FALSE);

  close_all_files;

end.
```

The following changes should be made and the following precautions should be taken in changing the template program:

1. Change the program name from userprog to a name which is more meaningful for the program being created.

2. Any user defined global variables should be entered after the VARS.PAS include file. If there are any variable definitions before the INCLUDE file, the program will not work.

3. If any of the user modifiable procedures which are supplied in the file UTILITIES.PAS are to be changed, then the line

   ```
   %INCLUDE 'sys$dial:UTILITIES.PAS'
   ```

   must be changed to:

   ```
   %INCLUDE 'UTILITIES.PAS'
   ```

   This will cause the utilities file which is in the local directory to be used during compilation rather than the utilities file which is in SYS$DIAL.

4. Create the user's portion of the program. Put the procedures and the calls to the procedures in the correct places in the main program.

5. The main program must be changed to set up the correct data base. There are three routines to set up the data base and only one can be used by the program. The correct one should be kept and the calls to the other two should be deleted.

MAKING AN EXECUTABLE DIAL PROGRAM

Now that a DIAL program has been designed and written, it must be compiled and linked. The process to do this procedure is very simple. To compile the program the line

```
pascal <user program name>
```

must be entered. Any of the standard VAX PASCAL compiler options can be added to this line as needed. If the program does not compile without errors, the problems must be fixed and the program must be recompiled.

When the program compiles correctly without any errors, it must be linked. The line

```
link <user program name>,SYS$DIAL:dial.obj
```

must be entered to do the linking. It is possible to use any of the standard VAX linker options while doing the linking. There should not be any problems with the linking as long as the two include files UTILITIES.PAS and VAXUSER.PAS are included in the source of the program. If there are any problems check to see if these two files are included. If the program links correctly, there is now an executable file of the program.

RUNNING A DIAL PROGRAM

Once the DIAL program has been compiled and linked, it is possible to run the program. The following must be done before the program can be run.

1. A DIAL directives file must be created. The easiest way to do this is to copy the file DIAL.CMD from SYS$DIAL and make the necessary changes for the particular design. The libraries which are needed to run the design must be specified in this file. If the routine READ_DATA_BASE is used to set up the data base, a INTERFACE_TYPE directive should be in the file. It may be desired to put other standard directives or user supplied directives into this file.

2. A test case must be created. If the program is a logical DIAL program a compiler expansion file must be present. If the program is a physical DIAL program an expanded net list and an expanded part list must be present.

3. The user may wish to change the logical file assignments. If this is to be done, the assignment file can be copied by entering:

   ```
   copy SYS$DIAL:dialassgn.com <user's name>
   ```

   The logical file assignment names can now be changed. It is suggested that the assignment names of the expansion files are kept as they are, since these are the names they are output as.

Now that all of these changes have been made, the program can be run. First the logical file assignments must be done. This is done by entering one of the following, depending on whether the file has been customized.

```
@SYS$DIAL:dialassgn         { file hasn't been changed }
@<user's name>              { file has been changed }
```

The interface program can now be run by entering:

```
run <user's program name>
```

If everything has been done correctly the program should terminate properly and a list file and output file should be written. If the program does not run correctly, check for logic errors in the program or check to see that all of the procedures to run the program have been followed correctly.

9.13 DIAL on UNIX

The files which are to be used to create a DIAL interface in UNIX are stored in /u0/scald/dial. The files which the directory contains are:

1. consts.pas - This file contains all of the global constant declarations for DIAL. This file should be used as reference in writing a DIAL program.

2. types.pas - This file contains all of the global types declarations for DIAL. This file should be used as reference in writing a DIAL program.

3. vars.pas - This file contains all of the global variable declarations for DIAL. This file should be used as reference in writing a DIAL program.

4. procs.pas - This file contains all of the global procedure declarations for DIAL. This file should be used as reference in writing a DIAL program.

5. dialint.obj - This is the object module which contains the constant, type, variable and procedure definitions. This is a module which is used by the user's program.

6. dial.obj - This is the pre-compiled DIAL library. It is used by the user's DIAL program to create an executable program. It contains all of the DIAL procedures and functions.

7. dialassign.com - This file contains assignments of DIAL logical environment variables to standard DIAL UNIX file names. It can be copied to the user's local directory if the run time bindings of logical file to UNIX file name are to be changed.

8. template.pas - A Pascal source file template for a DIAL program is provided. This program is provided to be used as the starting point for a user's DIAL program. It should be copied to a local directory and edited. Comments in the file describes how to make a logical or a physical DIAL program.

9. userunit.pas - This file is the source for the DIAL routines which can be modified by the user.

10. userunit.obj - This object module is the compiled version of the userunit.pas source program. This module should be used by the user if he has not made any changes to userunit.pas.

11. dial.crf - This is a cross reference file which is used when the user's program is shortened so the program can be properly compiled and linked.

CREATING A DIAL PROGRAM ON UNIX

Before starting to develop a DIAL program, the user should first create a local directory in which to work. The file template.pas should then be copied from the directory /u0/scald/dial into the local directory. This can be done with the following command:

```
cp /u0/scald/dial/template.pas <user program name>
```

The template program should look like the following:

```
program user(infile, outfile);


{ this is a template source file for a DIAL program.  The user should
  copy this file to a local directory and edit it to create a DIAL
  program.

  The USERUNIT described below is provided in source form so that the
  routines contained therein may be modified.  }


uses
  (*$U /u0/scald/dial/dialint.obj*) dialint,
  (*$U /u0/scald/dial/userunit.obj*) userunit;



  (****************************************)
  (*                                      *)
  (*      USER'S PROCEDURES GO HERE       *)
  (*                                      *)
  (****************************************)



  procedure close_all_files;
    { close all of the files which are still open }
  begin
    close_output_file(DIALLst, list_file);
  end; { close_all_files }


begin  { main program }

  init_DIAL;                   { initialize DIAL }
  read_directives_file;        { read the DIAL directives }


  { The correct data base must be set up for the program.  The
    following three routines are  provided.  Only one of the
    routines can be used in a program.  The calls to the other
    two routines must be deleted. }

  read_data_base;            { The directive INTERFACE_TYPE is used t
                               tell whether the program is a logical
                               or physical interface }

  read_logical_data_base;    { read the logical design and CHIP files
                               to set up data base structures }

  read_physical_data_base;   { read the physical design and CHIP file
                               to set up data base structures }



  { call to find_all_flag_nodes is not needed for physical DIAL or w
    the I/O signals for the design are not available or needed
    the directive INCLUDE_IO_LIST makes it possible to set the globa
    variable include_IO_list to either TRUE or FALSE depending on
    whether the flag_nodes are needed. }

  if (errors_encountered * fatal_errors = []) and
     include_IO_list then
    flag_nodes := find_all_flag_nodes(root_drawing_name);



  (*****************************************)
  (*                                       *)
  (*  CALLS TO USER'S PROCEDURES GO HERE   *)
  (*                                       *)
  (*****************************************)



  if errors_encountered * fatal_errors <> [] then
    error(112 { run stopped });

  display_error_summaries;        { report errors encountered }

  exec_time(DIALLst, starting_time, FALSE);

  close_all_files;

end.
```

The following changes should be made in the template program:

1. Change the program name from userprog to a name which is more meaningful for the program being created.

2. If any of the user modifiable procedures which are supplied in the file userunit.pas are to be changed, then the line

   ```
   (*$U /u0/scald/dial/userunit.obj*) userunit;
   ```

   must be changed to:

   ```
   (*$U userunit.obj*) userunit;
   ```

   This will cause the utilities file which is in the local directory to be used during compilation rather than the utilities file which is in /u0/scald/dial.

3. Create the user's portion of the program. Put the user's procedures and the calls to the procedures in the correct places in the main program.

4. The main program must be changed to set up the correct data base. There are three routines to set up the data base and only one can be used by the program. The correct one should be kept and the calls to the other two should be deleted.

Userunit.pas contains the routines which a user can modify to customize an interface. It may be desired to modify the routines which are in the file. If this is to be done, the file can be copied into the current directory by issuing the command:

```
cp /u0/scald/dial/userunit.pas user.pas
```

User.pas can now be edited to meet the user's needs. If there are global variables and types which are needed in both the userunit and the main program, the variables and types should be defined in the interface section of the userunit. Userunit.pas can then be compiled by issuing the following commands:

```
cp /u0/scald/dial/userunit.make makefile
make
```

If this command runs without any errors, then a file named userunit.obj will be present in the current directory. Now every time user.pas is to be compiled, only the second (make) command has to be input.

COMPILING AND LINKING A DIAL PROGRAM

Once the program has been edited to accomplish the job which is desired and userunit has been changed and compiled, the main program must be compiled and linked. To do this, the following must be done:

1. The makefile must be copied to the current directory by typing:

   ```
   cp /u0/scald/dial/usermakefile makefile
   ```

   If the makefile which compiles userunit exists, it should be moved to another file before issuing this command.

2. If a new userunit was made, the line

   ```
   shorten long.pas userprog.pas
   ```

   must be changed to:

   ```
   shorten long.pas userprog.pas shorten.crf
   ```

3. The makefile should be edited to change the occurrence of "long.pas" to the name of the user's PASCAL program.

4. All occurrences of the name "userprog" should be changed to a name the user would like the program to be run as.

5. If the userunit supplied has been changed, the makefile should be changed to reflect it. Every occurrence of ${DDIR}userunit.obj should be changed to userunit.obj.

The makefile is now ready to compile and link the main program. This is done by issuing the command:

```
make
```

If errors occur during the make, they should be fixed. If it compiles and links correctly the program is ready to run.

RUNNING A DIAL PROGRAM

After the user's program has been edited, compiled and linked properly, it is ready to run. Before the program is to be run, it may be desired to change the name of the files which the program is to produce. This can be done by copying the file assignment file to the current directory and modifying it to output more meaningful names. The file can be copied by issuing the command:

```
cp /u0/scald/dial/dialassign .
```

A file must now be made which will do the file assignment and run the program. This file will contain two lines and the first line will be:

```
. /u0/scald/dial~~/~~assign        { if file names are not changed }
```

period necessary!

OR

```
. <user's file assignment file>   { if changed }
```

The second line will be:

```
<user's program name>
```

The program can now be run by typing in the name of the two line command program which has just been edited. After the run is complete the program should be checked for accuracy and debugged.

9.14 DIAL ON IBM

The files which are used to create a DIAL interface are:

1. CONSTS PASCAL - This file contains all of the global constant declarations for DIAL. This file should be used as reference in writing a DIAL program.

2. TYPES PASCAL - This file contains all of the global type declarations for DIAL. This file should be used as reference in writing a DIAL program.

3. VARS PASCAL - This file contains all of the global variable declarations for DIAL. This file should be used as reference in writing a DIAL program.

4. PROCS PASCAL - This file has definitions for all of the routines which are provided with DIAL. This file should be used as reference in writing a DIAL program.

5. DIALINC MACLIB - This file is a macro library which has all of the constant, type, variable and routine declarations in it. It is used when the DIAL program is compiled.

6. TEMPLATE PASCAL - A PASCAL source file template for a DIAL program is provided. This program is provided to be used as the starting point for a user's DIAL program. It should be copied to a local directory and edited.

Comments in the file describes how to make a logical or a physical DIAL program.

7. UTILS COPY - This program contains the source for the DIAL routines which can be modified by the user.

8. UTILS MACLIB - This file contains the DIAL routines which can be modified by the user. This macro libray was made so the user could use it if it is not desired to change the routines in the utils file.

9. MAKEUTIL EXEC - This command file is to be used if the user decides to change UTILS COPY. This file will make the changed UTILS into a macro library. MAKEUSER EXEC - This command file is supplied to make it easy for a user to compile and link his program. The command file can have the name of the program to be compiled passed to it, or the default value of USER PASCAL will be used.

10. DIAL TEXT - This is the pre-compiled DIAL library. It is used by the user's DIAL program to create an executable program. It contains all of the DIAL procedures and functions.

11. DIAL CRF - This is a cross reference file which is used when the user's program is shortened so the program can be properly compiled and linked.

CREATING A DIAL PROGRAM ON THE IBM

To create a DIAL program, it is suggested to start the development from the template program which is provided. The file TEMPLATE PASCAL should be copied from the SCALD disk to the current disk. The template program should look like this:

```
program userprog(CmpExp, Chips, PstXNet, PstXPrt, PstXRef, monitor,
                 DIALLst, DIALSPEC, inprog, infile, outfile, DIALback
                 DIALStat, DIALSigB, DIALPrtB);
  { this is a template source file for a DIAL program.  The user
    should copy this file to a local directory and edit to create
    a DIAL program. }


const

%INCLUDE CONSTS

type

%INCLUDE TYPES
```

```
var

  {  input files  }

  infile,                 { input file }
  inprog,                 { user input file }
  CmpExp,                 { compiler expansion file }
  Chips,                  { library chips file }
  PstXNet,                { expanded net list }
  PstXPrt,                { expanded part list }

  { output files }


  PstXRef,                { cross reference file }
  monitor,                { output execution running summary }
  DIALLst,                { main list file - errors + summary }
  DIALSPEC,               { special output format }
  DIALback,               { back annotation file }
  DIALStat,               { State file }
  DIALSigB,               { Signal State file }
  DIALPrtB: textfile;     { Part State file }

%INCLUDE VARS

 { user defined global variables go here }

%INCLUDE PROCS


%INCLUDE UTILS



  (****************************************)
  (*                                      *)
  (*      USER'S PROCEDURES GO HERE       *)
  (*                                      *)
  (****************************************)



  procedure close_all_files;
    { close all of the files which are still open }
  begin
    close_output_file(DIALLst, list_file);
  end; { close_all_files }


begin { main program body }

  init_DIAL;                    { initialize DIAL }
  read_directives_file;          { read the DIAL directives }


  { Set up the correct data base: LOGICAL or PHYSICAL.
    Only one of the three routines can be used in a program. }


  read_data_base;             { The directive INTERFACE_TYPE can be
                                used to define whether the design
                                is to be logical or physical. }

  read_logical_data_base;     { read the logical design and CHIP
                                files to set up data base
                                structures }

  read_physical_data_base;    { read the physical design and CHIP
                                files to set up data base structures }


  { call to find_all_flag_nodes is not needed for physical DIAL or wh
    the I/O signals for the design are not available or needed.
    The directive INCLUDE_IO_LIST can be used to set the finding of
    flag nodes on. }

  if (errors_encountered * fatal_errors = []) and
     include_IO_list THEN
    flag_nodes := find_all_flag_nodes(root_drawing_name);


  (****************************************)
  (*                                      *)
  (*  CALLS TO USER'S PROCEDURES GO HERE  *)
  (*                                      *)
  (****************************************)


  if errors_encountered * fatal_errors <> [] then
    error(112 { run stopped });

  display_error_summaries;        { report errors encountered }

  exec_time(start_elapsed_time, start_CPU_time, FALSE);

  close_all_files;

end.
```

The user should then make the following changes to the template program:

1. Change the program name from userprog to a name which is more meaningful for the program being created.

2. Any user defined global variables should be entered after the VARS include file. If there are any variable definitions before the INCLUDE file, the program will not work.

3. Create the user's portion of the program. Put the procedures which do the work in the correct place. Then but the calls to the procedures in the correct place in the main program.

4. The main program must be changed to set up the correct data base. There are three routines to set up the data base and only one can be used by the program. The correct one should be kept and the calls to the other two should be deleted.

It may be desired to change some of the routines which are supplied in DIAL. In the file UTILS PASCAL, there are five routines which can be changed by the user. If any of the routines are to be changed the file UTILS PASCAL must be copied from the SCALD disk to the current disk. The file can then be edited.

After the file has been changed, it must be made into a macro library. There is a command file which is provided to do this. The file is MAKEUTIL EXEC. The macro library can be made by issuing the command:

```
MAKEUTIL
```

The macro library is now ready to be used when the main program is compiled.

Now that the program has been changed to do what it is supposed to do, and the necessary changes have been made to the routines in the UTILS PASCAL file, the program can be compiled and linked. This can be done by issuing the command:

```
MAKEUSER <user's program name>
```

If the command file completed without any errors, then the interface is ready to run.

RUNNING A DIAL PROGRAM

Once the DIAL program has been compiled and linked, it is possible to run the program. The following must be done before the program can be run.

1. A DIAL directives file must be created. The libraries which are needed to run the design must be specified in this file. If the routine READ_DATA_BASE is used to set up the data base, a INTERFACE_TYPE directive should be in the file. It may be desired to put other standard directives or user supplied directives into this file.

2. A test case must be created. If the program is a logical DIAL program a compiler expansion file must be present. If the program is a physical DIAL program an expanded net list and an expanded part list must be present.

3. The user may desire to copy the file RUNDIAL EXEC from the SCALD disk. This command file set the run time environment for the program. The user may wish to change some of the file names to names which are more meaningful.

After all of these things have been completed, the program can be run by issuing the command:

```
RUNDIAL <user's program name>
```

If the program runs to a successful conclusion, the results should be checked and changes should be made if needed.

CHAPTER 10

INTERFACES

SCALD Interface Program

10.1 INTRODUCTION

The SCALD Interface Program provides an interface between the SCALDsystem and the user's physical design environment. The files output by this program are intended to supply all of the information needed by physical design systems. The SCALD Interface Program reads the expanded net and part lists created by the Packager and produces files for the user's physical design system. These output files include concise net list, concise part list, etc..

10.2 INTERFACE FILES

All output files are text files. Each file has a header which identifies the file, and the date and time of the Packager run. Each output file described in this manual is of the form:

```
<header>
<some list of items>
END <name of list>
```

where <header> describes the name of the list and the run of the Packager that created it. The header is of the following form:

```
<name of list> - 1 <date>
<user information>
```

where <name of list> identifies the output file and the information it contains, <date> is a string containing the date and time when the Packager was run, and <user information> is user specified information (such as engineer, revision, etc.) that is supplied in the header input file (which may be as many lines as desired). An example header:

```
CONCISE NET LIST - 1  12-AUG-1982 13:18:10.21
Revision 2a.                            A. E. Steinmetz
TTL (Transistor-Transistor-Logic) Example
```

There are five interface files which can be produced by the SCALD Interface Program. These files are:

1. dialcnet - Concise Net List

2. dialcprt - Concise Part List

3. dialbonl - Body Ordered Net List

4. dialstf - Part Stuff List

5. dialpgnd - Power and Ground List

The user can specify which files are to be produced by a run of the SCALD Interface Program by using a directive. This will be described in the section on running the SCALD Interface Program. The following sections describe the format of each file which can be produced by the program.

CONCISE NET LIST

The concise net list is a list of the nets in the design that have at least two nodes; that is, nets connecting one pin (such as NC nets) are not present in this net list. Single node nets are included if specified with the SINGLE_NODE_NETS ON; Packager directive (see the section on DIAL directives in Chapter 9 for details).

The form of the concise net list is as follows:

```
<header>
<list of nets>
END CONCISE NET LIST
```

Each net entry in the <list of nets> appears on one line. A net entry is of the following form:

```
<net name> <part designator> <pin number> <part type>
```

where <net name> is the physical net name assigned by the Packager. <part designator> is the physical part designator assigned by the Packager. <pin number> is the pin number assigned by the Packager. <part type> is the physical part type Packager expanded part file. Each element in the entry is separated from the next by at least one space.

An example net list:

```
CONCISE NET LIST - 1  12-AUG-1982 13:18:10.21
XYZ                      U31              1       74LS00
XYZ                      U31              2       74LS00
```

```
END CONCISE NET LIST
```

CONCISE PARTS LIST

The concise parts list consists of a list of all the physical part types used in the design and the quantities. The physical part type is assigned in the libraries defining the part. The format of the file is as follows:

```
<header>
<list of parts>
END CONCISE PARTS LIST
```

An entry in the <list of parts> has the following form:

```
<physical part type> <internal part number> <quantity>
```

where <physical part type> is the part name from the libraries, <internal part number> is the value of the PART_NUMBER property attached to the part (see PART_NUMBER in the glossary), and <quantity> is the number of those parts used in the design.

An example of a concise parts list:

```
CONCISE PARTS LIST - 1  12-AUG-1982 13:18:10.21
74LS00                   1820-0121               12
74LS04                   2002-4312-2             2
END CONCISE PARTS LIST
```

STUFF LIST

The stuff list consists of a list of physical part types and physical part designators. The list is ordered by physical part type and has the following form:

```
<header>
<list of parts>
END STUFF LIST
```

An entry in the <list of parts> has the following form:

```
<part type> <internal part number> <part designator>
```

where <part type> identifies the physical part type, <internal part type> is the value of the PART_NUMBER property attached to the part in the library (see the section describing PART_NUMBER), and the <part designator> identifies the physical part designator. Physical part designators for a given physical part type are listed in order. A blank line and a line consisting of '------' separates entries of different part types.

An example stuff list:

```
STUFF LIST - 1  12-AUG-1982 13:18:10.21
74LS00                       1820-2002                     U31
74LS00                       1820-2002                     U32

------

74LS04                       2003-2-3333                   U27
74LS04                       2003-2-3333                   U29
END STUFF LIST
```

POWER AND GROUND LIST

This list consists of physical part designators, physical part types and their power and ground pins. The list is ordered by physical part designator and has the following form:

```
<header>
<column labels>
<list of parts>
END POWER AND GROUND LIST
```

The first line of the list of parts is <column labels> which identifies the meanings of the columns. The first two labels are DESIGNATOR and PART TYPE. The rest of the columns (every eight places) are the names given the power pins in the libraries (such as VCC, GND, VEE, etc.). For example:

```
DESIGNATOR       PART TYPE                    VCC     GND
```

An entry in the <list of parts> has the following form:

```
<part designator> <part type> <power pin list>
```

where <part designator> identifies the physical part designator and the <part type> identifies the physical part type. The elements of the <power pin list> are arranged in columns; one per power supply specified in the design (see the section on power and ground pin specifications). If the power pins have multiple pin numbers, the additional pin numbers are printed on successive lines in their proper columns with the <part designator> and <part type> fields left blank. The supply names are listed alphabetically.

An example power and ground list:

```
POWER AND GROUND LIST - 1  12-AUG-1982 13:18:10.21
DESIGNATOR       PART TYPE                    GND     VCC
U27              74LS04                       7       14
```

```
U31           74LS00                  7      14
U99           74LS75                  5      12
END POWER AND GROUND LIST
```

CONCISE BODY ORDERED NET LIST

This list contains the same information as the Concise Net List (described above) but is ordered by physical part designator (body) rather than by net.

The form of the concise body ordered net list is as follows:

```
<header>
<list of parts>
END BODY ORDERED NET LIST
```

An entry in the <list of parts> appears as follows:

```
<physical part designator> <physical part type>
<list of physical pins and nets>
```

where <physical part designator> is the physical part name of the part. <physical part type> is the name of the part type. The pins of the part are given in <list of physical pins and nets> each entry of which has the following form:

```
<physical pin number> <physical net name>
```

where the <physical pin number> specifies the pin number of the part with pin numbers listed in increasing order. The net connected to the pin is specified by <physical net name>.

An example body ordered net list:

```
BODY ORDERED NET LIST - 1  12-AUG-1982 13:18:10.21
U1            74LS00             1       XYZ
                                 2       OUT1
                                 3       OUT0
U2            74LS10             1       XYZ
                                 2       FDBCK
                                 3       CNTRL
                                 4       XTRYL
                                 5       OUTOUT1
                                 6       OUTOUT0
                                 8       FDBCK
                                 9       CNTRL
                                 10      BOTTOM
                                 11      DIS
                                 12      OUTPUT1
```

```
                                          13        OUTPUTO
     END BODY ORDERED NET LIST
```

10.3 RUNNING THE SCALD INTERFACE PROGRAM

The SCALD Interface Program is very easy to run. All that has to be done is to make certain that there are three files resident in the current directory (the files are described below). The three files which the program needs are:

1. Expanded Part List - This is an Expanded Part List which has been produced by a run of the Packager. This file is made by the Packager after a design has been edited, compiled and then packaged.

2. Expanded Net List - This is an Expanded Net list which has been produced by a run of the Packager. Note that the Expanded Net and Part lists must be produced by the same run of the Packager.

3. Command File - This file serves two purposes: to describe the libraries which are needed to run the design and to specify the files which are to be produced. The file will be named:

   ```
   scald.cmd   { VAX and UNIX }
   scald cmd   { CMS }
   ```

   See the next section for the definition of the directives which can be used to run the SCALD Interface Program.

After the expanded part and net lists are present in the current directory and the directives file contains the information which is needed to run the SCALD Interface Program, the program is run by entering the following command:

```
gscald
```

The program produces the files which are specified in the directives file and a list file which contains a summary of the run of the program.

10.4 THE SCALD INTERFACE DIRECTIVES FILE

The SCALD Interface Program uses a directives file to specify which libraries are to be used for the design and which files are to be output by the program. Two directives are used to accomplish this. These directives are:

LIBRARY_FILE

Specifies the names of files containing library components. These files are produced by the SCALD Compiler using the OUTPUT CHIPS directive. Any number of libraries can be specifed with this directive. The names can be placed in a list separated by commas or listed individually with separate LIBRARY_FILE directives. For example, the directive:

```
LIBRARY_FILE '100k.prt', 'lsttl.prt';
```

specifies two library files, 100k.prt and lsttl.prt, and is equivalent to the directives:

```
LIBRARY_FILE '100k.prt';
LIBRARY_FILE 'lsttl.prt';
```

The SCALD Interface Program checks to make sure that a file is not specified more than once.

OUTPUT

Controls which output files are produced by the Packager. Each of the various output listings can be individually suppressed or enabled. All files are generated by default. The first OUTPUT directive encountered causes all output files to be turned off (so that they may be individually turned back on) unless the '-' option is used, in which case files are deleted individually. The 'ALL' identifier can be used to turn all files on or off. For example, the directive:

```
OUTPUT;
```

is equivalent to the directive

```
OUTPUT -ALL;
```

which turns off all output files.

The names of these files are listed separated by commas in a single OUTPUT directive or can be specified with multiple OUTPUT directives. For instance, the directive:

```
OUTPUT CONCISENETLIST,CONCISEPARTLIST;
```

is equivalent to:

```
OUTPUT CONCISENETLIST;
OUTPUT CONCISEPARTLIST;
```

In both of the above examples, the only output files that will be generated are the concise net list and the concise part list.

Each of the OUTPUT files are listed below. See previous sections for a description of the format and content of each file.

CONCISENETLIST
: Causes the concise net list to be output to the file DIALCNET.

CONCISEPARTLIST
: Causes the concise part list to be output to the file DIALCPRT.

STUFFLIST
: Causes the stuff list to be output to the file DIALSTF.

POWERANDGNDLIST
: Causes the power and ground list to be output to the file DIALPGND.

CBODYORDEREDLIST
: Causes the concise body-ordered net list to be output to the file DIALBONL.

CROSSREFERENCES
: Causes all of the cross references to be output to the file SCALD.XREF.

LOCALPARTXREF
: Causes the local part cross reference to be output to the file SCALD.XREF.

GLOBALSIGNALXREF
: Causes the global signal cross reference to be output to the file SCALD.XREF.

GLOBALPARTXREF
: Causes the global part cross reference to be output to the file SCALD.XREF.

10.5 EXAMPLES OF FILES

The following sections contain examples of the files which can be produced by the SCALD Interface Program. Each section contains the following information:

1. The name of the file which contains the information. The file names are given for the three operating systems on which the program can be run.

2. An example of the format and information produced and placed into the file.

CONCISE NET LIST

The Concise Net List is named:

```
          dialcnet.dat          { in VMS }
          dialcnet.dat          { in UNIX }
          dialcnet data         { in CMS }
```

The format of the concise net list is:

```
CONCISE NET LIST - 1 TUE JAN 10 13:23:53 1984
UN12MERGE8PA0          U2              8         74LS374
UN12MERGE8PA0          U3              4         74LS283
UN12MERGE8PA1          U2              7         74LS374
UN12MERGE8PA1          U3              1         74LS283
UN12MERGE8PA2          U2              4         74LS374
UN12MERGE8PA2          U3              13        74LS283
UN12MERGE8PA3          U2              3         74LS374
UN12MERGE8PA3          U3              10        74LS283
UN12MERGE8PB0          U1              4         74LS283
UN12MERGE8PB0          U2              18        74LS374
UN12MERGE8PB1          U1              1         74LS283
UN12MERGE8PB1          U2              17        74LS374
UN12MERGE8PB2          U1              13        74LS283
UN12MERGE8PB2          U2              14        74LS374
UN12MERGE8PB3          U1              10        74LS283
UN12MERGE8PB3          U2              13        74LS374
UN1LS28310PCI0         U1              9         74LS283
UN1LS28310PCI0         U3              7         74LS283
ZERO                   U1              7         74LS283
ZERO                   U2              1         74LS374
END CONCISE NET LIST
```

CONCISE PART LIST

The Concise Part List is named:

```
          dialcprt.dat          { in VMS }
          dialcprt.dat          { in UNIX }
          dialcprt data         { in CMS }
```

The format of the concise part list is:

```
CONCISE PART LIST - 1  TUE JAN 10 13:23:53 1984
74LS283                                  2
74LS374                                  1

Total                                    3
END CONCISE PART LIST
```

BODY-ORDERED NET LIST

The Body-Ordered Net List is named:

```
        dialbonl.dat            { in VMS }
        dialbonl.dat            { in UNIX }
        dialbonl data           { in CMS }
```

The format of the body-ordered net list is:

```
BODY ORDERED NET LIST - 1  TUE JAN 10 13:23:53 1984
U1           74LS283        1        UN12MERGE8PB1
                            4        UN12MERGE8PB0
                            7        ZERO
                            9        UN1LS28310PCI0
                            10       UN12MERGE8PB3
                            13       UN12MERGE8PB2
U2           74LS374        1        ZERO
                            3        UN12MERGE8PA3
                            4        UN12MERGE8PA2
                            7        UN12MERGE8PA1
                            8        UN12MERGE8PA0
                            13       UN12MERGE8PB3
                            14       UN12MERGE8PB2
                            17       UN12MERGE8PB1
                            18       UN12MERGE8PB0
U3           74LS283        1        UN12MERGE8PA1
                            4        UN12MERGE8PA0
                            7        UN1LS28310PCI0
                            10       UN12MERGE8PA3
                            13       UN12MERGE8PA2
END BODY ORDERED NET LIST
```

POWER AND GROUND LIST

The Power and Ground List is named:

```
          dialpgnd.dat                 { in VMS }
          dialpgnd.dat                 { in UNIX }
          dialpgnd data                { in CMS }
```

The format of the power and ground list is:

```
POWER AND GROUND LIST - 1  TUE JAN 10 13:23:53 1984
DESIGNATOR       PART TYPE        GND      VCC
U1               74LS283          8        16
U2               74LS374          10       20
U3               74LS283          8        16
END POWER AND GROUND LIST
```

STUFF LIST

The Stuff List is named:

```
          dialstf.dat             { in VMS }
          dialstf.dat             { in UNIX }
          dialstf data            { in CMS }
```

The format of the stuff list is:

```
STUFF LIST - 1  TUE JAN 10 13:23:53 1984
74LS283                                     U1
74LS283                                     U3

----------------

74LS374                                     U2

----------------

END STUFF LIST
```