CAE Views HDL Programming Guide

Product Version 23.1
September 2023

Contents

Preface

About This Programming Guide

This Programming Guide explains how to use CAE Views to create custom interfaces to translate databases into a format that can be used by an external system.

Finding Information in This Programming Guide

This programming guide covers the following topics:

See...	For Information About...
Chapter 1, “CAE Views Overview,”	Introduction to CAE Views
Chapter 2, “Developing an Interface with CAE Views,”	Using template files and routines for creating customized interfaces
Chapter 3, “CAE Views Data Structures,”	Pre-defined data structures of CAE Views
Chapter 4, “CAE Views Routines,”	Pre-defined routines of CAE Views for developing customized interfaces
Chapter 5, “CAE Views Global Variables,”	Descriptions of global variables used in creating interfaces
Chapter 6, “CAE Views Directives,”	Using CAE Views pre-defined directives and creating new directives for interfaces
Chapter 7, “Command Line Arguments,”	Using pre-defined command-line arguments and defining custom arguments.
Appendix A, “CAE Views Error Messages,”	Error messages issued by CAE Views
Appendix B, “Part Properties Tables,”	Using part properties tables and creating your own tables

Typographic and Syntax Conventions

This list describes the syntax conventions used for tools used in the Allegro Design Entry HDL Rules Checker User Guide.

`literal (LITERAL)`	Nonitalic or (UPPERCASE) words indicate key words that you must enter literally. These keywords represent command (function, routine) or option names.
`argument`	Words in italics indicate user-defined arguments for which you must substitute a value.
|	Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other character. For example, `command argument	argument`
[ ]	Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.
{ }	Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.
...	Three dots (...) indicate that you can repeat the previous argument. If they are used with brackets, you can specify zero or more arguments. If they are used without brackets, you must specify at least one argument, but you can specify more. *`argument`*...: specify at least one argument, but more are possible [*`argument`*]...: you can specify zero or more arguments
,...	A comma and three dots together indicate that if you specify more than one argument, you must separate those arguments by commas.
`Courier font`	Indicates command line examples.

1 CAE Views Overview

This chapter describes the following:

- Overview
- Types of Interfaces
- Types of Translations

Overview

CAE Views lets you create custom interfaces to translate the database into a format that can be used by an external system and to update the database with changes from a physical design system.

CAE Views is written in C and has routines for tasks common to all database interfaces. To create a custom interface, you only need to write the routines that are unique to your application.

CAE Views reads the database and creates a CAE Views database subject to the directives you provide for your CAE Views application.

Your routines process the CAE Views database and output formatted information based on the requirements of the external system.

Types of Interfaces

The particular requirements for a specific interface depend not only on the format required by the external system, but also on the type of information to be translated.

In the past, an interface fell into one of two categories: *logical* or *physical*. A logical interface expressed design connectivity in terms of pin names, and a physical interface expressed design connectivity in terms of pin numbers.

A modern interface has a more complex array of choices. With CAE Views, you can create a customized logical or physical interface that implements translations that fall into several categories.

An HDL-centric environment supports *logical* and *physical* interfaces. A physical interface expresses design connectivity in terms of pin numbers. It performs a one-way translation, from the HDL database to an external system.

CAE Views implements the following translations:

- Forward Translation
- Flat Translation
- Hierarchical Translation

Logical or Physical Interface

The choice of a *logical* or *physical* interface depends on the type of information to be translated. The external system determines this choice. For example, most simulators require only logical information, while printed circuit board layout systems require physical information.

Logical Interfaces

A *logical* interface processes the database produced by Allegro Design Entry HDL and does SIZE replication for components that use the SIZE property. A *logical* interface can access part libraries if needed by a particular application.

If your CAE Views application is a *logical* interface, it can use the part properties table feature. This feature lets you to create new part types from a basic type or to attach new body properties to a part type without recreating or modifying the library files containing the part type definitions.

Since the Packager-XL adds physical information, a *logical* interface does not usually include physical information (pin numbers or location designators). If you want to translate physical design information, you must create a physical CAE Views application.

Physical Interfaces

A *physical* interface processes the output of the Packager-XL (the expanded part and signal lists). A physical interface can access all physical information in a design. The interface can also access required part libraries.

A *physical* interface is used with physical CAD tools. In general, such tools require a flat description of the physical design, complete with pin numbers and location designators.

Types of Translations

Forward Translation

A *forward* translation is the most common type of translation. It translates the HDL database to an external format. The requirements of the external system define this format. For example, an interface could generate the bill of materials for a specific design in the HDL database. For such an interface, you might add part numbers, using the part properties table feature.

A *forward* translation is a *physical* interface. The type of information an interface translates depends on the external system. Printed circuit board layout systems require physical information.

Flat Translation

A *flat* translation is the most straightforward way to describe an HDL design. All parts in a flat description are primitive parts, meaning that they come from Cadence or user-defined libraries.

A flat translation could be a physical interface. The type of information an interface translates depends on the external system.

Hierarchical Translation

In *hierarchical* translation, the information provided by CAE Views includes hierarchical modules and the primitives in the design. Such translations are useful for accessing the design hierarchy information.

2

Developing an Interface with CAE Views

CAE Views has template files and modifiable and precompiled routines that you use to create a customized interface. This chapter includes information on the following topics:

- CAE Views Templates and Files: How to access these files and their contents.
- Setting Up a Development Directory: How to create a custom interface.
- Program Flow Within an Interface: The various phases of interface program operation.
- Writing a CAE Views Program: How to modify CAE Views templates and files, and combine them with your own routines to create a custom interface.
- Predefined CAE Views I/O Files: The names of predefined files, their default values, and a description of their contents.

CAE Views Templates and Files

The directory location of the CAE Views templates and files varies according to the architecture you are using.

CAE Views contains the building blocks for creating a customized interface. CAE Views includes files that the interface uses and other files that you either modify or use as examples.

For example, the name of the CAE Views library file is:

- `caeviewshdl_sec.lib`

The file `user.c` includes three template routines and two template tables: *user_cmd_table* and *user_arg_table*. The CAE Views directory location is

`$<`*`your_install_dir`*`>/tools/caeviewshdl`

Table 2-1 on page 14 describes the files in the CAE Views directory for the HDL-centric environment.

Table 2-1 CAE Views Templates and Files for the HDL-centric Environment

File	Description
`main.c`	A file that contains a template for a C source program. You use this program as the starting point for creating your own interface. Comments in the file describe how to create the different interface types.
`user.c`	A file that contains three demo interfaces and two template tables: the *directives* table and the *line argument* table.
`makefile`	A file that contains a template for a make file to create an executable interface.
`caeviewshdl_sec.lib`	A file that contains the CAE Views library. This library is linked with your development code to create an executable interface.
`caeviewshdl.h`	A file that contains definitions for all CAE Views data structures, external function and global variable declarations. This file must be included (#include) in any module that uses a CAE Views data structure, global variable, or function (that is, in your modified `main.c` and `user.c` files).

Setting Up a Development Directory

To create a custom interface,

1. You can copy several files and templates in the CAE Views directory to a development directory where you can modify them for your specific application.
2. Create a development directory in your account and copy the following files from the `caeviewshdl` directory to your development directory:
   - ❑ `main.c`
   - ❑ `user.c`
   - ❑ `makefile`

Program Flow Within an Interface

The program flow within any interface consists of several phases. CAE Views contains routines for the first four phases. Since output generation is unique to each interface, you need to write the output generation routines. The program flow within an interface includes the following phases:

- Initialization Phase
- Database Input Phase
- Packaging Phase (optional)
- Design Analysis Phase (optional)
- Output Generation Phase

Initialization Phase

The first phase of an interface program initializes the CAE Views internal variables. During this phase, the directives file and any command line arguments are processed. If needed, you can also write additional initialization code for your specific application.

CAE Views contains the following routines to carry out initialization tasks
.

KV_init()	Initializes CAE Views package.
KV_print_welcome()	Prints a welcome message.

KV_set_hdl_architecture	Initializes CAE Views for the HDL-centric environment.
KV_read_directives()	Processes the directives file and command line arguments for the CAE Views application.

Database Input Phase

The second phase (database input) of an interface program creates the CAE Views database.

CAE Views sets up a database using the Packager-XL output or by calling the expander. CAE Views supplies a read database routine to set up the appropriate database.

KV_read_data_base()	Reads in the appropriate files and calls the expander to set up the CAE Views database.

Packaging Phase

The packaging phase sets up a physical database based on a logical database. If your CAE Views application is a logical interface, CAE Views gives you the option of modifying the logical database into a physical database.

To modify a logical database, you must write a packager function and have `KV_package_routine` set to point to this function. After the logical database is read, the packager function is automatically called.

Design Analysis Phase

This optional phase lets you verify site-specific design rules on a design. Your added routines analyze the data structures and apply tests to detect specific problems or violations.

Output Generation Phase

The last phase of a CAE Views application selects appropriate data and sets up the format of the output files. During this phase, CAE Views outputs the design to text files.

Writing a CAE Views Program

After you copy the CAE Views files and templates into your development directory, carry out the following tasks to create your customized CAE Views interface:

- Write routines specific to your application.

  You can write these routines in separate modules or include them in the `user.c` template file.

- Modify the `main.c` template file.

  Include calls to any of the routines that you added.

See Chapter 6, "CAE Views Directives," and Chapter 7, "Command Line Arguments," for detailed information on using the directives and command line argument tables.

- Add, delete, or modify the command directives and command line arguments by editing the command and argument tables in `user.c`.
- Add any unique error messages that you included in your new routines to your `main.c` file using the KV_add_error function.
- Edit the template "make" file. Then compile and link your modified `main.c` and `user.c` files and any new '`.c`' modules with the predefined CAE Views library file (`caeviews.a` or `caeviews.olb`) and, if you are creating a logical interface, the linker libraries to create an executable interface.

  Name your interface in the make file (that is, `interface.ex`).

- Move your executable interface to the `tools` directory (`$`*`CDS_INST_DIR`*`/tools/interfaces`*)*.

  Modify the template `caeviews.sh` or `caeviews.com` script so that your new interface is accessible. Rename the script *ginterface* and put it in your path (that is, `$`*`CDS_INST_DIR`*`/tools/bin`*)*.

- Create a template command file.

Modifying main.c

The `main.c` file is a template for you to follow in creating your main routine. The file contains the following:

- physical interface
- logical interface

The following sections provide examples on how to modify `main.c` for physical and logical interface applications.

Modifying main.c for a Physical Interface

A physical interface template is shown in Figure 2-1 on page 19.

Figure 2-1 Physical Interface Template

```
#include ”caeviews.h”
     extern KV_CMD_ENTRY user_cmd_table[];
     extern KV_ARG_ENTRY user_arg_table[];

     /****************************************************************
     **       PROGRAM  : Demo of Valid CaeViews
     **
     **       generates a physical interface example
     ******************************************************************

extern void init_demo();
     extern void read_demo_file();
     extern void write_demo_file();
     extern KV_STRING *DEMO_PART_prop;

/*****************************************************************
**
** main.c -- A template file for the user to write their main program.
The file contains three different interface examples
i.e. logical, physical interfaces.
*/
    extern void init_demo();

    extern void read_demo_file();

    extern void write_demo_file();

    extern KV_STRING *DEMO_PART_prop;

  A char      KV_program_name[]  = ”Demo Forward Interface”;

  B char      KV_program_date[]  = { DATE };
  C
    char      KV_program_version[] = { VERSION };

    KV_PART_TYPE  *top           = NULL;

    main(argc,argv)
         int argc;
         char **argv;
         {
  D KV_init(argc,argv);/* init CaeViews */
  E      init_demo();/* init the demo package */
  F KV_set_hdl_architecture()/* init CaeViews for the HDL environment */

  G
    KV_cmd_ptr = user_cmd_table;/* we use our own table */
         KV_arg_ptr = user_arg_table;/* we use our own table */

    KV_add_power_node = TRUE;/* adds power nodes to the database */

    KV_print_welcome();/* say hello to the world */
```

Figure 2-2 Physical Interface Template, *continued*

```
H  (void)KV_read_directives(argc,argv);/* reads directives file and
             cmd arg */

   (void)KV_read_data_base(&top);/* reads the data base */

   /* CALLS TO USER'S PROCEDURE GO HERE */
I
J  KV_exit(0);

K  }
```

If you are using the `main.c` file to create your physical interface, remove the portions of the file for the logical interfaces. These portions are located between lines:

```
#ifdef DEMO_LOGIC and #endif DEMO_LOGIC
#ifdef DEMO_DELTA and #endif DEMO_DELTA
```

The paragraph letters in the left column correspond to the callouts in the physical interface template shown in Figure 2-1 on page 19.

A	Change the *KV_program_name* string to a name that describes your interface.
B	The *make* script fills in the *KV_program_date* and *KV_program_version* strings. Don't delete.
C	The routine *KV_read_data_base*() returns the name of the design. Initialize the pointer top to NULL.
D	*KV_init* (*argc*, *argv*) is the function that initializes all global variables and tables used by CAE Views. Make sure to call this routine first in your interface.
E	*init_demo*() is the function that adds error messages to the demo interface error list and puts the specified strings into the string table. Modify this routine in `user.c` to add error messages to your interface.
F	*KV_set_hdl_architecture*() is the function that initializes CAE Views for the HDL environment.

G	These two lines are pointers to the command directives table and command line argument table in `user.c`:
	
	KV_cmd_ptr = user_cmd_table
	KV_arg_ptr = user_arg_table
	
	See Chapter 6, "CAE Views Directives," and Chapter 7, "Command Line Arguments," for more information on table entry format.

H	*KV_read_directives* (*argc*, *argv*) reads the directives in the command directive file and the command line arguments.
I	*KV_read_database*(&*top*) reads Packager-XL files and invokes CAE Views to set up the database elements.
J	Write the routines specific to your application and place calls to these routines in `main.c`.
K	Use *KV_exit*() to exit the program so that the output files are generated and properly closed.

Modifying main.c for a Logical Interface Application

A physical interface template is shown in Logical Interface Template on page 22.

Figure 2-3 Logical Interface Template

```
  /*************************************************************

      PROGRAM    : Demo of Valid CaeViews
            generates a Logical Hierarchical or Flat Interface
  *************************************************************/

  extern void read_demo_file();
  extern void write_demo_file();
  extern void write_delta_demo_file();
  extern void write_log_demo_file();
  extern void init_demo();
A charKV_program_name[]= "Demo Logical Interface";
B charKV_program_date[]= { DATE };
  charKV_program_version[]= { VERSION };
C KV_PART_TYPE   *top= NULL;
  main(argc,argv)
  int argc;
  char **argv;
  {
D KV_init(argc,argv);/* init CaeViews */
E init_demo();/* init the demo package */
F KV_cmd_ptr = user_cmd_table;/* we use our own table */
  KV_arg_ptr = user_arg_table;/* we use our own table */
G KV_read_logical_db = KV_build_in_read_logical_db;
      /* enable logical */
  KV_expansion_style = HIERARCHICAL;/* default:hierarchical
          netlist */
  KV_print_welcome();/* say hello to the world */
  (void)KV_read_directives(argc,argv);
          /* read directive file and
              cmd arg */
  KV_want_reference = FALSE;/* no reference files */
  KV_archive = FALSE;/* no archive */
  KV_cross_reference = FALSE;/* no cross reference */
  (void)KV_read_data_base(&top);/* reads the data base */
H /*CALLS TO THE USER'S PROCEDURE GO HERE */
I /* The following is an example of a logical interface: */
  KV_create_physical_net_names();/* more restricted name */
     write_log_demo_file(top);/* generates logical demo file */
  #ifdef lint
     write_demo_file(top);
     write_delta_demo_file(top);
     read_demo_file(top);
  KV_generate_feedback_netlist(top);
     return TRUE;
```

Figure 2-4 Logical Interface Template, *continued*

```
  #endif lint
J KV_exit(0)
  } /* end of main */
```

The paragraph letters in the left column correspond to the callouts in the logical interface template shown in Figure 2-3 on page 22.

A	Change the *KV_program_name* string to a name that describes your interface.
B	The make script fills in the *KV_program_date* and *KV_program_version* strings. Don't delete.
C	The routine *KV_read_data_base*() supplies the name of the design. Initialize the top to NULL.
D	*KV_init* (*argc*, *argv*) is the function that initializes all global variables and tables used by CAE Views. Make sure to call this routine first in your interface.
E	*init_demo*() is the function that adds error messages to the demo interface error list and puts the specified strings into the string table. Modify this routine in `user.c` to add error messages to your interface.

F	These two lines are pointers to the command directives table and command line argument table in `user.c`:
	
	KV_cmd_ptr = user_cmd_table
	KV_arg_ptr = user_arg_table
	
	See Chapter 6, "CAE Views Directives," and Chapter 7, "Command Line Arguments," for more information on table entry format.

G	This line sets the *KV_read_logical_db* function pointer for a logical interface application: KV_read_logical_db = KV_build_in_read_logical_db This line, *KV_expansion_style* = *HIERARCHIAL*, sets the default to hierarchical designs. This value can be overridden by including an EXPANSION_STYLE FLAT; directive in the command file.
H	This line, *KV_read_database*(&*top*), calls the expander to set up the database.

I	Write the routines specific to your application and place calls to these routines in `main.c`.
J	Use *KV_exit() to* exit the program so that the output files are generated and properly closed.

Writing Routines Specific to the CAE Views Application

You need to write your own routines to process the CAE Views database. These routines extract the information needed by the external system for which you are creating an interface.

CAE Views provides predefined utilities that are ready to link with the routines you write for your custom interface. Chapter 4, "CAE Views Routines," provides detailed information on CAE Views utilities and routines in *user.c*. These utilities fall into the following categories:

- User Interface
- Error Reporting
- Debugging
- Sorting
- Input/Output
- Heap Management
- Searching
- Time Keeping

In addition to the categories listed above, a miscellaneous category of utilities provides, among other things, routines that are specific to the task of translating or manipulating the database.

The file `user.c` provides three interface template routines that are examples of how to use CAE Views functions. You can modify the routines in `user.c` to create physical and logical interfaces.

Modifying Command Directive and Command Line Argument Tables

The file `user.c` also includes two tables: *user_cmd_table* and *user_arg_table*.

These template tables are provided as examples for you to use and modify depending on your interface. The entries contained in the command directive and line argument tables determine what directives and line arguments your particular interface recognizes. Chapter 6, “CAE Views Directives,” and Chapter 7, “Command Line Arguments,” explain how to create your own table entries.

Defining Error Messages

The interface you create probably requires unique error messages specific to its operation. Error messages inform you of any problems encountered in the translation of a database or running the interface. You can use CAE Views routines to add error messages to your interface.

Use the routine *KV_add_error*(*type*,*number*, *message*) to define a new error message. The parameters to this routine define the severity level of the error, assign an error number, and provide you a message. You can define any number of error messages for your interface. Appendix A, “CAE Views Error Messages,” has information on the currently-assigned error messages.

For example, the line:

```
KV_add_error(OVERSIGHT, 200, KV_translate("Not legal type");
```

adds error number 200 to the error list. When error number 200 is called (KV_error(200)), the message *Not legal type* is printed out in the listing file and on the screen.

Compiling and Linking

After you create your new interface program, compile and link it.

Note: When compiling and linking a Caeviews-based program, you might see an error about an undefined reference to `tel_prog_tag`. In such cases, add `char* tel_prog_tag="release";` to the source file then continue with the program linking.

The template makefile is set up to create two demonstration programs (`logic.ex` and `phys.ex`) by compiling `main.c` and `user.c,` and linking with `caeviews.a` and the required libraries.

Edit the makefile and make the appropriate changes for your interface. Be sure to include all other `.c` modules you have created and the CAE Views library reference.

Creating a Command Directive File

In order to run the interface, you need to have a local project file in the directory where the interface is called. This file is created by the Project Manager, when you start a new project, and has an extension `.cpm`. To provide CAE Views application with an appropriate set of directives, do the following:

- Edit the project file.
- Add a section for the interface application at the end of the `.cpm` file

  ```
  START_INTERFACE
  ```

  ```
  END_INTERFACE
  ```

  where INTERFACE gets replaced by the name of the interface application.
- Add the required directives between START_INTERFACE and END_INTERFACE entries.

Testing the Interface

Create a test case. For a logical interface, expand the design first. For a physical interface, run the Packager-XL to create the required expanded net list, expanded part list, and chips files.

To run your test case

➤ From the design directory, enter:

```
ginterface -proj <path to project file>
```

Predefined CAE Views I/O Files

A CAE Views interface reads from and writes to a number of different files. The table below lists the routines and their associated default files, and describes the file contents.

Table 2-2 Pre-Defined CAEViews I/O Files

Default Filename	**Description**
KV_chip_file[] = “pstchip.dat”	The chips input file used by a physical interface.

Default Filename	Description
KV_current_file[]	The current file being parsed.
KV_debug_file[] = "interface.dbg"	The debug output file used by all interfaces.
KV_header_file[] = ""	The header input file for all interfaces.
KV_list_file[] = "interface.lst"	The listing output file used by all interfaces.
KV_scale_factor_file = ""	The scale factor input file used by a logical interface.
KV_xnet_file[] = "pstxnet.dat"	The expanded netlist input file used a physical interface.
KV_xprt_file[] = "pstxprt.dat"	The expanded part list input file used by a physical interface.

* *interface* is the base name of the program. (such as, if your translator is `vhdl.ex`, the listing file is `vhdl.lst` by default).

3

CAE Views Data Structures

This chapter describes the following:

Overview

Interfaces developed with CAE Views retrieve information from predefined data structures. CAE Views general data structures apply to physical and logical, and interfaces.

The fields or “members” that are filled in these data structures vary in accordance with the type of interface you are designing. For example, logical CAE Views does not fill in the member package in the KV_INSTANCE structure.

CAE Views data structures are defined in the file `caeviewshdl.h`, along with CAE Views global variable and external function declarations. Each source file that refers to a CAE Views global variable, a CAE Views data structure, or a CAE Views function must contain the line:

```
    #include “caeviewshdl.h”
```

Most structures include the members reserved and value. If needed, your CAE Views application can use these members, but you need to keep in mind that:

- The contents of these members are undefined
- Some CAE Views routines use these members

The reserved member is used by the compare module and the following routines:

KV_set_alternate_part()
KV_find_alternate_part()
KV_generate_cross_reference_file()

CAE Views does its own heap memory management.

To optimize performance

➤ CAE Views does its own heap memory management.

All CAE Views structures are dynamically created by allocating them from 4K- byte pages. Cadence recommends that you use the built-in CAE Views allocation routines if you need to create any CAE Views structures. (See *KV_create()* and *KV_new_...()* functions.)

The KV_CMD_ENTRY and KV_ARG_ENTRY structures retrieve information contained in the directives and command line argument tables. These structures are described in Chapter 6, “CAE Views Directives,” and Chapter 3, “CAE Views Data Structures.”

General Structure Types

When you call *KV_read_data_base* from your main module, CAE Views automatically creates all of the structures required to describe your design. CAE Views has two KV_PART_TYPE pointers to walk through the data structures:

- a global pointer called *KV_part_type_list* which points to the first KV_PART_TYPE structure in a *define-before-use* list.
- a pointer filled by *KV_read_database* function which points to the KV_PART_TYPE structure that describes the root drawing (top module).

For example, after the call:

```
status = KV_read_data_base(&top)
```

`top` points to the root drawing description.

The structures that follow are arranged in alphabetical order.

KV_BODY_PIN Structure

For each pin of a body (even for vectorized pins), there is an associated KV_BODY_PIN structure that describes the logical pin. The KV_BODY_PIN structure and its members are defined below.

```
typedef struct KV_body_pin {
    struct KV_body_pin *next; /* next body_pin */
    struct KV_string *name; /* name of the body pin */
    struct KV_property *properties;/* body pin properties */
    int MSB; /* most significant bit offset */
    int LSB; /* least significant bit offset */
    int subscript; /* bit subscript */
    struct KV_pin*pins; /* pins for that body_pin */
    struct KV_part_type*part_type;/* part_type it belongs to */
    int common_pin;/* common!=0,total=#sections-1 */
    VOID*reserved; /* for your own usage */
    int value; /* for your own usage */
} KV_BODY_PIN;
```

The list of all logical pins on a body can be accessed by the body_pins pointer of the KV_PART_TYPE structure.

next	The logical pins are linked together by the next member of the KV_BODY_PIN structure.
name	The name member points to a KV_STRING structure that contains the logical name of the pin.
MSB LSB subscript	If the body pin is vectorized, the member MSB holds the upper value of the subscript, the member LSB holds the lower value of the subscript, and the member subscript holds the value of the subscript for this body pin. If the pin is not vectorized, the three members MSB, LSB, and subscript are all equal to -1.
pins	The pins member points to a list of physical pins for this logical pin. The next physical pin for this logical pin is given by the pointer next_section in the KV_PIN structure.
properties	The first body pin property for the body is accessed by the properties pointer. With the next member of KV_PROPERTY structures, you get all body pin properties for the body pin.
part_type	The *part_type* pointer is a back-link to the part type this pin belongs to.
common_pin	The *common_pin* member is a flag that indicates whether or not the pin is a common pin. If the pin is not a common pin, the flag is set to *(FALSE)*. If the pin is a common pin, the flag is set to *(TRUE)*. If a pin is common to all sections (global pin), the value of the flag is equal to the number of sections minus one.
reserved value	The reserved and value members are available for your own use in your particular interface.

Note: Special use of the KV_BODY_PIN structure for power rails

The list of all power rails can be accessed by the global KV_BODY_PIN pointer *KV_power_list*. The next member is used to find the next power rail. The name pointer gives the name of the power rail. The MSB, LSB, subscript, and common_pin members are always equal to zero. The pins, properties, and part_type pointers are set to *NULL*.

KV_INSTANCE Structure

The KV_INSTANCE structure describes an instance of a body. The KV_INSTANCE structure and its members are defined below.

```
typedef struct KV_instance {
    struct KV_instance *next; /* next instance */
    struct KV_string *name; /* instance name */
    struct KV_property *properties; /* body properties */
    struct KV_instance *next_on_package;/* next instance on package */
    struct KV_instance *next_same_type;/* next instance of same type */
    int section_number;/* section's number */
    struct KV_package *package; /* package it belongs to */
    struct KV_node *nodes; /* all nodes for this section */
    struct KV_part_type*part_type;/* part_type of this instance */
    struct KV_part_type*parent;/* module it belongs to*/
    struct KV_instance*hash_link;/* do not use */
    VOID*reserved; /* for your own usage*/
    int value; /* for your own usage*/
 } KV_INSTANCE;
```

`next`	The next member links the instances in a module together.
`name`	The name member points to a KV_STRING structure that contains the instance name string. This string contains the path name of the instance, which is created and assigned by Allegro Design Entry HDL.
`properties`	The first instance property for the body is accessed by the properties pointer. Use the next member in the KV_PROPERTY structure to capture all instance properties for that body.

`section_number`
`next_on_package`
`package`
	If instances are put in packages, the section_number member gives the number of the section in which this instance fits, next_on_package gives the next instance (if any) which fits in the same package.

`next_same_type`	All instances of the same type are linked together by the next_same_type member.
`nodes`	The nodes member points to the first node on the instance. The member next_on_instance in the KV_NODE structure lets you follow all nodes on the instance.
`part_type`	The part_type member points to the part type of the instance.
`parent`	The parent member is a back-link to the module the instance belongs to. The instance is part of the definition of the module parent.
`hash_link`	The hash_link should not be modified.

`reserved value`	The reserved and value members are available for your own usage in your particular interface.

KV_NET Structure

The KV_NET structure describes a 1-bit wide logical and/or physical signal. The KV_NET structure and its members are defined below.

```
typedef struct KV_net {
    struct KV_net *next; /* next net */
    struct KV_string *physical_name; /* physical name of net */
    struct KV_string *logical_name; /* logical name of the net */
    struct KV_property *properties; /* net’s properties */
    int number_nodes; /* number of nodes on net */
    struct KV_node *nodes; /* all nodes on net */
    struct KV_part_type*parent; /* module it belongs to */
    struct KV_net *hash_link; /* do not use */
    struct KV_net *reserved; /* for your own usage */
    int value; /* for your own usage */
} KV_NET;
```

`next`	The signals in a module are linked together with the next member.
`physical_name`	The physical_name member points to the signal physical name string.
`logical_name`	The logical_name member points to the signal logical name string.
`properties`	The properties member points to the list of net properties for the signal.
`nodes`	The nodes member points to the list of nodes attached to the signal.
`number_nodes`	The value contained in the number_nodes member indicates the number of nodes on the signal.
`parent`	The parent member is a back-link to the part type (module) the signal belongs to. The signal is part of the definition of the module parent.
`hash_link`	The hash_link member must not be modified.
`reserved value`	The reserved and value members are available for your own usage in your particular interface.

KV_NODE Structure

The KV_NODE structure describes a pin on a specific package, instance, or signal. There are three kinds of nodes:

- **physical nodes** that belong to an instance and package.

  So the package and instance members are not null.

- **logical nodes** that belong to an instance, but not to a package.

  The package member is NULL. In logical applications, all nodes are logical nodes. In physical applications, some common pins are logical nodes.

- **Interface nodes** that connect the module to the external world.

  Each interface signal (signal with \I attribute) includes an interface node. These nodes are found only in logical applications. The package and instance pointers are NULL.

The KV_NODE structure and its members are defined below.

```
typedef struct KV_node {
    struct KV_node *next; /* next node */
    struct KV_string *name; /* instance pin name */
    struct KV_property*properties; /* instance pin properties */
    struct KV_node *next_on_net; /* next node on net */
    struct KV_node *next_on_package;/* next node on package */
    struct KV_node *next_on_instance;/* next node on instance */
    struct KV_pin *pin; /* physical pin pointer */
    struct KV_net*net;/* net it belongs to/
    struct KV_package *package; /* package it belongs to */
    struct KV_instance*instance; /* instance it belongs to */
    VOID*reserved; /* for your own usage*/
    int value; /* for your own usage*/
} KV_NODE;
```

next	The next member links the nodes in a module together.
name	The name member points to the logical node name string.
pin	The pin member points to the physical pin this node is an instance of. This KV_PIN structure contains the pin number (alphanumeric). Through the body_pin member in this KV_PIN structure, you get the logical pin name (which might be different from the name member).
properties	The properties member points to a list of node properties.
next_on_net net	All nodes on a *net* are given by the nodes members in the KV_NET structure and are linked together by the next_on_net pointer. Each node points to the net to which it is connected (net member).
next_on_package package	All nodes on a *package* are given by the nodes member in the KV_PACKAGE structure and are linked together by the next_on_package pointer. The package member points to the package (if any) the node belongs to.

next_on_instance instance	All nodes on an *instance* are given by the nodes member in the KV_INSTANCE structure and are linked together by the next_on_instance pointer. The instance member points to the instance (if any) the node belongs to.
reserved value	The reserved and value members are available for your own use in your particular interface.

KV_PACKAGE Structure

The KV_PACKAGE structure describes a physical part, an instance of a specific part type. The KV_PACKAGE structure and its members are defined below.

```
typedef KV_struct package {
    struct KV_package *next; /* next package */
    struct KV_string *name; /* package’s name */
    struct KV_package *next_same_type;/* next package of same type*/
    struct KV_instance *instances; /* first instance for package*/
    int used_sections; /* number of used sections */
    struct KV_node *nodes; /* first node of package */
    struct KV_part_type*part_type; /* part_type of this package*/
    struct KV_part_type *parent; /* module it belongs to*/
    struct KV_package *hash_link; /* do not use */
    VOID*reserved; /* for your own use */
    int value; /* for your own use */
} KV_PACKAGE;
```

next	The next member links all the packages in a module together.
name	The name member points to the package name string.
next_same_type	The next_same_type member links all packages of the same type together.
instances	The instances member points to the first instance contained in a section of the package. The instances put in different sections of the same package are linked together by the next_on_package member in the KV_INSTANCE structure.
used_sections	The value in the member used_sections indicates the number of sections used in the package.
nodes	The nodes member points to the first node on the package. All nodes on the package can be found by following the next_on_package member in the KV_NODE structure.
part_type	The part_type member points to the part type of the physical part.

parent	The parent member is a back-link to the module the package belongs to. The package is part of the definition of the module parent.
hash_link	The hash_link member should not be modified.
reserved value	The reserved and value members are available for your own usage in your particular interface.

KV_PART_TYPE Structure

The KV_PART_TYPE structure describes either a primitive part type or a module in a hierarchical design. A primitive part type must be described in a library file or chips file. A module is an Allegro Design Entry HDL drawing that is translated into a single section part type, with a logical pin and related physical pin for each interface signal. A CAE Views database contains at least one module: the root drawing. The KV_PART_TYPE structure and its members are defined below.

```
typedef struct KV_part_type {
    struct KV_part_type *next; /* next part */
    struct KV_string *name; /* part name */
    struct KV_string *alternate; /* alternate part_type name */
    struct KV_property *properties; /* properties for this part */
    struct KV_body_pin *body_pins; /* all body_pins of part */
    int is_primitive; /* TRUE if part is primitive*/
    int number_sections;/* number of sections */
    int number_pins; /* number of pins on part */
    struct KV_pin *pins; /* all pins on part */
    struct KV_net *net_list; /* pointer to net list */
    struct KV_node *node_list; /* pointer to node list */
    struct KV_package *package_list; /* pointer to package list */
    struct KV_instance *instance_list;/* pointer to instance list */
    struct KV_package *packages; /* first package of type */
    struct KV_instance *instances; /* first instance of type*/
    struct KV_part_type*hash_link; /* do not use */
    VOID*reserved; /* for your own use */
    int value; /* for your own use */
} KV_PART_TYPE;
```

`next`	The list of all KV_PART_TYPE structures is accessed by the global KV_PART_TYPE pointer KV_part_type_list. This list is in a define-before-use order. Primitives are always at the beginning of this list. The next member links all part types together.
`name`	The name member points to the part type name string (for primitives, it is the name as found in the library).
`alternate`	The contents of the alternate member vary in accordance with the CAE Views application being a physical or logical translator.

- **Physical:** the alternate member contains the name assigned by the Packager-XL to the part type.

  If physical part tables were used, alternate can differ from the part type name. Otherwise, name and alternate are the same.

- **Logical:** the alternate member contains the context (parameters) of the module or, in case of primitives, the name of the part type.

If part property tables are used, alternate contains the name generated by CAE Views (similar to Packager-XL names, for example RESISTOR-1).

`properties`	The first property for the part type is accessed by the properties pointer. To get all part type properties, follow the next pointer in the KV_PROPERTY structure.
`body_pins`	The first pin of a body is accessed by the body_pins pointer. To get all pins on a body, follow the next pointer of the KV_BODY_PIN structure.
`is_primitive`	The is_primitive member is set to TRUE (1) if the part type is a primitive. In the case of a module (Allegro Design Entry HDL drawing), it is set to FALSE (0).
`number_sections`	The number of sections in the part type is given by the member number_sections.
`number_pins`	The number of pins, including power pins, is given by the member number_pins.
`pins`	The first pin on the part type is accessed by the pins member. To get all pins, including power pins, follow the next pointer in the KV_PIN structure. A power pin always has the section_number member in the KV_PIN structure equal to zero. In the case of a module, pins points to the list of pins generated by CAE Views.

A module is defined by the following members:

`net_list`	All nets of the module
`node_list`	All nodes of the module
`package_list`	All packages of the module

`instance_list`	All instances of the module In the case of a primitive, the net_list, node_list, package_list, and the instance_list are empty.
`instances`	The first instance of the part type is accessed by the instances pointer. To get all instances of that type, follow the next_same_type pointer on the KV_INSTANCE structure.
`packages`	If the Packager-XL has placed the instances in packages, the first package of the part type is accessed by the packages pointer. To get all packages of that type, follow the next_same_type pointer on the KV_PACKAGE structure.
`hash_link`	Do not modify the hash_link member.
`reserved value`	The reserved and value members are available for your interface specific operations.

KV_PIN Structure

For all pins of a part type, there is an associated KV_PIN structure that describes the physical pin. The KV_PIN structure and its members are defined below.

```
typedef struct KV_pin {
    struct KV_pin *next; /* next pin on part */
    struct KV_string *name; /* pin number */
    struct KV_pin *next_section;/* same pin next section */
    int section_number;/* section number */
    struct KV_part_type *part_type; /* part type it belongs to */
    struct KV_body_pin *body_pin; /* body_pin it belongs to */
    struct KV_pin *reserved; /* for your own use */
    int value; /* for your own use*/
} KV_PIN;
```

`next`	All pins, including power pins, on a part type are pointed to by the pins member of the KV_PART_TYPE structure. The next pin on a KV_PART_TYPE is found by following the next pointer of this KV_PIN structure.
`name`	The name member points to the pin name string. This string contains the (alphanumeric) pin number.
`next_section`	The next physical pin for the same logical pin (given by body_pin pointer) is given by the pointer next_section in the KV_PIN structure.

`section_number`	The section_number member holds the section's number. A value of 0 for this member indicates a power pin.
`part_type`	The part_type pointer is a back-link to the part type this pin belongs to.
`body_pin`	The body_pin pointer is a back-link to the logical pin this pin represents.
`reserved value`	The reserved and value members are available for your interface specific operations.

KV_PROPERTY Structure

Properties appear on almost every object in the design: instances, body pins, part types, nodes, and nets. All properties used in the design have an associated KV_PROPERTY structure. The KV_PROPERTY structure and its members are defined below.

```
typedef struct KV_property {
    struct KV_property *next; /* next property in the list*/
    struct KV_string *name; /* property name */
    struct KV_string *value; /* property value */
 } KV_PROPERTY;
```

`next`	The list of properties on an object is linked together by the next member. For example, if you want all properties on an instance, you start with the properties pointer in the KV_INSTANCE structure and follow the next member of the KV_PROPERTY structure until NULL is reached.
`name`	The name member points to the property name string. This name is always in uppercase letters, due to language specifications.
`value`	The value member points to the property value string.

KV_SCRATCH Structure

The KV_SCRATCH structure has a structure for general use. It uses the optimized memory allocation mechanism. This generic structure contains three pointers and three integer members. Call the *KV_create(scratch)* routine to dynamically create a KV_SCRATCH structure. Call the *KV_kill_scratch()* routine to release a SCRATCH structure. The KV_SCRATCH structure is defined below.

```
typedef struct KV_scratch {
    struct KV_scratch *next; /* a scratch pointer */
    struct KV_string *name; /* a string pointer */
    VOID*other; /* another pointer */
    int value; /* an integer value */
    int type; /* another integer value */
    int number; /* another integer value */
} KV_SCRATCH;
```

KV_SCRATCH members are provided for your use and therefore, do not have a special meaning. You can type cast all six members if you require different types.

KV_STRING Structure

The routine *KV_new_string()* uses the KV_STRING structure to dynamically create a string from the heap. The allocation mechanism assumes that the length of the string is fixed. Once allocated, the string might *not* be lengthened. You might change the contents of the string as long as you don't lengthen it. The KV_STRING structure and its members are defined below.

```
typedef struct KV_string {
    struct KV_string*next; /* do not use */
    int value; /* length of string*/
    char text[STRLEN]; /* content of the string */
} KV_STRING;
```

`next`	The next member is reserved for internal use and should not be changed.
`value`	The value member is initialized to the length of the string (value = strlen(text)). This member is not used by any routine. If you do not change it, it keeps the string length.
`text`	The text member contains the string in normal C format (NULL-terminated).

Since strings are often reused (the PATH property, for example, appears on every instance), all strings are placed in a hash table. This speeds up the search for a string, requires a minimum of memory for string storage, and lets strings to be compared by comparing pointers.

```
    KV_STRING *KV_enter_name(name)
    char *name;
```

This routine searches the string table for the specified name. If not found, *KV_enter_name()* adds the new name to the table and creates a new KV_STRING structure, copies the new string into the structure, and returns a pointer to the new string. If the string is already in the database, this routine only returns a pointer to the existing strings.

4

CAE Views Routines

CAE Views contains routines for you to use in the development of your customized interface. Using the predefined CAE Views routines reduces the number of routines you need to write to implement your application. This chapter groups CAE Views routines into the following categories:

- Input and Output Routines
- Heap Management and Allocation Routines
- Debugging Routines
- Searching Routines
- Sorting Routines
- Time Keeping Routines
- Miscellaneous Routines

Input and Output Routines

Routines in the I/O category control the files that are written to and read from.

void KV_close_file(fp)
FILE *fp;

Closes the file pointed to by `fp`. If CAE Views cannot close the file, it generates an error message. If CAE Views is in display mode, set by the routine *KV_open_file()*, it prints a message to the screen and listing file indicating the time required for processing. *KV_close_file()* resets the elapsed time counter.

int KV_copy_file(s1,s2)
register char *s1, *s2;

Copies the file specified by `s1` to the file specified by `s2`, without printing a message.

int KV_get_arg(fp,separators,arg)
register FILE *fp;
register char *separators;
register char *arg;

Reads the next argument from the specified file (*fp*). Arguments are separated by the characters specified by *separators*. When an argument is found, *KV_get_arg()* sets the value of *arg* equal to the uppercase version of the argument found and returns TRUE.

int KV_get_arg_noup(fp,separators,arg)
register FILE *fp;
register char *separators;
register char *arg;

Reads the next argument from the specified file (*fp*). Arguments are separated by the characters specified by *separators*. When an argument is found, *KV_get_arg_noup()* sets the value of *arg* equal to the value found and returns TRUE.

int KV_get_file_time (file)
char *file;

Retrieves the time the specified file (*file*) was last modified.

int KV_get_token(fp,separators,token)
register FILE *fp;
register char *separators:
register char *token;

Reads the next token from the specified file (*fp*) based on a list of separators (*separators*). *KV_get_token*() sets the value of *token* to the token found, and returns the separator between this token and the next.

The routine *KV_get_token()* bases its interpretation of a token on the following information:

- Ignores comments delimited by the values set by the global variables *KV_start_comment* and *KV_end_comment* (initialized to *{*and*}*).
- Interprets the new line character (\n) as a space unless it is included in the list of separators.
- Interprets the continuation character, as set by the global variable *KV_continue_char* (initialized to ~).
- Interprets tabs as spaces.
- Ignores leading spaces.

- Treats strings in quotes (both single (`'`) and double (`"`) quotes) as a unit.
- If found, sets the global variable *KV_quoted_token* to TRUE.
- Skips trailing spaces and comments if *KV_skip_trailing* is set to TRUE (initial value).

The routine *KV_get_token*() returns EOF if it encounters the end of the file. The current line number in the file is tracked by the global variables KV_to_line and KV_from_line, initialized by the routine *KV_open_file*(). KV_from_line indicates the starting line of the token, KV_to_line indicates the ending line of the token.

FILE *KV_open_file(name,mode,display)
char *name;
char *mode;
int display;

Opens the specified file (*`name`*), using the specified *`mode`* (`r` or `w`), and returns a pointer to the file. The global variable *KV_to_line* is set to 1. If the display mode (*`display`*) is set to TRUE, CAE Views prints a message to the screen and the listing file, both when the file begins processing and when the file is closed (*KV_close_file*()). The closing message also indicates the time required for processing. When CAE Views is in display mode and it cannot open the file, it generates an error message.

You can open only one file at a time using this call. Close the file before opening another file while using this call.

FILE *KV_pi_fopen(filename,style,file_type)
char *filename;
char *style;
int file_type;

Opens the specified file (*`filename`*), using the specified *`style`* (`r` or `w`), and returns a pointer to the file. The argument *`file_type`* indicates whether the file is ASCII (0) or BINARY (1).

int *KV_pi_open(filename,flags,rwmode,file_type)
char *filename;
int flags;
int rwmode;
int file_type;

Opens the specified file (*`filename`*) using the specified *`flags`* (see C function *open*()) and specified *rwmode* (0 for read, 1 for write, and 2 for read and write access). *KV_pi_open*() returns the file description for the opened file or -1 if the routine is unable to open the file. The argument *`file_type`* indicates whether the file is ASCII (0) or BINARY(1).

int KV_pi_mkdir(name,mode)
char*name;
int mode;

Creates the specified directory (`name`) with the specified `mode` (see C function *mkdir*()). The routine *KV_pi_mkdir*() is platform independent. *KV_pi_mkdir()* returns 0 if able to create directory and -1 if unable to create directory.

void KV_print_welcome()

Prints a welcome on the screen and in the listing file. Welcome messages typically include the program name, program version, program date, and copyright message.

void KV_skip_to_next_command(fp)
register FILE *fp;

Skips to next command (determined by global variable KV_end_of_line) in specified file `fp`.

Error Reporting Routines

The CAE Views error reporting routines allow you to define an error condition specific to your application and the corresponding message to be displayed to your users when the error is encountered.

void KV_add_error(type,number,message)
register int type, number;
register char *message;

Adds a new error to the list of errors your CAE Views application might encounter, along with an assessment of the severity of the error (`type`), error number (`number`), and error message (`message`).

Possible values for type include ERROR, OVERSIGHT, and WARNING. Possible values for number include any numbers over 200. `message` is the text of your new error message.

void KV_error(number)
register int number;

Prints the error message (if not suppressed) corresponding to the specified error number (`number`). The *KV_error*() routine also increments the related error counter.

void KV_error_message(str,[arg]...)
char *str;

Prints a formatted error message (if not suppressed) in the listing file and in the debug file (if enabled). This routine acts exactly like the C-function *printf*() and uses most of the format conversion characters available to a *printf*() function.

The first argument in the *KV_error_message*() routine is the control string ("%s%d") which specifies how the remaining arguments in the routine are displayed. The following arguments are variables specified in the control string. If the number of arguments is incompatible, the result is unpredictable. For example,

```
KV_error_message("%s %d\n", arg1, arg2);
```

expects the first argument (*arg1*) to be a string and the second argument (*arg2*) to be an integer. The value displayed for arg2 is in decimal format. Not all *printf*() formats are implemented, only %s, %c, %f, %e, %g, %d, and %n.

Heap Management and Allocation Routines

The routines in the heap management and allocation category allow you to add to the string table and power rail list, to release a scratch element, or to allocate memory for new structures. The routines use the CAE Views allocation method.

KV_SCRATCH *KV_create(type)
int type;

Creates a new CAE Views object and returns a pointer to it. The type of the desired CAE Views object is specified in the parameter type. This parameter is one of the following predefined keywords: INSTANCE_TYPE, NET_TYPE, PART_TYPE_TYPE, PIN_TYPE, PROPERTY_TYPE, BODY_PIN_TYPE, PACKAGE_TYPE, NODE_TYPE, SCRATCH_TYPE. You need to cast the returned pointer to the desired type.

void *KV_delete_scratch(ptr))
register KV_SCRATCH *ptr;

Releases the specified KV_SCRATCH element (*ptr*) to the list of free elements.

KV_STRING *KV_enter_name(name)
register char *name;

Searches the string table for specified string (*name*). If not found, adds the string to the table. Returns a pointer to the string.

KV_BODY_PIN *KV_enter_power_rail(name)
register char *name;

Searches for the specified power rail (`name`) in the power rail list (identified by global variable KV_power_rail_list). If not found, adds the specified power rail to the KV_power_rail_list. Returns a pointer to the power rail.

KV_BODY_PIN *KV_new_body_pin(part,name)
register KV_PART_TYPE * part;
register char *name;

Creates a new KV_BODY_PIN structure and returns a pointer to it. The new body pin is added at the beginning of the list of body pins of the specified part type, by means of the next member. The body pin name (`name`) is filled in. Other members are initialized as follows: subscript, -1; MSB, -1; and LSB, -1.

KV_INSTANCE *KV_new_instance(part,name)
register KV_PART_TYPE *part;
register char *name;

Creates a new KV_INSTANCE structure and returns a pointer to it. The new instance is added to the hash table (KV_instance_table) by means of the instance name (`name`). It is also added at the beginning of the list of instances of the specified part type (`part`) by means of the next member. The instance name (`name`) is filled in. The parent member is set to part. Other members are initialized to NULL or 0.

KV_NET *KV_new_net(part,physical_name,logical_name)
register KV_PART_TYPE *part;
register char *physical_name;
register char *logical_name;

Creates a new KV_NET structure and returns a pointer to it. The new signal is added to the hash table (KV_net_table) by means of the physical signal name (`physical_name`). It is also added to the beginning of the list of signals of the specified part type (`part`) by means of the next member. The signal physical name (`physical_name`) and logical name (`logical_name`) are filled in. The parent member is set to part. Other members are initialized to NULL or 0.

KV_NODE *KV_new_node(part,name)
register KV_PART_TYPE *part;
register char *name;

Creates a new KV_NODE structure and returns a pointer to it. The new node is added at the beginning of the list of nodes of the specified part type (`part`) by means of the next member. The node name (`name`) is filled in. Other members are initialized to NULL or 0.

KV_PACKAGE *KV_new_package(part,name)
register KV_PART_TYPE *part;
register char *name;

Creates a new KV_PACKAGE structure and returns a pointer to it. The new package is added to the hash table (KV_package_table), by means of the package name (`name`). It is also added to the beginning of the list of packages of the specified part type (`part`) by means of the next member. The package name (`name`) is filled in. The parent member is set to part. Other members are initialized to NULL or 0.

KV_PART_TYPE *KV_new_part_type(name)
register char *name;

Creates a new KV_PART_TYPE structure and returns a pointer to it. The new part type is added to the hash table (KV_part_type_table) by means of the part type name (`name`). It is also added to the beginning of the list of part types (KV_part_type_list) by means of the next member. The part type name and alternate are initialized to name. The part type is initialized as a primitive (is_primitive = TRUE). Other members are initialized to NULL or 0.

KV_PIN *KV_new_pin(body_pin,number)
register KV_BODY_PIN *body_pin;
register char *number;

Creates a new KV_PIN structure and returns a pointer to it. If a pin with the specified pin number (`number`) is not yet in the list of pins for this part type (*body_pin->part_type*), the new pin is added at the beginning of the list of pins by means of the next member. If it is found to already exist in the list, the common member in body_pin is updated.

The pin name (`number`) is filled in. The number_pins member in *body_pin->part_type* is incremented. The new pin is linked into the list of physical pins related to the logical pin *body_pin*. The link to the logical pin *body_pin* is made, and the backlink to the part type this pin belongs to (*body_pin->part_type*) is made. Other members are initialized to NULL or 0.

KV_PIN *KV_new_power_pin(part,number)
register KV_PART_TYPE *part;
register char *number;

Creates a new KV_PIN structure and returns a pointer to it. The new pin is added to the beginning of the list of pins of the specified part type (`part`) by means of the next member. The pin name (`number`) is filled in. The number_pins member in part is incremented. The backlink to the part type this pin belongs to (`part`) is made. Other members are initialized to NULL or 0.

KV_BODY_PIN *KV_new_power_rail(name)
register char *name;

Creates a new KV_PIN structure for a power pin and returns a pointer to it. The new power pin is added to the beginning of the list of power pins (KV_power_rail_list) by means of the next member. The power pin name (`name`) is filled in. Other members are initialized to NULL or 0.

KV_PROPERTY *KV_new_property(prop_list,prop_name,prop_value)
register KV_PROPERTY **prop_list;
register char *prop_name, *prop_value;

Creates a new KV_PROPERTY structure and returns a pointer to it if a property with the same name (`prop_name`) is not found in the specified property list (`prop_list`). The new property is added in alphabetical order to the specified property list. The property name (`prop_name`) and the property value (`prop_value`) are filled in.

If a property with the same name (`prop_name`) is already in the property list, its value is updated to the new value prop_value. *KV_new_property*() returns a pointer to the newly created or already existing property.

KV_STRING *KV_new_string(text)
char *text;

Creates a new KV_STRING structure and returns a pointer to it. The specified string (`text`) is copied into the new KV_STRING structure. The string length is stored in the value member. The memory allocated for this string is exactly the space required for storing the string text, an integer (the value member), and the next member. Thus, once allocated, the length of the string must not change.

Debugging Routines

Debugging your program is a very important step in the creation of any application. CAE Views includes a debugging mechanism, as well as routines that write various types of information to a debug file (`interface.dbg`).

A debugging statement takes the form of a test of a global debug array, followed by a call to a routine or a write to the debug file. You can set the debug array elements with directives or command flags.

void KV_debug_translate()

Writes information to the debug file about the translation file status.

void KV_dump_database()

Writes information from the database to the debug file. The output of the routine *KV_dump_database*() includes a list of part types and anything else indicated by the debug flags settings.

void KV_dump_error()

Writes a list of all the errors (with number, severity, and message) to the debug file.

void KV_dump_instances(instance_list)
KV_INSTANCE *instance_list;

Writes the specified list of instances (`KV_instance_list`) to the debug file, including properties and nodes for each instance.

void KV_dump_nets(net_list)
KV_NET *net_list;

Writes the specified list of nets (`KV_net_list`) to the debug file, including properties and nodes for each signal.

void KV_dump_nodes(node_list)
KV_NODE *node_list;

Writes the specified list of nodes (`KV_node_list`) to the debug file, including properties for each node.

void KV_dump_packages(package_list)
KV_PACKAGE *package_list;

Writes the specified list of packages (`KV_package_list`) to the debug file, including instances and nodes for each package.

void KV_dump_power_rails()

Writes the specified list of power rails (`KV_power_rail_list`) to the debug file.

void KV_dump_stat()

Writes CAE Views statistics into the debug file.

void KV_dump_strings()

Writes the list of strings (hash index, string contents) to the debug file.

Searching Routines

Routines in the search category allow you to identify and thus manipulate specific drawings, parts, nets, strings, and properties.

KV_PART_TYPE *KV_find_alternate_part(name)
register char *name;

Searches the part type list (pointed to by the global variable *KV_part_type_list*) for the part type with alternate name specified by `name`. Returns a pointer to the part type if found or NULL if not found. You can set the alternate member in the KV_PART_TYPE structure with the *KV_set_alternate_part*() routine, described under miscellaneous routines. *KV_find_alternate_part*() checks for ambiguity (more than one part type with same alternate name).

KV_INSTANCE *KV_find_instance(part,name)
register KV_PART_TYPE *part;
register char *name;

Searches for the specified instance (`name`) in the definition of the specified `part` (*part->instance_list*). If found, returns pointer to the instance. If not found, returns NULL.

KV_NET *KV_find_net(part,name)
register KV_PART_TYPE *part;
register char *name;

Searches for the specified instance (`name`) in the definition of the specified `part` (*part->net_list*). If found, returns pointer to the signal. If not found, returns NULL.

KV_PACKAGE *KV_find_package(part,name)
register KV_PART_TYPE *part;
register char *name;

Searches for the specified package (`name`) in the definition of the specified `part` (*part->package_list*). If found, returns pointer to the package. If not found, returns NULL.

KV_PART_TYPE *KV_find_part_type(name)
register char *name;

Searches the part type list (*KV_part_type_list*) for the specified part (`name`). If found, returns pointer to the part type. If not found, returns NULL.

KV_PIN *KV_find_pin_on_part(part,pin_number)
register KV_PART_TYPE *part;
register KV_STRING *pin_number;

Searches the specified part type (`part`) for the physical pin with specified pin number (`pin_number`). If found, returns pointer to the pin. If not found, returns NULL.

KV_PROPERTY *KV_find_property(property_list,property_name)
register KV_PROPERTY *property_list;
register KV_STRING *property_name;

Searches the property list (`property_list`) for a property with the specified name (`property_name`). If found, returns a pointer to the property. If not found, returns NULL.

KV_STRING *KV_find_string(name)
register char *name;

Searches the string table for specified string (`name`). If found, returns pointer to the string. If not found, returns NULL.

KV_PROPERTY *KV_find_value(property_list,property_value)
register KV_PROPERTY *property_list;
register KV_STRING *property_value;

Searches the property list (`property_list`) for the specified property value (`property_value`). If found, returns a pointer to the property value. If not found, returns NULL.

Sorting Routines

CAE Views contains routines to sort the database. These routines are based on a quick-sort algorithm and the KV_compare_string().

void KV_sort_instances(part)
register KV_PART_TYPE *part;

Sorts all instances in the definition of the specified part type (`part`) by instance name.

void KV_sort_instances_on_packages(part)
register KV_PART_TYPE *part;

Sorts the instances in each package in the definition of the specified part type (`part`) by instance name.

void KV_sort_instances_on_types()

Sorts the instances of each part type by instance name.

void KV_sort_logical_nets(part)
register KV_PART_TYPE *part;

Sorts all the nets in the definition of the specified part type (`part`) by logical name.

void KV_sort_packages(part)
register KV_PART_TYPE *part;

Sorts all packages in the definition of the specified part type (`part`) by package name.

void KV_sort_packages_on_types()

Sorts the packages of each part type by package name.

void KV_sort_part_alternates()

Sorts all part types by alternate names.

void KV_sort_part_types()

Sorts all part types by part name.

void KV_sort_physical_nets(part)
register KV_PART_TYPE *part;

Sorts all nets in the definition of the specified part type (`part`) by physical name.

void KV_sort_pins_on_instances(part)
register KV_PART_TYPE *part;

Sorts all nodes on all instances in the definition of the specified part type (`part`) by pin number.

void KV_sort_pins_on_packages(part)
register KV_PART_TYPE *part;

Sorts all nodes on all packages in the definition of the specified part type (`part`) by pin number.

Time Keeping Routines

CAE Views contains routines that allow you to track time.

void KV_get_time(string)
char *string;

Returns the date and time in the character string (`string`) (for example, Apr 5 09:20:09 1990).

void KV_print_clk(fp)
FILE *fp;

Writes the elapsed time since the last call to either the *KV_reset_clk()* routine or the *KV_start_clk()* routine to the specified file (*fp*).

void KV_reset_clk()

Resets the clock (also see *KV_print_clk()*.

void KV_start_clk()

The routine *KV_init()* calls *KV_start_clk()* to reset the start time. This routine is not normally called directly by a CAE Views application.

void KV_stop_clk(fp)
FILE *fp;

Prints time information to the specified file (*fp*).

Miscellaneous Routines

CAE Views has other miscellaneous routines that are described below. These routines include general functions such as initializing and exiting your application, reading command line arguments and directives, comparing strings, converting strings into lowercase letters or uppercase letters, and generating cross reference files.

void KV_add_power_pins(part)
register KV_PART_TYPE *part;

Adds all power pin nodes to the definition of the specified module (*part*). CAE Views automatically calls *KV_add_power_pins()* if the global variable *KV_add_power_node* is set to TRUE.

int KV_compare_strings(s1,s2)
register char *s1, *s2;

Compares the two specified strings (*s1* and *s2*) and returns a value greater than zero if *s1* is lexicographically greater than *s2*, returns zero if *s1* is equal to *s2*, and returns a value less than zero if *s1* is lexicographically less than *s2*. This routine takes care of handling numbers (for example, U23 > U3) and subscripts (for example, X<23> > X<3> AND Y[23] > Y[3]).

void KV_create_physical_net_names()

Deletes all current physical signal names and creates new ones (the routine takes care of the hash table).

KV_create_physical_net_names() calls the routines **KV_create_abbreviation()* and **KV_fix_name()*. This routine also updates the interface nodes. The interface nodes and related logical pins of the interface signals are updated to the same new name as generated for the signal.

void KV_exit(number)

Terminates CAE Views. *number* indicates the exit code (0 for success). If the exit code is 0, *KV_exit()* generates cross reference files (if requested), and displays the settings of the different flags in the listing file (see the print functions in the command directive and command line argument tables in Chapter 6, “CAE Views Directives,” and Chapter 7, “Command Line Arguments”).

After a successful exit, CAE Views always prints a program summary, execution time information, and an error summary. If requested, CAE Views also prints debug information. *KV_exit()* is the only CAE Views routine that calls the **C** *routine exit().*

void KV_generate_xref_file()

Generates the cross reference file for the design. Depending on the application, the cross reference file (*interface.xrf*) contains a physical part description, logical part description, physical signal description, logical net description, or a description of the changes found by the compare module.

int KV_hash(name)
register char *name;

Calculates a hash number between 0 and 4095 for the given string (*name*).

int KV_is_unnamed_net(netname)
char *netname;

Tests if syntax of the specified net name (*netname*) matches the syntax of an unnamed signal.

KV_is_unnamed_net() returns TRUE if *netname* matches the following syntax:

```
(1) unnamed_...
(2) (path name)unnamed_..
```

void KV_init(arc,argv)

int argc;

char**argv;

Initializes CAE Views. The routine *KV_init()* starts the clock (*KV_start_clk()*) and initializes all global variables and tables used by CAE Views. It opens the listing file (*interface.lst*). *KV_init()* needs to be the first CAE Views routine called by your CAE Views interface.

void KV_set_hdl_architecture()

Initializes CAE Views to run in HDL-centric environment.

char *KV_lower_case_string(string)
register char *string;

Converts the specified string (*string*) into lowercase letters.

int KV_not_power_net(net)
KV_NET *net;

Tests if the specified net (*net*) is connected to the power rails. *KV_not_power_net()* returns FALSE (or 0) if the specified net is connected to the power net, -1 if the specified net is an NC (not connected) net, or 1 for all other conditions.

void KV_property_to_look_for(name,which)
char *name;
int which;

Adds specified property (*name*) to the list of properties considered by the compare module. *which* indicates where the specified property is found, using predefined values. These values can be combined with the logical OR (|) and include the following choices: INSTANCE_TYPE, NET_TYPE, PART_TYPE_TYPE, and PIN_TYPE.

int KV_read_data_base(top)
register KV_PART_TYPE **top;

Reads the design and sets up the CAE Views database.

For a physical interface (see global variable *KV_physical_interface*), CAE Views reads the Packager-XL files. CAE Views sets up all database elements. Also see the following related global variables

KV_add_power_node	*KV_chip_file*
KV_delta_interface	*KV_skip_exp_file*
KV_xnet_file	*KV_xprt_file*
KV_want_reference	

For a logical interface, CAE Views calls the expander to set up the database (no packages are built). Also see the following related global variables:

KV_replace_one_and_zero
KV_compile_type
KV_expansion_style
KV_replace_globals

int KV_read_directives(arc,argv)
register int arc;
register char **argv;

Reads the directives in the command directive file (*interface.cmd*) and the command line arguments. Generates warning #20 if the command file does not exist. See Chapter 6, “CAE Views Directives,” and Chapter 7, “Command Line Arguments,” for more information on directives and line arguments as well as creating your own tables to process directives and line arguments.

void KV_remove_single_node(part)
KV_PART_TYPE *part;

Removes all single node nets (nets that are attached to only one node) from the database of the cell part. These nodes are added to the NC signal (the signal connects all non-connected pins). CAE Views automatically calls *KV_remove_single_node()* when the global variable *KV_keep_single_nets* is set to FALSE.

void KV_remove_unused_part()

Removes all part types that are not used in the design. CAE Views automatically calls *KV_remove_unused_part()* when the global variable *KV_remove_unused_types* is set to TRUE.

void KV_set_alternate_part(part_name_prop,err_num)
KV_STRING *part_name_prop;
int err_num;

Sets the alternate member in the KV_PART_TYPE structure. For each part type, *KV_set_alternate_part()* searches the part type property list for the specified property (*part_name_prop*). If the property is found, the property value is copied into the alternate member of the KV_PART_TYPE structure. If the property is not found and the error number (*err_num*) is not equal to zero, CAE Views generates the error message with *err_num*.

This routine also checks for ambiguity. Two different part types can have the same alternate name only if the parent part type is the same. Do not use this routine unless there is a one-to-one relationship between alternate and part type.

int KV_spice_to_double(item,real)
char*item;
double*real;

Converts the specified string (*item*) in SPICE format to a double number (*real*). Any mistake in format returns FALSE.

int KV_string_to_int(item,number)
register char *item;
register int *number;

Converts the specified string (*item*) to an integer (*number*). Any mistake in format returns FALSE.

char *KV_upper_case_string(string)
register char *string;

Converts the specified string (*string*) into uppercase letters.

char *KV_translate(s)
char *s;

Returns the translation of the specified string (*s*). If a translation is not found, returns original string. If the translation file (*translation_file*) is specified but not yet read in, it is read first.

void KV_user_exit(number)
register int number;

The value number indicates the type of exit required by your CAE Views application. A number 0 indicates normal (successful) CAE Views exit. A number other than 0 indicates erroneous exit. This routine is called by *KV_exit()*, to allow you to customize the exit for your CAE Views application.

5

CAE Views Global Variables

This chapter describes Global Variable Descriptions used by CAE Views. The variables you use in your interface depend on the type of interface you create.

Global Variable Descriptions

The global variable descriptions that follow are listed in alphabetical order. Each description includes its data structure type, name, default value, and the interface application (physical and logical) where applicable. If a variable can be used in all interfaces, it does not have a special notation.

See Chapter 3, "CAE Views Data Structures," for more information on data types.

Some of the variable types are standard C language data types, such as integer (int) and character (char). Other variable types are CAE Views specific variables, for example,

```
KV_ADDEL_NET   *KV_add_net_list
```

indicates that the variable *KV_add_net_list* is of type KV_ADDEL_NET. Variable types in all capital letters are predefined in CAE Views. Global variables are case-sensitive. The global variable declarations show the appropriate case for the variables.

The default listing in the following descriptions is the CAE Views internal default value for the variable. You can override the default value with command file directives or command line arguments.

Define statements in the file `caeviews.h` assign the symbolic names ON and TRUE to constant 1 and the names OFF and FALSE to the constant 0.

add_power_node

Type:	int
Name:	KV_add_power_node

Default:	FALSE

KV_add_power_node controls whether or not the application adds power nodes to its database after reading the database.

Set *KV_add_power_node* to TRUE to cause the application to add power nodes. Leave the default setting of FALSE if you don't want power nodes in the database.

arg_ptr

Type:	KV_ARG_ENTRY *
Name:	KV_arg_ptr
Default:	KV_arg_table

KV_arg_ptr is a pointer to the argument table. By default, it points to an internal table that contains all possible line arguments interpreted by CAE Views. Set this pointer to the line argument table you want to use.

chip_file

Type:	char
Name:	KV_chip_file[]
Default:	"pstchip.dat"

KV_chip_file[] identifies the chips file and should not normally be changed. This variable is limited to physical interfaces.

command_drawing

Type:	char
Name:	KV_command_drawing[]
Default:	"" (null string)

KV_command_drawing[] contains the name of the root_drawing entered either with the -*c*[ell] line argument or with the DESIGN_NAME directive in the project file.

command_file

Type:	char
Name:	KV_command_file[]
Default:	"" (null string)

KV_command_file[] identifies the name of the project file.

command_fp

Type:	FILE *
Name:	KV_command_fp
Default:	NULL

KV_command_fp points to the command file, *KV_command_file[]*. The routine *KV_read_directives()* opens, reads, and closes the command file.

cmd_ptr

Type:	KV_CMD_ENTRY *
Name:	KV_cmd_ptr
Default:	KV_cmd_table

KV_cmd_ptr is a pointer to the directives table. By default it points to an internal table that contains all possible directives interpreted by CAE Views. Set this pointer to your directives table.

continue_char

Type:	int
Name:	KV_continue_char
Default:	'~'

KV_continue_char defines the character the CAE Views parser interprets as a line continuation character.

copyright

Type:	char
Name:	KV_copyright[128]
Default:	"Copyright (C) 1989 Valid Logic Systems Inc."

KV_copyright[] contains the copyright information. CAE Views uses this string as part of the welcome message. You can write your own message up to 128 characters long.

create_abbreviation

Type:	PFV
Name:	KV_create_abbreviation
Default:	KV_DFT_create_abbreviation

KV_create_abbreviation points (pointer to a function returning void) to the abbreviation routine CAE Views uses to shorten names, for example, signal names.

The default routine:

```
KV_DFT_create_abbreviation(long_name,short_name,length)
```

creates an alphanumeric abbreviation with a specified maximum length of the long_name and puts it in short_name. *KV_create_abbreviation* is automatically called from *KV_create_physical_net_names()* and *KV_add_power_pins()*.

You can write a custom abbreviation routine to generate abbreviations. Set *KV_create_abbreviation* to point to your custom routine.

cross_reference

Type:	int
Name:	KV_cross_reference
Default:	FALSE

KV_cross_reference controls whether or not the application creates an expanded cross reference file (*interface.xrf*) when *KV_exit* function is called for a normal termination.

You can set *KV_cross_reference* to TRUE, with the **-***x*[ref] line argument or the *CROSS_REFERENCE ON* directive in the command file.

current_file

Type:	char
Name:	KV_current_file[]
Default:	*" "* (null string)

KV_current_file[] identifies the name of the last file opened with *KV_open_file()*.

debug

Type:	char
Name:	KV_debug[]
Default:	char KV_debug [DEBMAX]

KV_debug[] identifies the array of debug flags. Set a debug flag by using the *DEBUG x* directive in the command file (where *x* = debug number). You can also set a debug flag using the *-d x* command line argument.

debug_file

Type:	char
Name:	KV_debug_file[]
Default:	*“interface.dbg”*

KV_debug_file[] identifies the name of the debug file.

debug_fp

Type:	FILE *
Name:	KV_debug_fp
Default:	NULL

KV_debug_fp points to the debug file, *KV_debug_file[]*. The debug file is opened whenever a debug flag is turned on. You can then use the *KV_debug_fp* to write a message to the debug file.

debug_on

Type:	int
Name:	KV_debug_on
Default:	FALSE

KV_debug_on is set to TRUE when one or more debug flags have been turned on.

end_comment

Type:	char
Name:	KV_end_comment
Default:	*‘}’*

KV_end_comment defines the character the CAE Views parser interprets as the end of a comment. You may nest comments as long you define the start-of-comment and end-of-comment characters as different characters.

end_of_param

Type:	int
Name:	KV_end_of_param
Default:	`';'`

KV_end_of_param defines the character that *KV_get_arg()* or *KV_get_arg_noup()* interprets as the end of a parameter list.

error_count

Type:	int
Name:	KV_error_count
Default:	0

KV_error_count serves as a counter to track the number of errors encountered during the execution of the interface.

expansion_style

Type:	int
Name:	KV_expansion_style
Default:	FLAT

KV_expansion_style controls whether the CAE Views application generates a flat or hierarchical database. This variable is limited to logical interfaces.

Set *KV_expansion_style* to HIERARCHICAL to generate a database that reflects the hierarchy defined in the design, and leave the default setting of FLAT to generate a flat database.

fatal_error

Type:	int
Name:	KV_fatal_error
Default:	FALSE

When set to TRUE, *KV_fatal_error* reflects the presence of a fatal error. You are encouraged to set *KV_fatal_error* to TRUE when a fatal error is encountered.

from_line

Type:	int
Name:	KV_from_line
Default:	1

The CAE Views parser sets *KV_from_line* to the line number where it finds the first character of the current token.

header_file

Type:	char
Name:	KV_header_file
Default:	"" (null string)

KV_header_file identifies the name of the header file. The interface (not CAE Views) reads and/or writes this file as a header file.

Set the *KV_header_file* string with the HEADER_FILE directive in the command file.

instance_count

Type:	int

Name:	KV_instance_count
Default:	0

KV_instance_count serves as a counter to track the number of allocated instance structures.

keep_single_nets

Type:	int
Name:	KV_keep_single_nets
Default:	TRUE

KV_keep_single_nets controls whether or not the CAE Views application removes all single node nets (nets connected to only one node).

Set *KV_keep_single_nets* to FALSE using the *SINGLE_NODE_NETS OFF* directive to cause the CAE Views application to remove all single node nets. Leave the default setting at TRUE if you do not want the CAE Views application to remove the single node nets.

list_file

Type:	char
Name:	KV_list_file[]
Default:	*"interface.lst"*

KV_list_file[] identifies the name of the listing file. The listing file contains the welcome message, program summary, the global settings, error summary, and execution time information.

list_fp

Type:	FILE *
Name:	KV_list_fp

Default:	NULL

KV_list_fp points to the listing file, *KV_list_file[]*. The listing file contains the welcome message, program summary, the global settings, error summary, and the execution time information. A CAE Views application always leaves the listing file open.

log_fp

Type:	FILE *
Name:	KV_log_fp
Default:	stout

KV_log_fp points to the log file. The screen serves as the standard output for the log file.

max_error

Type:	int
Name:	KV_max_error
Default:	500

KV_max_error sets the maximum number of errors allowed before CAE Views terminates. You can modify this number with the MAX_ERROR *number* directive in the command file.

net_count

Type:	int
Name:	KV_net_count
Default:	0

KV_net_count serves as a counter to track the number of allocated net structures.

net_length

Type:	int
Name:	KV_net_length
Default:	24

KV_net_length sets the maximum length of signal names. This variable is used by *KV_create_abbreviation()* and *KV_fix_name()* functions.

The application user can modify the default with the *NET_NAME_LENGTH* directive in the command file.

node_count

Type:	int
Name:	KV_node_count
Default:	0

KV_node_count serves as a counter to track the number of allocated node structures.

output_dir

Type:	char
Name:	KV_output_dir
Default:	"" (null string)

KV_output_dir identifies the directory where files opened by CAE Views are written. By default, these files are written in the packaged view. This variable is limited to physical interfaces in HDL-centric environment.

oversight_count

Type:	int
Name:	KV_oversight_count
Default:	0

KV_oversight_count serves as a counter to track the number of oversights encountered during the application run.

oversight_output

Type:	int
Name:	KV_oversight_output
Default:	TRUE

KV_oversight_output controls whether or not the CAE Views application displays oversight messages.

Set *KV_oversight_output* to FALSE by using the OVERSIGHTS OFF directive to cause the application to turn off the display of oversight messages. Leave the default setting of TRUE if you want the application to display oversight messages. In either case, CAE Views counts the oversight messages and prints the count in the error summary.

package_count

Type:	int
Name:	KV_package_count
Default:	0

KV_package_count serves as a counter to track the number of allocated package structures.

package_routine

Type:	PFV
Name:	KV_package_routine
Default:	NULL

KV_package_routine serves to package a logical application. Set this variable to the name of the function that packages the design. This function is then called after reading the logical database. There is no default packaging function. This variable is limited to logical interfaces.

package_time

Type:	char
Name:	KV_package_time[]
Default:	"" (null string)

KV_package_time[] identifies the time the design was packaged.

part_type_list

Type:	KV_PART_TYPE *
Name:	KV_part_type_list
Default:	NULL

KV_part_type_list points to the first entry in the list of part types. This list contains primitives (physical parts described in a library or chips file) and modules. Primitives always precede modules. The part type list orders modules bottom up (define before use) with the "top drawing" module always the last part type in the list.

physical_interface

Type:	int

Name:	KV_physical_interface
Default:	TRUE

KV_physical_interface controls whether CAE Views creates its database by reading the Packager-XL output files or by calling the expander.

The directive INTERFACE_TYPE PHYSICAL sets the *KV_physical_interface* variable to TRUE (the default value), causing CAE Views to create a database by reading the Packager-XL files. The directive INTERFACE_TYPE LOGICAL sets *KV_physical_interface* to FALSE causing CAE Views to create a database by invoking the expander. CAE Views does not include packages in a database created by reading the database.

power_rail_list

Type:	KV_BODY_PIN *
Name:	KV_power_rail_list
Default:	NULL

KV_power_rail_list points to the first entry in the list of power rails in the design.

prog_root_name

Type:	char
Name:	KV_prog_root_name[]
Default:	"" (null string)

CAE Views sets the variable *KV_prog_root_name[]* at the beginning of the program. After initialization, this variable contains the base name of the program. The base name of the program is the name of the executable without the extension. For example, if your executable is named */usr/valid/tools/interfaces/vhdl.ex, KV_prog_root_name[]* is set to *vhdl*.

pst_dir

Type:	char

Name:	KV_pst_dir
Default:	"" (null string)

CAE Views sets the variable KV_pst_dir to the directory location from where the packager files are to be picked. This variable overrides the default location from where pst*.dat files are read in the packaged view. The variable is limited to physical interfaces in HDL-centric environment.

quoted_token

Type:	int
Name:	KV_quoted_token
Default:	FALSE

The CAE Views parser sets *KV_quoted_token* to TRUE when it encounters a token enclosed in either single (′) or double (″) quotes.

read_logical_db

Type:	PFI
Name:	KV_read_logical_db
Default:	NULL

If you are writing a logical application, set this variable to *KV_build_in_logical_db*. (See also *KV_interface_type*).

read_pstcmdb

Type:	int
Name:	KV_read_pstcmdb
Default:	FALSE

This is valid for physical interface. *KV_read_pstcmdb* controls whether or not the CAE Views application reads the pin and net constraints from the `pstcmdb.dat` file.

To allow reading of constraint information from the `pstcmd.dat` file, before a call to the *KV_read_data_base* routine, set *KV_read_pstcmdb* to TRUE.

real_file_name

Type:	char
Name:	KV_real_file_name
Default:	NULL

This variable contains the real name of the last file opened with *KV_open_file*. This variable can be different from *KV_current_file*.

remove_unused_types

Type:	int
Name:	KV_remove_unused_types
Default:	TRUE

KV_remove_unused_types controls whether or not the CAE Views application removes unused part types from the list.

Set *KV_remove_unused_types* to FALSE to cause the application to leave unused part types in the *KV_part_type_list*. Leave the default setting of TRUE to cause the application to remove unused part types from the *KV_part_type_list*.

replace_globals

Type:	int
Name:	KV_replace_globals
Default:	TRUE

KV_replace_globals controls whether CAE Views translates global signals in a hierarchical design bottom-up into interface signals. This variable is limited to logical interfaces.

Leave *KV_replace_globals* set to the default TRUE to cause CAE Views to translate global signals into interface signals. Set *KV_replace_globals* to FALSE to leave the global signals unresolved.

replace_one_and_zero

Type:	int
Name:	KV_replace_one_and_zero
Default:	TRUE

KV_replace_one_and_zero controls whether CAE Views bottom-up translates global signals, 1 and 0, into interface signals. This variable is limited to logical interfaces.

Leave *KV_replace_one_and_zero* set to the default TRUE to cause CAE Views to bottom-up translate global signals 1 and 0 into interface signals. Set *KV_replace_one_and_zero* to FALSE to leave the global signals 1 and 0 unresolved.

replicate_prim

Type:	int
Name:	KV_replicate_prim
Default:	TRUE

KV_replicate_prim controls whether CAE Views replicates primitives or not. If you set this variable to FALSE, primitives are not automatically sized.

root_drawing

Type:	char
Name:	KV_root_drawing[]
Default:	“” (null string)

KV_root_drawing [] identifies the name of the "root drawing." If you are using a logical interface, give this name as a command line argument:

```
     -c[ell] name or as ROOT_DRAWING name
```

in the command file. In a physical application, this name is found in the packager output files read by CAE Views.

run_time

Type:	char
Name:	KV_run_time
Default:	"" (null string)

When invoked, a CAE Views application sets *KV_run_time* to show the start time for the application.

scale_file

Type:	char
Name:	KV_scale_file[]
Default:	"" (null string)

KV_scale_file[] identifies the scale factor file. This variable is limited to logical interfaces.

sep

Type:	char
Name:	KV_sep
Default:	",; ="

KV_sep is a string of separators used during command file parsing. If you want to modify the separators list, do it before you call *KV_read_directive()*. See also *KV_end_of_param* variable.

skip_exp_file

Type:	int
Name:	KV_skip_exp_file
Default:	FALSE

KV_skip_exp_file controls whether or not physical CAE Views reads the Packager-XL expansion files *pstxnet.dat* and *pstxprt.dat*. If your application needs the information in these files, leave the default setting to cause CAE Views to read these files. If your application does not need the information in these files, change the setting of this variable to TRUE to speed up read time. This variable is limited to physical interfaces.

skip_trailing

Type:	char
Name:	KV_skip_trailing
Default:	TRUE

KV_skip_trailing controls whether or not the CAE Views parser skips all trailing blanks and comments after reading the current token.

Leave the default setting of TRUE to cause the parser to skip all trailing blanks and comments after reading the current token.

source_tool

Type:	char
Name:	KV_source_tool
Default:	"" (null string)

KV_source_tool stores the value of the *SOURCE_TOOL* property, which is set by Packager-XL in the *pstxprt.dat* file. This variable is used only by external applications. Do not modify it. The variable is limited to physical interfaces in HDL-centric environment.

start_comment

Type:	char
Name:	KV_start_comment
Default:	`'{'`

KV_start_comment defines the character the CAE Views parser interprets as the start of a comment character. You might nest comments as long you define the start-of-comment and end-of-comment characters as different characters.

time

Type:	char
Name:	KV_time[]
Default:	`""` (null string)

This time is usually kept in a state file.

to_line

Type:	int
Name:	KV_to_line
Default:	1

The CAE Views parser sets *KV_to_line* to the line number where it found the last character of the current token.

warning_count

Type:	int
Name:	KV_warning_count

Default:	0

KV_warning_count serves as a counter to track the number of warnings encountered during the application run.

warning_output

Type:	int
Name:	KV_warning_output
Default:	TRUE

KV_warning_output controls whether or not the CAE Views application displays warning messages.

Set *KV_warning_output* to FALSE by using the WARNINGS OFF directive to cause the application to turn off the display of warning messages. Leave the default setting of TRUE if you want the application to display warning messages. If you change the setting of *KV_warning_output* to false, CAE Views does not count warnings, and the warning count in the error summary remains zero.

xnet_file

Type:	char
Name:	KV_xnet_file[]
Default:	"pstxnet.dat"

KV_xnet_file[] identifies the name of the Packager-XL expanded netlist file. This variable is limited to physical interfaces.

xprt_file

Type:	char
Name:	KV_xprt_file[]

Default:	"pstxprt.dat"

KV_xprt_file[] identifies the name of Packager-XL expanded part list file. This variable is limited to physical interfaces.

xref_file

Type:	char
Name:	KV_xref_file[]
Default:	"*interface.xref*"

KV_xref_file[] identifies the name of the expanded cross reference file. If you include the CROSS_REFERENCE ON directive in the command file or if the application user invokes the CAE Views application with the *-x*[ref] line argument, CAE Views generates a cross reference file.

6

CAE Views Directives

This chapter describes the following:

- Overview
- Directives Table
- Directive Descriptions

Overview

Interfaces developed with CAE Views use a *directives table* to specify the directives available for your particular application. A template directives table is included in the `phys.c` file. This table makes it convenient for you to decide which directives to make available to the users of your interface.

This chapter describes how to define your own directives and how to add them to the template directives table or how to create your own directives table. The chapter also describes CAE Views predefined directives and how directives are used to override your program defaults.

The CAE Views directives table is a template table located in `phys.c` and referenced in `phys.c`. You use this table as a starting point to select which of the CAE Views predefined directives are used or to define your own directives.

The CAE Views directives table is accessed by a pointer (*KV_cmd_ptr*) in the main program. If you define your own custom directives table, you must set *KV_cmd_ptr* in your main program to point to your custom table. For example, if you create “my_cmd_table”, add the following line to your main program before the *KV_read_directives()* call

```
    KV_cmd_ptr = my_cmd_table;
```

and change the external structure reference to read:

```
    extern KV_CMD_ENTRY my_cmd_table[];
```

Directives Table

The directives table is an array of KV_CMD_ENTRY structures. A KV_CMD_ENTRY structure is defined as follows:

```
typedef struct KV_command_entry {
    char command[50];/* name of the command */
    void (*read_command)();/* function pointer to read the command */
    void (*print_command)();/* function pointer to print the command */
    int *value;/* pointer to the value to pass */
} KV_CMD_ENTRY;
```

In the structure
:

`command`	Name of the directive you want to process. Your command name can have a maximum of 49 characters.
`read_command`	A pointer to the function called when the directive is found in the directives file. When the function is called, two parameters are passed: A pointer to a buffer containing the directive name A pointer to a variable set by the *read_command*. You provide the address of this variable in the last argument (the value member) of the KV_CMD_ENTRY structure.
`print_command`	A pointer to a function that is called at the end of the program to display the “value” of the directives. When the function is called, three parameters are passed: A pointer to a stream file. A pointer to a buffer containing the directive name. A pointer to a variable you want to display. You provide the address of this variable in the last argument (the value member) of the KV_CMD_ENTRY structure. If you don’t want your program to display the value of a directive at the end of the program, include a NULL pointer for the print_command.
`value`	A pointer to a variable used by your read_command and print_command functions. This variable is not mandatory, and you can set it to NULL if you don’t need it.

Make the last entry of the directives table as:

```
“” , NULL, NULL, NULL
```

to indicate the end of the table.

Predefined Directives

CAE Views provides a set of predefined directives. To make these predefined directives available to your users, you must keep the original entry in directives table provided in the *phys.c* file. The following directives are predefined:

- *CHECK_PIN_NAMES*
- *CROSS_REFERENCE*
- *OVERSIGHTS*
- *PART_TABLE_FILE*
- *EXPANSION_STYLE*
- *REPLACE_CHECK*
- *SINGLE_NODE_NETS*
- *MAX_ERRORS*
- *DEBUG*
- *PROPERTY*
- *LIBRARY*
- *HEADER_FILE*
- *ROOT_DESIGN*
- *SUPPRESS*
- *WARNINGS*

The meaning and usage of these directives are described later in this chapter.

Read Directives Functions

CAE Views contains a set of both reserved and general-purpose *read_command* functions. Some of these functions are reserved for predefined directives; use them only with these directives. Some other functions are generic, and you can use them to interpret your specific directives.

Reserved Read Directive Functions

The following *read_command* functions are reserved for specific predefined directives:

- *KV_cmd_dir()*
- *KV_cmd_type()*
- *KV_cmd_lfile()*
- *KV_cmd_ppt()*
- *KV_cmd_suprs()*
- *KV_cmd_style()*
- *KV_cmd_lib()*
- *KV_cmd_master()*
- *KV_cmd_prop()*

General-Purpose Read Directive Functions

The following functions are available for general use in creating your own directives table:

`KV_cmd_str()`	This function reads a string from a directive. The string following the directive is copied into the buffer pointed to by the value argument. For example, if you have the following entry in the directives table: "MY_DIRECTIVE", KV_cmd_str, KV_pr_str, (int *)my_buffer and you put the following directive in the directives file: MY_DIRECTIVE a_string; CAE Views copies `a_string` into `my_buffer`. **Note:** Cast `my_buffer` as an int pointer.

`KV_cmd_flag()`	This function reads a flag from a directive. If the string following the directive is ON, the flag is set to TRUE. If the string following the directive is OFF, the flag is set to FALSE. A value other than *ON* or *OFF* is detected as an error (ERROR 8) and sets the variable *`KV_fatal_error`*. For example, if you have the following entry in the directives table “MY_DIRECTIVE”, KV_cmd_flag, KV_pr_flag, &my_flag and you include the following directive in the directives file MY_DIRECTIVE ON; CAE Views sets “my_flag” to TRUE. Similarly, if you include the following directive in the directives file: MY_DIRECTIVE OFF; CAE Views sets “my_flag” to FALSE. If you put the following directive in the directives file: MY_DIRECTIVE FOO; CAE Views prints an error message and sets *KV_fatal_error* to TRUE.
`KV_cmd_int()`	This function reads an integer from a directive. If the string following the directive is a valid number, the integer is set to this value. If the number is not legal, CAE Views prints an error message and sets *KV_fatal_error* to *TRUE*. For example, if you have the following entry in the directives table: “MY_DIRECTIVE”, KV_cmd_int, KV_pr_int, &my_int and you include the following directive in the directives file: MY_DIRECTIVE 3; CAE Views sets “my_int” to 3. If you put the following directive in the directives file: MY_DIRECTIVE FOO; CAE Views reports an error and sets *KV_fatal_error* to TRUE.

User-Defined Read Directive Functions

If you don’t want to use a generic read directive function, you can write your own read command function. This function expects two parameters and returns void. The first parameter is a pointer to the directive string, and the second parameter is a pointer to the variable passed from the directives table.

As an example, assume you want to read a directive *“MY_DIRECTIVE”* that expects an argument with either a TRUE or FALSE value. You want the variable “*my_var*” to be set to TRUE if the argument is TRUE, and set to FALSE if the argument is FALSE.

To implement the read function

➤ Add the following line to the directives table

```
“MY_DIRECTIVE”, my_read_func, my_print_func, &my_var
```

and write code to read this directive. The following is an example of the required code.

```
char token[100];

void my_read_func(command, value)
     char *command;
     int *value;
     {

if (KV_get_arg(KV_command_fp,KV_sep,token) == FALSE)
       KV_parse_error(4,strcat(command,

               XR(” command expects an argument”)));
     else {
       if (strcmp(token,”TRUE”) == SAME) *value = TRUE;
       else if (strcmp(token,”FALSE”) == SAME) *value = FALSE;
         else KV_parse_error(8,strcat(command, XR(” Invalid option”)));
        } /* found an argument */

}
```

Print Directive Functions

CAE Views has a list of predefined print command functions. Some of these functions are specific to predefined directives and should only be used with these directives; some other functions are generic; you can use them to print your specific directives.

Reserved Print Directive Functions

The following print commands are reserved for specific predefined directives:

- *KV_pr_dir()*

- *KV_pr_style()*
- *KV_cmd_dir()*
- *KV_pr_lib()*
- *KV_pr_master()*
- *KV_pr_prop()*
- *KV_pr_debug()*
- *KV_cmd_dir()*
- *KV_pr_type()*
- *KV_pr_lfile()*
- *KV_pr_ppt()*
- *KV_pr_suprs()*

General-Purpose Print Directive Functions

Use the following functions to print your directives:

`KV_pr_str()`	This function prints a string from a directive. The string pointed to by the value argument is printed. For example, if you include the following entry in the directives table: “MY_DIRECTIVE”, KV_cmd_str, KV_pr_str, (int *)my_buffer and “my_buffer” contains “a_string”, CAE Views prints the following line in the output file: MY_DIRECTIVE a_string; **Note:** Cast “my_buffer” to an int pointer.

`KV_pr_flag()`	This function prints a flag from a directive. If the flag is TRUE, then ON is printed. If the flag is FALSE, then OFF is printed. For example, if you have the following entry in the directives table: “MY_DIRECTIVE”, KV_cmd_str, KV_pr_str, (int *)my_buffer and “my_buffer” contains “a_string”, CAE Views prints the following line in the output file: MY_DIRECTIVE a_string; **Note:** Cast “my_buffer” to an int pointer.
`KV_pr_int()`	This function prints an integer from a directive. CAE Views prints the directive followed by the value of the integer. For example, if you have the following entry in the directives table: “MY_DIRECTIVE”, KV_cmd_int, KV_pr_int, &my_int and my_int equals 3, CAE Views prints the following line in the output file: MY_DIRECTIVE 3

User-Defined Print Directive Functions

If you don’t want to use the default print directive, you can write your own print command. This function returns void and expects three parameters. The first parameter is a pointer to a stream file, the second parameter is a pointer to a buffer containing the directive string, and the third parameter is a pointer to the variable passed from the directives table.

As an example, assume you want to print a directive `“MY_DIRECTIVE”` that expects an argument with either a TRUE or FALSE value. You want to print TRUE, if the argument is TRUE, and print FALSE if the argument is FALSE.

To implement the print function, add the following line to the directives table

```
“MY_DIRECTIVE”, my_read_func, my_print_func, &my_var
```

and write code to read this directive. The following is an example of the required code.

```
void my_print_func(fp, command, value)
FILE *fp;
char *command;
int *value;
{

if (*value)
    (void)fprintf(fp,” %s TRUE\n,command);
else
    (void)fprintf(fp,” %s FALSE\n,command);
    }
```

The Template Directives Table

The *phys.c* file contains the template directives table where you to add or remove entries as necessary depending on the application you are developing. Figure 6-1 on page 89 shows the default template table.

You modify this table for your application. Just remove the lines for the predefined commands that you don't want, and add extra lines for your application-specific commands.

Some general comments on adding a directive to the table are as follows:

- Enclose the directive (command name) in double quotes
- Include a comma after each field entry
- Use tabs or spaces between columns for legibility

Note: Cadence recommends that you put the directives in alphabetical order, because this is the order used to print the directives at the end of the program. Also, if you have an alternate name for a directive set the print command pointer to NULL for the alternate names to eliminate printing the same directive several times. For example:

```
    “DIRECTORY”, KV_cmd_dir, KV_pr_dir, NULL,
    “DIR”, KV_cmd_dir, NULL, NULL,
    “USE”, KV_cmd_dir, NULL, NULL,
```

Figure 6-1 Default user_cmd_table

```
KV_CMD_ENTRY user_cmd_table[] = {
/*
COMMAND NAMEREAD FUNCTPRINT FUNCTVARIABLE POINTER
*/
”COMPILE”,KV_cmd_str,KV_pr_str,(int *)KV_compile_type,
”CHECK_PIN_NAMES”,KV_cmd_flag KV_pr_flag,&KV_check_pin_names,
”CROSS_REFERENCE”,KV_cmd_flag,KV_pr_flag,&KV_cross_reference,

”DEBUG”,NULL,KV_pr_debug, NULL,

“DESIGN_LIBRARY”, NULL, KV_cmd_str,(int *)KV_hdl_lib,
“DESIGN_NAME”, KV_cmd_str, NULL,(int *)KV_hdl_cell,
”DIR”, KV_cmd_dir,NULL,NULL,
”DIRECTORY”,KV_cmd_dir,KV_pr_dir,NULL,
”EXPANSION_STYLE”KV_cmd_style, KV_pr_style, NULL,
”HEADER_FILE”,KV_cmd_str,  KV_pr_str,  int *)KV_header_file,
”INTERFACE_TYPE”KV_cmd_type,  KV_pr_type,  NULL,
”LIBRARY”,KV_cmd_lib,  KV_pr_lib,  NULL,
”LIB”,KV_cmd_lib, NULL,     NULL,
”LIBRARY_FILE” KV_cmd_lfile, KV_pr_lfile, NULL,
”MASTER_LIBRARY,KV_cmd_master,KV_pr_master,NULL,
”MAX_ERRORS”,KV_cmd_int,  KV_pr_int,  &KV_max_error,
”MERGE_MINIMUM”,KV_cmd_int,  KV_pr_int,  &KV_merge_min,
”NET_NAME_LENGTH”,KV_cmd_int,  KV_pr_int,  &KV_net_length,
”OVERSIGHTS”,KV_cmd_flag,KV_pr_flag,&KV_oversight_output,
”PART_TABLE_FILE”,KV_cmd_ppt,KV_pr_ppt,NULL,
”PROPERTY”,KV_cmd_prop,KV_pr_prop,NULL,
”ROOT_DRAWING”,KV_cmd_str,NULL,(int*)KV_command_drawing,
”REPLACE_CHECK”,KV_cmd_flag,KV_pr_flag,&KV_replace_check,
”SCALE_FACTOR_FILE”,KV_cmd_str,KV_pr_str,(int *)KV_scale_file,
”SINGLE_NODE_NETS”,KV_cmd_flag,KV_pr_flag,&KV_keep_single_nets,
”SPLIT_MINIMUM”,KV_cmd_int,KV_pr_int,&KV_split_min,
”SUPPRESS”,KV_cmd_suprs,KV_pr_suprs,NULL,
”USE”, KV_cmd_dir,  NULL,     NULL,
“VIEW_CONFIG”, NULL, KV_cmd_str,(int *)KV_hdl_view,
”WARNINGS”,KV_cmd_flag,KV_pr_flag,&KV_warning_output,
””, NULL,NULL,NULL
};
```

Directive Descriptions

This section describes the predefined CAE Views directives. The following format conventions are used to describe the directives

text	Items to be entered and variable arguments are shown in italics.
{}	Curly braces enclose choices that you pick from a list of specific entries. The curly braces are for document clarity only and are not entered as part of the command.
...	Ellipses show that you can repeat information as necessary.
[]	Optional directive arguments are enclosed in square brackets (for example, [*filename*]). The square brackets are for document clarity only and are not entered as part of the command.
	The directives and directive arguments are not case-sensitive. The directive descriptions that follow use all uppercase letters for both directives and directive arguments.

When you refer to the file *caeviews.h* (#include) in your program, the symbolic names *ON* and *TRUE* are assigned the program constant 1 and the names *OFF* and *FALSE* are assigned the program constant 0.

The directive descriptions that follow include the syntax and the global variable (if any) modified by the directive. The directives that specify libraries and files do not affect global variables, so there is no default value for them. Other directives such as *PROPERTY* and *PART_TABLE_FILE* do not have a corresponding global variable.

CHECK_PIN_ NAMES

The *CHECK_PIN_NAMES* directive controls whether the compare module compares pin names (as well as the number of pins and number of sections in a package) before replacing one package with another.

The variable affected by the *CHECK_PIN_NAMES* directive is *KV_check_pin_names*. The syntax for the *CHECK_PIN_NAMES* directive is as follows:

Syntax

```
CHECK_PIN_NAMES {ON | OFF} ;
```

DEBUG

The *DEBUG* directive turns on (by number) a debug flag in CAE Views.

The variable affected by the *DEBUG* directive is *KV_debug* [*number*]. The syntax for the *DEBUG* directive is as follows:

Syntax

```
DEBUG number ;
```

Example

If, for example, you set the *DEBUG* directive to

```
DEBUG 100;
```

debug flag 100 is specifically requested.

DESIGN_NAME

The *DESIGN_NAME* directive specifies the name of the root design. This directive is placed in the global section of a project file.

Syntax

```
DESIGN_NAME‘design_name’
```

DESIGN_LIBRARY

The *DESIGN_LIBBRARY* directive specifies the name of the library, which contains the root design. This directive is usually placed in the global section of a project file.

Syntax

```
DESIGN_LIBRARY‘library_name’
```

HEADER_FILE

If you create an application that uses a header file, specifying a file with the *HEADER_FILE* directive sets the value of the global variable *KV_header_file*. The syntax is as follows:

Syntax

```
HEADER_FILE ’filename’ ;
```

Example

The content of the specified file (*filename*) is used as the header for the interface’s output files. For example, the directive

```
HEADER_FILE ’top.txt’;
```

causes the CAE Views application to use the contents of the file *top.txt* as the header for its output files.

MAX_ERRORS

The *MAX_ERRORS* directive sets the value of the global variable *KV_max_error*. The syntax for the *MAX_ERRORS* directive is as follows:

Syntax

```
MAX_ERRORS number ;
```

The *MAX_ERRORS* directive specifies the number of errors allowed before the CAE Views application terminates. If the maximum number of allowed errors is reached, the CAE Views application terminates program execution after printing a summary of the execution.

NET_NAME_LENGTH

The *NET_NAME_LENGTH* directive limits the length of a name assigned by CAE Views to a physical net. You can always use this directive in logical CAE Views, you can also use it in physical CAE Views if the net names are reassigned. If the maximum length is too short, the CAE Views application is unable to generate unique names for all nets and prints an error message.

The variable affected by the *NET_NAME_LENGTH* directive is *KV_net_length*. The syntax for the *NET_NAME_LENGTH* directive is as follows:

Syntax

```
NET_NAME_LENGTH length ;
```

where *length* is the maximum number of characters allowed for a physical net name.

OVERSIGHTS

The *OVERSIGHTS* directive sets the value of the global variable *KV_oversight_output* to *TRUE* or *FALSE*. The syntax for the OVERSIGHTS directive is as follows:

Syntax

```
OVERSIGHTS {ON | OFF} ;
```

With the *OVERSIGHTS* directive set to *ON*, the global variable *KV_oversight_output* is set to *TRUE* and oversight messages are displayed. With the *OVERSIGHTS* directive set to OFF, the global variable *KV_oversight_output* is set to *FALSE* and oversight messages are not displayed.

Errors categorized as *oversights* should be corrected, but the design might work without fixing them. Since the total number of oversights detected is always reported following interface execution, displaying oversight messages is a matter of preference. You can use the *SUPPRESS* to turn off specific oversight messages by number.

REPLACE_CHECK

The *REPLACE_CHECK* directive controls whether the compare module checks for replaceable packages. This directive sets *KV_replace_check* to *TRUE* or *FALSE*.

Syntax

```
REPLACE_CHECK {ON | OFF} ;
```

With the *REPLACE_CHECK* directive set to *ON*, the compare module checks for replaceable packages. With the *REPLACE_CHECK* directive set to *OFF*, the compare module translates replaced packages as delete and add package operations. Also see the *CHECK_PIN_NAMES* directive.

SINGLE_NODE_NETS

The *SINGLE_NODE_NETS* directive sets the value of the global variable *KV_keep_single_nets* to *TRUE* or *FALSE*.

The syntax for the SINGLE_NODE_NETS directive is as follows:

Syntax

```
SINGLE_NODE_NETS  {ON | OFF} ;
```

With the SINGLE_NODE_NETS directive set to *ON*, the global variable *KV_keep_single_nets* is set to *TRUE*, and nets with only one node are left as single node nets.

With the *SINGLE_NODE_NETS* directive set to *OFF*, the global variable *KV_keep_single_nets* is set to *FALSE*, and all single node nets are removed.

SUPPRESS

The *SUPPRESS* directive controls whether specific error messages are displayed. Errors detected by a CAE Views application are categorized according to the severity of the condition encountered. Depending on the particular CAE Views application, errors categorized as *warnings* or *oversights* may occur in a good design. The *SUPPRESS* directive allows an application to selectively choose not to display messages associated with errors of less importance to a particular CAE Views application.

The total number of warnings, oversights, and errors is reported following interface execution. Only warnings and oversights can be suppressed. A suppressed warning is not counted in the total of warnings reported. A suppressed oversight is counted in the total number of oversights. Thus the CAE Views application user can still be aware of oversights, even if the associated message is not displayed.

The syntax for the *SUPPRESS* directive is as follows:

Syntax

```
SUPPRESS number [, number]…;
```

where *number* represents a specific error number. See Appendix A, "CAE Views Error Messages," for a description of error messages.

VIEW_CONFIG

The *VIEW_CONFIG* directive specifies the view of the root design to use. This directive is usually placed in the global section of a project file.

Syntax

```
VIEW_CONFIG‘view_name’
```

WARNINGS

The *WARNINGS* directive sets the global variable *KV_warning_output* to *TRUE* or *FALSE*.

Syntax

```
WARNINGS   {ON | OFF} ;
```

With the *WARNING* directive set to *ON*, the global variable *KV_warning_output* is set to *TRUE*, and warning messages are displayed. With the *WARNING* directive set to *OFF*, the global variable *KV_warning_output* is set to *FALSE*, and warning messages are not displayed.

Errors categorized as *warnings* often involve information that should be added to the drawing. Often, by adding this information to the drawing, you can avoid such warnings. The *SUPPRESS* directive can be used to turn off specific warning messages by number.

7

Command Line Arguments

This chapter describes the following:

- Overview
- Line Argument Table
- Predefined Command Line Arguments

Overview

CAE Views has a command argument table for defining the set of command line options available to your users. These command line arguments allow your users to override command file directives and global variable defaults directly from the command line when invoking the interface.

This chapter describes the predefined command line arguments in the table and how to define custom arguments for your users.

A template command line argument table is located in the `phys.c` file. You can use this table as a starting point to choose which of the predefined CAE Views command line arguments are made available to your users or to define custom command line arguments.

The command line argument table is accessed by a pointer (*KV_arg_ptr*) in the main program. If you define your own custom table, set *KV_arg_ptr* in your main program to point to your custom table. For example, if you create my_arg_table, add the following line to your main program before the *KV_read_directives()* call

```
KV_arg_ptr = my_arg_table;
```

and change the external structure reference to read:

```
extern KV_ARG_ENTRY my_arg_table[];
```

Line Argument Table

The command line argument table is an array of KV_ARG_ENTRY structures. A KV_ARG_ENTRY structure is defined as:

```
typedef struct KV_arg_entry {
     char argument[50];
     /* name of the argument */
     void (*read_arg)();/* function pointer to read the argument */
     int *value;/* pointer to the value to pass */

} KV_ARG_ENTRY;
```

In the structure:

`argument`

Holds the name of the argument you want to process.

`read_arg`

Points to the function called when the argument is found in the command line. When the function is called, three parameters are passed:

- Points to *argc*
- Points to *argv*
- Points to the variable to be set by the *read_arg* function.

  You provide the address of this variable in the last argument (the value member) of a KV_ARG_ENTRY structure.

`value`

Points to the variable used by your read_arg function. This variable is not mandatory. You can set it to NULL when it is not required.

The last entry of the command line argument table must be

```
      “” , NULL, NULL
```

to show the end of the table.

Predefined Arguments

CAE Views has a list of predefined arguments. To make these predefined arguments available to your users, keep the original entry in the template command line argument table as shown in the *phys.c* file. The following arguments are predefined:

- `b, backward`
- `compile`
- `hier`
- `logic`
- `phys`
- `u, use`
- `c, cell`
- `flat`
- `i`
- `l, lib`
- `x, xref`
- `r, ref`

The definition and usage of these arguments are detailed later in this chapter.

Read Argument Functions

CAE Views includes both reserved and general-purpose *read_arg* functions.

Reserved Read Argument Functions

The following *read_arg* functions are reserved for specific arguments and should not be used to develop custom command line arguments:

- `KV_arg_flat()`
- `KV_arg_lib()`
- `KV_arg_hier()`
- `KV_arg_dir()`

General-Purpose Read Argument Functions

The following functions are available for general use in creating your own command line arguments:

KV_arg_str()

This function reads a string from a command line argument. The string following the argument is copied into the buffer pointed to by the value argument. For example, if you have the following entry in the line argument table

```
    "my_arg", KV_arg_str, (int *)my_buffer
```

and you include the following argument with the command:

```
    ... -my_arg a_string ...
```

CAE Views copies “a_string” into “my_buffer.”

Note: Cast my_buffer to an int pointer.

KV_arg_set()

This function sets a flag to *TRUE* from a command line argument. For example, if you have the following entry in the directive table

```
    "my_arg", KV_arg_set, &my_flag
```

and you include the following argument with the command

```
    ... -my_arg ...
```

CAE Views sets “my_flag” to *TRUE*.

KV_arg_reset()

This function sets a flag to *FALSE* from a command line argument. For example, if you have the following entry in the directives table

```
    "my_arg", KV_arg_reset, &my_flag
```

and you include the following argument with the command

```
    ... -my_arg ...
```

CAE Views sets “my_flag” to FALSE.

KV_arg_int()

This function reads an integer from a command line argument. If the string following the command line argument is a valid number, the integer is set to this value. If the string is not an integer, CAE Views prints an error message and stops. For example, if you have the following entry in the directives table

```
    "my_arg", KV_arg_int, &my_int
```

and you include the following argument in the command

```
    ... -my_arg 3 ...
```

CAE Views sets “my_int” to 3. If you include the following argument with the command:

```
    ... -my_arg foo ...
```

CAE Views reports an error and stops.

User-Defined Read Argument Functions

If you do not want to use the internal line argument defaults, you will need to write your own *read_arg* function. This function must return void and expects three parameters. The first parameter is a pointer to the number of remaining arguments, the second parameter is a pointer to the pointer of the argument string, and the third parameter is a pointer to the variable passed from the line argument table.

For example, to read a command line argument “my_arg” that expects a parameter with two possible values: *TRUE* or *FALSE*, you want the variable my_var to be set to *TRUE* when the argument is *TRUE* and set to *FALSE* when the argument is *FALSE*.

To define “my_arg”

➤ Add the following line to your command line argument table

```
     "my_arg", my_read_func, &my_var
```

and write the code to read this line argument.

For example:

```
void my_read_func(argc, argv, value)
int *argc;
char **argv[];
int *value;
{
/* we update argc and argv to remove the line argument */
(*argc)--;
(*argv)++;
/* now we check that a parameter has been passed for this argument
*/
if ((*argc < 1) || (**argv[0] == '-')) {
(void)fprintf(LOGFP,XR("Invalid command line"));
(void)exit(EXIT_CODE(2));
}
```

```
    /* now we test the line argument */
    if (strcpm(**argv, ”TRUE”) == SAME) *value = TRUE;
    else if (strcpm(**argv, “FALSE”) == SAME) *value = FALSE;
    else {
    (void)fprintf(LOGFP,XR(“Invalid option in command line”));
    (void)exit(EXIT_CODE(2));
    }
    }
```

The Template Line Argument Table

The following template command line argument table is provided in the file *phys.c*:

```
KV_ARG_ENTRY user_arg_table[] = {
/*
LINE ARGUMENT READ FUNCT VARIABLE POINTER
*/
”c”,         KV_arg_str,    (int *)KV_command_drawing,
”cell”,      KV_arg_str,    (int *)KV_command_drawing,
”i”,         KV_arg_set,    &KV_delta_interface,
”l”,         KV_arg_lib,    NULL,
”lib”,       KV_arg_lib,    NULL,
”logic”,     KV_arg_reset,  &KV_physical_interface,
”phys”,      KV_arg_set,    &KV_physical_interface,
“proj”,      KV_arg_str,(int *)KV_global_cpm_file,
”r”,         KV_arg_set,    &KV_want_reference,
”ref”,       KV_arg_set,    &KV_want_reference,
”compile”,   KV_arg_str,    (int *)KV_compile_type,
”u”,         KV_arg_dir,    NULL,
”use”,       KV_arg_dir,    NULL,
”x”,         KV_arg_set,    &KV_cross_reference,
”xref”,      KV_arg_set,    &KV_cross_reference,
””,          NULL,          NULL
};
```

You can modify this table for your application. Just remove the lines for the predefined arguments that you don’t want to support and add extra lines for the custom command line arguments that you define.

Predefined Command Line Arguments

Command line arguments allow your users to change the setting of a directive or internal global variable directly from the command line. If a command line argument requires a variable, the variable follows the line argument separated by a space.

Command line arguments are case-sensitive. Invoking an interface with an upper-case version of any command line argument results in a CAE Views error message.

The command line argument descriptions that follow include a short description of the line argument and the variable affected by the line argument. For global variable defaults, see Chapter 5, “CAE Views Global Variables.”

-d
Debug Command Line Argument

The *-d* command line argument turns on debug switches. This line argument is for software developers, not for application users.

-h[elp]
Help Command Line Argument

The *-h* command line argument is not built into CAE Views. You can implement the line argument by adding the command to the line argument table. If implemented, the *-h* command line argument provides a help facility for application users. The help facility gives a brief explanation of all command line arguments and, if not too numerous, directives implemented in the CAE Views application. Explanations specific to the implementation are displayed on the screen and/or the listing file.

See “The Template Line Argument Table” on page 102 in this chapter for more information on adding line arguments in your application.

-proj
Project File Command Line Argument

The *-proj* command line argument identifies the project file to be used by the application. This command line must always be provided.

The syntax for the -proj command line argument follows:

Syntax

```
    interface -proj project_file
```

In the command line, *project_file* is the path name of the project file.

-v
Version Command Line Argument

The *-v* command line argument causes the current version of CAE Views to be displayed. The CAE Views program version is different from the version of your interface, which is displayed as part of the welcome banner.

A

CAE Views Error Messages

This chapter describes the following:

- Overview
- Syntax Errors
- File Errors
- CAE Views Errors
- Compilation Errors
- Part Properties Table Errors
- Miscellaneous Errors

Overview

CAE Views issues error messages. Each message is numbered and categorized according to the severity of the problem. Categories include:

- Warnings—A warning is a minor problem that CAE Views should successfully overcome.
- Oversights—An oversight is a problem that CAE Views can overcome but might lead to confusion.
- Errors—An error is a problem that must be fixed before the translation is complete.

  Some errors are fatal and cause CAE Views program execution to terminate.

CAE Views tracks the number of Errors, Oversights, and Warnings separately and prints them in the CAE Views listing file after program execution. An example of how CAE Views reports these errors is as follows:

```
3 errors detected
4 warnings detected
No oversights detected
```

You can suppress warnings with the SUPPRESS and WARNING OFF directives. See Chapter 6, “CAE Views Directives,” for more information on using these directives.

The following sections describe the standard CAE Views error messages. The description includes the error number, category of severity, text of message, and some comments on the cause of the problem. The error messages are listed in numerical order.

Syntax Errors

The following section gives a brief description of syntax error messages.

`1    ERROR`

Syntax error: unexpected character

CAE Views generates this error message when it expects a character and finds another character while parsing an input file. CAE Views prints the expected character, the name of the processed file, and the line number as part of the error message.

`2    ERROR`

Syntax error: unexpected token

CAE Views generates this error message when it expects a specific token and finds another token while parsing an input file. CAE Views prints the expected token, the name of the processed file, and the line number as part of the error message.

`3    ERROR`

Syntax error: expected a number

CAE Views generates this error message when it expects a number (integer or real) and finds some other data. CAE Views prints the found data, the name of the processed file, and the line number as part of the error message.

`4    ERROR`

Syntax error: expected a string

CAE Views generates this error message when it expects a string and finds some other data. CAE Views prints the found data, the name of the processed file, and the line number as part of the error message.

`5    ERROR`

Syntax error: expected a file name

CAE Views generates this error message when it expects a file name and finds some other data. CAE Views prints the found data, the name of the processed file, and the line number as part of the error message.

`6    ERROR`

Syntax error: comment not closed

CAE Views generates this error message when it does not find the end of a comment before the end of the file. A comment begins with the *KV_start_comment* character and ends with the *KV_end_comment* character.

`7    ERROR`

Syntax error: string not closed

CAE Views generates this error message when it does not find the end of a quoted string before the end of the current line. A string can be quoted by single (`'`) or double (`"`) quotes. The end of the line is set by the *KV_end_of_line* global variable. CAE Views prints the name of the processed file and the line number as part of the error message. Check the specified line to ensure that the string is enclosed in quotes.

`8    ERROR`

Syntax error: unexpected option

CAE Views generates this error message when it expects a predefined option (for example, ON/OFF, LOGICAL/PHYSICAL) and finds some other data. CAE Views prints the expected list of options, the name of the processed file, and the line number as part of the error message.

File Errors

The following section gives a brief description of file error messages. In cases where the warning and fatal error messages are the same, the difference in program execution is that a mandatory or critical file is missing in the application program.

`10   ERROR`

Unknown directive

CAE Views generates this error message if it finds an unknown directive in the command file. CAE Views prints the unknown directive as part of the error message. Usually, this error does not prevent CAE Views from reading the rest of the directives.

`20   WARNING`

Cannot open file

CAE Views generates this warning message when it is unable to open a non-critical input or output file, for example, the directives file. This usually occurs due to a protection problem. The name of the file is printed. Check the protections for the file and the user to ensure that they are compatible.

`21 ERROR`

Cannot open file

CAE Views generates this error message when it is unable to open a critical input or output file. This usually occurs due to a protection problem, (that is, no read or write access to the file). The name of the file is printed. Check the protections for the file and the user to ensure that they are compatible.

`22 ERROR`

Cannot close file

CAE Views generates this error message when it is unable to close a critical input or output file. This usually occurs when the application tries to close a file that was never opened or that has already been closed. The name of the file is printed. Check the application program to be sure that the file being closed has been opened or that it has not been previously closed and rerun the program.

`23 WARNING`

File not found

CAE Views generates this warning message when it cannot find a non-critical file. The name of the file is printed. Check the protections for the file and the user to ensure that they are compatible.

`24 ERROR`

File not found

CAE Views generates this error message when it cannot find a critical file, for example, Packager-XL output files. This usually occurs due to a protection problem (that is, you do not have read or write access to the file). The name of the file is printed. Check the protections for the file and the user to ensure that they are compatible.

`25 ERROR`

File has wrong type

CAE Views generates this error message when it finds an input file with the wrong FILE_TYPE specification. All system data files, except the command file, must be identified with a FILE_TYPE specifier; this lets CAE Views to check the validity of input data. To fix this problem, be sure that the program is calling the correct file, and then change the FILE_TYPE specification within the file.

`26 ERROR`

Write on output file failed

CAE Views generates this error message when it is unable to write data to an output file. This usually occurs due to a protection problem (that is, you do not have write access to the file). The name of the file is printed. Check the protections for the file and the user to ensure that they are compatible.

`27 ERROR`

Read on input file failed

CAE Views generates this error message when it is unable to write data to an output file. This usually occurs due to a protection problem (that is, you do not have read access to the file). The name of the file is printed. Check the protections for the file and the user to ensure that they are compatible.

CAE Views Errors

The following section gives a brief description of CAE Views errors.

`40 ERROR`

Object not found in database

CAE Views generates this error message when it finds a CAE Views object (part type, package, signal, and so on) that is not described in the CAE Views database. Part types in a design for example need to have a corresponding part definition in a library file. If this error occurs, check the command file to ensure that all required files are listed.

`41 OVERSIGHT`

Compare request not accepted

CAE Views generates this oversight message when compare is unable to match the current state of the design with a previous state of the design. This can happen if the reference files do not exist or if they are from another design. It can also happen if the packages or nets cannot be matched. CAE Views cannot generate Engineering Change Orders when this oversight occurs.

`42 OVERSIGHT`

Packages have different pin names

CAE Views generates this oversight message when the compare module finds a replaced package with pin names that differ from the previous package.

43 ERROR

Ambiguous part types found

CAE Views generates this error message when different part types have the same alternate name. This checking is done by the routine *KV_find_alternate_part()*. The built-in mechanism of alternate names should only be used if there is a one-to-one mapping between part types and the alternate names. If this mapping is not guaranteed, you should implement your own mechanism.

44 WARNING

Ambiguous feedback part types found

CAE Views generates this error message when different part types have the same alternate name. The routine *KV_find_alternate_part()* generates this message in a feedback application.

45 ERROR

UNSUPPORTED feature

CAE Views generates this error message if you try to run an unsupported feature. The current version of CAE Views generates this error when you invoke TIMES expansion or if you use the POWER_GROUP feature (use POWER_PINS instead).

46 ERROR

This error can't be suppressed

CAE Views generates this error message if the parameter of the *SUPPRESS* directive corresponds to an error which cannot be suppressed. It is only possible to suppress OVERSIGHTS and WARNINGS, which indicate user flaws that do not prevent the CAE Views run. More serious errors cannot be suppressed.

47 ERROR

Wrong description in libraries for part

CAE Views generates this error when the physical description found in the libraries does not match the actual description of the instance. Check that the physical description of pins for the part matches the body. This error frequently occurs when you update the body of a part, but forget to update the corresponding *chips.prt* file.

48 ERROR

Error on Memory primitive

CAE Views generates this error when the width of the address bus connected to the memory primitive is incorrect. The Memory primitive needs to have a DEPTH property attached to it. The value of this property needs to be coherent with the width of the address bus connected. For example, if you specify a DEPTH of 64 the width of the bus has to be 6.

Compilation Errors

The following section describes compilation error messages.

```
50  ERROR
```

Compilation errors

CAE Views generates this error message when the Compiler/Linker reports an error. These errors must be fixed before linking, otherwise, CAE Views cannot read the design database. The expander automatically collects all errors in your design and produces a *cmplst.dat* error file.

```
51  WARNING
```

Two different flags on same net

CAE Views generates this warning message in a logical application when two flag bodies on the same net are of different type (INPUT, OUTPUT, BIDIRECTIONAL). CAE Views prints the instance name (plus parameters) and the net name. You can ignore this warning if those different flag bodies are put on the same net for a reason (for example implementing a bidirectional flag).

```
52  OVERSIGHT
```

Different values for same property

CAE Views generates this oversight message in a logical application when a body pin has (in the definition of the part type) a different load value from the related node on an instance (instance of body pin). CAE Views prints the instance name, the part type name, the pin name, the property name and the conflicting property values. To correct this problem, remove the load value on the definition of the pin or instance of the pin.

```
53  WARNING
```

Cell description does not match body

CAE Views generates this error in a hierarchical design when the number of pins on a body does not equate to the number of interface signals in the cell description. CAE Views prints the missing pin and leaves this pin unconnected.

Part Properties Table Errors

The following section describes the part properties error messages.

`70  WARNING`

Instance property already used before

CAE Views generates this warning message when a property is specified more than once in a part properties table. CAE Views prints the name of the property following the error message.

`71  ERROR`

Unknown property attribute

CAE Views generates this error message when an unknown property attribute is specified in the part properties table. Currently, four attributes are understood by CAE Views. The OPT attribute indicates whether a property is optional on an instance of a part. The S attribute (default value) indicates that the property value is a string. The N attribute indicates that the property value is a real number. The R attribute indicates that the property value is within a range. Any other property attributes in the part properties table generates this error.

`72  ERROR`

Duplicate physical part table entries

CAE Views generates this error message when a part properties table entry is specified more than once in the part properties table. CAE Views prints the entry after the error message.

`73  ERROR`

Subtype name is illegal

CAE Views generates this error message when it detects an illegal user-specified subtype suffix in the part properties tables. When CAE Views creates a new part type from the properties in the part properties table, it creates a new name for this *subtype* by appending a hyphen (-) and a suffix to the original part type. You can define the suffix in the part properties table. The subtype suffix must follow the instance property value and be enclosed in parentheses.

`74  WARNING`

Property not found on instance

CAE Views generates this warning message when an instance property in the part properties table is missing a value. You may add or modify the values of instance-specific properties on instances by giving new property values in the part properties table. If such a property is not optional (as indicated by the attribute OPT following the property name), CAE Views expects to find a value given for each instance property. Make sure that the appropriate values are given in the table for non-optional properties.

`75  ERROR`

Physical part table entry not found

CAE Views generates this error message when it cannot find an instance with the instance properties specified in the part properties table. If this error occurs, check the part properties table to ensure that the entry CAE Views printed has the correct properties.

`76  OVERSIGHT`

Invalid range specification

CAE Views generates this oversight message when a range specification in the part properties table is wrong. CAE Views prints more information about the kind of error made and the line number on which it occurred.

Miscellaneous Errors

The following section gives a brief description of miscellaneous error messages.

`100 ERROR`

CAE Views Internal Error

CAE Views generates this error message when something goes wrong internally in CAE Views. If this error occurs, save all files and contact your local Valid office.

`101 ERROR`

Error limit exceeded

CAE Views generates this error message when the number of errors exceeds the error limit set with the *MAX_ERRORS* directive. The default value is 500. CAE Views terminates after printing this message.

`102 ERROR`

Run stopped because errors were detected

CAE Views generates this error when it encounters fatal errors in the processing of the design. CAE Views writes the error message to the screen and listing file and terminates the CAE Views application.

`103 E1RROR`

Interrupt requested by user

CAE Views generates this error when the user interrupts the CAE Views program by pressing *<CTRL> C*. CAE Views stops processing after printing this message.

B

Part Properties Tables

This chapter describes the following:

Overview

Part properties tables allow you to create new part types from a basic type. They also allow you to attach new body properties to a part type, without recreating or modifying the library files containing the part type definitions.

This appendix begins with a description of part properties table uses and a sample file. This overview information is followed with detailed descriptions of how to create your own part properties tables.

Part Properties Table Uses

Create New Part Types from Basic Types

Using part properties tables, you can create new part types from a basic type. For example, you can create many different types of resistors and capacitors from a single basic resistor or capacitor. The various resistor types may have different resistance values, power dissipation, cost, or reliability characteristics. All of these characteristics can be specified in a part properties table.

Part properties tables are upwards compatible with Physical Part Tables. There is only one library definition for the part, and therefore only one copy of the model. Logical CAE Views uses the properties attached to the part to differentiate multiple instances of the same part.

Attach New Body Properties to a Part Type

Another use of part properties tables is to attach new body properties to a part type, without recreating or modifying the library files containing the part type definitions. An important use of this capability is the addition of new properties to the libraries for specific interfaces. For example, properties that describe the type and shape of each component can be added for a SCICARDS Interface.

By using several part properties tables, you can change the way part types are handled without changing the library files. If you are creating several different interfaces, you can use a part properties table for each interface, without adding the properties for each interface to the library. You then specify the part properties table to be used by each interface.

You can create a part properties table with any text editor. Since the files are kept in tabular form, they can easily be read and updated.

Figure B-1 on page 116 provides an example of a part properties table file for 1/4-watt resistors. The line numbers on the left are not actually in the file, but are used to describe the format of the table. Note that comments are enclosed in braces, and in this example they precede the elements they describe.

Figure B-1 Sample PART_PROPERTIES_TABLE File

```
 1. FILE_TYPE=PART_PROPERTIES_TABLE;
 2.
 3. { 1/4-watt resistor table }
 4.
 5. PART '1/4W RES'
 6.
 7. { SCICARDS specific properties }
 8.
 9. SCI_PART = RES1/4W
10. SCI_SHAPE = CR1/4W
11.
12. { table format }
13.
14. :VALUE = PART_NUMBER, COST;
15.
16. { actual table entries for the resistors }
17.
18. 1K  = CB1025,$0.05
19. 1.2K = CB1225,$0.05
20. 1.5K = CB1525,$0.05
21. 2.2K = CB2225, $0.05
22. 2.7K = CB2725,$0.05
23. 3.3K= CB3325,$0.05
```

```
24. 3.9K= CB3925,$0.05
25. 4.7K = CB4725, $0.05
26. 5.6K = CB5625, $0.05
27. 6.8K = CB6825, $0.05
28. 8.2K = CB8225, $0.05
29.
30. { end of the 1/4W RES entries }
31.
32. END_PART
33.
34. { end of the part properties table file }
35.
36.  END.
```

Line 1	Identifies the file as a part properties table (FILE_TYPE=MULTI_PHYS_TABLE is still supported).
Line 2	Separates the first line from the lines that follow. Blank lines are ignored by CAE Views and can be used to make your properties table easier to read.
Line 3	Adds a comment to the table. The character {introduces a comment and the character} terminates a comment. Comments can cross line boundaries, but they cannot be nested.
Line 5	Begins the part properties table entries for the 1/4-watt resistor (part type ‘1/4W RES’). Notice that the part type name is enclosed in single quote marks. Either single or double quote marks are required if the part name includes spaces.
Lines 9 and 10	Indicate that all 1/4-watt resistors have the body properties SCI_PART and SCI_SHAPE added to the part type, with the values “RES1/4W’ and ‘CR1/4W’ respectively.
Line 14	Describes the format for each line in the table for the 1/4-watt resistor. In this example, the format identifies the VALUE property as the property that can be modified for each new part type and adds the new properties PART_NUMBER and COST. Notice that a comma delimits the properties PART_NUMBER and COST. This defines a comma as the separator character between the PART_NUMBER and COST values within the table that follows.

Lines 18 to 28	Define new parts, following the format set up in line 14. Logical CAE Views searches through this table to determine the new part types to create. For example, line 18 specifies that all 1/4-watt resistors with a VALUE property of ‘1K’ are assigned to a new part type. This new part type has the same definition as a 1/4-watt resistor without a VALUE property, plus the additional properties PART_NUMBER and COST. The values of CB1025 and $0.05 are respectively assigned to these additional properties. Likewise, if the 1/4-watt resistor has a VALUE of 4.7K, line 25 specifies the new part type has values of CB4725 for the PART_NUMBER property and $.05 for the COST property.
Line 32	Denotes the end of the part table for the 1/4-watt resistors.
Line 36	Denotes the end of the file.

Using Part Properties Tables

If a part has a table associated with it, logical CAE Views reads the table format definition line to find properties that can be used to alter the part. If any of these properties are found on an instance, their values are checked against the entries in the table. If CAE Views cannot find an entry in the table for the given values on a part, it generates an error message. You must either change the property values in the drawings or update the part properties tables.

Logical CAE Views creates a unique library part definition for each entry in the table that matches an instance in the drawings. These are treated as though they were unique physical part types. The associated information from the tables is added to each new library part created. For example, attaching a POWER_PINS property to a new part type forces CAE Views to replace the previous power supply with the specified alternate power supply.

PART_TABLE_FILE Directive

The *PART_TABLE_FILE* directive identifies the files containing properties tables. You can specify any number of tables with this directive. You can specify each file on a separate line using multiple *PART_TABLE_FILE* directives or list multiple files (separated with commas) on one line with one *PART_TABLE_FILE* directive.

For example, the directive

```
    PART_TABLE_FILE ’res.tab’,’cap.tab’;
```

specifies two part properties table files, *'res.tab'* and *'cap.tab'*, and is equivalent to the directives

```
    PART_TABLE_FILE 'res.tab';
    PART_TABLE_FILE 'cap.tab';
```

For CAE Views interfaces, which use the CAE Views database after reading in part properties tables, different part types with the same name but different alternate name can be found in the database. Their properties differ, and if the POWER_PINS property was used the pins maybe different.

Scale Factors

When CAE Views searches the part properties table entries, the property values on the instances in a design must meet some requirements. Property attributes (*S*, *N*, *R*, and *OPT*), referred to below, are discussed in detail in the next section, Part Properties Table File Format.

- If a property value is specified as a string, the property value on the instance must exactly match the value defined in the entry.

  A property value is assumed to be a string by default or if you use the attribute *S*.

- If a property value is specified as a number, the property value is translated into a real value.

  This value should match the value of the table entry or be within a table entry range. A property is assumed to be a real value if you use the property attribute *N* (number) or *R* (range).

CAE Views recognizes by default a common set of scale factors (as defined by SPICE) to translate property values into real values.

The figure below lists these scale factors.

Table B-1 CAE Views Scale Factors

Scale Factor	Value
T	1E12
G	1E9
MEG	1E6
K	1E3
M	1E-3
U	1E-6

Table B-1 CAE Views Scale Factors, *continued*

Scale Factor	Value
N	1E-9
P	1E-12
F	1E-15

As an example of using scale factors, the table specifies the value of K as 1E3. This means that the property values ‘1.234K’, ‘1234’, ‘1.234KOhm’, ‘1234ohm’ all translate to the same real value ‘1234’.

You can define your own scale factors.

If the default scale factors are not appropriate for your environment, you can define your own complete set of scale factors. Follow these steps to create your own scale factors.

1. Create a scale factor file defining your scale factors. Scale factors must be expressed in SPICE notation.

   Figure B-2 on page 120 provides an example of a scale factor file

.
Figure B-2 Sample SCALE_FACTORS File

```
FILE_TYPE=SCALE_FACTORS;
TERRA=  1E12
GIGA =  1E9
MEGA=   1E6
KILO=   1E3
MILLI=  1E-3
MICRON= 1E-6
NANO=   1E-9
PICO=   1E-12
FEMTO=  1E-15
END.
```

2. Specify the name of your scale factor file with the *SCALE_FACTOR_FILE* directive.

   ```
   SCALE_FACTOR_FILE 'own_fators.dat';
   ```

   CAE Views recognizes the scale factors defined in your file as long as this entry is in the directives file.

Part Properties Table File Format

A part properties table file begins with a line that identifies the type of file it is

```
FILE_TYPE = PART_PROPERTIES_TABLE;
```

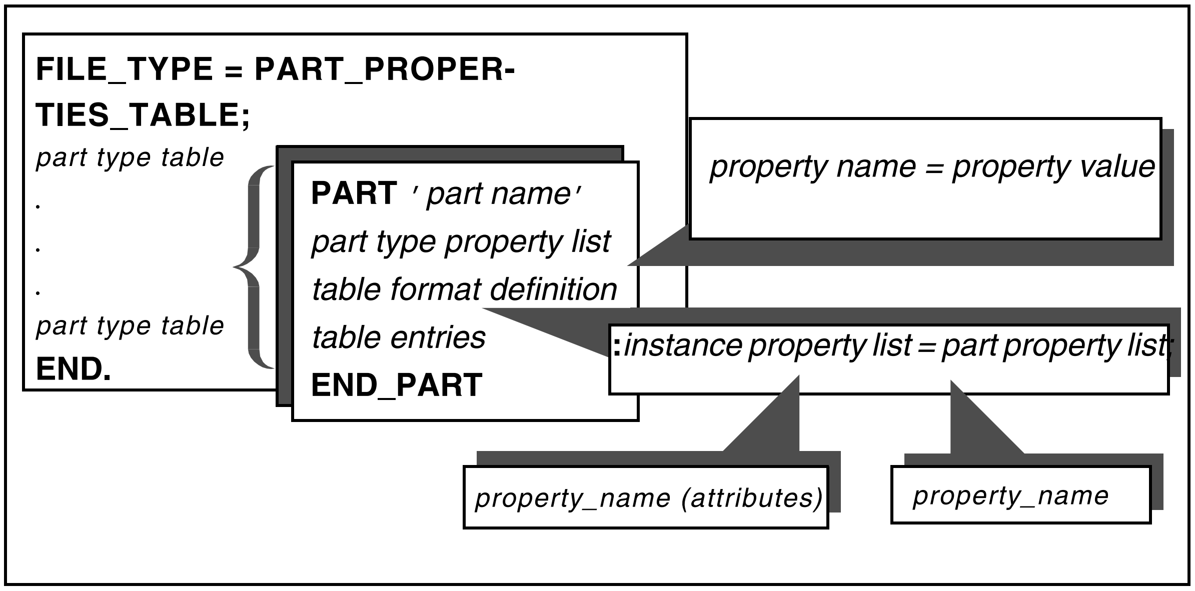
and ends with the keyword

```
END.
```

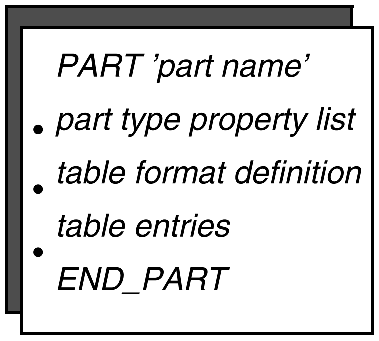
Between these two lines you can include information for more than one part type. Each part type definition is a separate *part type table*. Each table begins with a line with the keyword *PART* followed by the name of the part type being redefined by the table entries, and ends with the keyword *END_PART* (notice the absence of a period).

Figure B-3 on page 121 shows a generalized picture of a part properties table file, along with the format of an individual part type table.

Figure B-3 Conceptual View of PART_PROPERTIES_TABLE Syntax

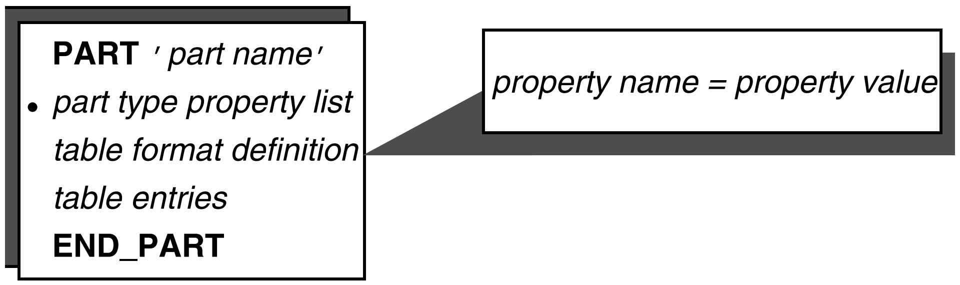


The subsections that follow provide detailed syntax information on the format of a part type table. Each line marked with a bullet in the part type table outline below corresponds to a subsection that follows.



Part Type Property List

The *part type property list* adds new properties to a part, without modifying the chips files or library drawings. This is useful if you wish to add properties independent of any set of properties attached to a logical part. Each part type property list entry appears on a line by itself.



Syntax

```
property name = property value
```

property name

A standard SCALD property name. It is a string of no more than 16 alphanumeric characters, beginning with an alphabetic character. The underscore (_) is considered an alphanumeric character.

property value

A string of any characters, terminated by the end of the line. If the desired string of characters exceeds the line length, you can use the tilde (~) as a continuation character. The tilde must be the last character on the line. For example,

```
SCI_PART = RES1/4W
```

is equivalent to

```
SCI_PART = RES~
1/4W
```

Multiple spaces are considered to be one space, and leading and trailing spaces around property values are ignored. Enclose a property value in quotes to indicate that spaces should be interpreted literally. Thus the line

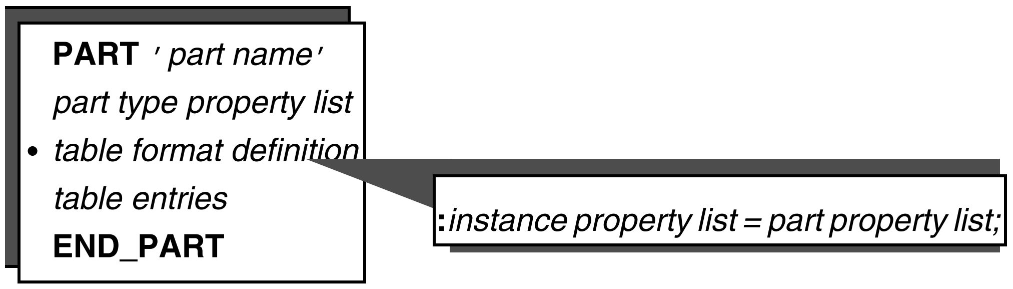
```
SCI_PART = ’ RES1/4W ’
```

defines a SCI_PART value with a leading and trailing space.

You may use either single quotes (′) or double quotes (″). This allows you to use quotes in a property value, by using the other quote character to enclose the entire property value.

Table Format Definition

The table format definition defines the format of each table entry that follows.


PART *′part name′*
part type property list
• *table format definition*
table entries
END_PART
:*instance property list = part property list;*

Syntax

```
:instance property list = part property list;
```

:	Identifies the line as a table format definition.

`instance property list`	A list of property names that may be attached to an instance of a part. These properties allow you to customize a part. The entries that follow the format definition provide the values for the new part types. If more than one property is specified in the instance property list, each value must be matched in a table entry before a part is selected.
`part property list`	A list of the properties to be associated with the new part type by CAE Views.
`;`	Terminates the statement, marking the end of the table format definition.

If your instance property list or part property list contains more than one entry, you must choose a *separator* character. You indicate your choice of separator character simply by using an eligible character in your definition. Your character choice as a separator eliminates the use of that character in expressing a property value. You may use the same separator character in the instance and part property lists or define a different character for each list.

A *separator* may be any keyboard character (including a space) that does not have a conflicting definition. This eliminates all alphanumeric characters (including the underscore), which are interpreted as letters or numbers. It also eliminates the characters listed below.

Characters ineligible as separators to delimit property names in an instance or part property list:

()	Opening and closing parentheses, which delimit attributes.
{}	Opening and closing braces, which delimit comments.
[]	Opening and closing square brackets, which enclose a range for an *R* attribute. These characters are ineligible only when the *R* attribute is used.
=	Equals sign, which is an assignment character.
:	Colon, which introduces the table format definition.
;	Semicolon, which is a statement terminator.
’ ”	Quote marks (single or double), which indicate that spaces should be interpreted literally.
~	Tilde, which is a continuation character.

The following example uses a comma to separate the property names VALUE and TOLERANCE.

Example

```
:VALUE, TOLERANCE = PART_NUMBER, COST;
```

Instance Property List

The *instance property list* identifies the property names the table is redefining. In essence, it identifies the variable(s) modified to create each new part type. The instance property list follows the pattern

```
    property_name (attributes)
```

which is repeated for each identified property. Delimit each identified property (and any attributes) with a *separator*.

When present, the *attribute list* describes special characteristics CAE Views must be aware of during the processing of the part properties table. Attributes are enclosed in parentheses; and if there is more than one attribute, they are separated by commas. CAE Views currently understands four different attributes.

Attributes recognized by CAE Views:

OPT

Indicates a property is optional on an instance of a part. If this attribute is not assigned to a property and a part is missing the property, CAE Views generates a warning message.

You can also use this attribute to indicate a default value for a property. Then if a part is missing a property, CAE Views uses the default value assigned with the *OPT* attribute. This default value should appear as a quoted string if it contains spaces.

Example

```
:VALUE(OPT='1K') = PART_NUMBER;
```

The example table format definition specifies that the VALUE property is optional on the part. If not present on the part, CAE Views assumes a default value of 1000 without generating a warning messages.

S

Indicates that the property value is a string of characters. The property value on an instance of a part should exactly match the value specified in the table. If you don’t specify an attribute, CAE Views assumes the *S* attribute by default.

N

Indicates that the property value is a number. All property values with number attributes are translated into real number values. A property value found on an instance of a part is translated into a real number before comparing it with the table entries. If the translation into a real value fails, CAE Views generates an error message.

R

Indicates that the property values specified in the table are ranges and the property value found on an instance of the part is a number which should fall within a specified range. As an example, the definition

Example

```
:VALUE(OPT='1K', R) = PART_NUMBER;
```

specifies that the VALUE property is optional. If missing, CAE Views assumes a default value of 1000. The table entries that follow this definition can specify a range of values for the VALUE property. An instance matches a table entry when its VALUE property falls within the specified range.

If you use the *R* property and your instance property list contains more than one property, be sure to define something other than an opening or closing square bracket [] as the separator character. These characters are used in your table entries to specify ranges.

Parts Property List

The parts property list identifies the properties CAE Views associates with the new part type. There is no limit to the number of property names you can include in your list. If your list exceeds the length of your line, you can continue on the next line without using a continuation character. The semicolon (;) marks the end of the part property list.

If your part property list contains just one property name, simply list the property name and follow it with a semicolon to terminate the definition.

```
    property_name;
```

If your part property list contains more than one property name, simply use a separator between property names and follow the last listed name with a semicolon.

```
    property_name | property_name;
```

Table Entries

Each table entry follows the pattern set by the table format definition. Since a part properties table file usually contains more than one part type table, this pattern likely changes from part to part.

PART *′ part name′*
part type property list
table format definition
- *table entries*

END_PART

Each table entry must appear on one line. If an entry is too long, you can use tilde (~) as a continuation character. Each table entry follows the pattern set by the table format definition and takes one of the following two forms:

Syntax

```
instance values = part type values
```

–or–

```
instance values = part type values : new properties
```

instance values

A list of property values for the instance properties specified in the table format definition line. Property values must conform to attributes specified in the definition. The *S* (string) attribute is assumed by default.

If the attribute for the instance property is *R*, the instance value can either be a discrete real number or a range of numbers. An instance value with an **R** attribute and a range specification takes the following form:

```
    border_char value separator value border_char
```

border_char

Maybe either an opening or closing square bracket. The orientation of the square bracket indicates either inclusion or exclusion. An opening bracket on the right side or a closing bracket on the left indicates exclusion. For example,

[2, 10]	Indicates a range from 2 to 10
[2, 10[	Indicates a range from 2 to less than 10
[2, 2]	Indicates the number 2. You can express an instance value of 2 as a discrete number, without brackets and separator.

value

A real number. You may use scale factors to express a value. The at-sign (@) indicates infinity.

separator

Either a colon (:) or a comma(,).

part type values

A list of values for the new part, one for each property associated with the part type in the table format definition.

: new properties

A list of new properties and associated values to be added for this specific table entry. This is similar to adding properties using the *part type property list*, except that the property is added only for this specific part. The format for *new properties* follows.

```
    property_name ='property value'
```

If there is more than one new property, this pattern is repeated, using a *separator* between properties. Since a colon is used to separate the last part type property value from any new properties, enclose any property values containing colons in quotes. A property value must be enclosed in quotes if it contains spaces.

Part Properties Table Examples

The following part properties table is an abbreviated version of an earlier example. It provides a concise illustration of table entries conforming to the table format definition.

Example

In this resistor table example, each line is defined to start with a VALUE property followed by an equals sign (=). The next two fields are the PART_NUMBER and COST property values. The placement of the comma between the two part property fields defines the comma as a *separator* character. As such, you cannot use a comma to express a property value within this part type table.

```
FILE_TYPE=PART_PROPERTIES_TABLE;
PART '1/4W RES'
:VALUE = PART_NUMBER, COST;

1K  = CB1025,  $0.05
1.2K = CB1225,$0.05
1.5K = CB1525,$0.05
2.2K = CB2225, $0.05
2.7K = CB2725,$0.05

END_PART
END.
```

Example

The next example is the same as the previous, with two exceptions. It uses a vertical bar (|) as a separator. This change allows you to use a comma to express a property value. This example also adds a new property to one of the table entries. Since the property is added as a table entry, the new property only affects the definition of 1K resistors.

This part properties table thus defines resistors with a VALUE of 1000 to have a PART_NUMBER of CB1025, a COST of $0.05, and a TOLERANCE of 5%. Resistors with VALUES of 1200, 1500, 2200 each have PART_NUMBER and COST values defined, but they do not have TOLERANCE properties.

```
FILE_TYPE=PART_PROPERTIES_TABLE;
PART '1/4W RES'
:VALUE = PART_NUMBER, COST;

1K  = CB1025| $0.05 : TOLERANCE = 5%
1.2K = CB1225|$0.05
1.5K = CB1525|$0.05
2.2K = CB2225|$0.05
2.7K = CB2725|$0.05

END_PART
END.
```

Example

The next part properties table provides an example of a table format definition with more than one property name in the instance property list. Each property name has an attribute attached to it, one of which is a range.

When the format definition of a part type table includes more than one property name in the instance property list, the instance values for both properties must match the table entry before CAE Views selects the entry. Thus, as is shown below, a table can include multiple entries for one property value while varying the value for other properties in the instance property list.

```
FILE_TYPE=PART_PROPERTIES_TABLE;
PART '1/4W RES'
:VALUE(N),TOLERANCE (R) = PART_NUMBER, COST;

1K, ]@,1%[= CB1025, $1.00
1K, [1%,10%[=CB1025, $0.75
1K, 10% =   CB1025$0.50
1K, ]10%,@[=CB1025$0.05
2K, ]@,1%[= CB1225, $1.00
2K, [1%,10%[=CB1225, $0.75
2K, 10% =   CB1225$0.50
2K, ]10%,@[=CB1225$0.05
END_PART
END.
```

This file creates four categories of 1K resistors and 4 categories of 2K resistors, based on TOLERANCE values. The four TOLERANCE categories are values less than 1%, values between 1% and less than 10%, values equal to 10%, and values greater than 10%.

Each category could have different PART_NUMBER and COST values associated with it, but this particular file specifies just one part number (CB1025) for 1K resistors and another part number (CB1225) for 2K resistors. COST value assignments vary depending on the TOLERANCE levels. Again, since this is an independent assignment, the table could specify a different value for each entry.

Example

CAE Views compares properties on an instance of a part with entries in a part type table one-by-one in the order specified in the table. If a table includes ranges, CAE Views does not check the combination of ranges for validity. The following example demonstrates that table entries specifying ranges can result in redundant entries, and perhaps some unexpected consequences.

```
FILE_TYPE=PART_PROPERTIES_TABLE;
PART '1/4W RES'
:VALUE(N),TOLERANCE (R) = PART_NUMBER, COST;

1K, ]@,10%[=CB1025, $1.00
1K, [2%,5%[=CB1025$0.50
1K, [10%,@]=CB1025$0.05
2K, [10%,@[=CB1225$0.05
END_PART
END.
```

The first entry in the preceding table specifies that 1K resistors with a TOLERANCE less than 10% COST $1.00. The second entry specifies that 1K resistors with a TOLERANCE between 2% and less than 5% COST $0.50. Since CAE Views processes the table entries one line at a time, in the order specified, the first entry matches all instances of 1K resistors with a TOLERANCE less than 10%. CAE Views assigns a COST value of $1.00 to 1K resistors with a TOLERANCE between 2% and less than 5%, certainly not the intent of the person constructing the part type table. If the two table entries were reversed, CAE Views would match 1K resistors with a TOLERANCE between 2% and less than 5% and assign a COST value of $0.50, then match the remaining 1K resistors with a TOLERANCE less than 10% with a COST value of $1.00.

Modified Part Types in Part Properties Tables

CAE Views normally generates new part types with the same part type name as the original one. Packager-XL, on the other hand, generates new part type names for modified part types (subtypes) by appending a dash (-) and an integer to the part type name. Examples of subtype names for the part type RESISTOR are RESISTOR-1, RESISTOR-2 and RESISTOR-3.

CAE Views generates new names the same way the Packager-XL does, but stores this name in the alternate field of the part type structure. CAE Views also generates the property PARENT_PART_TYPE with a value equal to the original part type name, and adds it to the property list of the new part type.

Subtype part type names are not guaranteed to be associated with the same parts from one run to another. Therefore, you can assign your own subtype name in the part properties table by attaching it to the property name. By using this feature, the subtype name remains the same each run of the program. The following part properties table includes subtype names.

```
FILE_TYPE=PART_PROPERTIES_TABLE;
PART '1/4W RES'
:VALUE(N),TOLERANCE (R) = PART_NUMBER, COST;
1K, 2%(1K)= 1285, $.50
2.3K,1%(2.3K)=1300,$.50
1K, 5%(1K,5%)=1024,$.24
5K, 1%(!)=  1000,$.43
1K  3%(!)=  1028,$.24
1K  4%  =   1028,$.24
END_PART
END.
```

The subtype names (suffixes) are enclosed in parentheses. The suffixes are of two types: implicit and explicit. Implicit subtypes are denoted by an exclamation point (!). They cause CAE Views to append the property values to the part type name. CAE Views places commas between property values. For example, lines 7 and 8 above result in the following subtypes.

```
line 7 '1/4W RES-5K,1%'
line 8 '1/4W RES-5K,3%'
```

Explicit subtypes appear directly within the parentheses. For example, lines 5 and 8 above result in the following subtypes.

```
line 5 '1/4W RES-2.3K'
line 6 '1/4W RES-1K,5%'
```

You can use any alphanumeric characters within a suffix (including upper- and lowercase and the underscore), as well as the following characters:

- , (comma)
- % (percent sign)
- & (ampersand)
- + (plus sign)
- $(dollar sign)
- # (number sign)
- * (asterisk)
- . (period)

If you don't specify a suffix, CAE Views creates a numeric suffix and appends it to the part type name. As an example, based on the table entry on line 9 in the previous example, CAE Views might assign a subtype of '1/4W RES-1'.

Whether assigned by default or by you, the subtype name cannot exceed the length of a legal part type. If the name is longer than this limit, CAE Views generates an error message and the name is truncated. The *PART_TYPE_LENGTH* directive controls the subtype name length limit.

Index