JPL Publication 88-32, Rev. 1, Vol. II

62 P

Concurrent Image Processing Executive (CIPE)

Volume II: Programmer's Guide

Winifred I. Williams

March 15, 1990

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL Publication 88-32, Rev. 1, Vol. II

Concurrent Image Processing Executive (CIPE)

Volume II: Programmer's Guide

Winifred I. Williams

March 15, 1990

NASA

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

ABSTRACT

This manual is intended as a guide for application's programmers using the Concurrent Image Processing Executive (CIPE). CIPE is intended to become the support system software for a prototype high performance science analysis workstation. In its current configuration CIPE utilizes a JPL/Caltech Mark IIIfp Hypercube with a Sun-4 host. CIPE's design is capable of incorporating other concurrent architectures as well. CIPE provides a programming environment to applications' programmers to shield them from various user interfaces, file transactions, and architectural complexities. A programmer may choose to write applications to use only the Sun-4 or to use the Sun-4 with the hypercube. A hypercube program will use the hypercube's data processors and optionally the Weitek floating point accelerators. The CIPE programming environment provides a simple set of subroutines to activate user interface functions, specify data distributions, activate hypercube resident applications, and to communicate parameters to and from the hypercube.

Table of Contents

PAGE IV INTENTIONALLY BLANK

PRECEDING PAGE BLANK NOT FILMED

LIST OF FIGURES

1. INTRODUCTION TO CIPE

Concurrent systems provide greatly enhanced computational power by integrating large numbers of processors via various interconnection topologies. However, they present significant programming difficulties due to their architectural complexities. CIPE is designed to shield the architectural complexities from programmers, to provide an interactive image processing environment for users, and to utilize the architectural characteristics of a concurrent system (interconnection topology, processing power, memory) for securing the computational power of the system.

In a traditional image processing environment, a user activates a program and provides file names for input and output, thus involving an explicit file transaction for each program. The file transaction oriented data manipulation is very inefficient and cumbersome in an interactive image processing environment. In CIPE, image processing programs and datasets are viewed as subroutines and variables that a user can manipulate interactively without file transactions. Such an environment is achieved via a combination of interactive user interface modes, incremental loading of image processing modules, symbolic data management, and an automatic data distribution/retrieval process.

In order to shield an applications programmer from various user interfaces, file transactions, and architectural complexities, the CIPE programming environment offers a simple set of subroutines for activating user interface functions, specifying data distributions, activating hypercube resident application routines, and communicating parameters to and from the hypercube routines. An applications programmer does not have to be aware of actual user interface modes, or data distribution and hypercube communication protocols.

This manual is designed to provide all the necessary information for a programmer to write application programs to be run within CIPE. The current implementation of CIPE uses a Sun-4 host computer and a JPL/Caltech Mark IIIfp 8-node hypercube. Application programs may run without the use of a coprocessor, and use only the resources of the Sun-4, or they may run with a coprocessor, and use the resources of the Mark IIIfp hypercube, with or without the Weitek floating point accelerators. Other attached concurrent systems may be supported in the future.

Throughout this manual the term programmer is used to refer to a CIPE applications programmer. The term user is used to refer to a CIPE user.

2. PROGRAMS WITHOUT A COPROCESSOR

An application program consists of three basic parts: the user interface, data I/O, and data processing. CIPE provides a programmer with commands for accomplishing the user interface and data I/O, and expects the programmer to provide the algorithm for data processing. Without the use of a coprocessor, all of the parts of an application program run within the host system. If a coprocessor is used with CIPE, the user interface is implemented in the host, the data I/O takes place between the host and the coprocessor, and the data processing takes place in the coprocessor.

2.1. Include Files

All programs to be run in the host must include the file *cipeappl.h*, found in /usr/local/include. This file should provide a CIPE applications programmer with all the definitions and declarations necessary to use CIPE-provided routines. A copy of the file has been included in Appendix A for programmer reference. Any other files to be included are at the programmer's discretion. For an example, see the *Include Files* section of the sample program in Figure 2.1.

2.2. User Interface

CIPE provides two operational user modes, command line interpreter (CLI) mode and menu mode. Differences in user modes are made transparent to the applications programmer through the use of two routines, **cipedef** and **cipepar**, which are able to interface to either of the user modes.

2.2.1. Use of *cipedef*

The routine **cipedef** associates a user's input with a variable declared in the application. A sample **cipedef** statement is shown below, and a description of its parameters follows.

```
cipedef(param_num,prompt,param_type,param_storage,
            num_of_elements,if_req);
```

A parameter number should be the first value passed to **cipedef**. The first call to **cipedef** should provide a **param_num** of 1. Successive calls to **cipedef** should provide successive integer values for **param_num**.

The second value passed to **cipedef** should be a **prompt** string. In CLI mode a user is expected to know the order of parameters and will receive no prompts. In menu mode, however, the prompt will appear on the screen to assist the user.

powerspec

```
/****************************** INCLUDE FILES ********************************** /
#include <stdio.h>
#include <ctype.h>
#include "cipeappl.h"
powerspec()
{
      cipe_sym_name input_real, input_imagi, output;
      int p;
      char option[3];
      p = 1;
      strcpy(option,"y");
/****************************** USER INTERFACE ******************************** /
      cipedef (p++, "real symbol              ", INPUT_SYMBOL, input_real,
            sizeof(cipe_sym_name), REQ);
      cipedef (p++, "imaginary symbol         ", INPUT_SYMBOL, input_imagi,
            sizeof(cipe_sym_name), REQ);
      cipedef (p++, "power spectrun symbol ", OUTPUT_SYMBOL, output,
            sizeof(cipe_sym_name), REQ);
      cipedef (p++, "fold for display (y/n)", STRING, option,
            sizeof(option), DEFAULT);
      if (cipepar(check_power, help_power)) return(-1);

      if (strcmp(option,"y")==0)
            host_power_spec(input_real,input_imagi,output,1);
      else
            host_power_spec(input_real,input_imagi,output,0);
}
```

check_power

```
check_power(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
      int error,whereis;
      error = 0;
      switch(number){
            case 0: error = cipe_error_read_symbol(value.s);
                        break;
            case 1: error = cipe_error_read_symbol(value.s);
                        break;
            case 2: error = cipe_warn_write_symbol(value.s);
                        error = 0;         /* User is warned but not forced to give * /
                                             /* a new value. * /
                        break;
            case 3: /* no check, "y" will be taken as yes, everything else as no * /
                        break;
      }
      return(error);
}
```

help_power

```
help_power(params,number)
struct param *params;
int number;
{
      switch (number) {
            case 0,1:
                  printf("A valid input symbol must be associated");
                  printf("with two dimensional image data.");
                  break;
            case 2:
                  printf("A valid output symbol must not already exist.");
                  printf("You may see if it does by using List from the");
                  printf("Symbol menu.");
                  break;
            case 3:
                  printf("Use a y to indicate you would like it folded ");
                  printf("for display. Otherwise, use another character.");
                  break;
      }
}
```

Figure 2.1 Application Program Without a Coprocessor

host_power_spec

```
host_power_spec(input_real,input_imagi,output,disp)
cipe_sym_name  *input_real,*input_imagi,*output;
int disp;
{
      struct cipe_symbol *sin_real, *sin_imagi, *sout;
      float *power_disp_data, *p1;

/******************************** DATA I/O ************************************/
      sin_real = cipe_get_symbol (input_real);
      sin_imagi = cipe_get_symbol (input_imagi);
      sout = cipe_create_symbol_and_data(output,
             CIPENL(sin_real),CIPENS(sin_real),1,FLOAT_TYPE);

/******************************** DATA PROCESSING ******************************/
      power_spec(CIPEDATA(sin_real),CIPEDATA(sin_imagi),CIPEDATA(sout),
             CIPENL(sin_real),CIPENS(sin_real));

      if (disp) {
             power_disp_data = (float *)malloc(CIPEDATASIZE(sout));
             if (power_disp_data == NULL) {
                    printf("power_disp_data malloc failure\n");
                    return(-1);
             }
             p1 = (float *)CIPEDATA(sout);
             p1[0] = 0.0;           /* turn off the sum for better contrast */
             fold_image (CIPEDATA(sout),power_disp_data,CIPENL(sout));
             free (CIPEDATA(sout));
             CIPEDATA(sout) = (unsigned char *)power_disp_data;
      }
}
```

power_spec

```
power_spec(real,imagi,power,nl,ns)
float *real,*imagi, *power;
int nl,ns;
{
      register float *p1,*p2, *p3;
      int i;
      double temp;

      p1 = real;
      p2 = imagi;
      p3 = power;
      for (i=0; i<nl*ns; i++) {
             temp = (*p1 * *p1) + (*p2 * *p2);
             *p3++ = temp;
             p1++;
             p2++;
      }
}
```

fold_image

```
fold_image(array,brray,n)
float *array,*brray;
int n;
{
      int ioff, joff,i,j;
      register float *p1;

      p1 = brray;
      for (i=n/2; i<n+n/2; i++) {
             ioff = i%n;
             for (j=n/2; j<n+n/2; j++) {
                    joff = j%n;
                    *p1++ = array[ioff*n+joff];
             }
      }
  }
```

Figure 2.1 (contd)

The value of **param_type** indicates the type of value the user is expected to enter. CIPE provides keywords for the programmer's use to express the type. Figure 2.2 is a list of **cipedef**'s valid type keywords.

INPUT_SYMBOL		
OUTPUT_SYMBOL		
STRING		
INT	-	**scalar and vector**
FLOAT	-	**scalar and vector**
DOUBLE	-	**scalar and vector**
BOOL	-	**scalar and vector**
INT_ARRAY	-	**two-dimensional**
FLOAT_ARRAY	-	**two-dimensional**
DOUBLE_ARRAY	-	**two-dimensional**

Figure 2.2 Valid Type Keywords

A *symbol*, as in INPUT_SYMBOL and OUTPUT_SYMBOL, is CIPE's means of representing data and will be discussed in Section 2.3. STRING may be any type of string. INT, FLOAT, DOUBLE, and BOOL may represent single values of that type, or a one-dimensional array of values of that type. INT_ARRAY, FLOAT_ARRAY, and DOUBLE_ARRAY permit two-dimensional arrays of that type.

The variable **param_storage** is a pointer to the location where the user's input is to be placed. The space pointed to by **param_storage** must have been previously allocated by the programmer. CIPE is unable to verify that the programmer has provided adequate space. When **cipepar** is called, the user's input is placed at the specified location.

The value needed by **num_of_elements** is dependent upon the **param_type** specified. In the case of an INPUT_SYMBOL, OUTPUT_SYMBOL, or STRING, **num_of_elements** should be the maximum number of characters that the user is permitted to give. The combined length of **prompt** and the user's input string must be less than seventy characters. In the following example of a **cipedef** command for an INPUT_SYMBOL, the user is permitted to give an input symbol name of up to 32 characters.

```
typedef cipe_sym_name char[32];     /* defined in cipeappl.h */

cipe_sym_name in;

cipedef(1,"input name", INPUT_SYMBOL, in, sizeof(in), REQ);
```

In the case of INT, FLOAT, DOUBLE, and BOOL, **num_of_elements** is expected to be the number of integers, floats, doubles, or Booleans, expected of the user. The value for **num_of_elements** may not exceed ten since that is the maximum number that will fit in a parameter window. A **cipedef** command for two INTs might look like:

```
int window[2];

cipedef(1,"filter window(nlw,nsw)", INT, window, 2, REQ);
```

In the case of INT_ARRAY, FLOAT_ARRAY, or DOUBLE_ARRAY, two values are expected for **num_of_elements**, one for the number of rows and one for the number of columns. The parameter window again imposes the limit on the number of rows and columns. Arrays may have up to ten rows and seven columns. The argument declarations and **cipedef** command for an INT_ARRAY might look like:

```
int *weights;
int window[2];

weights = (int *)malloc(sizeof(int) * window[0] * window[1]);
cipedef(2, "weights", INT_ARRAY, weights, window[0], window[1], REQ);
```

The last value passed to **cipedef** indicates whether a user is required to provide a value where one is requested. If the value is required, the keyword REQ should be used. If default values are available and a user is not required to provide a value, the keyword DEF should be used. If DEF is used, a programmer should store the default values in **param_storage**. There is no means of ensuring that a programmer has done this. When the parameter entry screen in menu mode appears the default value will appear beside the prompt. The user may leave the default if he/she does not care to provide another value. If REQ is used, a user will not be allowed to complete the menu entry screen and continue running an application without providing a value for the parameter. In CLI mode a user must provide values for all parameters whether defaults are available or not. This is because the parser parses in order and is unable to skip parameters.

2.2.2. Use of *cipepar*

The ultimate task of **cipepar**, in either user mode, is to store a user's input in the provided storage locations so it is available to the application. In menu mode, **cipepar** also creates a parameter entry screen for the user of the prompts provided to **cipedef** and blanks corresponding in length to the **num_of_elements** the user may input. An application may provide multiple parameter entry screens by multiple sets of **cipedef** and **cipepar** calls.

The **cipepar** routine provides some user input error checking for the programmer. For all parameter types except strings, **cipepar** is able to verify that the user has given the proper type of input. For parameters declared to be INPUT_SYMBOLs, **cipepar** checks to make sure the symbol exists, and issues an error if it does not. For parameters declared to be OUTPUT_SYMBOLs, **cipepar** checks to see if the symbol exists and issues a warning if it does. If the user chooses to ignore the warning the old symbol will be overwritten.

Cipepar also allows the programmer to provide his/her own error checking instead of using that provided by CIPE. Moreover, **cipepar** allows programmers to provide a user help for parameter input. If a programmer chooses to use only **cipepar**'s provided error checking and chooses not to provide help, the keywords NOECHK and NOHELP may be given to **cipepar** to indicate the absence of any specialized error checking or help routine(s).

```
if (cipepar(NOECHK, NOHELP) == -1) return(-1);
```

If a programmer desires to provide error checking or help, he/she must provide error checking and help routines. These routines must also be declared. The routines should include a switch statement which switches on the parameter numbers, as specified in **cipedef**, to provide error checking or help as desired.

If a programmer provides an error checking routine, it is necessary for it to perform all the error checking desired, including that which **cipepar** normally provides. The routines **cipepar** uses to do its error checking are available to the programmer to use. The routine **cipe_error_read_symbol** takes a symbol name as input and checks to see if it is present. If the symbol does not exist it returns -1 and prints an error message on the user's parameter screen. Typically, **cipe_error_read_symbol** would be used for INPUT_SYMBOLs.

```
typedef cipe_sym_name char[32];          /* defined in cipeappl.h */

cipe_sym_name sname;

ok = cipe_error_read_symbol(sname);
```

The routine **cipe_warn_write_symbol** also takes a symbol name as input and checks for its existence. Typically **cipe_warn_write_symbol** would be used for OUTPUT_SYMBOLs. If the symbol exists it prints a warning message. No error is returned if the symbol exists because it is conceivable that a user could choose to write one symbol over another. If a user chooses to ignore the warning, the output symbol will be overwritten.

```
typedef cipe_sym_name char[32];          /* defined in cipeappl.h */

cipe_sym_name sname;

ok = cipe_warn_write_symbol(sname);
```

All other error checking the programmer must provide for explicitly. As the user enters each parameter, **cipepar** does its own checking or goes to the error routine provided. If the error routine returns a -1 for that parameter, **cipepar** deletes the entry and does not allow the user to finish the parameter entry screen until a valid value is entered or the default, if provided, is accepted.

A help routine simply switches on the parameter number and provides help through **printf** statements. If the user is in menu mode and requests help the print statements will appear on the bottom of the parameter entry screen. CLI mode assumes a very knowledgeable user and does not provide a way for the user to see the help message provided in a help routine. Figure 2.3 provides sample help and error routines.

See the *User Interface* section of Figure 2.1 for an example of the use of **cipedef** and **cipepar** which have been explained. The parameter entry screen in Figure 2.4 is that produced by the calls to **cipedef** and **cipepar** in Figure 2.1.

2.3. Data I/O

CIPE employs a symbolic data representation where a dataset is identified with a name, and the name and the dataset's attributes (data dimensions, data type, etc.) are stored in a symbol structure. The symbol structure is declared in the include file *cipeappl.h*, discussed in Section 2.1, which may be found in *Appendix A*. The use of names and attribute structures allows programmers to access datasets in a consistent manner and allows users to manipulate datasets interactively with a minimum of file transactions. Symbols may be created by associating a file name with a symbol (read A from "filename"), by copying an existing dataset (B=A), by assigning a set of values (C={2,5}), or by activating an application (D=func(A)). If a symbol is created by associating a file name with a symbol, the attributes of the dataset are read from the file header and stored in the symbol structure. If a symbol is created by copying an existing data set, the output symbol is assigned the same attributes as the input symbol. If a symbol is created as the output of a function, it receives its attributes from the input symbol and the function applied to it. In addition to the symbol name and symbol attributes, the data associated with a symbol name may also be stored in the symbol structure. CIPE makes an effort wherever possible to limit file transactions. Therefore, the data associated with a symbol name is not stored in a symbol structure until a user's actions require that it be there. As other attributes of a symbol become useful, they will be described in more detail.

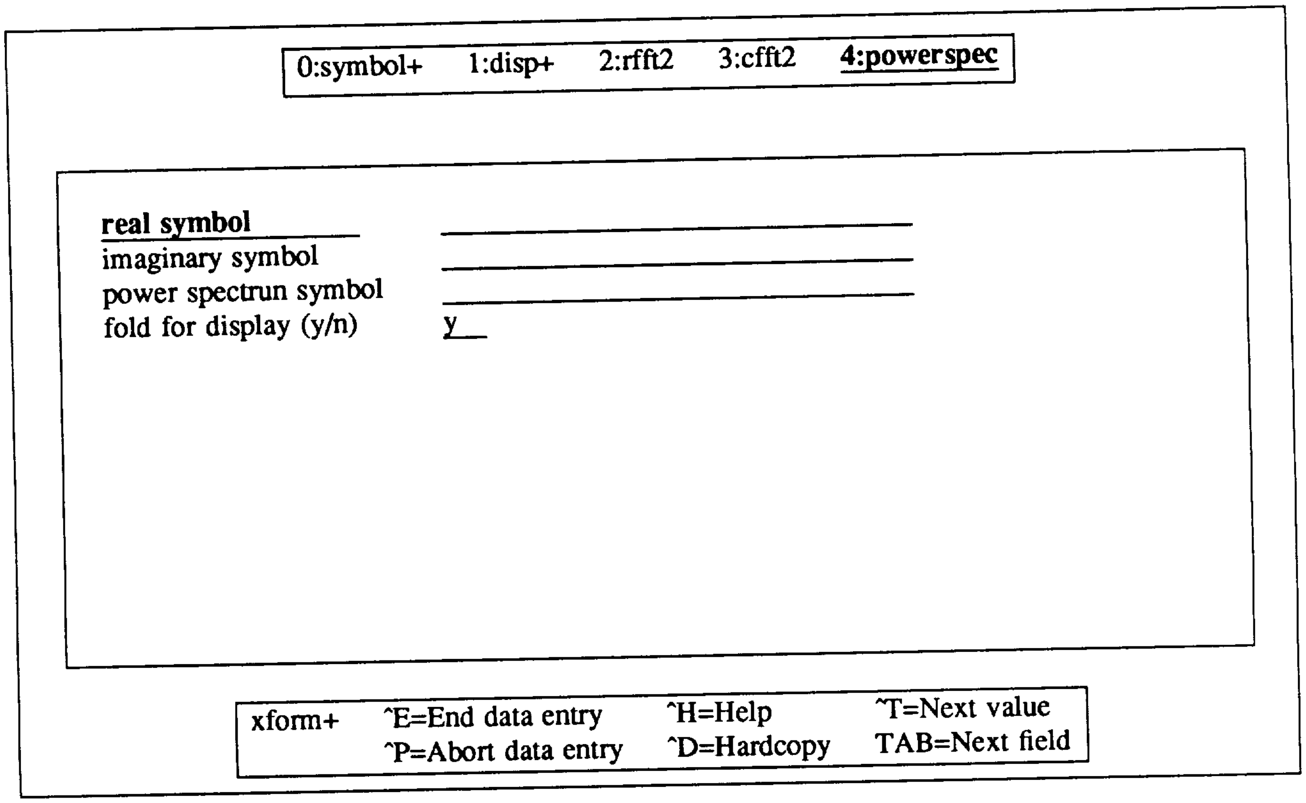
```
int error_routine(), help_routine();

cipepar(error_routine(), help_routine());

error_routine(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
   int error,whereis;
   error = 0;
   switch(number){
       case 0: error = cipe_error_read_symbol(value.s);
               break;
       case 1: error = cipe_warn_write_symbol(value.s);
               break;
       case 2: if (((value.i) % 2) == 0) error = -1;
               if (error) printf("must be odd number");
               break;
   }
   return(error);
}

help_routine(params,number)
struct param *params;
int number;
{
   switch (number) {
       case 0:
          printf("A valid input symbol must be associated");
          printf("with two dimensional image data.");
          break;
       case 1:
          printf("A valid output symbol must not already exist.");
          printf("You may see if it does by using list from the");
          printf("Symbol Menu.");
          break;
       case 2:
          printf("Dimensions of the filter window must be odd ");
          printf("numbers.");
          break;
   }
}
```

Figure 2.3 Sample Help and Error Routines



0:symbol+ 1:disp+ 2:rfft2 3:cfft2 4:powerspec

real symbol	______________
imaginary symbol	______________
power spectrun symbol	______________
fold for display (y/n)	y_

```
xform+   ^E=End data entry     ^H=Help       ^T=Next value
         ^P=Abort data entry   ^D=Hardcopy   TAB=Next field
```

Figure 2.4 Parameter Entry Screen

Since CIPE provides a user with the ability to associate data with a symbol name, thereby creating a symbol, an applications programmer can assume that a user will provide a symbol name for input data. As was already explained in Section 2.2.2, CIPE will automatically check for the existence of the symbol if default user input error checking is used, or the programmer can use CIPE provided routines to check for a symbol's existence explicitly. Knowing that the symbol exists, the programmer needs only to retrieve it. Symbol retrieval is accomplished by use of the command, **cipe_get_symbol.**

```
typedef cipe_sym_name char[32];

cipe_sym_name       input_symbol_name;
struct cipe_symbol  *input_symbol_ptr;

input_symbol_ptr = cipe_get_symbol( input_symbol_name );
```

The routine, **cipe_get_symbol,** takes the name of a symbol and returns a pointer to the symbol of that name. In the event of an error **cipe_get_symbol** returns NULL.

Output symbol names may be existing or new symbol names. If a user gives a name for an output symbol, and the name is already in use, the user is warned. If the user chooses not to heed the warning, the old symbol will be overwritten with a new one. If an old symbol name is used, its attributes will need to be modified to reflect the new data it will store. If a new symbol name is used, the symbol must be created. The creation of symbols is accomplished by the commands **cipe_create_symbol** and **cipe_create_symbol_and_data.** If a symbol's data area is to be used in the application program it is necessary to create the symbol with the command **cipe_create_symbol_and_data.** This is typically the situation in application programs which do not use a coprocessor.

```
typedef cipe_sym_name char[32];

cipe_sym_name       output_symbol_name;
struct cipe_symbol  *output_symbol_ptr;
int                 number_of_lines, number_of_samples,
                    number_of_bands, type_of_data;

output_symbol_ptr = cipe_create_symbol_and_data(output_symbol_name,
        number_of_lines, number_of_samples, number_of_bands,
        type_of_data);
```

The routine **cipe_create_symbol_and_data** takes a symbol name, the dimension in each of three directions of the data to be associated with the symbol, and a data type, and returns NULL in the event of an error.

Commonly, the dimensions of data for an output symbol will be the same as those for the input symbol, though this is not always the case.

The variable **type_of_data** should be one of following keywords: **CHAR_TYPE, SHORT_TYPE, INT_TYPE, FLOAT_TYPE, DOUBLE_TYPE**, according to the kind of data the symbol is expected to store.

If for any reason the symbol's data area is not to be used by the host program, then the command **cipe_create_symbol** may be used. This creates space for all the symbol's attributes but does not create space for the data. **Cipe_create_symbol** takes the same attributes as **cipe_create_symbol_and_data**, and also returns NULL on error.

The routines **cipe_create_symbol_and_data** and **cipe_create_symbol** may also be used to create temporary symbols for a programmer's use. Temporary symbols may be deleted by the command **cipe_delete_symbol**.

```
typedef cipe_sym_name char[32];

cipe_sym_name symbol_name;

ok = cipe_delete_symbol(symbol_name);
```

The routine **cipe_delete_symbol** takes a symbol name as an argument and returns -1 if it was unable to delete the symbol. Programmers should only delete temporary symbols created for their own purposes. Application programs should not delete user symbols. CIPE provides for users the ability to delete symbols when they no longer need them.

See the *Data I/O* section of Figure 2.1 for an example of the use of **cipe_get_symbol** and **cipe_create_symbol_and_data**.

2.4. Data Processing

With all the necessary parameters received through the user interface, and the data I/O complete, only the data processing code remains to be written. CIPE provides a set of macros to access symbol attributes which the applications programmer may find useful for writing the data processing code. The attributes of the symbol structure were assigned values upon creation of the symbol. Figure 2.5 is a list of the available macros. The symbol attribute definitions CIPE provides can make it easier for a programmer to write a new application. For an example of a data processing subroutine which uses the macros, see the *Data Processing* section of Figure 2.1.

CIPEDATA(symb_ptr)	Pointer to the data area of the symbol
CIPENAME(symb_ptr)	Pointer to the name of the symbol
CIPEFILE(symb_ptr)	Pointer to the name of the file (if any) associated with the symbol
CIPENDIM(symb_ptr)	Number of dimensions of the data
CIPESB(symb_ptr)	Starting band of the data
CIPENB(symb_ptr)	Number of bands in the data
CIPESL(symb_ptr)	Starting line of the data
CIPENL(symb_ptr)	Number of lines in the data
CIPESS(symb_ptr)	Starting sample of the data
CIPENS(symb_ptr)	Number of samples in the data
CIPEDATATYPE(symb_ptr)	Type of data elements, e.g. INT_TYPE, CHAR_TYPE
CIPEELEMENTSIZE(symb_ptr)	Size of an element of the data
CIPENUMELEMENT(symb_ptr)	Total number of data elements
CIPEDATASIZE(symb_ptr)	Total size of the data area

Figure 2.5 Symbol Attribute Definitions

2.5. Compiling

A sample makefile for an application program without a coprocessor has been provided below in Figure 2.6. The CIPEDIR given is the location of CIPE as of the writing of this manual.

```
CC=cc
CIPEDIR=/judy/ufs/cipe
CIPE = $(CIPEDIR)/cipe
LINK= $(CIPEDIR)/appl/CPLINK

all: function1

function1: function1.o subfunction.o $(CIPE)
        $(LINK) $(CIPE) function1.o subfunction.o
```

Figure 2.6 Application Without a Coprocessor Makefile

3. PROGRAMS WITH A COPROCESSOR

Programs that run with the assistance of a coprocessor require two coordinating pieces of code to be written, one to run in the host, and one to run in the coprocessor. As the hypercube is the only attached processor fully implemented in CIPE, it is the only one which will be described here. CIPE's programming environment is designed so that the host module interacts with the hypercube module as a main routine interacts with a subroutine in generic programming. In other words, the host module passes the data and parameters to the hypercube module, and the hypercube module processes the data and returns the results.

3.1. Host Module

The host module of an application program with a coprocessor is identical to the application program without a coprocessor with respect to user interface and symbol retrieval/creation. Data processing will be given to the hypercube module, and the host module will be responsible for transferring the data and parameters to the hypercube module. Appendix E provides a reference for all host resident executive subroutines.

3.1.1. User Interface

The user interface is identical in application programs with and without coprocessors. For this reason, no discussion of the user interface is provided here. Refer to Section 2.2 for further information.

3.1.2. Data I/O

Programming using a hypercube as a coprocessor requires two types of data I/O: symbol creation and retrieval in the host, and data distribution and retrieval with the hypercube. Input symbols may be retrieved by using **cipe_get_symbol**, and output symbols may be created by using **cipe_create_symbol** or **cipe_create_symbol_and_data**. Explanations and examples of the use of these routines have been provided in Section 2.3. Applications with a coprocessor, unlike those without, typically do not need a symbol to have data space in the host since the data processing will be done within the coprocessor. Therefore, it will be most efficient to create a symbol using **cipe_create_symbol** instead of **cipe_create_symbol_and_data**.

In order to download data to the hypercube a programmer must choose a decomposition for the data which is appropriate for the problem being solved. Seven data types are supported in CIPE; they are listed below in Figure 3.1 with the keywords necessary to achieve them.

BCAST_DIST	**A broadcast distribution**
HORIZ_DIST	**A horizontal distribution**
HORIZ_OVERLAP	**A horizontal distribution with overlap**
GRID_DIST	**A grid distribution**
GRID_OVERLAP	**A grid distribution with overlap**
VERT_DIST	**A vertical distribution**
VERT_OVERLAP	**A vertical distribution with overlap**
CUSTOM_DIST	**A customized distribution, anything goes!**

Figure 3.1 Distribution Type Keywords

For all symbols which will be passed to the hypercube module, the programmer must choose a distribution. For all but the CUSTOM_DIST this is accomplished by a call to *cipe_compose_standard_loadmap*. There are a variable number of arguments to this routine, depending on the chosen distribution. The routine returns -1 on error. The call syntax and its arguments' declarations are illustrated below.

```
struct cipe_symbol *symbol_ptr;

ok = cipe_compose_standard_loadmap(symbol_ptr, distribution, ...);
```

If a BCAST_DIST, HORIZ_DIST, GRID_DIST, or VERT_DIST is desired **cipe_compose_standard_loadmap** requires only two arguments. The first argument, **symbol_ptr**, can be obtained by a call to **cipe_get_symbol**, **cipe_create_symbol**, or **cipe_create_symbol_and_data** (See Section 2.3). The argument **distribution** must be one of the previously given distribution type keywords. A request for a horizontal distribution might look as follows.

```
struct cipe_symbol *symbol_ptr;

ok = cipe_compose_standard_loadmap(symbol_ptr, HORIZ_DIST);
```

If an overlapped distribution of any type is requested it is necessary to specify the number of lines of overlap in the appropriate planes. For a HORIZ_OVERLAP one integer is expected following the distribution to represent the number of lines of overlap in the vertical plane. There is no way of requesting a different number of lines of overlap up and down. A valid request for a HORIZ_OVERLAP, with three lines of overlap up and down might look like:

```
int ok;
struct cipe_symbol *symbol_ptr;

ok = cipe_compose_standard_loadmap(symbol_ptr, HORIZ_OVERLAP, 3);
```

Similarly, a GRID_OVERLAP requires values for the number of lines of overlap in vertical and horizontal planes, and a VERT_OVERLAP requires values for the number of lines of overlap in the horizontal plane.

A custom distribution allows a programmer to create any distribution that CIPE has not provided. The creation of a CUSTOM_DIST requires the use of the routine **cipe_compose_custom_loadmap**. One call must be made to **cipe_compose_custom_loadmap** for each node of the hypercube. The routine requires a programmer to know exactly what data he/she wants in each node.

```
struct cipe_symbol symbol_ptr;
int procnum;
int starting_line, starting_sample, starting_band, number_of_lines,
   number_of_samples, number_of_bands;

ok = cipe_compose_custom_loadmap(symbol_ptr, procnum, starting_line,
   starting_sample, starting_band, number_of_lines, number_of_samples,
   number_of_bands):
```

The first argument, **symbol_ptr**, can be obtained by a call to **cipe_get_symbol**, **cipe_create_symbol**, or **cipe_create_symbol_and_data**. The second argument, **procnum**, should be the processor number whose data region is being specified. The last six arguments define the data region for the processor.

3.1.3. Parameter Passing and Node Module Initiation

CIPE provides a programmer a routine, **cipe_cube_param_def**, to define which parameters will be needed for data processing within the coprocessor. The arguments required allow CIPE to reduce the tasks a programmer must do by performing them for him/her.

```
int param_num, size;

cipe_cube_param_def(param_num, param_type, direction, size, storage_ptr);
```

The argument **param_num** must be an integer between 1 and 32. For each activation of a hypercube module no two parameters may have the same **param_num**. CIPE requires a programmer to label the first parameter 1 and then increase **param_num** by one for each consecutive call to

cipe_cube_param_def.

CIPE provides two keywords SYMBOL_PARAM and NON_SYMBOL_PARAM for a programmer to specify **param_type**. If the parameter being passed is a symbol, the keyword SYMBOL_PARAM should be used. Otherwise, the keyword NON_SYMBOL_PARAM should be used.

The **direction** of a parameter may be specified with the keywords INPUT, OUTPUT, or BOTH. INPUT indicates that a parameter will be used only as input to the data processing routine. OUTPUT indicates the parameter should have a value only after the data processing is complete. In this case a value will be sent back from each node of the hypercube after data processing. CIPE has no way of verifying that the programmer has provided sufficient space for this to happen. BOTH should be used to provide a distinct value to each node of the hypercube for input and to receive a distinct value back from each node of the hypercube. The BOTH option is available only for NON_SYMBOL_PARAMs.

The size of the parameter, in bytes, should be passed as the argument **size**.

The argument **storage_ptr** is a pointer to the parameter being defined.

A programmer requests activation of a host module's corresponding hypercube module by the command, **cipe_cube_execute_module**. It takes as its only argument the name of the hypercube module that is to be run. An example call to **cipe_cube_execute_module** might look like:

```
ok = cipe_cube_execute_module("/judy/ufs/cipe/appl/node_module");
```

When a programmer requests activation of a host module's corresponding hypercube module CIPE completes several tasks for the programmer. CIPE first checks to be certain that the programmer has chosen a data decomposition for symbols. It then creates space in the hypercube for each of the parameters specified. The load maps of all symbols are sent to the nodes. The data of input symbols is sent to the nodes. The values of non-symbol input parameters are sent to the nodes. And the values of non-symbol output parameters are uploaded after execution of the hypercube module. If any errors occur in these processes **cipe_cube_execute_module** returns a -1. A sample host module is shown in Figure 3.2.

3.1.4. Compiling and Linking

The process of compiling and linking a host module is nearly identical to the process required for an application program without a coprocessor, explained in Section 2.5. The only difference is that programs which run with a coprocessor need to be compiled with **ccc** instead of **cc** in order to resolve hypercube related subroutine calls. See the sample makefile in Figure 3.3.

```
/****************************** INCLUDE FILES ********************************* /
#include <stdio.h>
#include <ctype.h>
#include "cipeappl.h"
                                                                        powerspec
powerspec()
{
     cipe_sym_name  input_real, input_imagi, output;
     int  p;
     char  option[3];

     p = 1;
     strcpy(option,"y");
/****************************** USER INTERFACE ******************************** /
     cipedef (p++, "real symbol                ", INPUT_SYMBOL, input_real,
          sizeof(cipe_sym_name), REQ);
     cipedef (p++, "imaginary symbol           ", INPUT_SYMBOL, input_imagi,
          sizeof(cipe_sym_name), REQ);
     cipedef (p++, "power spectrun symbol ", OUTPUT_SYMBOL, output,
          sizeof(cipe_sym_name), REQ);
     cipedef (p++, "fold for display (y/n)", STRING, option,
          sizeof(option), DEFAULT);
     if (cipepar(NULL, NULL)) return(-1);

     if (strcmp(option,"y")==0)
          cube_power_spec(input_real,input_imagi,output,1);
     else
          cube_power_spec(input_real,input_imagi,output,0);
}
                                                                   cube_power_spec
cube_power_spec(real,imagi,power,disp)
cipe_sym_name  real,imagi,power;
int  disp;
{
     struct cipe_symbol *realp, *imagip, *powerp;
     float *power_disp_data, *p1;
/********************************* DATA I /O ********************************** /
     realp = cipe_get_symbol(real);
     imagip = cipe_get_symbol(imagi);
     powerp = cipe_create_symbol(power,CIPENL(realp),CIPENS(realp),1,FLOAT_TYPE);

     cipe_compose_standard_loadmap(realp,HORIZ_DIST);
     cipe_compose_standard_loadmap(imagip,HORIZ_DIST);
     cipe_compose_standard_loadmap(powerp,HORIZ_DIST);

     cipe_cube_param_def(1,SYMBOL_PARAM,INPUT,sizeof(cipe_sym_name),realp);
     cipe_cube_param_def(2,SYMBOL_PARAM,INPUT,sizeof(cipe_sym_name),imagip);
     cipe_cube_param_def(3,SYMBOL_PARAM,OUTPUT,sizeof(cipe_sym_name),powerp);

/****************************** DATA PROCESSING ****************************** /
     cipe_cube_execute_module("/judy/ufs/cipe/appl/xform/elt/power_spec");

     if (disp) {
          cipe_cube_read_data(powerp);
          cipe_delete_symbol_from_cube(powerp);
          power_disp_data = (float *)malloc(CIPEDATASIZE(powerp));
          if (power_disp_data == NULL) {
               printf("power_disp_data malloc failure\n");
               return(-1);
          }
                                                                ...cube_power_spec

          p1 = (float *)CIPEDATA(powerp);
          p1[0] = 0.0;          /* turn off the sum for better contrast * /
          fold_image (CIPEDATA(powerp),power_disp_data,CIPENL(powerp));
          free (CIPEDATA(powerp));
          CIPEDATA(powerp) = (unsigned char *)power_disp_data;
     }
}
```

Figure 3.2 Host Module, Application With a Coprocessor

```
CC=ccc
CIPEDIR=/judy/ufs/cipe
CIPE = $(CIPEDIR)/cipe
LINK= $(CIPEDIR)/appl/CPLINK

all: function1

function1: function1.o subfunction.o $(CIPE)
        $(LINK) $(CIPE) function1.o subfunction.o
```

Figure 3.3 Application With a Coprocessor Makefile

3.2. Hypercube Module

The hypercube module of a CIPE application needs to accomplish two tasks, retrieving the data processing parameters defined by the host module and processing the data. Appendix F provides a reference for all hypercube resident executive subroutines.

3.2.1. Include Files

All hypercube modules of CIPE application programs must include the file *eltappl.h*, found in /usr/local/include. This file provides the necessary information to use CIPE-provided routines and definitions. A copy of this file has been include in Appendix B for programmer reference. As in the host module, any other include files are at the programmer's discretion. See the *Include Files* section of Figure 3.5 for an example.

3.2.2. Parameter Retrieval

When the host module requests activation of the hypercube module the parameters needed for data processing are sent to a monitor in the hypercube. It is necessary for the hypercube module to request the retrieval of those parameters from the hypercube monitor. Application parameters necessary for data processing may be retrieved by using the command **elt_get_param.**

```
int param_num;

elt_get_param(param_num,address_of_ptr);
```

The first argument, **param_num,** is a parameter number. For a given parameter, the programmer needs to give the same number in the host and hypercube modules in order to retrieve the parameter.

The second argument to **elt_get_param, address_of_ptr**, is an address of a pointer. When the data processing parameters were sent to the hypercube monitor, the monitor created space for them. Rather than creating additional space, **elt_get_param** stores the address of the parameter requested in **address_of_ptr** so the application can use the parameter where it resides.

3.2.3. Data Processing

As in the host, macros have been defined for use within the hypercube to assist the programmer in accessing symbol attributes. See Figure 3.4 for a list of these macros.

DATA(symbol)	**Data are of the symbol**
NAME(symbol)	**Name of the symbol**
DATATYPE(symbol)	**Type of data elements, e.g. INT_TYPE, CHAR_TYPE**
SB(symbol)	**Starting band of data in the node**
NB(symbol)	**Number of bands of data in the node**
SL(symbol)	**Starting line of data in the node**
NL(symbol)	**Number of lines of data in the node**
SS(symbol)	**Starting sample of data in the node**
NS(symbol)	**Number of samples of data in the node**
ELEMENTSIZE(symbol)	**Size of an element of the data**
NUMELEMENTS(symbol)	**Total number of data elements in the node**
DATASIZE(symbol)	**Total size of data in the node**

Figure 3.4 Hypercube Symbol Attribute Definitions

The programmer can also create temporary symbols for his/her use. Symbols may be created by the commands **elt_create_symbol** and **elt_create_symbol_and data**. The commands take the same arguments as their counterparts in the host and function similarly. The routine **elt_create_symbol** creates a symbol with no data space. The routine **elt_create_symbol_and_data** creates a symbol and data space. The former, **elt_create_symbol**, is not likely to be used since a symbol is usually created temporarily in order to use its data space. Again, similar to the host, a temporary symbol may be deleted with the command, **elt_delete_symbol**. A programmer should only delete temporary symbols created for their own purposes. User created symbols should not be deleted but left for the user to determine when they should be deleted.

Very little should be required to make a data processing subroutine which runs with CIPE in the host run within the hypercube. All that should be required is to change any symbol attribute macros which were used to their equivalent macros within the cube. A sample hypercube module is shown in Figure 3.5.

```
/**************************** INCLUDE FILES ********************************** /
#include <stdio.h>
#include <ctype.h>
#include "eltappl.h"

power()                                                                          power
{
      struct elt_symbol *realp, *imagip, *powerp;
      int call_wtk_power_spec();

/************************* PARAMETER RETRIEVAL ******************************* /
      elt_get_param(1,&realp);
      elt_get_param(2,&imagip);
      elt_get_param(3,&powerp);
      printf("%s  %s  %s\n",NAME(realp),NAME(imagip),NAME(powerp));

/**************************** DATA PROCESSING ******************************** /
      power_spec(DATA(realp), DATA(imagi), DATA(power), NL(realp), NS(realp));


}

power_spec(real,imagi,power,nl,ns)                                               power_spec
float *real,*imagi, *power;
int nl,ns;
{
      register float *p1,*p2, *p3;
      int i;
      double temp;

      p1 = real;
      p2 = imagi;
      p3 = power;
      for (i=0; i<nl*ns; i++) {
            temp = (*p1 * *p1) + (*p2 * *p2);
            *p3++ = temp;
            p1++;
            p2++;
      }
}
```

Figure 3.5 Hypercube Module, Application With a Coprocessor

3.2.4. Data Processing Using the Weitek

To use the Weitek floating point processors for data processing requires a little more work. Using the Weitek ultimately requires three separate files, one each to run in the host, the hypercube's data processor, and the hypercube's Weitek processor. The host file does not need to change to call the Weitek. The data processor code will be the code described in previous sections as the hypercube module, but it will now lack the subroutine that actually does the data processing. The data processing subroutine itself now needs to be placed in a third file which will be run in the Weitek.

The Weitek file needs to include the file *wtkdefs.h*. Arguments are passed from the hypercube's standard data processor to the Weitek via an argument list. For this reason a small subroutine must be created to call the data processing subroutine and pass the pieces of the argument list to the data processing subroutine as it is expecting them. For the following data processing subroutine and arguments:

```
filt_sub(sin,sout,window,weight)
struct elt_symbol *sin, *sout;
int *weight, *window;
{ }
```

the following calling subroutine would have to be created:

```
call_filt_sub(arg)
struct {
   struct elt_symbol *sin, *sout;
   int *weight, *window;
} *arg;
{
filt_sub(arg->sin, arg->sout, arg->weight, arg->window);
}
```

In the data processor code, the calling subroutine would have to be declared as an integer, and then it would have to be called, with all of its parameters, by the use of the command **call_wtk**. The additions to the data processor code for the previous Weitek example would look like:

```
int call_filt_sub();

call_wtk(call_filt_sub, sin, sout, window, weight);
```

The sample data processor and Weitek code may be found in Figures 3.6 and 3.7.

power

```
/****************************** INCLUDE FILES ********************************** /
#include <stdio.h>
#include <ctype.h>
#include "eltappl.h"

power()
{
      struct elt_symbol *realp, *imagip, *powerp;
      int call_wtk_power_spec();

/************************** PARAMETER RETRIEVAL ******************************** /
      elt_get_param(1,&realp);
      elt_get_param(2,&imagip);
      elt_get_param(3,&powerp);
      printf("%s %s %s\n",NAME(realp),NAME(imagip),NAME(powerp));

/**************************** DATA PROCESSING ********************************** /
      callwtk(call_wtk_power_spec,DATA(realp),DATA(imagip),DATA(powerp),
            NL(realp),NS(realp));
}
```

Figure 3.6 Hypercube Module, Data Processor Code Calling Weitek

call_wtk_power_spec

```
call_wtk_power_spec(arg)
struct {
      float *array;
      float *brray;
      float *crray;
      int nl;
      int ns;
} *arg;

{
      power_spec(arg->array,arg->brray,arg->crray,arg->nl,arg->ns);
}
```

power_spec

```
power_spec(real,imagi,power,nl,ns)
float *real,*imagi, *power;
int nl,ns;
{
      register float *p1,*p2, *p3;
      int i;
      double temp;

      p1 = real;
      p2 = imagi;
      p3 = power;
      for (i=0; i<nl*ns; i++) {
            temp = (*p1 * *p1) + (*p2 * *p2);
            *p3++ = temp;
            p1++;
            p2++;
      }
}
```

Figure 3.7 Hypercube Module, Weitek Code

3.2.5. Compiling and Linking

The hypercube module shown in Figure 3.5, which does not use the Weitek, may be compiled and linked with the following makefile shown in Figure 3.8. It uses cc68 which links in the necessary hypercube libraries. The location specified in CIPEDIR is the location of CIPE as of the writing of this manual. As stated earlier, CIPE is currently implemented with a Sun-4 as a host to the hypercube. Cross-compilers on the Sun-4 enable it to generate 68020/Sun-3 code to run in the hypercube.

```
CC = cc68
CIPEDIR = /judy/ufs/cipe
CIPE = $(CIPEDIR)/cipe

ELT_MONITOR = $(CIPEDIR)/cube/elt_monitor/elt_mon
LINK = $(CIPEDIR)/appl/ELTLINK
TARGET_ARCH = -sun3
CFLAGS = -g

function1: function1.o subfunction.o $(ELT_MONITOR)
        $(LINK) -v -e entryname -o outputname
        $(ELT_MONITOR) function1.o subfunction.o
```

Figure 3.8 Hypercube Module Without Weitek Makefile

The data processor and Weitek modules shown in Figure 3.6 and 3.7 may be compiled and linked with the following makefile shown in Figure 3.9. It uses cc68 to compile the data processor code and w68 to compile the Weitek code.

```
CC = cc68
W68 = w68
CIPEDIR = /judy/ufs/cipe
CIPE = $(CIPEDIR)/cipe

ELT_MONITOR = $(CIPEDIR)/cube/elt_monitor/elt_mon
LINK = $(CIPEDIR)/appl/ELTLINK
TARGET_ARCH = -sun3
CFLAGS = -g
WFLAGS = -O

dpfunction: dpfunction.o wtk_function.o $(ELT_MONITOR)
        $(LINK) -v -e entryname -o outputname
        $(ELT_MONITOR) dpfunction.o wtk_function.o

wtkfunction.o: wtkfunction.c
        $(W68) -c $(WFLAGS) wtkfunction.c
```

Figure 3.9 Hypercube Module With Weitek Makefile

4. PROGRAMS WITH AND WITHOUT A COPROCESSOR

It may serve a programmer best to combine some of the information provided and write one program which can run either with or without a coprocessor. Figure 4.1 accomplishes the same tasks as the sample programs in both Figures 2.1 and 3.2.

The only new piece of information necessary is the use of the definition **COPROCESSOR** to determine if the user has activated the cube. If the cube is activated the application assumes it is to be used and calls the same node program as that in Figure 3.5. If the cube has not been activated the application is run entirely in the host.

```
/****************************** INCLUDE FILES ********************************* /
#include <stdio.h>
#include <ctype.h>
#include "cipeappl.h"

powerspec()                                                          powerspec
{
     cipe_sym_name input_real, input_imagi, output;
     int p;
     char option[3];

     p = 1;
     strcpy(option,"y");

/****************************** USER INTERFACE ******************************* /
     cipedef (p++, "real symbol               ", INPUT_SYMBOL, input_real,
          sizeof(cipe_sym_name), REQ);
     cipedef (p++, "imaginary symbol          ", INPUT_SYMBOL, input_imagi,
          sizeof(cipe_sym_name), REQ);
     cipedef (p++, "power spectrun symbol ", OUTPUT_SYMBOL, output,
          sizeof(cipe_sym_name), REQ);
     cipedef (p++, "fold for display (y/n)", STRING, option,
          sizeof(option), DEFAULT);
     if (cipepar(NULL, NULL)) return(-1);

     if (COPROCESSOR == CUBE_KWD)
          if (strcmp(option,"y")==0)
               cube_power_spec(input_real,input_imagi,output,1);
          else
               cube_power_spec(input_real,input_imagi,output,0);
     else
          if (strcmp(option,"y")==0)
               host_power_spec(input_real,input_imagi,output,1);
          else
               host_power_spec(input_real,input_imagi,output,0);
}

host_power_spec(input_real,input_imagi,output,disp)            host_power_spec
cipe_sym_name *input_real,*input_imagi,*output;
int disp;
{
     struct cipe_symbol *sin_real, *sin_imagi, *sout;
     float *power_disp_data, *p1;

/******************************** DATA I/O ********************************** /
     sin_real = cipe_get_symbol (input_real);
     sin_imagi = cipe_get_symbol (input_imagi);
     sout = cipe_create_symbol_and_data(output,
          CIPENL(sin_real),CIPENS(sin_real),1,FLOAT_TYPE);
```

Figure 4.1 Host Module, Application With and Without a Coprocessor

PRECEDING PAGE BLANK NOT FILMED

PAGE 28 INTENTIONALLY BLANK

```
/******************************* DATA PROCESSING ******************************* /
     power_spec(CIPEDATA(sin_real),CIPEDATA(sin_imagi),CIPEDATA(sout),
          CIPENL(sin_real),CIPENS(sin_real));

     if (disp) {
          power_disp_data = (float *)malloc(CIPEDATASIZE(sout));
          if (power_disp_data == NULL) {
               printf("power_disp_data malloc failure\n");
               return(-1);
          }
          pl = (float *)CIPEDATA(sout);
                                                                        ...host_power_spec
          pl[0] = 0.0;          /* turn off the sum for better contrast * /
          fold_image (CIPEDATA(sout),power_disp_data,CIPENL(sout));
          free (CIPEDATA(sout));
          CIPEDATA(sout) = (unsigned char *)power_disp_data;
     }
}

cube_power_spec(real,imagi,power,disp)                                  cube_power_spec
cipe_sym_name real,imagi,power;
int disp;
{
     struct cipe_symbol *realp, *imagip, *powerp;
     float *power_disp_data, *pl;

/********************************* DATA I/O *********************************** /
     realp = cipe_get_symbol(real);
     imagip = cipe_get_symbol(imagi);
     powerp = cipe_create_symbol(power,CIPENL(realp),CIPENS(realp),1,FLOAT_TYPE);

     cipe_compose_standard_loadmap(realp,HORIZ_DIST);
     cipe_compose_standard_loadmap(imagip,HORIZ_DIST);
     cipe_compose_standard_loadmap(powerp,HORIZ_DIST);

     cipe_cube_param_def(1,SYMBOL_PARAM,INPUT,sizeof(cipe_sym_name),realp);
     cipe_cube_param_def(2,SYMBOL_PARAM,INPUT,sizeof(cipe_sym_name),imagip);
     cipe_cube_param_def(3,SYMBOL_PARAM,OUTPUT,sizeof(cipe_sym_name),powerp);

/******************************* DATA PROCESSING ****************************** /
     cipe_cube_execute_module("appl/xform/elt/power_spec");

     if (disp) {
          cipe_cube_read_data(powerp);
          cipe_delete_symbol_from_cube(powerp);
          power_disp_data = (float *)malloc(CIPEDATASIZE(powerp));
          if (power_disp_data == NULL) {
               printf("power_disp_data malloc failure\n");
               return(-1);
          }
          pl = (float *)CIPEDATA(powerp);
          pl[0] = 0.0;          /* turn off the sum for better contrast * /
          fold_image (CIPEDATA(powerp),power_disp_data,CIPENL(powerp));
          free (CIPEDATA(powerp));
          CIPEDATA(powerp) = (unsigned char *)power_disp_data;
     }
}
```

Figure 4.1 (contd)

5. TESTING PROGRAMS IN CIPE

A programmer may test his/her application programs in either CLI or menu mode. In CLI mode a user needs to specify a pathname to a function, and then the arguments required by the function. A CLI call for a filter function might look like:

```
/judy/ufs/winnie/cipe/appl/filter(A,B,3,3,{1,1,1,1,1,1,1,1,1});
```

In menu mode a programmer needs to choose the **Bltin** option from the top level menu, and **myfunc** from the following menu. The option **myfunc** will prompt the programmer for an application with full pathname. When this has been given the programmer will receive the parameter entry screens provided for in his/her application.

If a programmer wants to repeatedly test a new application it might be useful to temporarily add the function to menu or CLI mode rather than having to always type a full path name. A function may be temporarily added to menu mode by choosing the **Bltin** option from the top level menu and **add_func** from the following menu. The added function may then be accessed by once again choosing the **Bltin** option from the top level menu, and then choosing myfunc and providing a function name without a full pathname. A function may be temporarily added to CLI mode by the command:

```
add_func("function","path","help");
```

Once added the function may be accessed like any other CLI function by simply using the function name and arguments. Temporary additions to CLI mode or menu mode endure only for the length of the CIPE session.

If a programmer will need to continuously test an application in distinct CIPE sessions it may be most convenient to keep a personal menu configuration file and function dictionary so temporary additions to CLI or menu mode do not have to be added during each CIPE session. Copies of CIPE's default versions of these files are provided in Appendices C and D. The functions referred to by a programmer's menu configuration file must be present in the dictionary. For detailed information about these files refer to the *CIPE User's Manual*. The **setup** option in CIPE's top level menu allows the programmer to specify the locations of personal menu configuration and function dictionary files. For the new menu configuration to be used the programmer must exit to CLI mode and re-enter the menu mode by typing menu.

An applications programmer may provide traces in code by using the keyword APPL_TRACE. APPL_TRACE may be turned on by typing the following in CLI mode:

turn on appl trace

In menu mode, the **setup** option in the top level menu provides an opportunity to turn APPL_TRACE on.

This manual has attempted to provide all the information necessary for a programmer to write and test applications to run with CIPE. Appendices of application include files, *cipeappl.h* and *eltappl.h*, a menu configuration file, a function dictionary, and programmer callable routines have been provided for programmer reference.

Appendix A – INCLUDE FILE cipeappl.h

```
#ifndef CIPEAPPL_H
#define CIPEAPPL_H

/* data type definitions */
#ifndef DATATYPE_H
#define DATATYPE_H

#define UNDEF_TYPE        0
#define CHAR_TYPE         1
#define BYTE_TYPE         1
#define SHORT_TYPE        2
#define INT_TYPE          3
#define FLOAT_TYPE        4
#define DOUBLE_TYPE       5
#define BOOL_TYPE         6
#define STRING_TYPE       7
#endif DATATYPE_H

/* symbol structure and its definitions */
#ifndef SYMBOL_H
#define SYMBOL_H
#define CIPEFILESIZE      45
#define CIPENAMESIZE      32
#define FUNCNAMESIZE      45

typedef char cipe_file_name[CIPEFILESIZE];
typedef char cipe_sym_name[CIPENAMESIZE];
typedef char cipe_func_name[FUNCNAMESIZE];

struct loadmap {
    int loadtype;
    int sb;
    int nb;
    int sl;
    int nl;
    int ss;
    int ns;
};

struct cipe_symbol {
unsigned char *data;
    cipe_sym_name name;
    cipe_file_name file;
    int ndim;
    int sb;
    int nb;
    int sl;
    int nl;
    int ss;
    int ns;
    int datatype;
    int dataloc;
    int loadtype;
    struct loadmap *cubeload;
    struct loadmap *oldloadmap;
};

extern int cipe_size[];

/* macros for use with pointers to symbol table entries */
#define CIPEDATA(s)          ((s)->data)
#define CIPENAME(s)          ((s)->name)
#define CIPEFILE(s)          ((s)->file)
#define CIPENDIM(s)          ((s)->ndim)
#define CIPESB(s)            ((s)->sb)
#define CIPENB(s)            ((s)->nb)
#define CIPESL(s)            ((s)->sl)
#define CIPENL(s)            ((s)->nl)
#define CIPESS(s)            ((s)->ss)
#define CIPENS(s)            ((s)->ns)
#define CIPEDATATYPE(s)             ((s)->datatype)
#define CIPEELEMENTSIZE(s)          cipe_size[CIPEDATATYPE(s)]
#define CIPENUMELEMENTS(s)          (CIPENL(s)*CIPENS(s)*CIPENB(s))
#define CIPEDATASIZE(s)             (CIPENUMELEMENTS(s)*CIPEELEMENTSIZE(s))

/* function types */
extern struct cipe_symbol *cipe_create_symbol();
extern struct cipe_symbol *cipe_create_symbol_and_data();
extern struct cipe_symbol *cipe_get_symbol();
#endif SYMBOL_H

/* data distribution types */
#ifndef ELT_DATA_H
#define ELT_DATA_H

#define NOT_DIST          0    /* data not in cube */
#define BCAST_DIST        1    /* broadcast - all nodes have a copy */
#define HORIZ_DIST        2    /* row major decomposition */
#define VERT_DIST         3    /* column major decomposition */
#define GRID_DIST         4    /* grid distribution */
#define HORIZ_OVERLAP     5    /* horizontal distribution with overlap */
#define GRID_OVERLAP      6    /* grid distribution with overlap */
#define VERT_OVERLAP      7    /* vertical distribution with overlap */
#define CUSTOM_DIST       8    /* a custom distribution */

#endif ELT_DATA_H

/* cube parameter definitions */
#ifndef CUBEPARAM_H
#define CUBEPARAM_H

#define MAX_PARAM_COUNT 32
struct appl_param {
    int param_count;
    struct {
        int type;
        int direction;
        int length;
        unsigned char *ptr;
    }param[MAX_PARAM_COUNT];
};

/* type */
#define NON_SYMBOL_PARAM        0
#define SYMBOL_PARAM            1

/* direction */
#define OUTPUT          0
#define INPUT           1
#define BOTH            2

#endif CUBEPARAM_H

/* user interface parameter definitions */
#ifndef PARAM_TYPE_H
#define PARAM_TYPE_H

#define INPUT_SYMBOL    (-1)
#define OUTPUT_SYMBOL   (-2)
#define INT_ARRAY       (-3)
#define FLOAT_ARRAY     (-4)
#define DOUBLE_ARRAY    (-5)
#define CHAR            1
#define SHORT           2

#define INT             3
#define FLOAT           4
#define DOUBLE          5
#define BOOL            6
#define STRING          7
#define KEYWORD         8

#define REQ             (-264)
#define DEFAULT         (-262)
#define NOHELP          NULL
#define NOECHK          NULL
#endif PARAM_TYPE_H

/* cipe attribute definitions */
#ifndef CIPE_H

/* system attributes */
extern struct {
    char *name;
    int value_type;
    int value;
    int access;
} cipe_attr[];

#define COPROCESSOR_ATTR        0
#define CUBE_DIMENSION_ATTR     1
#define DISPLAY_DEVICE_ATTR     2
```

```
#define MOUSE_ATTR              3
#define DEBUG_LEVEL_ATTR        5
#define APPL_TRACE_ATTR         12
#define MENU_MODE_ATTR          13

#define COPROCESSOR         cipe_attr[COPROCESSOR_ATTR].value
#define CUBE_DIMENSION      cipe_attr[CUBE_DIMENSION_ATTR].value
#define DISPLAY_DEVICE      cipe_attr[DISPLAY_DEVICE_ATTR].value
#define MOUSE               cipe_attr[MOUSE_ATTR].value
#define DEBUG_LEVEL         cipe_attr[DEBUG_LEVEL_ATTR].value
#define APPL_TRACE          cipe_attr[APPL_TRACE_ATTR].value
#define MENU_MODE           cipe_attr[MENU_MODE_ATTR].value
#define CUBE_KWD            1
#define GAPP_KWD            2
#define NODISPLAYDEVICE     3
#define IVAS0_KWD           4
#define IVAS1_KWD           5
#define ORBITY512_KWD       6
#define ORBITY800_KWD       7
#define JUDY512_KWD         8
#define JUDY800_KWD         9
#endif

/* cube related global variables */
extern int doc,nproc;
#endif CIPEAPPL_H
```

Appendix B – INCLUDE FILE eltappl.h

```
#ifndef ELT_APPL_H
#define ELT_APPL_H

#include <cipe.h> /* cipe_attr and APPL_TRACE_ATTR definitions here */
#define APPL_TRACE          cipe_attr[APPL_TRACE_ATTR].value

/* data type definitions */
#ifndef DATATYPE_H
#define DATATYPE_H

#define UNDEF_TYPE       0
#define CHAR_TYPE        1
#define SHORT_TYPE       2
#define INT_TYPE         3
#define FLOAT_TYPE       4
#define DOUBLE_TYPE      5
#define BOOL_TYPE        6
#define STRING_TYPE      7

#endif DATATYPE_H

/* elt_symbol structure and its field definitions */
#define NAMESIZE      32     /* max length of symbol name */

#ifndef ELT_SYMBOL_H
#define ELT_SYMBOL_H
typedef char elt_name[NAMESIZE];        /* type of symbol names */

struct elt_loadmap{
    int loadtype;           /* load type if a simple type, -1 if unknown */
    int sb;                 /* starting band */
    int nb;                 /* number of bands */
    int sl;                 /* starting line */
    int nl;                 /* number of lines per band*/
    int ss;                 /* starting sample */
    int ns;                 /* number of samples per line */
};

struct elt_symbol{
    unsigned char *data;              /* pointer to data area */
    elt_name name;                    /* symbol name */
    int global;                       /* 1 if symbol is global, 0 if local */
    int datatype;                     /* data type code */
    struct elt_loadmap loadmap;       /* distribution pattern */
    struct elt_loadmap oldloadmap;    /* distribution pattern */
};

/* macros for use with pointers to symbol table entries */
#define DATA(s)          ((s)->data)
#define NAME(s)          ((s)->name)
#define DATATYPE(s)      ((s)->datatype)
#define LOADTYPE(s)      ((s)->loadmap.loadtype)
#define SB(s)            ((s)->loadmap.sb)
#define NB(s)            ((s)->loadmap.nb)
```

```
#define SL(s)           ((s)->loadmap.sl)
#define NL(s)           ((s)->loadmap.nl)
#define SS(s)           ((s)->loadmap.ss)
#define NS(s)                  ((s)->loadmap.ns)
#define ELEMENTSIZE(s)   (elt_elementsize[DATATYPE(s)])
#define NUMELEMENTS(s)  (NB(s)*NL(s)*NS(s))
#define DATASIZE(s)        (NUMELEMENTS(s)*ELEMENTSIZE(s))

#endif ELT_SYMBOL_H

#ifndef CUBE_PARAM_H
#define CUBE_PARAM_H
#define MAX_PARAM_COUNT    32
struct appl_param {
    int param_count;
    struct {
        int type;
        int direction;
        int length;
        unsigned char *ptr;
    }param[MAX_PARAM_COUNT];
};

/* type */
#define NON_SYMBOL_PARAM  0
#define SYMBOL_PARAM          1

/* direction */
#define OUTPUT       0
#define INPUT          1
#define BOTH           2
#endif CUBE_PARAM_H

#endif ELT_APPL_H

extern int doc, procnum, nproc;
```

Appendix C – CIPE MENU CONFIGURATION FILE

```
MENU mainmenu
setup/setup
Symbol/Symbol
Disp/Display
MssDisp/Mssdisp
Builtin/Builtin
Xform/Xform
Filter/Filter
Restore/Restore
Geom/Geom
Stretch/Hstretch
END

MENU Builtin
Symbol/Symbol
Disp/Display
add_func/appl
myfunc/appl
typecast/appl
pattern/appl
math/appl
stat/appl
Matrix/Matrix
END

MENU Matrix
matop/appl
cmatop/appl
constop/appl
END

MENU Xform
Symbol/Symbol
Disp/Display
rfft2/appl
cfft2/appl
powerspec/appl
END

MENU Filter
Symbol/Symbol
Disp/Display
kernel/appl
spfilter/appl
freqfilter/appl
medfilter/appl
prep/appl
END

MENU Geom
Symbol/Symbol
Disp/Display
surfit/appl
gentie/appl
```

```
tiept/appl
rotate/appl
scale/appl
END

MENU Restore
Symbol/Symbol
Disp/Display
feature_psf/appl
image_psf/appl
kernel/appl
invfilter/appl
ME/appl
ML/appl
END

MENU Hstretch
Symbol/Symbol
Disp/Display
perc/appl
END

MENU Mssdisp
Symbol/Symbol
Stretch/Stretch
mssdraw/mssdisp
mssplot/mssdisp
erase/display
zoom/display
END

MENU Display
Symbol/Symbol
Alloc/Alloc
Stretch/Stretch
zoom/display
draw/display
draw_color/display
erase/display
histo/display
cursor/display
hardcopy/display
END

MENU Stretch
linear/display
table/display
END

MENU Alloc
alloc/display
select/display
dealloc/display
disp_list/display
```

```
END

MENU Symbol
list/list_symbol
read/read_image
copy/copy_symbol
assign/assign_data
save/save_image
delete/delete
print/print_data
END
```

Appendix D – CIPE FUNCTION DICTIONARY

```
! cipe dictionary file          March/1/1990
function add_func
pathname "bltin_function"
help "add_func (function_name, pathname, help_msg) "
! cube
function cube_reset
pathname "bltin_function"
help "cube_reset"
! symbol stuff
function copy
pathname "bltin_function"
help "output = copy (input, {start_line, start_sample, number_of_line, number_of_sample})"
function delete
pathname "bltin_function"
help "delete (input)"
!builtin 2arg
function matop
pathname "bltin_function"
help "output =  matop (operation, input1, input2)"
function add
pathname "appl/bltin/cp/matop"
help "output = add (input1, input2) "
function sub
pathname "appl/bltin/cp/matop"
help "output = sub (input1, input2) "
function mult
pathname "appl/bltin/cp/matop"
help "output = mult (input1, input2) "
function div
pathname "appl/bltin/cp/matop"
help "output = div (input1, input2) "
!typecast
function typecast
pathname "appl/bltin/cp/bltintype"
help "output = typecast (output_data_type, input) "
function char
pathname "appl/bltin/cp/bltintype"
help "output = char (input) "
function int
pathname "appl/bltin/cp/bltintype"
help "output = int (input) "
function float
pathname "appl/bltin/cp/bltintype"
help "output = float (input) "
! math functions
function math
pathname "appl/bltin/cp/bltinmath"
help "output = math (operation, input) "
function sqrt
pathname "appl/bltin/cp/bltinmath"
help "output = sqrt (input) "
function log
pathname "appl/bltin/cp/bltinmath"
help "output = log (input) "
```

```
function log10
pathname "appl/bltin/cp/bltinmath"
help "output = log10 (input) "
function square
pathname "appl/bltin/cp/bltinmath"
help "output = square (input) "
function abs
pathname "appl/bltin/cp/bltinmath"
help "output = abs (input) "
function negate
pathname "appl/bltin/cp/bltinmath"
help "output = negate (input) "
!statistics functions
function stat
pathname "appl/bltin/cp/bltinstat"
help "output = stat (operation, input) "
function min
pathname "appl/bltin/cp/bltinstat"
help "output = min (input)"
function max
pathname "appl/bltin/cp/bltinstat"
help "output = max (input)"
function mean
pathname "appl/bltin/cp/bltinstat"
help "output = mean (input)"
function median
pathname "appl/bltin/cp/bltinstat"
help "output = median (input)"
function mode
pathname "appl/bltin/cp/bltinstat"
help "output = mode (input)"
function std
pathname "appl/bltin/cp/bltinstat"
help "output = std (input)"
function var
pathname "appl/bltin/cp/bltinstat"
help "output = var (input)"
! complex 2arg matrix operation
function cmatop
pathname "appl/bltin/cp/cmatop"
help "{out_real, out_imagi} = cmatop (operation, input1_real, input1_imagi, input2_real, input2_imagi)
-- not implemented in CLI mode yet"
function cmpadd
pathname "appl/bltin/cp/cmatop"
help " {out_real, out_imagi} = cmpadd (input1_real, input1_imagi, input2_real, input2_imagi)
-- not implemented in CLI mode yet"
function cmpsub
pathname "appl/bltin/cp/cmatop"
help " {out_real, out_imagi} = cmpsub (input1_real, input1_imagi, input2_real, input2_imagi)
-- not implemented in CLI mode yet"
function cmpmult
pathname "appl/bltin/cp/cmatop"
help " {out_real, out_imagi} = cmpmult (input1_real, input1_imagi, input2_real, input2_imagi)
-- not implemented in CLI mode yet"
```

```
function cmpdiv
pathname "appl/bltin/cp/cmatop"
help " {out_real, out_imagi} = cmpdiv (input1_real, input1_imagi, input2_real, input2_imagi)
-- not implemented in CLI mode yet"
! matrix arithematic operation with a constant
function constop
pathname "appl/bltin/cp/constop"
help "output =  constop (operation, input1, input2)"
function cadd
pathname "appl/bltin/cp/constop"
help "output =  cadd (input1, input2)"
function csub
pathname "appl/bltin/cp/constop"
help "output =  csub (input1, input2)"
function cmult
pathname "appl/bltin/cp/constop"
help "output =  cmult (input1, input2)"
function cdiv
pathname "appl/bltin/cp/constop"
help "output =  cdiv (input1, input2)"
! display utilities
function alloc
pathname "display"
help "alloc (host_name, device_type, window_size) "
function select
pathname "display"
help "select (unit_number) "
function dealloc
pathname "display"
help "dealloc -- no argument needed"
function disp_list
pathname "display"
help "disp_list -- no argument needed"
function draw
pathname "display"
help "draw (input, {start_line, start_sample}) "
function draw_color
pathname "display"
help "draw_color (input_red, input_green, input_blue, {start_line, start_sample}) "
function erase
pathname "display"
help "erase (i/o/a, {start_line, start_sample, number_of_line, number_of_sample}) "
function lstretch
pathname "display"
help "lstretch (min, max)"
function zoom
pathname "display"
help "zoom (i/o/a, zoom_factor, {start_line, start_sample}) "
! multi spectral data display
function mssdisp
pathname "disp/mssdisp"
help "mssdisp (input, band, {start_line, start_sample}) "
! pattern generator
function pattern
```

```
pathname "appl/bltin/host/pattern"
help "output = pattern (pattern_type, pattern_size{length,width}, inten{dark,light},size{length,width})
-- consult menu mode for the param of specific pattern"
! spatial filter
function spfilter
pathname "appl/filter/cp/spfilter"
help "output = spfilter (input_image, input_kernel)"
function medfilter
pathname "appl/filter/cp/medfilter"
help "output = medfilter (input_image, {nlw(3), nsw(3)}, thresh(0))"
! frequency filter
function freqfilter
pathname "appl/filter/cp/freqfilter"
help "output = freqfilter (input_image, input_psf, mode)"
! preprocessing
function reseau
pathname "appl/filter/cp/prep"
help "output = prep (input, reseau_file) -- this program requires hypercube"
! kernel generator
function kernel
pathname "appl/filter/host/kernel"
help "output = kernel (psf_type, size {nl, ns})"
! power spectrum
function Power
pathname "appl/xform/cp/Power"
help "output = Power (input)"
function powerspec
pathname "appl/xform/cp/powerspec"
help "output = powerspec (real_fft_result, imagi_fft_result, fold(y/n))"
! complex input fft2
function cfft2
pathname "appl/xform/cp/cfft2"
help "(output_real, output_imagi) = cfft2 (input_real, input_imagi, mode)
-- not implemented in CLI mode yet"
! real input fft2
function rfft2
pathname "appl/xform/cp/rfft2"
help "(output_real, output_imagi) = rfft2 (input, mode) -- not implemented
in CLI mode yet"
! restoration using inverse filter
function invfilter
pathname "appl/restore/cp/invfilter"
help " output = invfilter (input_image, input_psf, noise_level(float), niter, lambda(float), del_lamda(float))"
! restoration using maximum likelihood constraint
function ML
pathname "appl/restore/cp/ML"
help " output = ML (input_image, input_psf, noise_level(float), niter)"
! restoration using maximum entropy constraint
function ME
pathname "appl/restore/cp/ME"
help " output = ME (input_image, input_psf, noise_level(float), d_lamda(float), niter)"
!psf
function feature_psf
pathname "appl/restore/host/gen_psf"
```

```
help "it needs interactive graphic device -- not available in CLI mode"
function image_psf
pathname "appl/restore/host/gen_psf"
help "it needs interactive graphic device -- not available in CLI mode"
! pyramid reduce
function reduce
pathname "appl/geom/cp/reduce"
help " output = reduce (input, pyramid_level)"
! pyramid expand
function expand
pathname "appl/geom/cp/expand"
help " output = expand (input, pyramid_level)"
! pyramid merge
function merge
pathname "appl/geom/cp/merge"
help " output = merge (input1, input2, sample1, sample2, pyramid_level)"
! concatenate two images
function concat
pathname "appl/geom/host/concat"
help " output = concat (input1, input2, istat(0 for horiz, 1 for vertical), iave(1 for averaging))"
! rotate an image
function rotate
pathname "appl/geom/cp/rotate"
help " output = rotate (input, angle(float), clip_option(1 for clipping))"
function scale
pathname "appl/geom/cp/scale"
help " output = scale (input, x_scale_factor(float), y_scale_factor(float))"
function surfit
pathname "appl/geom/cp/surfit"
help "output = surfit (input, tiept_file, order_of_fit)"
function gentie
pathname "appl/geom/cp/gentie"
help "output_tiept_file = gentie (input_tie_file, order_of_fit, tiept_param{nptx,npty,gapx,gapy})
-- use menu mode"
function tiept
pathname "appl/geom/cp/tiept"
help "output = tiept (input, tiept_file)"
function data_dist
pathname "appl/diag/cp/data_dist"
help "output = data_dist (input, dist_type)"
function percent
pathname "appl/stretch/cp/perc_stretch"
help "output = perc_stretch (input, lower_percent, upper_percent)"
function sar_proc
pathname "appl/geom/cp/sar_proc"
help "output = sar(input, res{x,y,z}, sat_param(h,n,i))"
```

Appendix E – HOST RESIDENT EXECUTIVE SUBROUTINES

Subroutine: **cipe_compose_custom_loadmap(symbol, procnum, starting_line, starting_sample, starting_band, number_of_lines, number_of_samples, number_of_bands);**

Input:
struct cipe_symbol *symbol;
int starting_line, starting_sample, starting_band;
int number_of_lines, number_of_samples, number_of_bands;

Output: Loadmap attribute of **symbol** for the specified procnum is filled with the new loadmap.

Function: Creates the loadmap for the specified node using the parameters provided.

Subroutine: **cipe_compose_standard_loadmap(symbol, distribution, [vertical, horizontal]);**

Input:
struct cipe_symbol *symbol;
int distribution;
int vertical, horizontal;

Output: Loadmap attribute of **symbol** is filled with new loadmap.

Function: Automatically creates the loadmap for the specified distribution. **Distribution** should be one of the keywords, BCAST_DIST, HORIZ_DIST, GRID_DIST, VERT_DIST, HORIZ_OVERLAP, GRID_OVERLAP, or VERT_OVERLAP. BCAST_DIST, HORIZ_DIST, GRID_DIST, and VERT_DIST require only the first two arguments. Overlap loadmap requests require additional parameters to express the number of lines of overlap in the appropriate planes, e.g. HORIZ_DIST requires parameters for the number of lines overlap in the **vertical** plane.

Subroutine: **cipe_create_symbol(sname,nl,ns,nb,type)**

Input:
cipe_sym_name sname;
int nl, ns, nb;
int type;

Output: **struct cipe_symbol *cipe_create_symbol();**

Function: Creates an entry in the symbol table under the symbol name stored in **sname.** Space will be allocated for the symbol structure but no data space will be created. Each field in the symbol structure will be initialized with default values or those provided by the programmer to the routine. A pointer to a symbol structure will be returned.

Subroutine: **cipe_create_symbol_and_data(sname,nl,ns,nb,type)**

Input: **cipe_sym_name sname;**
int nl, ns, nb;
int type;

Output: **struct cipe_symbol *cipe_create_symbol_and_data();**

Function: Creates an entry in the symbol table under the symbol name stored in **sname**. Space will be allocated for the symbol structure, including data space. Each field in the symbol structure will be initialized with default values or those provided by the programmer to the routine. A pointer to a symbol structure will be returned.

Subroutine: **cipe_cube_param_def(param_num, param_type, direction, size, storage_ptr);**

Input: **int param_num, param_type; int direction, size; char *storage_ptr;**

Output: none

Function: Defines a parameter which is to be sent to the hypercube to be used in data processing. **Param_num** for the first parameter defined should be one, and successive parameters should use successive integers. **Param_type** should be one of the keywords, SYMBOL_PARAM or NON_SYMBOL_PARAM. **Direction** should indicate whether the parameter is used for INPUT, OUTPUT, or BOTH. BOTH is intended for use when distinct parameters are to be given to distinct nodes of the hypercube, and distinct values are to be returned. Storage pointed to by **storage_ptr** should be allocated by the programmer. Corresponding routine in the hypercube to retrieve parameters is **elt_get_param.**

Subroutine: **cipe_delete_symbol(sname)**

Input: **cipe_sym_name sname;**
int sindex;

Output: None

Function: Deletes the symbol table entry for the symbol of name **sname**, deallocates the symbol structure, and reorganizes the symbol table. Programmers should only delete temporary symbols created for their own purposes. Users have the ability to delete their own symbols.

Subroutine:	**cipe_error_read_symbol(sname);**
Input:	**cipe_sym_name sname;**
Output:	none
Function:	Takes the symbol name given as input and checks to see if it is present. If the symbol does not exist it returns a -1 and prints an error message for the user.

Subroutine:	**cipe_get_symbol(sname)**
Input:	**cipe_sym_name sname;**
Output:	**struct cipe_symbol *cipe_get_symbol();**
Function:	Returns a pointer to a symbol structure with the name **sname**.

Subroutine:	**cipe_warn_write_symbol(sname);**
Input:	**cipe_sym_name sname;**
Output:	none
Function:	Takes the symbol name given as input and checks to see if it is present. If the symbol exists it returns -1 and prints a warning message for the user.

Subroutine:	**cipedef(param_num, prompt, param_type, param_storage, num_of_elements, if_req);**

Input:	**int param_num;**
	char *prompt;
	int param_type;
	int num_of_elements, if_req;

Output:	**char *param_storage;**
Function:	Provides for the user interface for either CLI or menu mode and associates a user's input with a variable declared in the application. A parameter entry screen will be created in menu mode, and user input will be placed in the space pointed to by **param_storage**, when **cipedef** is followed by a call to **cipepar**. The first call to **cipedef** should use a **param_num** of 1. Successive calls to **cipedef** should use successive integers for **param_num**. **Param_type** should be one of the keywords INPUT_SYMBOL, OUTPUT_SYMBOL, STRING, INT, FLOAT, DOUBLE, BOOL, INT_ARRAY, FLOAT_ARRAY, or DOUBLE_ARRAY. **If_req** should be one of the keywords REQ, required, or DEF, default.

Subroutine:	**cipepar(error_routine(), help_routine());**
Input:	int error_routine(); int check_routine();
Output:	none
Function:	Produces a parameter entry screen in menu mode and stores a user's input in the provided storage locations so it is available to the application. **Cipepar** also allows a programmer to provide help and error checking for user input. Help provided here is not available in CLI mode. Error checking is available for both CLI and menu modes.

Appendix F – HYPERCUBE RESIDENT EXECUTIVE SUBROUTINES

Subroutine:	**elt_create_symbol(symbol_name, number_of_lines, number_of_samples, number_of_bands, type_of_data);**

Input:	**elt_name symbol_name;**
	int number_of_lines, number_of_samples, number_of_bands;
	int type_of_data;

Output:	**struct elt_symbol *elt_create_symbol();**
Function:	Creates a symbol of name **symbol_name**, fills all the symbol attributes with default values or provided subroutine arguments, but does not create any symbol data space. **Type_of_data** should be one of the keywords: CHAR_TYPE, SHORT_TYPE, INIT_TYPE, or FLOAT_TYPE.

Subroutine:	**elt_create_symbol_and_data(symbol_name, number_of_lines, number_of_samples, number_of_bands, type_of_data);**

Input:	**elt_name symbol_name;**
	int number_of_lines, number_of_samples, number_of_bands;
	int type_of_data;

Output:	**struct elt_symbol *elt_create_symbol_and_data();**
Function:	Creates a symbol of name **sname**, fills all the symbol attributes with default values or provided subroutine arguments, and creates data space for the symbol. **Type_of_data** should be one of the keywords: CHAR_TYPE, SHORT_TYPE, INIT_TYPE, or FLOAT_TYPE.

Subroutine:	**elt_delete_symbol(symbol_name);**
Input:	**elt_name symbol_name;**
Output:	none
Function:	Deletes the symbol of name **symbol_name**. Programmers should only delete temporary symbols created for their own purposes. User-created symbols should not be deleted by programmers.

Subroutine:	**elt_get_param(param_num, pointer_to_pointer);**
Input:	**int param_num; char **pointer_to_pointer;**
Output:	none
Function:	Retrieves parameters which **cipe_cube_param_def** designated to be sent to the cube. Programmer does not have to allocate the space pointed to by pointer_to_pointer. The same **param_num** should be used in the host and node for the same parameter.