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

358

Concurrent Image Processing Executive (CIPE)

Volume I: Design Overview

Meemong Lee
Steven L. Groom
Alan S. Mazer
Winifred I. Williams

March 15, 1990

NASA
National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

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

Concurrent Image Processing Executive (CIPE)

Volume I: Design Overview

Meemong Lee
Steven L. Groom
Alan S. Mazer
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 report describes the design and implementation of a Concurrent Image Processing Executive (CIPE), which is intended to become the support system software for a prototype high performance science analysis workstation. The target machine for this software is a JPL/Caltech Mark IIIfp Hypercube hosted by either a MASSCOMP 5600 or a Sun-3, Sun-4 workstation; however, the design will accommodate other concurrent machines of similar architecture, *i.e.*, local memory, multiple-instruction-multiple-data (MIMD) machines. The CIPE system provides both a multimode user interface and an applications programmer interface, and has been designed around four loosely coupled modules: (1) user interface, (2) host-resident executive, (3) hypercube-resident executive, and (4) application functions. The loose coupling between modules allows modification of a particular module without significantly affecting the other modules in the system. In order to enhance hypercube memory utilization and to allow expansion of image processing capabilities, a specialized program management method, incremental loading, was devised. To minimize data transfer between host and hypercube, a data management method which distributes, redistributes, and tracks data set information was implemented. The data management also allows data sharing among application programs. The CIPE software architecture provides a flexible environment for scientific analysis of complex remote sensing image data, such as planetary data and imaging spectrometry, utilizing state-of-the-art concurrent computation capabilities.

ACKNOWLEDGMENTS

The authors would like to acknowledge Dr. Jerry Solomon for his support and direction of this work. The rest of the authors would also like to thank Winifred Williams for her selfless effort in editing this report for publication.

The research described in this report was carried out by the Observational Systems Division of the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration (NASA). Special thanks are due to the office of Space Science and Applications (Code EC) of NASA for their encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

The current and projected growth in volume and complexity of the National Aeronautics and Space Administration (NASA) mission remote sensing image data is placing a considerable strain on the computational resources available to the science community for analyzing and interpreting this valuable source of information. By the end of this century, given the Earth Observing System (EOS) and other NASA space exploration missions, the wealth of data available to scientists in a variety of disciplines will be truly astounding. In order to address the problems associated with analyzing this data, a new generation of computational resources and tools must be developed and made available to the NASA science community. This task report describes work carried out during FY88 and FY89 which addresses one aspect of this problem, *i.e.*, the development of user environments and tools for high-performance science analysis workstations. This work has been carried out in the context of implementing a Concurrent Image Processing and Analysis Testbed (CIPAT), which will form a prototype high-performance workstation. The utilization of emerging concurrent processing technology, in particular multiple-instruction-multiple-data (MIMD) architectures which provide large amounts of local memory, is a natural approach to handling compute-intensive problems with large data sets. This technology presents its own unique set of problems, however, particularly with respect to system software design and interactive user environments. The work described in this report provides one solution to these problems through the implementation of a general executive software system, the Concurrent Image Processing Executive (CIPE).

As the name implies, CIPE was designed for hardware systems which utilize concurrent computational resources for the processing of image and other multidimensional data; more specifically, the concurrent computational resource is assumed to be an attached processor which is connected to a *host* computer. In this context, the host machine provides the usual broad range of operating system services, file I/O, display device hardware, network connections, *etc.* Although the executive described in this report could be extended to cover commercial *multiprocessor* computers such as Alliant, Convex, Elxsi, Flex, *etc.*, that was not a design objective. The hardware environment within which CIPE was developed consists of a MASSCOMP 5600 dual-processor host machine, an International Imaging Systems (I^2S) IVAS image display processor, and an 8-node JPL/Caltech Mark IIIfp hypercube. The host machine is currently changed to a Sun-4 to accommodate much higher data transfer rates utilizing new concurrent I/O capabilities. Numerous display devices (Sun consoles and IVASes) are added to the CIPE environment employing a remote process control (RPC) protocol for accessing remotely connected devices.

CIPE was designed around four major software modules: (1) user interface, (2) host system monitor, (3) hypercube monitor, and (4) application functions. A major design objective was to provide a powerful and flexible user interface for efficient utilization of the system's considerable computational resources. System modularity provides relatively easy functional extensibility, both at the user level and the applications programming level. CIPE provides a number of utilities which greatly ease the application programmer's difficulties in dealing with a new machine architecture, *i.e.*, the hypercube topology. We believe that the system described in this report provides a flexible framework within which to begin building a powerful user-oriented computing environment for high-performance science analysis workstations.

This report is intended to serve both as an overview of the CIPE design and implementation philosophy. For more detailed information in usage of CIPE functions and the CIPE programming environment, a programmer's guide (Volume II of this report) and a user's guide (Volume III) are provided separately. This report is organized in four sections. Section 1 describes the design objectives of CIPE. Section 2 describes the two user interface modalities of the command line interpreter and menu mode. Section 3 describes the host system monitor functions with respect to generic operating system functions and concurrent system interface functions. Finally, Section 4 describes the hypercube monitor functions, including the interface mechanism to the host system and CIPE management of data and application programs.

1. OVERVIEW

CIPE was designed and implemented by the Image Analysis Systems Group for the purpose of utilizing various concurrent architectures in a high rate data processing environment. Concurrent systems provide greatly enhanced computational power by integrating large numbers of processors into systems via various interconnection topologies. However, they do present significant programming difficulties due to their architectural complexities. In particular, utilization of such systems for interactive image processing requires a unique kind of software environment which shields users and programmers from architectural complexities while offering the computational advantages of concurrent systems.

1.1. WHAT IS AN EXECUTIVE?

Though there is no written definition for an executive, in general, an executive is software which resides on top of an operating system and provides common resources to users and programmers within a well-defined application area. The common goal of an interactive image processing environment in a concurrent system configuration is the utilization of a concurrent architecture for faster processing of image data, visual presentation, and real-time user interaction. The shared resources include user interfacing, data manipulation, display, file I/O, and interfacing to concurrent systems. Therefore, an executive approach was chosen to provide a software environment (user and programmer) for concurrent image processing.

CIPE has distinctive characteristics as a concurrent system executive, as a high rate data processing executive, and as an interactive image processing executive. As a concurrent system executive, it provides a programming environment which shields a programmer from architectural complexities while it utilizes the maximum potential of a concurrent system. As a high rate data processing executive, it provides a flexible file management scheme and efficient data manipulation and representation mechanisms. As an interactive image processing executive, it offers a friendly, flexible, and efficient user environment.

1.2. DESIGN OBJECTIVES

As CIPE is an executive serving three distinctive purposes, utilization of a concurrent system, interactive image processing, and high rate data processing, it has three corresponding design objectives and one overall objective.

First, the design objective as a concurrent system executive was to utilize a concurrent system to its full potential. The maximum concurrency of the system for data processing was pursued via appropriate data and program management schemes which utilize the architectural characteristics of a concurrent system (interconnection topology, processing power, and memory system).

Second, the design objective as a high rate data processing executive was to manage data efficiently so that its processing can be performed without wasteful file transactions or data traffic between the host and a concurrent system. The main goal of the data management scheme was to overcome the data manipulation difficulties in a concurrent system that has a localized memory and a serious data transfer bottleneck.

Third, the design objective as an interactive image processing executive was to provide an easy-to-use image processing environment which provides a user a wider range of expressional tools and choices of interaction levels. The expressional tools allow a user to carry out tasks efficiently. The multiple choices of a user interaction level, according to experience level and application needs, allow a user to be acquainted with CIPE easily while providing room to grow with it.

Finally, the overall design objective of CIPE is to provide an architecture-independent environment where neither a user nor a programmer needs to be involved in the system-dependent details related to the file system, display devices, or concurrent systems. Such an environment is pursued by separating the executive into device-dependent and independent modules and coupling them with a set of generalized interface routines. The decoupling of the architecture dependency from the functionalities allows CIPE to be easily upgraded to future system configurations as well as to maintain the transportability of the application programs.

1.3. DESIGN IMPLEMENTATION

The current implementation of CIPE uses the combination of a Sun-4 host computer and a MARK IIIfp 8-node hypercube (Figure 1.1). CIPE consists of four modules: user interface, host system monitor, hypercube monitor, and application (a set of generic image processing functions). Each module is designed and implemented to achieve the overall design objectives of CIPE.

As a user environment, CIPE provides a friendly user interface in two forms: a command line interface and a menu interface. The command line interface provides a simple interpreted programming language allowing interactive function definition and evaluation of algebraic expressions. The menu interface employs a hierarchical screen-oriented menu system for executing CIPE commands.

The host system monitor handles overall system interactions and provides generic image processing executive functions. The system management involves peripheral device handling, the concurrent system interface, and application program execution. In order to develop an image processing executive for a virtual concurrent system environment, system setup procedures, data management schemes among multiple systems, and concurrent system interface methods were designed and implemented. Also, a data representation method was devised to optimize the large volume data manipulation among file, host, and

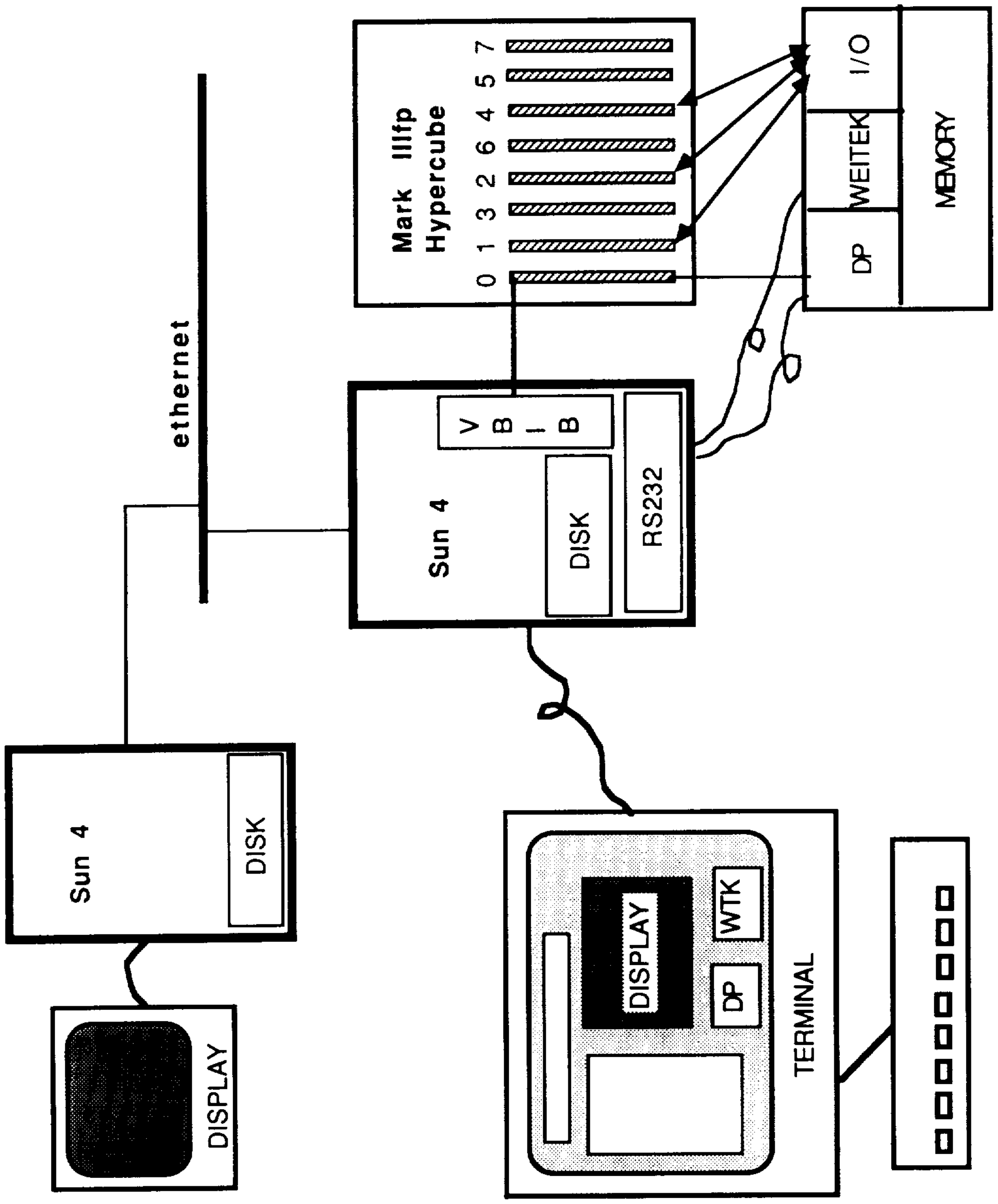

Figure 1.1 Concurrent Image Processing System Configuration

concurrent systems.

Each concurrent system resident executive must implement unique architecture-dependent data and program management methods for optimal utilization of the architecture. The hypercube resident monitor allows CIPE to avoid data communication bottlenecks by keeping active data resident in the hypercube and minimizing application program space by downloading code as required to operate on the data. In order for data to be shared among several application programs with different data distribution requirements, an automatic data redistribution method is devised.

The image processing function module is a non-resident component of CIPE, since each function is dynamically loaded only for execution. The executive-provided functions (partly implemented) include generic image processing functions, systematic flight data processing, multi-spectral data processing, and a set of data processing primitives for interactive algorithm development.

The overall relation among modules is displayed in Figure 1.2. The example concurrent system resident executive in Figure 1.2 is a hypercube resident executive. The hypercube resident software is developed employing the Crystalline Operating System (CrOS) using the C Programming Language.

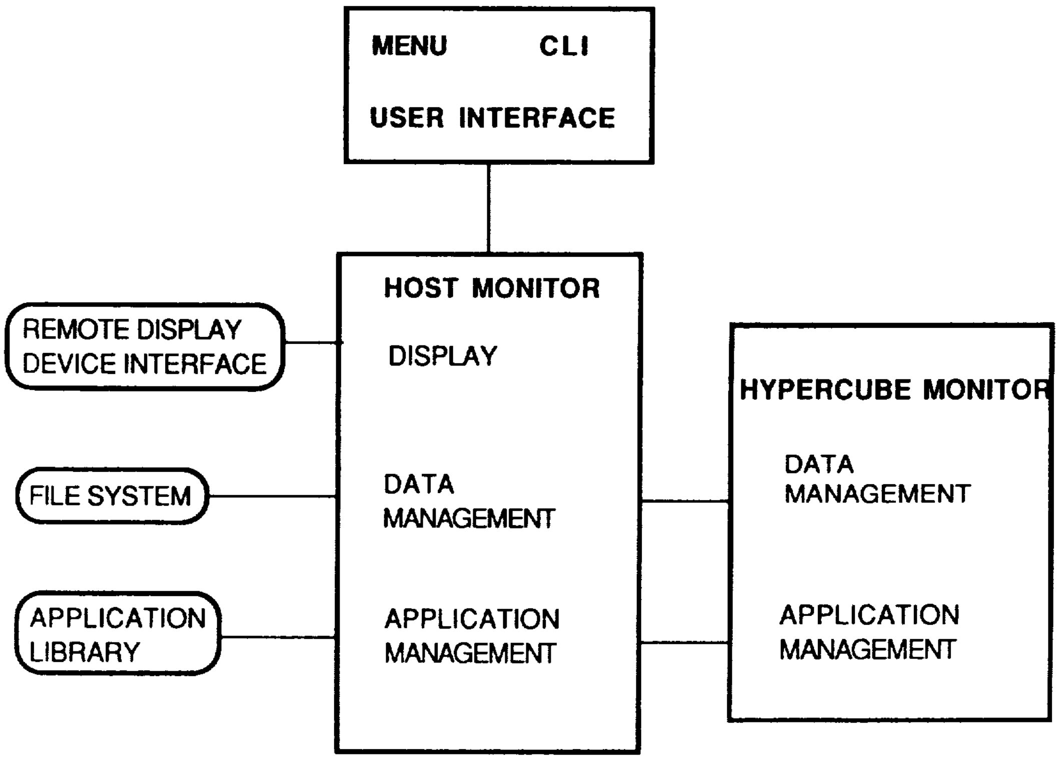

Figure 1.2 CIPE Software Structure

2. USER INTERFACE

The design philosophy behind the CIPE user environment is to provide the user a choice of interfaces. Though numerous "user friendly" interfaces have been developed, there is no single interface scheme that all users agree on; individual users have different conceptions of "friendliness" just as they have different levels of expertise and different processing needs.

Basically, there are three types of user interface modes: command line, menu, and windowing. The command line interface (used by computer operating systems, for example) imposes a rigid syntax structure and usually demands complete information about the operation to be performed when the command is entered. It is especially useful in a batch-oriented environment where a set of precomposed command lines can be utilized repeatedly, or when functions and expressions need to be interactively defined and evaluated. The menu interface offers a hierarchical organization and preformatted prompts for necessary information which a user can easily follow in real time. Its usage is for interactive data processing environments with limited amounts of user-provided information. The windowing interface provides another dimension to user interaction by offering a graphical expression. Its usage is for visually oriented data processing environments where a simple graphical expression can replace a complicated command line or menu input.

CIPE provides the first two types of user interface, allowing the user the advantages of each. The command line interpreter (CLI) provides a simple interpreted programming language allowing interactive function definition and evaluation of algebraic expressions. The menu interface employs a hierarchical screen-oriented menu system. User input error-checking is performed prior to execution to reduce the probability of execution error and to prevent wasteful execution. In this section, the discussion of each user interface mode is focused on describing CIPE's usage of each mode with respect to user interface organization and functionality through descriptions and examples.

2.1. COMMAND LINE INTERPRETER

The first of the user interface approaches implemented in CIPE is the Command Line Interpreter (CLI). The CLI makes available the basic capabilities of CIPE, including environment control, program execution, and the evaluation of functions and expressions. It also provides for the definition of CLI-interpreted functions and scripts, allows the creation of *workspace* collections of user-defined procedures, and provides more terminal independence than the other user interfaces. Interpreted functions may be expressed in terms of built-in primitives, compiled code, or other interpreted functions. The command line approach allows operations to be performed repetitively through looping and simplifies interactive evaluation of complex expressions using built-in and user-defined functions. Since most operating systems are command line based, the CLI approach also allows a user to escape CIPE execution temporarily and run operating system utilities such as those used to list a directory or to edit a file.

This section discusses the design and implementation of the CLI, which has three parts: command line input and lexical analysis, parsing, and code generation. The execution of functions is discussed in the CIPE User's Guide (Volume III). Commands are taken from the keyboard or workspace, separated and unaliased into tokens in the command language, and parsed; commands taken from the keyboard may have been edited. The parser uses syntax-directed translation to build a pseudo-code program, which is then executed. The following subsections discuss each of these functions in detail.

2.1.1. Input and Lexical Analysis

The input and lexical analysis phases of command line interpretation are straightforward, centered around the concept of *input streams*. An input stream corresponds to a source of command input. As new streams are added, descriptors for previous streams are pushed onto a stack until such time as the new stream runs out of input. The first input stream in any session is the *standard input*, by default, the keyboard. As a user typing at the keyboard loads a previously saved workspace, the workspace file becomes the current stream and the descriptor for the previous stream is pushed onto the stack. Workspace execution continues until the workspace loads another workspace or ends, at which time the stack is adjusted accordingly. Edited command lines are handled similarly, written to a file which is then opened as a stream and afterwards discarded.

Command line interpretation starts with initializations: the allocation of initial buffer space, and the creation of an initial input stream descriptor. The remainder of interpretation is then a loop in which input handling is simply determining the current stream, managing pointers and status variables, providing for editing, and reading a line. If the user requests editing of a previous or current command using CTRL E, the input code writes out what has been typed so far, or if nothing, the last command, and allows the user to edit the command using a standard full screen editor. This approach was chosen over more simple, line-based command editors both because of the ease with which a user can access a favorite editor and because CIPE command lines, such as those containing function definitions, are potentially so complex that a line editor approach is very difficult to use. Once edited to the user's satisfaction, the descriptor for the previous stream is temporarily pushed onto the stack, and input is taken from a temporary file containing the user's edited command. Implicit in the process of command input are management requirements such as the monitoring of buffer space (which may be automatically increased if necessary), the removal of unnecessary space and comments, and for edited command lines, feedback of the final command line to the user.

Actual lexical analysis is done by the function, *yylex*, generated from a legal-token specification by the Unix* utility *lex*. Supporting this lexical analysis is a variety of functions within CLI for tracing, line continuation handling, input buffer access, recognition of reserved words and their types, and error handling. Line continuations are usually provided automatically. Whenever a user enters a command line which is incomplete, either because some terminating delimiter has not been entered (such as a *for* loop started but not finished with an *end* or a function call with an incomplete argument list), or because the line ended with a two-argument operator, the CLI prints a continuation prompt, properly indented, to show that more input is expected. Alternatively, a user can indicate that more is to come by ending an input line with a backslash(\). Reserved words are recognized using a variety of lists stored within the lexical analyzer. Because reserved words such as multiword system attributes may be specified in several ways (with underscores between words of the attribute, or spaces, and with component words abbreviated), their recognition requires some local parsing to determine whether they are reserved words, legal variable names or simply errors. In addition to system attributes, reserved words include commands, command support words (e.g., *step* in *for i=0 to 10 step 5*), and boolean and keyword values (e.g., *on*). While error detection is largely done by the parsing, alerting the user is a function of lexical analysis, largely because the lexical analyzer has more information about the actual command lines as entered than does the parser. On errors, the lexical analyzer either prints the command line causing the error, together with an explicit error message and a carat (^) pointing to the location of the error within the line, or displays a workspace name and the number of the line within the workspace causing the error.

2.1.2. Parsing

The tokens taken from the input stream by the lexical analyzer are next fed into the parser, which attempts to assemble the tokens into a legal form, while at the same time generating pseudo-code for the eventual execution of the command. The parser itself is generated by the Unix utility *yacc* from the CLI grammar definition file *parser.y*.

The *yacc* language definition consists of two parts: productions and parse actions. The productions are defined formally by the following grammar in Backus-Naur Form:

```
    cmd_list ← cmd_list command
              | command

    command ← simple_cmd
```

* Unix is a registered trademark of AT&T Bell Laboratories.

```
               | cntl_struct

 simple_cmd ← exec_cmd
               | ! unix_cmd
               | output = expr
               | func_name ( actual_args )
               | func_name
               | quit

 cntl_struct ← for var_name = expr to expr scmd_list end
               | for var_name = expr to expr step expr scmd_list end
               | while expr scmd_list end
               | if expr then scmd_list endif
               | if expr then scmd_list else scmd_list endif
               | if expr then scmd_list else_if_list else_list endif

   exec_cmd ← set entity attr to expr
               | set attr to expr
               | turn bool entity attr
               | turn bool attr
               | functions
               | symbols
               | traces
               | print expr_list
               | show func_name
               | define func_name ( formal_args ) cmd_list end
               | load ws_name
               | save ws_name
               | edit ws_name
               | read var_name from filename
               | write var_name to filename
               | menu
               | nomenu
               | quit

     output ← var_name
               | var_name [ index_list ]

  scmd_list ← scmd_list ; simple_cmd
               | simple_cmd

else_if_list ← else_if_list else if expr then scmd_list
               | else if expr then scmd_list

  else_list ← else scmd_list
               | ε
```

```
      entity ← cube | gapp | system

        attr ← dimension | num_procs | topology | priority | logging
               | lextrace | parsetrace | codegentrace
               | exectrace | symtabtrace | functabtrace

 formal_args ← ε
               | var_name
               | formal_args , var_name

     ws_name ← expr

    filename ← expr

      at_loc ← ε
               | at expr , expr

  index_list ← index_list , index
               | index

 actual_args ← ε
               | expr_list

       index ← expr
               | expr : expr

   expr_list ← expr
               | expr_list , expr

        expr ← expr op2 expr
               | op1 expr
               | var_name
               | var_name [ index_list ]
               | func_name ( actual_args )
               | ( expr )
               | constant

    constant ← single_value
               | { value_list }

  value_list ← value_list , single_value
               | single_value

single_value ← integer
               | float
               | string
               | enum
```

```
op2 ← &&  |  |  |  &  |  |  |  ^  |  <  |  <=
      |  ==  |  !=  |  >=  |  >  |  *  |  /  |  +  |  -

op1 ← -  |  !  |  ~
```

A few formats and details are omitted here where the rules are generally agreed upon, such as for the formation of *floats*. Variables are not explicitly typed; types are determined at run-time and functions do type conversion as necessary.

Yacc's output code uses *shift-reduce* parsing with single-token lookahead to reduce every input line to a valid command. As tokens are received from the lexical analyzer and *shifted* onto a stack, the parser attempts to *reduce* (combine) sequences of tokens appearing on the right side of one of the listed productions. When several tokens can be reduced to a higher-level element of the grammar, called a *non-terminal* to distinguish it from a *terminal* or reserved word in the command language, they are replaced on the stack by the non-terminal, which is in turn compared against the right side of the productions above. Ultimately, the group of tokens forming the input line reduces, through many intermediate steps, to a *cmd_list*. Note that a legal input line may actually have multiple commands according to this grammar. No command delimiter is necessary; the grammar is designed such that successive commands may be expressed on the same line unambiguously.

In addition to the productions listed above, the yacc input definition contains parse actions. Parse actions are invoked whenever a sequence of tokens or non-terminals reduces to a higher-level non-terminal, and they represent most of the code in the yacc input definition. They fulfill two functions: to assist in parsing, and to create the pseudo-code which is the ultimate product of the parser. For example, when an integer value is sent by the lexical analyzer, the parser recognizes that and reduces it to the non-terminal *single_value*. The parse action for the reduction is to print out a trace, if the user has requested parse tracing, allocate storage and save the value in the symbol table, and then associate the index of the value in the symbol table with the new non-terminal on the parse stack. Similarly, the non-terminal single_value will eventually be used along with other input in the formation of one of the non-terminals *constant* or *value_list*, depending on whether or not it was preceded by a brace ({). If the single_value is a part of the non-terminal value_list, the parse action combines the value represented by single_value with the other values in the value list, associating with the new non-terminal this expanded list of values. Ultimately, the instruction using this value will be generated by yet another parse action looking at a correspondingly higher-level group of non-terminals.

These and similar parse actions are applied at every step in the parsing of the grammar until a program has been created. Other parse actions handle such things as intelligent error detection and recovery, table lookups and management, backpatching (changing previously created code based on new and previously unavailable information), code segment management (described in more detail in the next section), and the creation of temporary variables.

2.1.3. Code Generation

A large part of parser action work is in the generation of pseudo-code instructions for execution of the command. The code called by these actions forms the third part of the CLI, code generation. As the idea of streams is fundamental to understanding command input and lexical analysis, so *code segments* are central to code generation. A code segment is a data structure containing the name of a user-defined function, a pointer to the compiled code, a symbol table and function table, and a pointer to a *parent* code segment. One code segment is set up by default when the user starts up CIPE. Commands typed at the keyboard are compiled and described by this code segment, including any functions defined at the keyboard level and any symbols created interactively. When a workspace is loaded, a descriptor is created for it, with the original code segment as parent, and execution shifts over to the workspace. The use of code segments allows easy grouping of symbols and subordinate functions in the proper context, and allows CIPE to quickly find the correct data or function when a specification is potentially ambiguous.

The code generation functions fall into two groups: code segment maintenance and support, and instruction generation. Code segment maintenance routines include code to create new code segments (as when executing a DEFINE command), to add instructions to the current code segment, and to return to a previous code segment. Support routines store and manage constants in the symbol table, perform compile-time type conversions, disassemble instructions for user-defined workspace and function listings, and create temporary variables. Instruction generation routines take information grouped by the parser and translate it into individual instructions. Most commands recognized by the parser translate into a specific instruction which is generated by a corresponding code generation routine. For example, the routine *cipe_arith_op* is called when the parser gets a token group of the form *constant* + *constant*. It generates an addition instruction, creating a temporary variable, if necessary. Similar code generation routines create instructions for assignments, subscripting, function calls, and the other language features. In the case of branching as part of a looping construct, where all the available information for instruction generation may not be available until the loop is completely parsed, parser actions direct the routines to produce skeletal code which is then backpatched, or filled in as the information becomes available. Code generation for the assignment instruction attempts to modify previous code to make the assignment instruction and the resulting overhead unnecessary.

2.2. MENU

The menu interface of CIPE serves several purposes: self-orientation to CIPE's organization, a simple environment for inexperienced users, interactive input verification, and functional grouping of CIPE capabilities. The entire CIPE functionality is accessible by flipping through menus using menu control keys, allowing one to quickly grasp the organization and usage of individual CIPE commands. Each activated command displays prompts for required user inputs and help can be made available for each input field so that inexperienced users can easily follow the procedure. The validity of input may also be checked in real-time and proper error messages displayed so that users can correct input before actual execution takes place. The hierarchical organization of the menus allows grouping of related functions. This functional grouping enhances the menu entry selection process, since a user can scope CIPE according to his/her processing needs.

Menu mode is implemented using Yamm (Yet Another Menu Manager), a general purpose menuing package. Design details of Yamm are included in Appendix A and an implementation example is provided in the CIPE Programmer's Guide. This section is devoted to describing the utilization of the package in CIPE to provide a menuing user interface to serve the purposes presented above.

Yamm employs three windows: a menu window, an application I/O and data entry window, and a status window. The menu window displays the submenus and functions available at the current level of the menu tree. The menu hierarchy is specifiable on a per-user basis using *menu configuration* files, allowing individual users to customize the environments. The menu configuration is structured in a multibranched tree fashion, where the leaf node is connected to a corresponding routine name as shown in Figure 2.1.

Currently, accessible submenus and applications are selected either by number or by using arrow keys to move to the desired selection. If a submenu is selected, its set of menus and accessible functions replaces those of the previous menu in the menu window. When the selected submenu entry is a leaf node in the menu configuration, the corresponding function becomes activated. The interaction between a user and the activated function takes place in the application I/O and data entry window.

The application I/O and data entry window serves a dual purpose. During execution of user programs, it displays application output and system error messages. During application-requested data entry, the window displays a data entry form for specification of a given command's inputs. A parameter may be classified as required when a value must be given by a user, or defaulted. For the parameters with default values, the default values are automatically displayed and a user may accept or change them. Individual parameters may also have help and parameter-specific error checking. For parameters with just

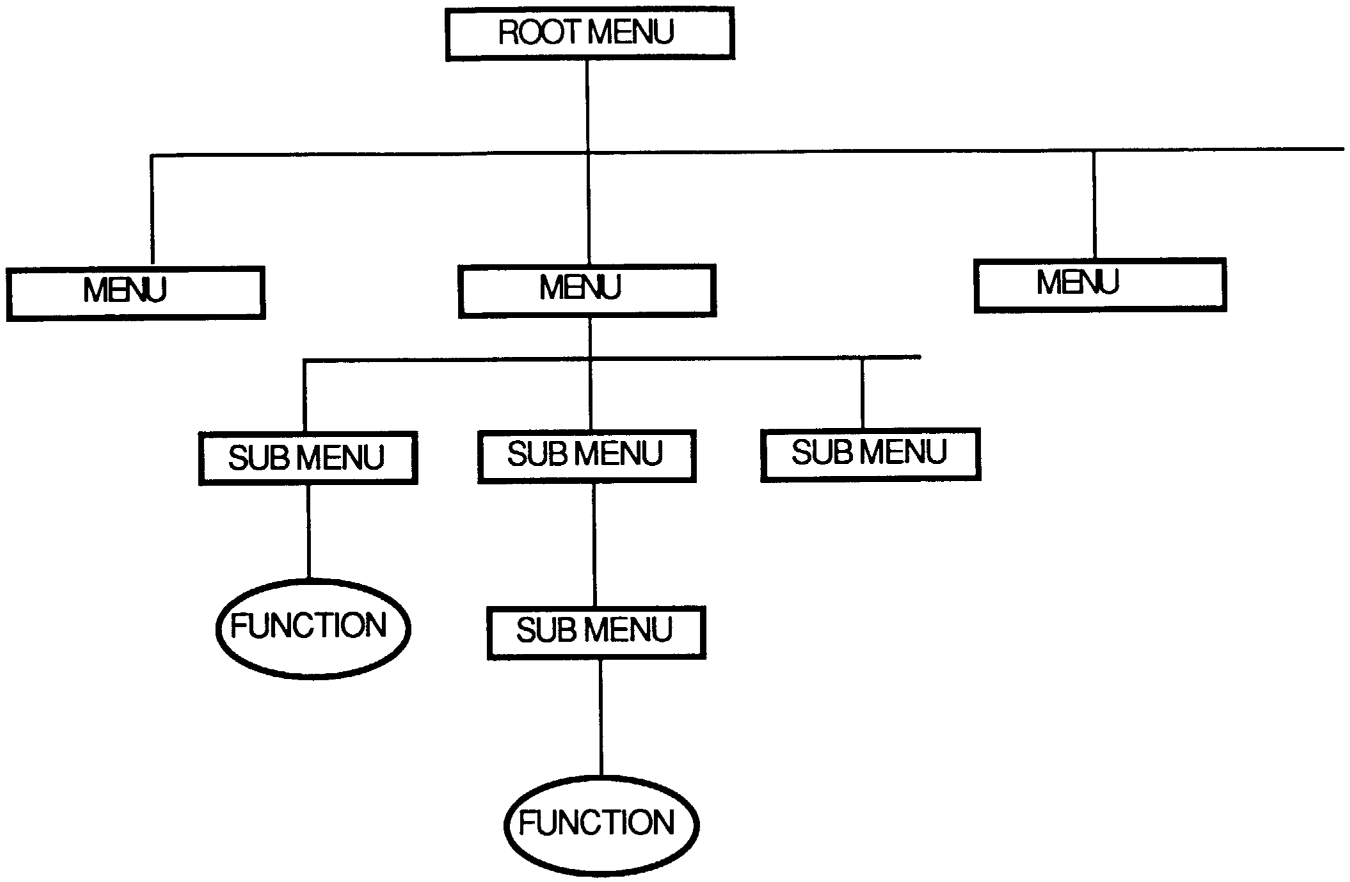

Figure 2.1 Menu Structure

a fixed number of valid values, the key combination shown in the status window next to *Next Value* may be used to step through the possibilities. See Appendix A for the complete set of Yamm capabilities.

The status window displays the name of the current menu and the keys used for the available functions in context. For example, during parameter entry the window shows keyboard mappings for *Return to Menu*, *Help*, *End Data Entry*, etc. When waiting for a submenu or application selection, the menu status window shows mappings for such things as *Previous Menu* and *Exit*. Keyboard mappings are determined automatically based on the type of terminal in use, but may also be specified in the code by the programmer or in user-specific menu configuration files.

2.3. APPLICATION PROGRAM INTERFACE

CIPE offers a set of resident functions which are always loaded into the system when CIPE is activated. The resident functions include symbol manipulation and utilities for display and system setup. Besides the system resident functions, CIPE provides a large set of generic image processing functions including various spatial/frequency filters, geometric transformations, restoration algorithms, and algebraic data manipulation functions. These functions are supplied to CIPE as application programs. The CIPE allows the application functions to be added without requiring any modification to CIPE by employing a dynamically loaded function dictionary file. The function access can also be tailored for an individual user via a user composed menu configuration file.

Function Dictionary - At a given time in the CIPE session, the available function list is determined by the current function dictionary file. A user may change the function list at any time by providing a new dictionary file. The dictionary file contains a list of functions where each function is described with respect to its function name, file name of the executable code, and help message (calling procedure, etc.). When a function is invoked by a user, CIPE searches the function name from the dictionary, loads the corresponding executable file, and executes the code. When the function name does not exist in the dictionary, the function name is assumed to be an executable code file name in order to allow activation of a function that has not been added to the dictionary file. When the executable file does not exist in the system, or the user input parameters do not match with the function call, the function call will be aborted.

Menu Configuration - The MENU mode of CIPE is controlled by a menu configuration file which contains the hierarchical structure of functions. A user may design his/her menu configuration file to enable a better function access for a given task. A menu entry is either followed by a sub-menu or a function. For a CIPE resident function, the resident function name is accompanied by a specific routine name while all of the application functions are accompanied by a string called "appl" (ex.

function name/appl). When a menu entry accompanied by "appl" is selected, CIPE performs the application function execution in the manner described above using the menu entry name as a function name.

With these two external files, CIPE achieves a great flexibility to accommodate a growing list of application functions and offers a user environment that can be designed by a user for his/her individual need at a given time. The default function dictionary file and menu configuration file are appended in the CIPE user's guide (Volume III of this report). For more detailed information on application programming, refer to the CIPE programmer's guide (Volume II of this report).

3. HOST SYSTEM MONITOR

The CIPE system configuration includes a host system and one or more concurrent systems. Thus, the executive is divided into a host resident part and the concurrent system resident part. The host resident part, referred to as a host system monitor, handles overall system interactions and provides generic image processing functions. The system management involves peripheral device handling, the concurrent system interface, and application program execution.

In order to develop an image processing executive for a virtual concurrent system environment, system setup procedures, data management schemes among multiple systems, and concurrent system interface methods were designed and implemented. A data representation method was also devised to optimize the large volume data manipulation among file system, host system, and concurrent systems. In this section, the host system monitor is described with respect to its four functions: system setup, data management, file management, and concurrent system interface.

3.1. DATA MANAGEMENT

In an image processing environment, frequently, a sequence of functions is applied to a data set to remove systematic noises and to enhance the data. It is common practice in image processing to apply a function to a data set, save the result to a file, retrieve the file for the next function, and so on until the data set has been completely processed. Such a scenario is extremely time consuming when there is more than one system involved for data processing and each system requires downloading and uploading of data for each function.

In order to achieve efficient data processing with minimized file I/O and host-concurrent system data traffic, a data sharing mechanism among application functions is devised via a cross-referable data management scheme among the host monitor, concurrent system monitor, and application functions. In a conventional image processing system, a data set is stored in a file and a file name is used as a reference to a data set. Since a data set may be distributed among several systems in CIPE, a data set can not be referred to as a file. Therefore, CIPE employs a symbol-oriented data management system, where a data set is represented by a symbol structure.

3.1.1. Symbol structure

The symbol structure is designed so that a data set can be a subset of an existing file, of another symbol, or a set of assigned values. A three-dimensional data set is allowed to incorporate scalar, vector, image, and multi-spectral data. All data types (ex. char, int, float) are allowed and data can be stored on a disk in the host memory, and/or in the concurrent system's memory. The current symbol structure assumes that a concurrent system has multiple nodes and local memory like a hypercube. The symbol structure contains the following four types of information.

Data Location - Data location is indicated by a file name if a data set is on a disk, a load map if it is in the concurrent system, and/or a memory location if it is in the host system memory. When a function requests a data set, the data in the nearest location will be applied. For example, when an application function requests a data set to be distributed among the hypercube nodes and the data set is in the hypercube nodes as well as in the host memory, CIPE will redistribute the data in the hypercube accordingly instead of downloading the data from the host memory. More detailed data distribution and redistribution schemes are discussed in section 4.

Size field - The data size is described in three-dimensional terms: number of lines, samples, and bands. When the data is a subset of an existing file, starting locations are included for each dimension.

Data type - Data types are described as *char, short, int, float,* and *double*. The combination of the data size and the data type allows the proper memory allocation for each symbol.

Data Distribution - When data are distributed among nodes in a concurrent system, a map describing the distribution is composed and attached to the symbol structure. The load map is discussed in more detail in the CIPE programmer's guide (Volume II of this report).

3.1.2. Symbol Table Manipulation

The symbols are managed through a symbol table. The symbol table is a one-dimensional array which contains pointers to the symbol structures. When a symbol is created, its structure pointer is added to the symbol table; when a symbol is deleted, its structure pointer is deleted from the symbol table and the table is updated. The host monitor provides three symbol manipulation functions.

Create Symbol – A symbol can be easily created using a *cipe_create_symbol* function with its name, size (no. of lines, no. of samples, and no. of bands), and data type. The routine returns a pointer to the symbol table entry of the created symbol. When there is a symbol with the same name that exists in the symbol table, it will be overwritten. A *cipe_create_symbol_and_data* function is also provided so that necessary memory space is allocated for the symbol in case the data is to be created in the host system.

Get Symbol – Information for a symbol can be retrieved using a *cipe_get_symbol* with a desired symbol name. The symbol table is searched for a given symbol name and the corresponding pointer to the symbol structure is returned. Using the pointer and macro definition of each field of the symbol structure (defined in symbol.h), an application program can access the relevant symbol information.

Delete Symbol – A symbol may be deleted using *cipe_delete_symbol* with a symbol name when the symbol is determined to be no longer in use. For obvious reasons, a program may delete only the symbols that are created by the program for temporary use. When CIPE receives the symbol deletion command, it checks the data distribution status of the symbol and deletes the symbol from all coprocessors that have any portion of the symbol prior to deleting the symbol from the host monitor.

3.2. FILE MANAGEMENT

The main purpose of file management is to isolate a programming environment from the operating system and executive-specific file formats so that application programs are unaffected by changes in the operating system and/or file formats. The disadvantages of executive file management are the added layers of indirection, non-standard (executive dependent) programming, and the requirement of learning an extra set of file I/O routine formats. The CIPE file management tries to avoid these problems by maintaining standard C language I/O function call formats as much as possible and minimizing the indirection. More general purpose data file formats are also devised to be compatible with other executive environments.

3.2.1. Header File Structure

First, CIPE employs a header file separated from a data file. Most executives employ their own file structures using peculiar label formats which require a label conversion/removal process for other programs to access the data. For very large data files (50 Mbytes or more), the label conversion process becomes not only time consuming, but also unfeasible due to limited disk space. Second, the header file is designed to be an ascii file which can be edited and displayed using standard edit and type operating system commands, *e.g.* vi and cat. The ascii file format was chosen to minimize label retrieval/parse/conversion related chores. Finally, the header consists of only systematic information including size, type, and the starting location of data in the file. Since CIPE is designed to demonstrate the computational power of concurrent systems in a generic image processing environment, any specific operation related information was not included.

Executive Indicator - The executive indicator field contains the name of the executive for which the file was produced. For example, a file generated by CIPE will have "CIPE" as an executive indicator while a file generated by the VICAR (MIPL/JPL) will have "VICAR" instead. The indicator field is implemented so that the origin of a file can be preserved as well, so that a proper label processor can be activated if the label information should be extracted.

Offset - The offset field contains an integer which indicates where the actual data starts in the data file. This field is implemented so that CIPE can handle data sets with various types of label information. When CIPE opens a file, it skips the offset number of bytes so that the file index points to the beginning of the data area. The offset field is always zero for the files that are generated by CIPE.

Data size - The data set size is described by the number of bands, number of lines, and the number of samples in a line in order to incorporate three-dimensional data. A scalar value is described as a three-dimensional data set with one band, one line, and one sample. Similarly, a two-dimensional image is described as three-dimensional data with one band. Three-dimensional data sets are stored in a band interleaved fashion where a line from each band is concatenated to form a record in a file.

Data type - The type of a data set is expressed as **byte, short, int, float,** or **double**. The data type information is used for determining data size, for checking arithmetic operation compatibility, and for data conversion.

Others - Besides the systematic and general description of a data set, there can be data-set-related information which may or may not apply for a given application. Such information is an optional part of the header. Currently, the starting and ending wavelength fields are implemented in the CIPE file header for multispectral data sets.

The following example header file shows that this is a CIPE header for a data file "aisa" which contains a 256 byte label, for a three-dimensional multispectral data set of size 200 by 28 by 32. The data type is byte type which ranges between 0 and 255.

aisa.hdr

```
CIPE
offset = 256
number of lines = 200
number of samples = 28
number of bands = 32
type = byte
```

3.2.2. CIPE File I/O

The objective of CIPE's file I/O functions is to isolate the programming environment from file structure specifics. In general, a programmer does not have to perform explicit file I/O in CIPE since the data are managed by CIPE as if they were variables in a program. However, if a programmer wishes to perform an explicit file I/O which will read and create CIPE format files, the following functions may be utilized. CIPE file manipulation consists of an external level with which the programmer interfaces, and an internal level that CIPE uses to resolve file structure-dependent procedures.

For the external file I/O, CIPE provides a file read function (*cipe_read_from_file*) which reads header file information into a symbol structure and reads the data into the system memory. A user provides a symbol name and file name along with the area of interest. This function activates the CIPE internal file open function to open the header file and data file, and the read function to read the data.

CIPE also provides a file write function (*cipe_write_to_file*) which writes data associated with a given symbol into a file. This function gathers the data from the hypercube when the data is distributed among the hypercube nodes and composes a header structure. It activates the CIPE internal file create function to create a header file and data file, and the CIPE internal write function to write the data to the data file.

The CIPE internal file I/O consists of file open/create, header file read/write, and data file read/write functions. The file create function (*cipe_create*) creates a data file using a specified file name and also creates a header file. The header file name is composed by concatenating ".hdr" to the data file name. The file open function (*cipe_open*) opens a specified data file and the corresponding header file. This function seeks the actual data starting position in the data file using the offset information of the header file.

CIPE provides a header file read (*cipe_get_image_header*) function which parses the header file information and constructs an image header structure, and a header file write (*cipe_put_image_header*) function which writes the header structure content out to the header file. The read/write/close functions call operating system-provided functions.

3.3. CONCURRENT SYSTEM INTERFACE

CIPE is designed so that it can be applied as a testbed for various types of concurrent systems. The concurrent system testbed requires a unique programming environment which allows a programmer to access routines that are shielded from architectural complexities while each function is being executed by a specific concurrent architecture using its full potential. Such a programming environment was approached by implementing a concurrent system interface mechanism between a host system and a concurrent system.

Currently, an 8-node JPL/Caltech Mark III hypercube system is implemented in CIPE. A concurrent system is activated by a user command (*set coprocessor to CUBE*) or a menu selection. The activated concurrent system dynamically alters the CIPE execution path by loading the corresponding set of modules. A concurrent system may be activated at a given time and may be released and reaccessed within a CIPE session.

The hypercube-related executive software is divided into two parts: the host resident part, and the hypercube resident part. The host resident part of the software interfaces with the rest of the executive via standard subroutine calls and interacts with the hypercube resident part via a set of predefined commands. The hypercube resident part is called the hypercube monitor which waits for a command from the host process as a slave process. The host resident part of the hypercube software is discussed here with respect to command protocol, data distribution, and application program execution.

3.3.1. Hypercube Command Protocol

The host system monitor and the hypercube monitor have a master and slave relationship. The host monitor issues a command and the hypercube monitor performs the command. The common program environment for generic image processing applications was investigated to design a set of relevant commands and their execution protocols. Within CIPE, a hypercube application program is viewed as a routine that performs a specific data processing function on existing data sets. CIPE passes the parameters that are composed by a programmer (see CIPE programmer's guide, Volume II of this report, for detail) before executing the hypercube program. There are two types of parameters: symbol and non-symbol, where a symbol parameter represents a dataset whose name is specified by a user and a non-symbol parameter represents a variable internal to the program.

A typical scenario that takes place in the host system monitor when a hypercube application program is activated is illustrated below.

Pass the parameters to the hypercube monitor
(1) create the input symbols in hypercube,
(2) download the data of the input symbols according to the load maps,
(3) create the output symbols in the host,
(4) create the output symbols in the hypercube,
(5) pass the non-symbol parameter values to the hypercube.

Execute the function
(1) download the function module,
(2) execute the function,
(3) wait till execution is completed,
(4) readback the processed non-symbol parameter values.

A set of thirteen commands was designed and implemented in order for the host monitor to control the hypercube monitor and to carry out the application program execution following the above scenario. When the host monitor sends a command to the hypercube monitor, the two monitors synchronously execute a set of pre-arranged program steps for the command. The command syntax was designed in terms of the hypercube action.

Init - initialize the hypercube monitor. The hypercube monitor initializes its symbol table and trace flags.

Set_trace - set a debug trace flag. The hypercube monitor utilizes three debug flags: *cube_symbol_trace*, *cube_command_trace*, and *cube_data_trace*. to activate debug statements related to three different functions: *symbol table management*, *command execution*, and *data distribution/redistribution*.

Create_symbol - create a symbol in the hypercube symbol table. This command is sent when the host resident program expects the data of an output symbol to be generated by a hypercube resident module.

Delete_symbol - delete a symbol from the hypercube symbol table.

Read_loadmap - download a load map to each node of the hypercube. This command must be sent prior to data distribution and/or prior to data generation in order to let each node know the portion of data it is to receive and/or to generate.

Read_data - read data according to the corresponding load map.

Write_loadmap - upload a load map to the host. This command may be issued in case the host monitor does not have the updated data distribution load map.

Write_data - upload data to the host. The hypercube monitor writes all of the data of a given symbol name to the host. The host monitor then assembles the data in a contiguous manner.

Redist_data - redistribute data among nodes. When the data distribution is requested by an application function different from the distribution currently applied to the data, the host monitor sends this command with a new data distribution type. The hypercube monitor redistributes data to the specified data distribution type via inter-node data exchange.

Load_module - load an application module in the hypercube monitor's application program area.

Execute_module - executes the loaded application module.

Load_parameters - send the parameters for the loaded application module. The parameters include input and output symbol names and input and output variables.

Retrieve_parameters - send the output variable values back to the host.

Exit - terminate the hypercube monitor program execution.

3.3.2. Application Program Management

As mentioned above, an application program for a hypercube coprocessor consists of a host resident part and a hypercube resident part. The host resident part is loaded as discussed in Section 2.3, employing a function dictionary file. The host resident program then passes a corresponding hypercube resident module name to be loaded to CIPE (*cipe_cube_execute_module("module_name")*. CIPE reads the hypercube module and requests the hypercube monitor to load the module in a predetermined area in the cube. The relation between a host resident part and the hypercube resident part is similar to the relation between a main routine and a subroutine. As a main routine may call several subroutines, a host resident part may interact with several hypercube resident modules. The difference is that only one of the hypercube resident modules may be loaded at a time. The loading of an application program utilizes the incremental loading mechanism provided by Unix.

3.3.3. Hypercube Data Distribution

The image processing environment involves large volumes of data. Utilization of a hypercube system with many nodes significantly enhances the computation by processing the data concurrently. However, large data volume manipulation is somewhat complicated since the local memory architecture does not allow direct data sharing among nodes, and the data must be transferred via interconnecting channels.

In order to minimize the application programmer's effort in data manipulation, and to allow a very flexible data distribution scheme, a load map structure is composed where a programmer can specify an exact area in a given data set for each node. A three-dimensional data set is assumed and a subarea is described by starting and ending points in each dimension (line, sample, and band).

CIPE also provides a set of standard load maps for the frequently applied data distribution schemes. The currently supported standard distribution types are a broadcast distribution (BCAST_DIST), a horizontal distribution (HORIZ_DIST), a horizontal distribution with overlap (HORIZ_OVERLAP), a vertical distribution (VERT_DIST), a vertical distribution with overlap (VERT_OVERLAP), a grid distribution (GRID_DIST), and a grid distribution (GRID_OVERLAP). An application programmer simply requests a proper distribution type, and the load map is automatically composed.

The data can be redistributed from one standard distribution to another standard distribution without involving the host data download, except to BCAST_DIST.

Data distributions other than the standard distribution types are considered to be customized data distributions (CUSTOM_DIST) and require a user-composed load map for each node. The customized data distribution cannot take advantage of hypercube internal data redistribution. The actual data distribution and redistribution are discussed in more detail in Section 4.2.3. An application program determines how the data should be distributed and calls a proper data distribution routine. Details are described in CIPE programmer's guide (Volume II of this report).

4. HYPERCUBE RESIDENT MONITOR

A JPL/Caltech MARKIIIfp hypercube is employed as a concurrent coprocessor for CIPE. Each node is equipped with 4 Mbyte of memory, a Weitek floating point processor board, a Motorola 68020 I/O processor, and a Motorola 68020/68881 data processor. The implementation details of CIPE with regard to the hypercube are specific to the MARK III/CrOS hypercube, but the basic concept associated with utilization of a multi-node interconnection topology, localized memory system, and multiple programming capacity is shared among all MIMD systems. Therefore, the monitor design can be easily generalized to other MIMD systems.

The localized memory configuration of a hypercube system allows a very high level of concurrency, since a large number of nodes can be connected without creating a memory access bottleneck. However, three problems arise: distribution of data among multiple nodes is more complicated; data transfer between a host system and a hypercube system performed through a single hypercube node, node 0, creates a bottleneck; and the memory is shared by program and data, one affecting the other for the limited memory space. All of these are serious limitations for image processing applications where large data sets are commonplace.

To aid in distributing data to the hypercube, CIPE supports a set of standard data distributions and an easy-to-compose data load map structure for customized data distributions. These simplify the programming effort required in hypercube data manipulation.

CIPE employs a global data management scheme to permit data sharing among successive application programs which reduces the data transfer between the host and the hypercube. Data can be redistributed from some distribution types to others within the hypercube using the hypercube interconnection topology. Redistributing data within the cube reduces data traffic between the host and the hypercube, by allowing data to be shared among successive application programs with different data distribution requirements. Near-term enhancements to the hypercube include concurrent I/O capabilities between the host and multiple nodes of the hypercube. This enhancement should significantly reduce the data transfer bottleneck.

A specialized program management method, incremental loading, is devised for efficient hypercube memory utilization for program and data. When an entire data set cannot be present in the hypercube at one time, another level of data distribution difficulty is added, and the programming is much more complicated. Moreover, only the activated function module is required to be present in the hypercube node for data processing. Therefore, a trade-off has been made between the program area and data area. By loading only an activated function module to a preallocated program area, the most memory may be used for data.

The hypercube monitor performs these data and program management functions upon the host system monitor's command. There are eleven commands including seven data management related commands, three program management related commands, and one termination command. In this section, the execution of these commands is examined in detail.

4.1. INTERFACE TO HOST EXECUTIVE

When a CIPE user initializes the hypercube as the coprocessor of choice, the hypercube is reset and the hypercube monitor software is downloaded to the cube. Each node of the hypercube has an identical copy of the monitor and this monitor will remain resident in each node of the hypercube throughout the use of the hypercube as a coprocessor. The host and hypercube interact as a master and slave where the hypercube monitor waits to receive a command from the host, acknowledges its receipt, executes it, and waits for the next command.

The monitor interfaces to the host executive providing for the execution of eleven commands as mentioned in Section 3.4. The commands are for program and data management including *create_symbol* and *delete_symbol*, for symbol management, *read_load_map, read_data, write_load_map, write_data*, and *redist_data* for data distribution, and *load_module, execute_module*, and *set_trace* for program management. For each received command, the hypercube monitor and the host monitor follow predetermined communication steps to receive/transfer necessary information for the command execution. An execution sequence is described below, using a *create_symbol* command as an example.

Host Monitor	**Hypercube Monitor**
1. Send symbol name	*1. Receive a symbol name.*
2. Wait for acknowledgment	*2. Check if the symbol exists already. If so, then send error acknowledgment (NACK) and go back to wait; otherwise, send acknowledgment (ACK). Create the symbol entry in the symbol table.*
3. If NACK abort the command	*3. Receive the data size and allocate the data area; else, send data size*

4.2. DATA MANAGEMENT

A data set is distributed among nodes in the hypercube for processing. Each node may have the entire data set or a subarea of the data set according to the processing need. The data set distribution information is composed by the host monitor for each node and passed down to the hypercube monitor prior to actual data transfer. A data set mentioned in this section refers to a portion of data which resides in a given node.

Similar to the data management of the host monitor discussed in Section 3.2, a data set in the hypercube monitor is represented as a symbol which is associated with an attribute structure describing its name, data type, and load map (distribution scheme). A data set is accessed by its symbol name by application programs. The hypercube monitor employs a symbol management scheme very similar to that of the host monitor.

4.2.1. Symbol Structure

The symbol structure employed by the hypercube monitor receives the symbol information through a *create_symbol* command for the name and data type, and through a *read_load_map* command for the load type and data distribution scheme. The symbol structure contains name, data type, and a load map field. The load map field contains load type, location, and size field. The symbol structure declaration can be found in Appendix B under *elt_symbol.h*.

Data type - An element within a symbol is described for its type as *char, short, int, float, and double*. The combination of the data size and the data type allows a proper memory allocation for each symbol.

Load type - Data distribution type is described as BCAST_DIST, etc. (see Section 3.4 for the type definition). The load type information combined with the load map allows the data redistribution within the hypercube.

Load map - The load map describes the absolute data location of a symbol in a node in reference to the whole data set in three-dimensional terms. The starting location and its size for each dimension are expressed in lines, samples, and bands.

Data - Data area is allocated separately from the symbol attribute structure, and its address is stored in the data location field of the symbol structure.

4.2.2. Symbol Table

The symbols are managed through a symbol table. The symbol table is a one-dimensional array which contains the pointers to symbol structures. When a symbol is created, its structure pointer is added to the symbol table. When a symbol is deleted, its structure pointer is deleted from the symbol table and the table is updated. The hypercube monitor provides three symbol manipulation functions for hypercube module application programs to access the symbol table identical to those of the host monitor described in section 3.1.2.

4.2.3. Data Distribution

Data is distributed through the use of the monitor's *read_data* and *write_data* commands. Before data may be read, a symbol must exist and a load_map must have been read to know the data decomposition type and the specific load map parameters. Data is read into the cube in one of five distribution types: a BCAST_DIST, which sends the entire data set to each node; a HORIZ_DIST (HORIZ_OVERLAP), which breaks the data into horizontal regions (with overlap); a GRID_DIST (GRID_OVERLAP), which breaks the data into a grid (with overlap); a VERT_DIST(VERT_OVERLAP), which decomposes the data into vertical regions (with overlap); or a CUSTOM_DIST, which permits the decomposition of data in any manner that may be specified by six parameters (starting line, sample, and band, and number of lines, samples, and bands). Figure 4.1 illustrates data decomposition of above data distribution types in a four node hypercube system.

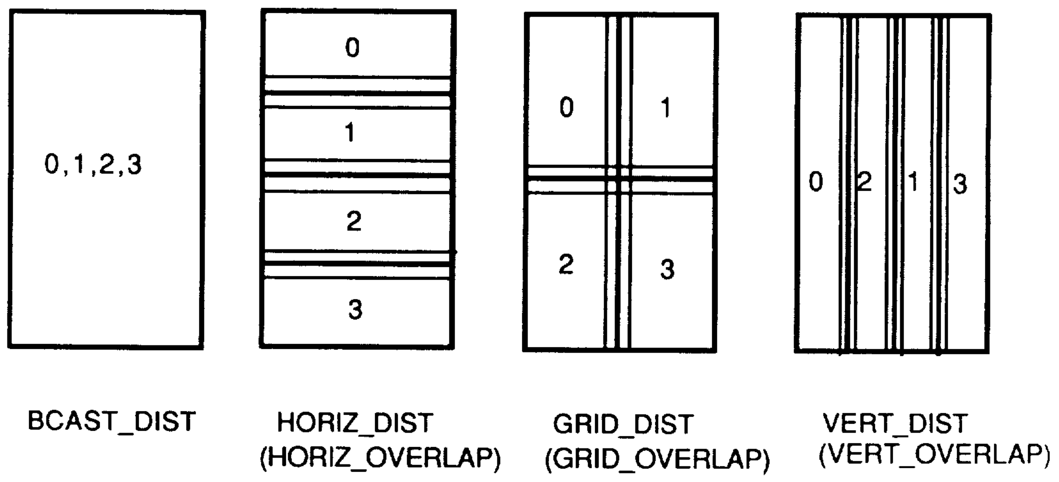

Figure 4.1 Data Decomposition

4.2.4. Data Redistribution

The data distribution schemes discussed above shield a programmer from the architectural complexities of the hypercube system. However, without a proper data handling, they may easily degrade the system performance by wasteful data traffic. In order to minimize redundant data traffic, CIPE devised an automatic data redistribution scheme without involving a host. Once data is distributed to the hypercube as one of the standard data distribution types, the data can be redistributed into another type via node-node data exchange.

Figure 4.2 illustrates the procedure involved in redistributing the data from HORIZ_DIST type to VERT_OVERLAP type. First, CIPE shuffles data into GRID_DIST type by exchanging the right side half with the left side half of a neighboring node. It repeats the process until each node has a vertical strip (VERT_DIST). Then, it exchanges the overlapped portion with a neighboring node to achieve VERT_OVERLAP data distribution. When the neighboring nodes are not connected physically, all of the nodes send out the overlapped portion, and each node extracts a needed portion from the pool. Figure 4.3 illustrates overall data redistribution relationship. Such a data redistribution scheme resolves the host-cube data transfer bottleneck and thus enhances the computational efficiency of the system significantly.

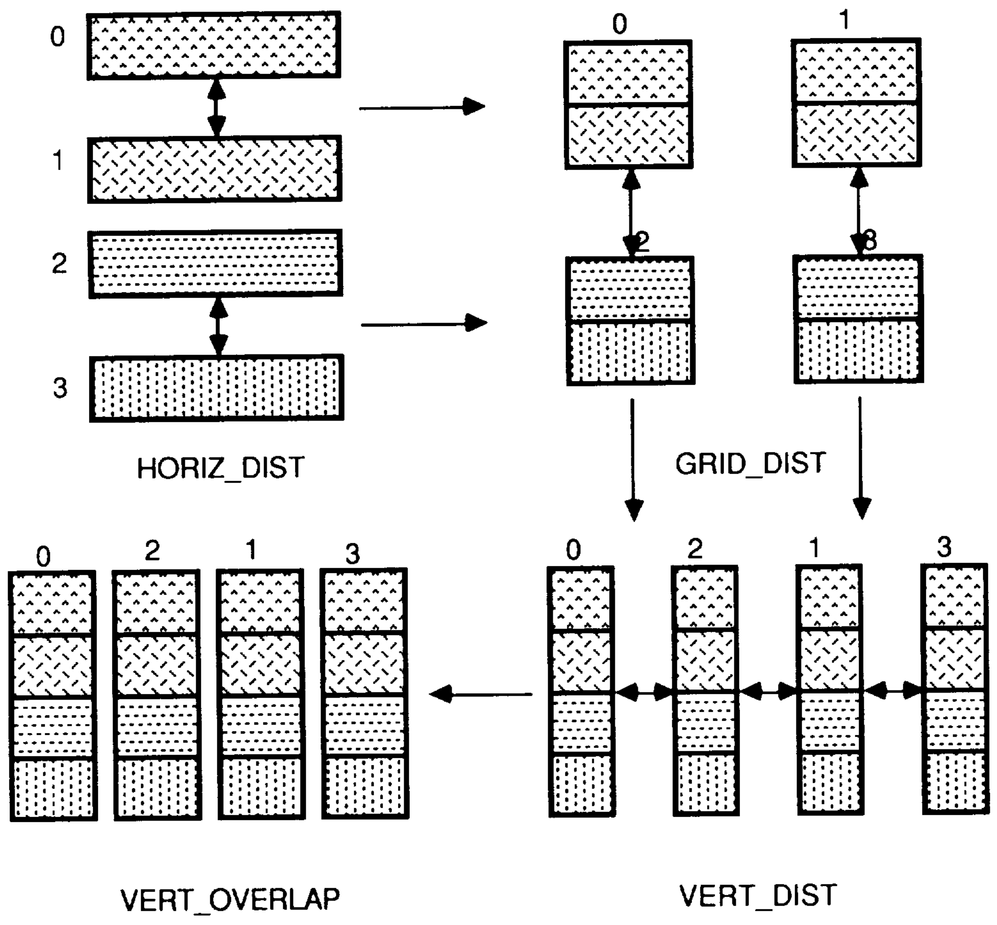

Figure 4.2 Data Redistribution Example (HORIS_DIST->VERT_OVERLAP)

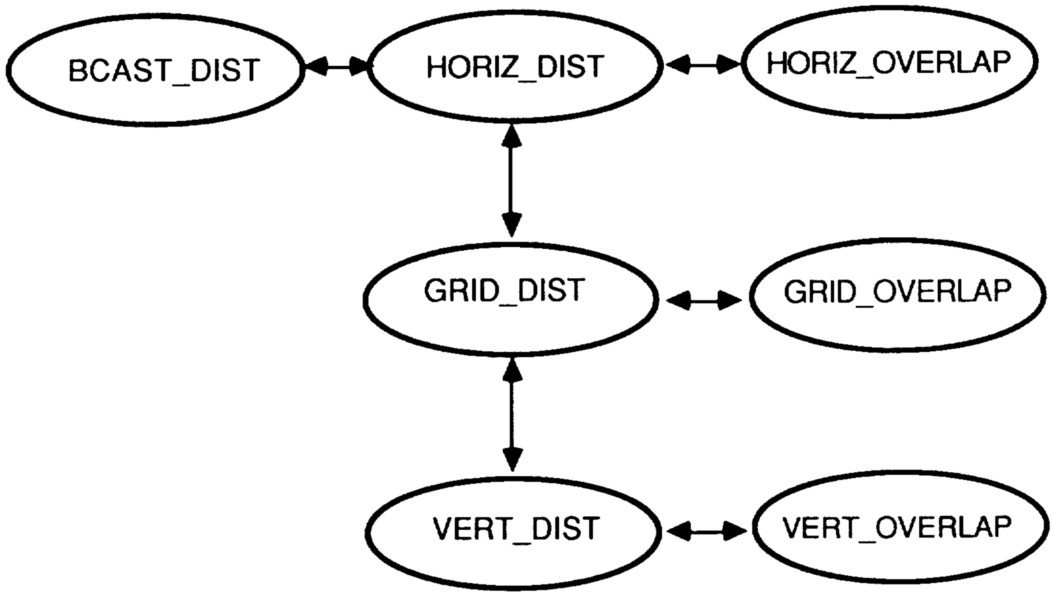

Figure 4.3 Data Redistribution Scheme

5. CONCLUSION

The previous four sections described the features and implementation details of CIPE with respect to the user environment, programming environment, and internal management of data and image processing functions. The user environment emphasizes friendly access to provided functions, and the programming environment provides system architecture-independent subroutine layers for user interface, file I/O, hypercube utilization, and display device manipulation. A symbol-oriented data management and an incremental program loading mechanism are implemented for more efficient utilization of the concurrent system, and to overcome the system I/O bottleneck common in a high rate data processing environment.

The CIPE development phase has been completed as of the fiscal year 1989. In the future, CIPE will be utilized as an environment for various computationally demanding data processing applications. Currently, CIPE provides over fifty image processing functions. The future direction of CIPE will focus on enlarging the function list for more complete coverage of data processing functionalities.

Appendix A – YET ANOTHER MENU MANAGER (YAMM) PROGRAMMER'S GUIDE

1. INTRODUCTION

One of the most time-consuming yet necessary tasks of writing any piece of interactive software is the development of a user-interface. Frequently, the development of even a very simple interface takes up valuable time that might be much better devoted to the design and implementation of the underlying task. Yamm (*Y*et *a*nother *m*enu-*m*anager) is an application-independent menuing package, designed to remove much of the difficulty and save much of the time inherent in the implementation of front-ends for large software packages.

This appendix gives an overview of Yamm's structure and design philosophy. Section 2 describes the menu configuration specification, a potentially user-specific description of an application's menu structure(s) and terminal interface. Section 3 describes the Yamm routines usable by an application. The steps involved in implementing a program using Yamm are covered in Section 4, and an example tying everything together follows in Section 5.

2. MENU CONFIGURATION

Applications running under the menu package consist of two parts: a description of the menu configuration (sometimes including terminal characteristics), which may be user-specific, and the body of application code, which includes a call to start up the menu package as well as calls to menu package interface routines.

The menu configuration is used at runtime to define the menu structure and any non-standard terminal characteristics; it may be read in from a programmer-specified file or included in the source. The example program in Section 5 uses both methods: if the menu configuration file (*menuconfig*) exists, the program uses it, otherwise it relies on the coded version.

Menu configuration specifications are composed of menu and terminal definitions.

Menu definitions define specific menus within the menu tree. Each definition contains the word *MENU* and a menu name, a series of prompt-name pairs, and an *END*. The name in each pair may be either a reference to an application function or the name of another menu defined within the menu configuration.

Terminal definitions are optional and allow the user to specify non-standard keyboard mappings and terminal capabilities. Each definition contains the word *TTY* and a series of terminal types (e.g., *vt100 Sun*) followed by key and capability definitions. Key definitions specify a generic key-name (e.g., *up_arrow, help*), what the user calls the key to be used (e.g., *go up, CTRL H*), and the actual ASCII sequence generated by the key. For example:

```
help:CTRL H:^H
```

The redefinable keys are:

up_arrow	help	enddataentry	func_0	func_4
down_arrow	root	endselect	func_1	func_5
right_arrow	previous	refresh	func_2	func_6
left_arrow	exit	toggle	func_3	func_7

Terminal capability definitions give the capability name and then the corresponding key sequence, as in:

```
clear_screen::\033H\033J                # ESC H ESC J – for VT52
```

The redefinable terminal capabilities are:

move	set_inv_video	reset_inv_video	clear_screen
set_underline	reset_underline	down_line	clear_eol
scroll_region	configure	reconfigure	set_gcs
reset_gcs	upper_left_cor	horiz_bar	upper_right_cor
vert_bar	lower_right_cor	lower_left_cor	

Character sequences may be formed using the usual \octal-value (e.g., *\033* for escape) and inserting the actual control or escape characters into the definition. The carat (^) may also be used to form easily-readable control characters. If system-special characters such as CTRL-C or CTRL-S are used, their normal functions are disabled inside the program.

Null padding may be specified for terminal capability definitions by preceding the character sequence with the number of following nulls. For example, if the definition for *clear_screen* above required a following padding of 10 nulls, the sequence could be changed to *10\033H\033J.* Neither type of terminal definition may be used in the menuconfig of a dynamically-created (nested) menu structure.

Note that the formats for the *move* and *scroll_region* sequences are different from those used by termcap. Specifically, the values must come in the reverse of the default termcap order, and start at 1 rather than 0. This simplifies integration of other terminals but may be confusing when compared with termcap entries.

Comments and white space may be used freely within the menu configuration commands; see the example for more information.

3. APPLICATION ROUTINES

Here are the functions callable from application programs. Most of these return −1 on error and 0 otherwise; see the example for information on actual usage. The structures and definitions used here are defined in menuapp.h.

```
menu_start(config_name,config,func_table,localinit,localreinit,logfile)
char *config_name;
char *config[];
struct func_name_pair func_table[];
int (*localinit)( ),(*localreinit)( );
FILE *logfile;
```

invokes the menu interface. It must be called before any of the other functions listed here. *Config_name* gives the name of the menu configuration file; this may be *NULL* if the configuration is hardcoded and no user override is to be permitted. *Config* gives a hardcoded default configuration; this may be *NULL* if there will always be a configuration file. *Func_table* is a list of function name-code pairings; this is illustrated below. *Localinit* and *localreinit* point to functions invoked automatically when the specified menu tree is entered and left, respectively. Either or both of these may be *NULL*, but if they are used, should return −1 on error and 0 otherwise. If session logging is desired, *logfile* should contain a file pointer to an open file. Otherwise it should be *NULL*. If logging is used, closing the file is left to the application.

```
getpar(params,num_params,error_code,help_code)
struct param *params;
int num_params;
int (*error_code)( ),(*help_code)( );
```

requests parameter values from the user using a data-entry screen. *Params* points to a series of parameter definition structures, *num_params* contains the number of parameters defined, *error_code* points to a function able to detect errors or is *NOECHK* if none, and *help_code* points to a function which prints parameter information for the user on request or is *NOHELP* if none.

Parameter definition is done using *def( )*, which is described below.

The programmer-supplied error code receives three parameters: a pointer to the parameter definition structures, the number of the parameter to check, and the value supplied, an *int-double-char* * union. (Booleans are treated as integers for purposes of error checking.) Error checking functions typically use *unspecified( )*, *new_value( )*, and *unspecify( )*, described below, to detect and correct errors. The function must return -1 if the parameter value is unacceptable, and 0 otherwise.

Help code receives two parameters: a pointer to the parameter definition structures, and the number of the parameter in question. One additional special case is also provided for. On entry into *getpar( )*, Yamm calls the help routine specifying parameter number -1 to enable output of any initial help or welcome message. This is illustrated in the example.

```
def(params,param_num,prompt,type,value_ptr,...)
struct param *params;
int param_num;
char *prompt;
int type;
unsigned char *value_ptr;
```

defines parameters on the data-entry screen. The first five parameters are required, in the order shown; subsequent parameters may come in any order and are optional, with appropriate defaults.

params is the name of a block of storage reserved for parameters, as in *struct param params[5];* to reserve space for five parameters,

param_num is the number of the parameter being defined, starting with 0,

prompt is a pointer to a short textual description of the value which will be used as a prompt string,

type is the parameter type, *INT*, *FLOAT*, *DOUBLE*, *STRING*, or *BOOL* (YES/NO), and

value_ptr is a memory location to receive the specified value(s).

In addition, a variety of parameters may be defined by specifying a keyword followed by a comma and the relevant value(s):

INCR precedes an increment in bytes from *value_ptr* to use when multiple values are specified – the default is parameter-type dependent: 4 for INT and FLOAT, 8 for DOUBLE, 0 for STRING, and 4 for BOOL,

WID precedes the field width in columns needed to specify the value – the default is parameter-type dependent: 6 for

INT, 8 for FLOAT and DOUBLE, 20 for STRING, and 3 for BOOL,

COUNT precedes a pointer to an integer variable which will receive the number of values specified – the default is to not supply a count,

LINE precedes the line location (where 0 is top of window) of the prompt – the default is 0,

COL precedes the column location of the prompt (where 0 is the left edge) – the default is 0,

DUP precedes the number of values expected, or the negative of the maximum number of values if the number is variable – the default is 1,

DEFAULT indicates that the area referenced by the *value_ptr* parameter contains default values which should be displayed to the user – the default is "no default,"

GROUP precedes the group number (greater than or equal to zero) of the parameter; if the user specifies a value for any parameter within a group, all parameters must be specified. – the default is "no group,"

REQ indicates that a value for the parameter is required – the default is "not required,"

ENUM precedes two values, a number, and a pointer to that many values, from among which the user must choose in valuing the parameter; when the values are STRINGs (for STRING-type parameters), the pointer is to a list of pointers to these values, i.e., the list is defined: *static char *legal_values[] = {"value1","value2",...};*

ENTRYLINE precedes the line number (starting with 0) of the first data-entry blank for this parameter – the default is to use the line number of the prompt, except when multiple data items are grouped together in a table format, in which case the default is the number of the line below the prompt,

ENTRYCOL precedes the column number (starting with 0) of the first data-entry blank for this parameter – the default is that column to the right of the prompt, except when multiple data items are grouped together in a table format, in which case the default is the column number such that the prompt is centered over the column, and

END functions as an end-of-definition keyword and is required.

See the example for numerous parameter definitions.

```
unspecified(param)
int param;
```

returns true if no value has been specified by the user for parameter *param*. This is most useful in the application-supplied error-checking routine, where it can be used to make sure that a specified value does not conflict with other related parameter values. This is typically used with *new_value( )* and *unspecify( )* below.

```
new_value(params,entry)
struct param *params;
int entry;
```

notifies the menu package that the value of the parameter numbered by *entry* has been changed, usually because of a change in the value of a related parameter.

```
unspecify(param)
int param;
```

removes the value of the specified parameter. This is usually only useful inside error-checking routines, when a newly specified value is incompatible with the value of another parameter and the latter must be removed.

```
create_new_subtree(config_name,config,func_table,localinit,localreinit)
char *config_name;
char *config[];
struct func_name_pair func_table[];
int (*localinit)( ),(*localreinit)( );
```

dynamically creates a supplemental menu tree. The arguments are similar to the arguments for *menu_start( )* above. The code pointed to by *localinit* will be executed immediately, and if executed without error (i.e., code returns 0) then the user will be placed at the root of the new menu tree. The function pointed to by *localreinit* is run on exit.

Recursion is permissible with care, but the use of *static* variables should be avoided.

4. TO IMPLEMENT A PROGRAM USING YAMM

The following steps are sufficient to implement most programs using Yamm:

1. Determine where the menu configuration file, if any, will be kept, and if desired, create a default configuration. Next, declare the functions which will be called through the menus and assign to each menu-callable function a name. Write any application-specific initialization and reinitialization code. If logging is desired, *fopen* a file to contain the log. Finally, write the call to *menu_start( )*, using the data structures above as parameters.

   Example:

```
char *config_name = "./menuconfig";
char *config[] = NULL;

int function1(),function2(),function3();

struct func_name_pair funcs[]={
    {function1,"func1"},
    {function2,"func2"},
    {function3,"func3"},
    {0,"/keep last"},
};

my_init() { /* application init code here */ }
my_reinit() { /* reinit code here */ }

main() { menu_start(config_name,config,funcs,my_init,my_reinit,NULL); }
```

2. Create a menu configuration file. An example for the above function table might be:

```
MENU mainmenu
DoFunc1/func1           # DoFunc1 is the prompt; if chosen, calls func1.
OtherFuncs/submenu
END

MENU submenu
```

```
DoFunc2/func2          # func2 and func3 must be associated with a
DoFunc3/func3          # function using a function table as above.
END
```

3. In each application routine called by the menus, define parameters using *def( )*, and write *chk_* and *help_* routines. (You may want to leave writing these routines until the application runs correctly. Use *NOECHK* and *NOHELP* in *getpar( )* calls meanwhile.)

4. Compile the application code, linking with the menu library and libtermcap.a. For example,

   ```
   cc mycode.c -o mycode -lyamm -ltermcap
   ```

   The resulting executable should display the menus as configured in your configuration file and allow you to execute the application functions. Once everything is working, you may want to change *config_name* and keep the file in a different place or keep the file contents exclusively in the code.

5. AN EXAMPLE

This section contains a detailed step-by-step implementation of a program using the menu package.

1. First create menuconfig:

```
# This is the menu configuration file. Each menu consists of MENU with
# the menu name, a number of menu entries, and an END; a menu entry is
# made up of the text to display for the menu selection and the
# function or submenu name, as appropriate. Terminal-specific
# sequences and keystroke-function mappings start with the TTY keyword
# and end with END. See the programmer documentation for more
# specifics. Comments should be prefaced with a #; all such text is
# ignored. Indentation may be used to illustrate menu structure. This
# text may also be put into the code. SCCS yamm_prog_guide 4.22 - 7/12/88

MENU mainmenu
Info/information      # function
Two Legs/twolegs      # submenu
Four Legs/fourlegs    # submenu
END
```

```
MENU twolegs
Cockatiel/bird
Parakeet/bird
Pigeon/bird
END

MENU fourlegs
Alsatian/dog
Collie/dog
German Shepherd/dog
END
```

2. Add in the call to *menu_start( )*, defining the menu config filename and menu config, function table, local (re)initialization routines, and logging file pointer. Make sure you include menuapp.h where necessary. Compile the code, linking with the menu (-lyamm) library and libtermcap.a (-ltermcap).

   Here is the code from the demo program contained in demo.c:

```
#ifndef lint
static char sccsid[]="@(#)yamm_prog_guide     4.22 7/12/88";
#endif lint

#include <ctype.h>
#include <stdio.h>
#include <strings.h>
#include "menuapp.h"

int information(),dog(),bird();

char *config_name = "./menuconfig";
char *config[] = {
    "MENU mainmenu","Information/information","Two Legs/twolegs",
       "Four Legs/fourlegs","END",
    "MENU twolegs","Cockatiel/bird","Parakeet/bird","Pigeon/bird","END",
    "MENU fourlegs","Alsatian/dog","Collie/dog","German Shepherd/dog","END",
    "/keep last"
};

struct func_name_pair func_table[]={
    {information,"information"},
    {dog,"dog"},
    {bird,"bird"},
    {0,"/keep last"},
};
```

```
main()
{
   FILE *logfile;
   logfile = fopen("demo.log","w");
   (void)menu_start(config_name,config,func_table,(int (*)())NULL,
      (int (*)())NULL,logfile);
   (void)fclose(logfile);
}

information()
{
   (void)printf("This is the Pets pet registry program.  ");
   (void)printf("Since this is for ");
   (void)printf("menu system demonstration only, no information is ");
   (void)printf("actually saved.  You are now at the top of a ");
   (void)printf("short menu tree.  By selecting menu entries ");
   (void)printf("(use the help key for more info) you can ");
   (void)printf("descend the tree in a manner compatible with your ");
   (void)printf("particular animal.  ");
   (void)printf("When you've reached the bottom of ");
   (void)printf("the menu tree, the function selected will display a ");
   (void)printf("parameter ");
   (void)printf("entry screen.  Feel free to move about the screen ");
   (void)printf("(use the help key for specifics) ");
   (void)printf("and fill in values.  If the program erases something ");
   (void)printf("immediately after you've typed it, the input is ");
   (void)printf("illegal.  ");
   (void)printf("Type the end-data-entry key ");
   (void)printf("when you're done entering parameter values.  Note that ");
   (void)printf("the program may refuse to leave if some parameter ");
   (void)printf("value is required and unspecified.  ");
   (void)printf("When you've reached the bottom of ");
   (void)printf("the menu tree, the function selected will display a ");
   (void)printf("parameter ");
   (void)printf("entry screen.  Feel free to move about the screen ");
   (void)printf("(use the help key for specifics) ");
   (void)printf("and fill in values.  If the program erases something ");
   (void)printf("immediately after you've typed it, the input is ");
   (void)printf("illegal.  ");
   (void)printf("Type the end-data-entry key ");
   (void)printf("when you're done entering parameter values.  Note that ");
   (void)printf("the program may refuse to leave if some parameter ");
   (void)printf("value is required and unspecified.  ");
}

/*-----------------------------------------------------------------
This function demonstrates the basic structure.  Note that some values
```

```
are statics. These are initialized to default values the first time, and
then when this function is called in the future, the last values used are
the defaults. Pet's name goes into name (up to 20 characters), and is
required; there is also a default. Foods takes up to 5 values, stored
starting at location foods and every 20 bytes thereafter; the number of
foods selected is returned in food_count. Eye color is simply a string,
stored in eyes. The next two are boolean (YES or NO); note that booleans
should always be initialized. The weights parameter illustrates how to
request input of a matrix (in this case, a weights matrix for your dog).
Note the use of the GROUP keyword (to enable table formation); this is
somewhat awkward but very simple and flexible. The getpar request
displays the menu and returns with the values, which in this example, are
ignored.
---------------------------------------------------------------------*/
dog()
{
   int chk_dog(),help_dog(),food_count;
   char name[25],foods[20*5];
   static double size[2];
   static char eyes[20];
   static int license=0,plays=0,first_time=1;
   static float wgts[9];
   struct param params[9];
   (void)strcpy(name,"Rover");
   if (first_time) {
      (void)strcpy(eyes,"brown");
      first_time=0;
   }
   (void)def(params,0,"Pet's name",STRING,name,WID,20,LINE,1,DEFAULT,REQ,
      END);
   (void)def(params,1,"Favorite food(s)",STRING,foods,WID,15,INCR,20,
      COUNT,&food_count,LINE,1,COL,40,DUP,-5,END);
   (void)def(params,2,"Eye color",STRING,eyes,WID,10,LINE,3,DEFAULT,REQ,
      END);
   (void)def(params,3,"Licensed?",BOOL,&license,LINE,5,DEFAULT,END);
   (void)def(params,4,"Likes to chase cars?",BOOL,&plays,LINE,7,DEFAULT,
      END);
   (void)def(params,5,"Ht, Length (fractional) ?",DOUBLE,size,
      WID,10,LINE,9,DUP,2,REQ,END);
   (void)def(params,6,"Weights",FLOAT,wgts,INCR,sizeof(float)*3,LINE,7,
      COL,50,DUP,3,GROUP,0,ENTRYLINE,8,ENTRYCOL,40,END);
   (void)def(params,7,"Weights",FLOAT,wgts+1,INCR,sizeof(float)*3,LINE,7,
      COL,50,DUP,3,GROUP,0,ENTRYLINE,8,ENTRYCOL,50,END);
   (void)def(params,8,"Weights",FLOAT,wgts+2,INCR,sizeof(float)*3,LINE,7,
      COL,50,DUP,3,GROUP,0,ENTRYLINE,8,ENTRYCOL,60,END);
   if (getpar(params,9,chk_dog,help_dog) == -1) return;
/* continue with rest of program here - there isn't any in these examples */
```

```
   (void)printf("%f %f0,size[0],size[1]);
}


/*----------------------------------------------------------------------
This is the error-checking routine for dog() above, which is very
straightforward. Number is the number of the parameter, and value is the
value specified (a union of three differently typed variables). If this
returns a non-zero value, getpar will erase what the user typed, indicating
that the input was erroneous.
----------------------------------------------------------------------*/
/*ARGSUSED*/
chk_dog(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
   int error;
   switch (number) {
      case 0: error = alphawhite(value.s); break;
      case 1: error = alphawhite(value.s); break;
      case 2:
         if ((error=color(value.s))!=0) {
            (void)printf("Unknown color. Use blue, gray, green ");
            (void)printf("or brown.");
         }
         break;
      default: error = 0; break;
   }
   return(error);
}

/*----------------------------------------------------------------------
This is the help routine for the dog() function above. When the user is
entering parameter values and presses the help key, getpar passes the
number of the parameter to this routine which prints the information
desired. Also, the help function is called by getpar immediately on startup
with a parameter value of -1 in case application has some specific prompt
for user.
----------------------------------------------------------------------*/
/*ARGSUSED*/
help_dog(params,number)
struct param *params;
int number;
{
   switch (number) {
      case -1: (void)printf("Welcome to the Dog menu"); break;
      case 0: (void)printf("Name of dog; the default is Rover"); break;
      case 1: (void)printf("Favorite foods"); break;
      case 2: (void)printf("Eye color"); break;
      case 3: (void)printf("Is your dog licensed?"); break;
      case 4:
         (void)printf("Do its ears perk up whenever it hears a VW?");
         break;
   }
}

/*--------------------------------------------------------------------
This is somewhat more complicated than the previous function. Parameter
2, and parameters 3 and 4 are mutually exclusive. (If a bird has its
wings clipped, it cannot fly; if it can fly, its wings are not clipped.)
This is enforced by chk_bird(). Also, if parameter 3 is specified, then
parameter 4 must be, and vice versa; hence, they are specified as
belonging to group 0 instead of NOGRP. This is enforced by the getpar
routine. The first parameter illustrates the use of legal-value
sets. Finally, if the third name is selected, the program dynamically
creates another similar menu below the current one and goes into it.
This one contains no information option.
--------------------------------------------------------------------*/
bird()
{
   int clipped=0,dist,chk_bird(),help_bird();
   float height;
   static char name[25];
   static int first_time = 1;
   char units[20];
   struct param params[5];
   static char *poss_values[] = {"rover","king","ohnooo!"};
   static char *newconfig[] = {
      "MENU mainmenu","Two Legs/twolegs","Four Legs/fourlegs","END",
      "MENU twolegs","Cockatiel/dog","Parakeet/dog","Pigeon/dog","END",
      "MENU fourlegs","Alsatian/bird","Collie/bird","German Shepherd/bird",
         "END","/keep last"
   };
   if (first_time) {
      (void)strcpy(name,"king");
      first_time = 0;
   }
   (void)def(params,0,"Pet's name",STRING,name,WID,20,LINE,2,REQ,ENUM,3,
      (unsigned char *)poss_values,DEFAULT,END);
   (void)def(params,1,"Height (fractional inches)",FLOAT,&height,WID,6,
      LINE,4,COL,40,END);
   (void)def(params,2,"Wings clipped?",BOOL,&clipped,LINE,4,DEFAULT,END);
   (void)def(params,3,"Maximum flying distance (integer)",INT,&dist,WID,4,
        LINE,8,GROUP,0,END);
    (void)def(params,4,"Units",STRING,units,WID,10,LINE,8,COL,40,GROUP,0,
        END);
    if (getpar(params,5,chk_bird,help_bird) == -1) return;
    if (strcmp(name,"ohnooo!") == 0) {
        (void)printf("Ohnooo!");
        (void)create_new_subtree((char *)NULL,newconfig,func_table,
            (int (*)())NULL,(int (*)())NULL);
    }
    else (void)printf("What kind of name is
/* continue with rest of program here - there isn't any in these examples */
}

/*-----------------------------------------------------------------
Note in this function the handling at cases 2 through 4.  If parameter 2
becomes true (wings clipped), specified values for parameters 3 and 4 are
removed.  Similarly, if parameter 3 or 4 is given a value, then parameter
2 is cleared.  This is done using unspecify, which removes the value for
the specified parameter, and new_value, which tells the menu package that
the error routine has changed a value and it should be redisplayed.
-----------------------------------------------------------------*/
/*ARGSUSED*/
chk_bird(params,number,value)
struct param *params;
int number;
union {int i; double f; char *s;} value;
{
    int error,i,pr_msg;
    char locbuf[30];
    switch (number) {
        case 0: error = alphawhite(value.s); break;
        case 1:
            error = (value.f<2. | value.f>15.? -1:0);
            if (error == -1)
                (void)printf("Birds must be between 2 and 15 inches tall.");
            break;
        case 2:
            error = 0;
            pr_msg = 0;
            if (value.i && !unspecified(3)) {
                (void)unspecify(params,3);
                pr_msg=1;
            }
            if (value.i && !unspecified(4)) {
                (void)unspecify(params,4);
                pr_msg=1;
            }
        if (pr_msg)
           (void)printf("Changing flying distance to be undefined.");
        break;
     case 3:
        error = (value.i<0? -1:0);
        if (error == 0 && !unspecified(2) && *((int *)params[2].value)) {
           *((int *)params[2].value) = 0; /* or clipped=0; */
           (void)new_value(params,2);
           (void)printf("Changing wings to
        }
        break;
     case 4:
        error = 0;
        (void)strcpy(locbuf,value.s);
        for (i=0;i<strlen(locbuf);i++)
           if (isupper(locbuf[i])) locbuf[i]+=32;
        if (strcmp(locbuf,"inches")!=0 && strcmp(locbuf,"feet")!=0 &&
           strcmp(locbuf,"yards")!=0 && strcmp(locbuf,"miles")!=0)
              error = -1;
        if (error == -1)
           (void)printf("Please use inches, feet, yards, or miles.");
        else if (!unspecified(2) && *((int *)params[2].value)) {
           *((int *)params[2].value) = 0; /* or clipped=0; */
           (void)new_value(params,2);
           (void)printf("Changing wings to
        }
        break;
     default: error = 0; break;
  }
  return(error);
}

/*ARGSUSED*/
help_bird(params,number)
struct param *params;
int number;
{
  switch (number) {
     case 0: (void)printf("Name of bird."); break;
     case 1: (void)printf("Height, between 2 and 15 inches"); break;
     case 2: (void)printf("Are wings clipped?"); break;
     case 3:
        (void)printf("Maximum flying distance.  ");
        (void)printf("If this is specified, distance units must also ");
        (void)printf("be specified and vice versa.");
        break;
     case 4:
            (void)printf("Distance units.  ");
            (void)printf("If this is specified, maximum flying distance ");
            (void)printf("must also be specified and vice versa.");
            break;
    }
}

/*--------------------------------------------------------------------
Error checking routines:
alphawhite(buf) returns 0 iff buf is only letters and white space
color(buf) returns 0 iff buf is a valid color
--------------------------------------------------------------------*/
alphawhite(buf)
char *buf;
{
    int i;
    for (i=0;i<strlen(buf);i++)
       if (!isalpha(buf[i]) && !isspace(buf[i])) return(-1);
    return(0);
}

color(buf)
char *buf;
{
    int i;
    char locbuf[30];
    (void)strcpy(locbuf,buf);
    for (i=0;i<strlen(locbuf);i++) if (isupper(locbuf[i])) locbuf[i]+=32;
    if (strcmp(locbuf,"brown")==0 | strcmp(locbuf,"blue")==0 |
       strcmp(locbuf,"green")==0 | strcmp(locbuf,"gray")==0) return(0);
    else return(-1);
}
```

6. INTERNALS

There are a few internal issues that a programmer should be aware of.

Yamm changes the handling of stdout and stderr in order that program output can be intercepted and properly placed in the menu windows. In the unlikely event of a bug in the output handling routines, traces and other output statements might not function correctly. To bypass this processing, the file pointer *mm_termfp* may be referenced using *extern*; this pointer directly accesses the screen.

Also, Yamm starts up a subprocess (*menuwatch*) and allocates a semaphore in order to properly handle application I/O. Otherwise application I/O would be limited to 4096 bytes (the size of the buffered I/O buffer) and flushed only on application completion. The subprocess waits for application I/O and then signals Yamm to print it in the proper place. Normally, when an application dies, the subprocess will also die, freeing the semaphore in the process. However, if the subprocess is killed, the semaphore may not get removed; when enough semaphores build up, Yamm will not start properly. To avoid this, the user should never kill the subprocess explicitly. If semaphores are left around, *ipcrm(1)* may be used to remove them.

In order to dump the screen, Yamm creates a temporary file in the user's current directory and then attempts to troff this file using the pipeline *tbl | ptroff -ms*. If this succeeds, the file is then deleted; otherwise the file remains. The user may specify an alternative command (or no command) by setting the environment variable *MENU_HARDCOPY_CMD*. For example, the command *setenv MENU_HARDCOPY_CMD 'tbl %s | ptroff -ms'* would print hardcopies without deleting the originals. The names of temporary files are determined by the day and time of the screen dump; e.g., *menupic1091252* was dumped Monday (1), at time 9:12:52.