

# ornl

ORNL/TM-10679  
(CESAR-88/04)

OAK RIDGE  
NATIONAL  
LABORATORY

MARTIN MARIETTA

## A Computer Vision System for a Hypercube Concurrent Ensemble

J. P. Jones  
R. C. Mann  
E. M. Simpson

OAK RIDGE NATIONAL LABORATORY  
CENTRAL RESEARCH LIBRARY  
CIRCULATION SECTION  
BOWEN ROOM 175  
**LIBRARY LOAN COPY**  
DO NOT TRANSFER TO ANOTHER PERSON  
If you wish someone else to see this  
report, send it along with report and  
the library will arrange a loan.

OPERATED BY  
MARTIN MARIETTA ENERGY SYSTEMS, INC.  
FOR THE UNITED STATES  
DEPARTMENT OF ENERGY

Printed in the United States of America. Available from  
National Technical Information Service  
U.S. Department of Commerce  
5285 Port Royal Road, Springfield, Virginia 22161  
NTIS price codes—Printed Copy: A04; Microfiche A01

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Engineering Physics and Mathematics Division

A COMPUTER VISION SYSTEM FOR A HYPERCUBE CONCURRENT ENSEMBLE

J.P. Jones,\* R.C. Mann,\* and E.M. Simpson<sup>†</sup>

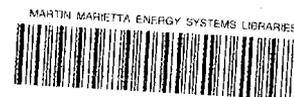
Date Published - August 1988

Research sponsored by the U.S. Army  
Human Engineering Laboratory and the  
U.S. Department of Energy

\*Advanced Computing and Integrated Sensor Systems Group  
Center for Engineering Systems Advanced Research

<sup>†</sup>University of West Florida  
Pensacola, Florida

Prepared by the  
OAK RIDGE NATIONAL LABORATORY  
Oak Ridge, Tennessee 37831  
operated by  
Martin Marietta Energy Systems, Inc.  
for the  
U.S. DEPARTMENT OF ENERGY  
Under Contract No. DE-AC05-84OR21400



3 4456 0280503 4



## CONTENTS

Summary.....	v
1. Introduction.....	1
2. Hypercubes.....	2
3. A Programming Model.....	5
3.1 The I/O System.....	5
3.2 Concurrent Utilities.....	10
4. Decomposition and Communications.....	14
5. Discussion.....	21
6. Acknowledgements.....	23
7. References.....	24
A. Appendix.....	26
A.1 Concurrent Utilities.....	26
A.1.1 Requirements.....	26
A.1.2 Initializations and Concurrent Data Structures....	27
A.1.3 I/O Utilities.....	29
A.1.4 Very Low Level Utilities.....	32
A.1.5 Graphics.....	36
A.1.6 Low Level Utilities.....	38
A.1.7 Intermediate Level Utilities.....	43
A.2 Host I/O Server and Utilities.....	46
A.2.1 The I/O server and its requirements.....	46
A.2.2 VME Subsystem.....	48
A.2.3 SBX Graphics Device.....	51
A.2.4 Disk Access.....	52
A.2.5 Hypercube image I/O.....	53
A.2.6 Newport Rotation Stages.....	53
A.2.7 Console.....	56
A.2.8 Miscellaneous.....	57



## Summary

A system of image processing and analysis utilities for a general-purpose hypercube topology concurrent multiprocessor system is described. The purpose of the system is to provide an efficient development and run-time environment for theoretical and experimental inquiry into computational vision for mobile and manipulative robots. An additional objective is to provide an environment with rich support for the development of concurrent computer vision algorithms. A number of principles for programming concurrent systems are embodied in the implementation of this system. In particular, all significant computation is performed by the hypercube proper, reserving the host processor exclusively for input/output functions. One consequence of this principle is that applications are developed by viewing the concurrent multiprocessor as a single computer with a special internal structure, rather than as a number of independent machines. Applications derive from a single source, rather than many, reducing development time without sacrificing run-time efficiency. This document serves as an introduction to the image processing and analysis system, its principles of operation, and as a reference manual for its use.



## 1. Introduction

Due to large data sets, high throughput and low turnaround time requirements, single processor computers are insufficient for many applications in image processing and computer vision. A variety of alternative computer architectures have been proposed, some quite specialized [12, 15, 18, 21]. These systems are characterized by parallel or concurrent distributed processors.

In the Advanced Computing and Integrated Sensor Systems Group of the Center for Engineering Systems Advanced Research, we have been exploring the application of a specific general-purpose concurrent multiprocessor system [16, 13] to a variety of problems in low and intermediate level image processing and analysis, focussing specifically on robotics applications [1, 9, 10]. This exploration has resulted in the implementation of a set of image processing and analysis utilities which exploit the multiprocessor ensemble with efficiency approaching 100%.

The objectives of this effort were threefold. First, to develop and implement on a hypercube multiprocessor network a multi-purpose high-performance system which is easy to learn, easy to program, and flexible enough to support a variety of computer vision applications. Second, to provide a development environment for research in computer vision problems relevant to mobile and manipulative robots. Third, to provide a development environment with

rich support for research in image processing and analysis algorithms for concurrent computers.

This document serves as an introduction to the image processing and analysis facility and its principles of operation, and as a reference manual for its use. Section 3 reviews certain features of current hypercube multiprocessors. Section 4 describes the principles around which the system is organized. Section 5 describes image mapping onto the hypercube ensemble and some fundamental communications algorithms. Section 6 presents conclusions. An appendix describes the local hardware configuration and the detailed operation of each system component.

## 2. Hypercubes

A D-dimensional hypercube network (see Fig. 1) is composed of  $P=2^D$  processors. Each processor (node) is assigned a D-bit number between 0 and  $2^D-1$ . Direct connections are implemented between nodes whose numbers differ in exactly one bit. A pair of such nodes are said to be adjacent. Nodes which are not adjacent have a Hamming distance of H, where H is equal to the number of bits which are different in the the node numbers. The maximum Hamming distance in a D dimensional hypercube is therefore D.

The communication channels between nodes of the cube are referred to as links. The collection of all links between nodes whose logical node numbers differ in a single specific bit is called an "axis", since these channels are

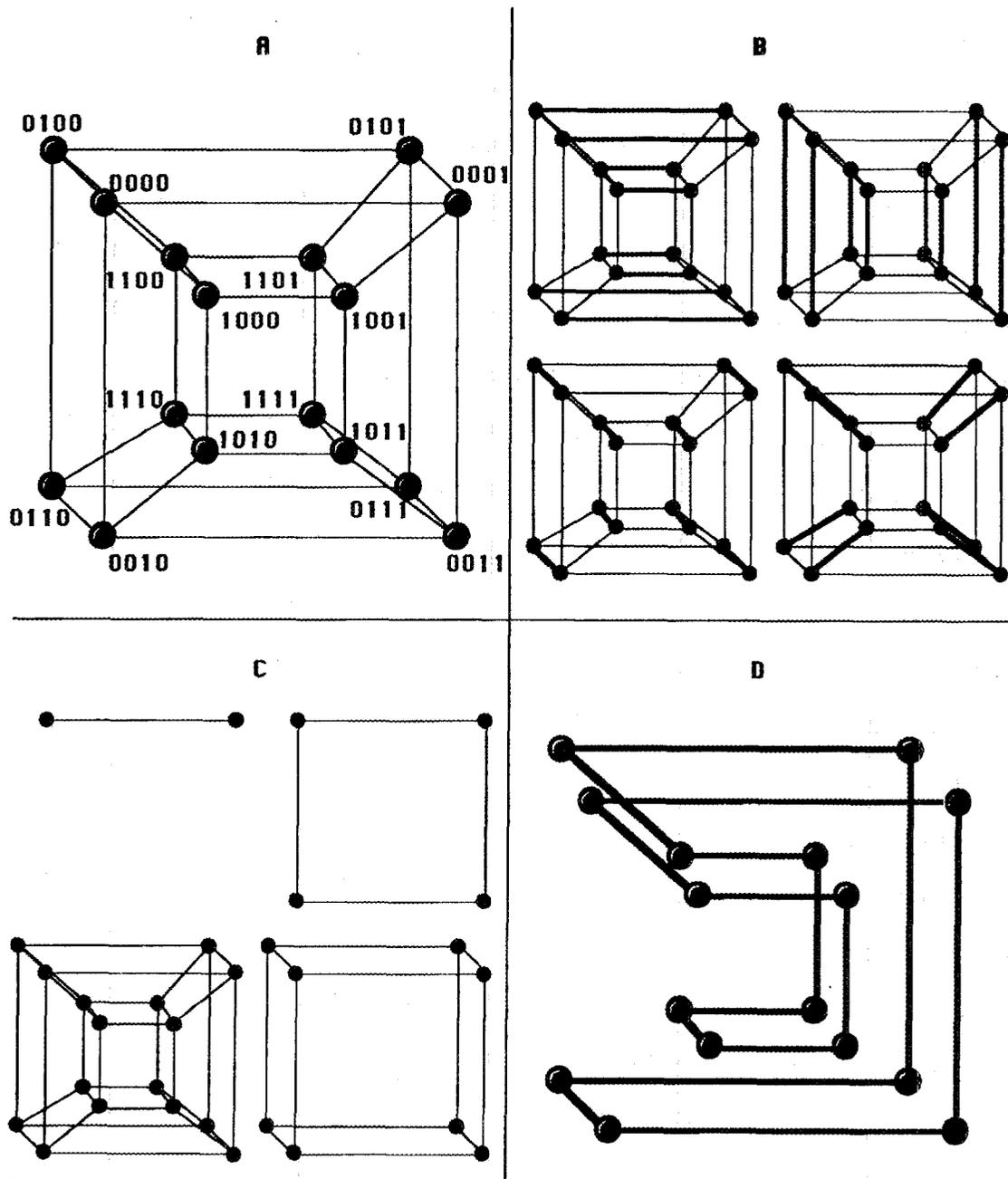


Figure 1. (a) A hypercube network is composed of  $P=2^D$  processors. Direct connections are implemented between processors whose numbers differ in exactly one bit. (b) An axis of the hypercube is the collection of all links between nodes whose numbers differ in a single specific bit. (c) Hypercubes of large dimension are recursively constructed from pairs hypercubes of smaller dimension. (d) An embedded graycode ring is a mapping of integers onto the hypercube such that successive integers are mapped onto adjacent nodes.

parallel to a coordinate axis in  $D$  dimensional space. For instance, all nodes whose numbers differ in the least significant bit are said to communicate over the least significant axis, or the 0th axis. In a  $D$  dimensional cube, the axes are numbered from 0 to  $D-1$ . A  $D+1$  dimensional hypercube is recursively constructed from  $D$  dimensional cubes by linking nodes whose numbers differ in bit  $D+1$ .

A mapping of node numbers onto the set of integers  $R=\{0, \dots, P-1\}$  such that successive elements of the map designate adjacent nodes is a graycode ring or simply a ring. The graycode computed by  $g=n \wedge n/2$  for  $n =$  the node number,  $\wedge$  the exclusive or, and  $g$  an element of  $R$  is called the standard graycode ring or the standard ring.

In the current generation of hypercubes, there is a distinguished processor in the system, not part of any hypercube, called the host. Aside from communications between nodes in the hypercube proper, the host is responsible for all of the I/O operations in the system. For instance, it is the only processor capable of reading a disc file or printing a character on a terminal. In the present system, the host processor runs a multi-user multi-tasking operating system. The host communicates directly with only a subset of the nodes in any hypercube.

Communication between two nodes, or between a node and the host, is called message passing, and the information communicated is called a message. Message passing typically

consumes CPU time on both sending and receiving processors. It is a relatively time consuming operation, typically 1 to 2 orders of magnitude slower than floating point arithmetic operations [2]. The exact time consumed depends upon the intrinsic speed of the hardware, the length of the message, the Hamming distance between communicating processors, overhead on the sending and receiving nodes, and overhead in any intermediate nodes. A simple approximation is a linear model where the message passing time is given by a fixed overhead plus the product of the message length, the Hamming distance, and a proportionality constant. In opposition to previous assumptions [11], communication time is frequently dominated by overhead. Optimal communications algorithms jointly minimize the length of messages and the number of messages, with preference presently given to minimizing the number of messages. For more details concerning system architecture, see [13].

### 3. A programming model

#### 3.1 The I/O system

The hypercube based image processing and analysis facility is an integrated system which supports a number of conventional I/O devices (terminals, discs), image-oriented I/O devices (cameras, monitors) and effector devices (rotation stages, and for robots, motion controllers). Figure 2A illustrates many of the components of the current

system, which forms a hierarchy of processors and I/O devices.

It is potentially difficult to program such a system in the absence of a well-defined assignment of function to each level of the hierarchy, and a well-developed communications interface between levels. To develop a system which is both easy to learn and easy to program we adopt a more generic model of the system, and abstract a number of operational principles.

Figure 2B presents a somewhat abstracted view of this hierarchy. The hypercube occupies the highest level, and the I/O devices the lowest level. The hypercube communicates only with the host processor, which in turn communicates with one or more device controllers, each of which in turn communicates with one or more I/O devices. The device controllers can be of heterogeneous functionality, spanning the range from limited function special purpose devices (such as SCC controllers) through general purpose computers (e.g. the M68020 in the present system), to additional concurrent multiprocessors.

Illustrated in this way, it is apparent that the host is a single resource, shared by potentially many devices, and represents a potential bottleneck in the system. Therefore, the host should be strictly reserved for I/O operations.

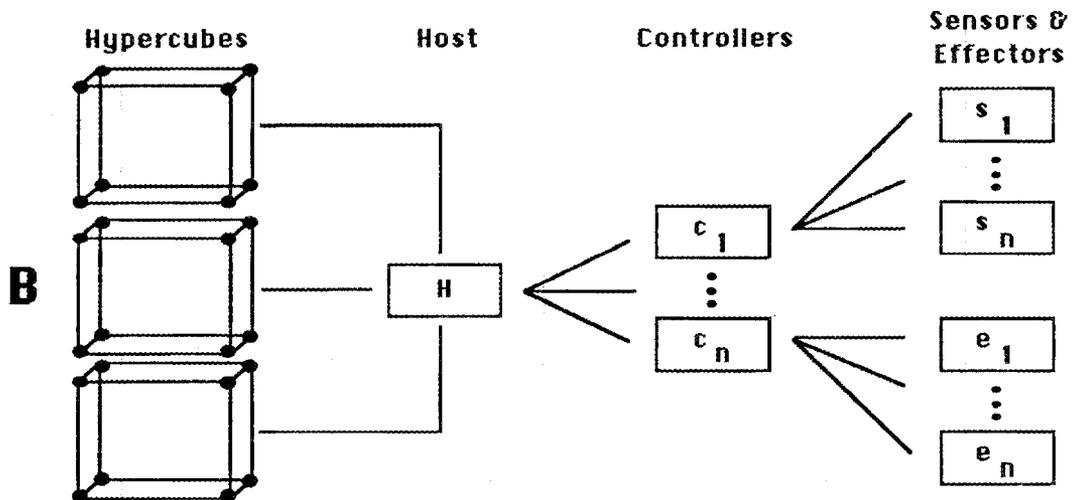
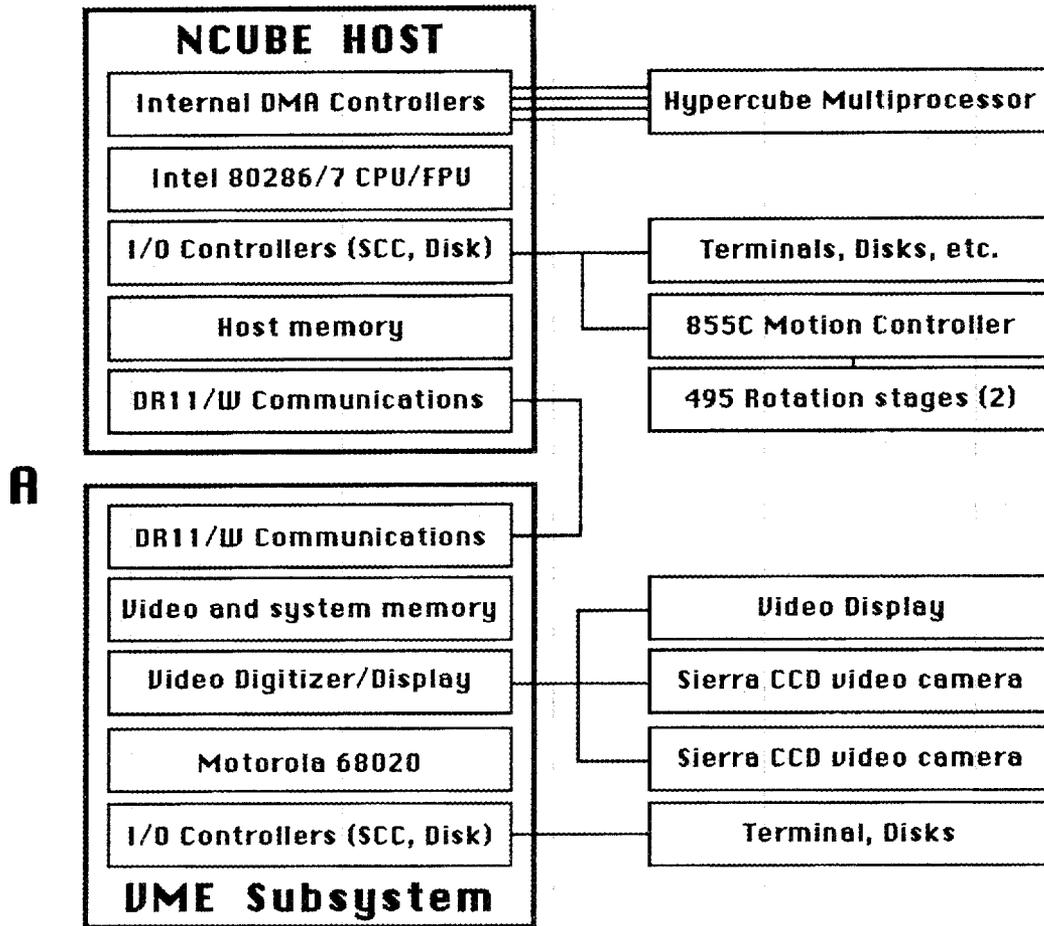


Figure 2. (a) The image processing and analysis facility is composed of two coupled systems, a hypercube multiprocessor and a VME-based system. (b) An abstraction of this system, which reveals a bottleneck at the level of the hypercube's host processor.

This principle has two immediate consequences. First, it demands that all significant computation (applications programs) take place on the hypercube proper. Second, since I/O operations can come in arbitrary order, the host must become a "slave" of the hypercube, executing I/O instructions in the order applications program request.

Although these two consequences may appear somewhat restrictive, they give rise to a programming environment of great power and simplicity. As long as the I/O devices remain fixed in number and function, it is possible to design a single host program of remarkably simple structure which satisfies all I/O operations independent of application. Once developed, this program need never change. In the current system a single host program supports the vast majority of our applications. Applications differ only in the program(s) running on the hypercube proper.

Access from hypercube applications to I/O devices is provided by function calls which issue I/O request messages to the host processor and send and receive data. Thus, the I/O system appears to the applications program as an operating system interface, much in the spirit of UNIX. The details of message passing between the host and the hypercube is hidden within this interface.

I/O device controllers are similarly slaved to the host processor. Therefore, the interface between the hypercube and the host is isolated from the interface between the host

and its various slaves. Applications programs are therefore protected from a potentially dynamic hardware environment.

The advantage of this model is that each resource other than the hypercube proper has a well-defined function, its operation follows a well-defined logical structure, and a well-defined communications protocol can be developed. General-purpose programs can be implemented serving the entire spectrum of functions for each non-hypercube processor in the system. If a loosely synchronous programming model is adopted [4], concurrent applications consist of a single source program: after the communications and I/O control software has been developed once, it is unnecessary to change it.

There are two main problems with this model, particularly with respect to real-time operation. First, the host processor is a single shared resource, and represents a potential bottleneck during communications between the hypercube and the I/O processors. We have observed the limitations imposed by this bottleneck particularly in image I/O operations, where large quantities of data must be communicated in a short time. Second, the host processor currently runs a multi-user multi-tasking UNIX-like operating system, which does not permit user interrupt service routines. This makes it impossible to guarantee response within a given time. These difficulties are ameliorated to some extent by the I/O controllers, which can

execute real-time operations, but the majority of the computational resources in the system are excluded from real-time response.

These problems can be solved in principle through elaborations on the abstract programming model illustrated in Figure 2b. First, the host processor is replaced by a butterfly communications network, or an "I/O cube" which has the ability to route data between any of  $N=2^M$  hypercube processors and  $N=2^M$  I/O device controllers in  $M \leq \log(N)$  steps. In this system the host processor acts as a device for program development and loading. This architectural modification also has the advantage that the "operating system" in the communications network can be specialized for data-driven or real-time communications.

### 3.2 Concurrent Utilities

The I/O system outlined above effectively isolates I/O operations from computation, and places the burden of computation on the hypercube proper. Current hypercube computers contain as many as 1024 processors [19], and it is expected that by 1990 systems will be available with 4096 or more general purpose processors. It is unreasonable to expect that each processor in such a system will be individually programmed. Therefore, it is desirable to minimize the number of programs which must be written to control the system for a given application.

Furthermore, it is desirable to support a class of applications through the development of utilities generic to that class. For example, in numerical linear algebra, a wide variety of specific applications are supported by LINPAK. This strategy places the burden of certain low-level considerations, such as round-off errors, on the developer of library functions. In a distributed-memory message-passing multiprocessing environment, we identify the messages, the communications topology, and the distributed nature of the memory as "low-level" features. These features should properly be hidden from applications.

These considerations lead us to adopt, in the present system, an SPMD (single program, multiple data) loosely-synchronous approach, in the spirit of the Crystalline system [4], supported by a number functions generic to computer vision applications. A single program runs on all processors simultaneously, implicitly exploiting the concurrency of the machine through function calls. Image processing and analysis operations are implemented through these function calls. Distributed data structure management, concurrency and its optimization, and message passing are handled inside the functions. Thus, the machine is programmed using a high level set of "instructions". To applications programs, this causes the machine to appear as an ordinary sequential computer, but very fast.

There are a number of advantages to this approach. Only one program need be written to control an arbitrary number of processors, and thus the software development time and associated costs are reduced. Most applications of significant size can be reduced to a sequence of relatively high level functions, and thus the approach is fairly general. Individual functions are small enough to be easily understood, and performance can frequently be optimized through the use of deterministic communications algorithms. With a relatively well-developed I/O interface, error diagnostics can be easily reported, and typically apply to the program as a whole, rather than to individual nodes.

Naturally, this approach suffers from certain limitations. In some cases, computational bottlenecks develop due to the need for global communication in the course of executing a given function. This effect is most profound in hypercubes of large dimension. The burden upon the host processor is exacerbated, since all nodes request I/O at roughly the same time. In addition, as the number of processors in the largest systems increases some degree of instruction parallelism will be inevitable. In such a system we anticipate creating several concurrent virtual machines, each balanced, each running in SPMD mode. The scheduling problem in such a system would be greatly simplified.

A simple example illustrates the points made in this section. A copy of the following program runs on each node

of the hypercube. The program reads an image, finds the edges in the image using the Sobel operator [17], and outputs the image.

```

main()
{
    IMAGE src, dst;                /* 0 */

    src = imalloc(256,256,1)      /* 1 */
    dst = imalloc(256,256,1)      /* 2 */

    imagein(src);                 /* 3 */
    sobel(src,dst);               /* 4 */
    imageout(dst);                /* 5 */
}

```

IMAGES (see below) are concurrent data structures. They are declared in line 0 and actively allocated in lines 1 and 2. Line 3 acquires an image, line 4 convolves the image with a Sobel operator, placing the result in the destination object, line 5 outputs the image.

The host processor is reserved for I/O. The I/O functions `imagein` and `imageout` pass messages to the host processor instructing it to perform the requisite I/O. Otherwise, the host processor is not involved in the computation. I/O functions can occur in arbitrary order. Naturally, each node on the hypercube must request I/O in the same order as all other nodes.

The details of the architecture (message passing, communications topology, distributed data) are hidden from the applications program. No explicit reference is made to these features. In particular, note that no explicit reference is made to the dimension of the hypercube, and

that data parallelism (the distribution of IMAGE over the cube) is implicit.

#### 4. Decomposition and communications

There are two popular strategies for mapping 2 dimensional images onto the nodes of a hypercube. In the first [8, 11], the image is decomposed into a set of square or rectangular subimages. Each subimage is assigned to a node in a graycode grid. In the second, the image is decomposed into a set of strips, each strip composed of several complete rasters. Each strip is assigned to a node in a graycode ring. In both strategies elements which are adjacent in the original image are mapped onto adjacent nodes.

Grid mapping attempts to minimize communication time during image processing. To justify grid mapping [11], note that it is the perimeter of the sub-image which must be communicated. To minimize communication delays, decompose the image into rectangular subimages of minimum perimeter. Since a square has the minimum perimeter of any rectangular region of a given area, decompose the image into subimages which are as square as possible.

This argument rests upon two incorrect assumptions. First, it is assumed that the time consumed by communication is dominated by message length. Benchmarks [2] indicate that a fixed startup cost consumes most of the time in passing a message. Therefore, it is more desirable to

minimize the number of messages which must be passed, rather than their length. Second, it is assumed that the majority of communications will be nearest-neighbor, as in convolution. Many computer vision operations, such as histogramming or component labeling, require global communications, and many image oriented I/O devices structure the data in a specific, fixed format which is incompatible with grid mapping.

Therefore, we chose ring mapping. Ring mapping requires only two messages to exchange data between adjacent sub-images. Grid mapping requires four. Ring mapping requires no rearrangement of data on image input or output to standard raster scan digitizers or display devices. Direct memory access devices can be exploited to acquire images from cameras and display images on monitors. To support grid mapped images, CPU time must be consumed somewhere to arrange the data for I/O.

Regardless of the decomposition chosen, it is desirable to endow a concurrent data structure with certain properties. In the present system, the concurrent data structure IMAGE has the following properties:

- (1) The run-time maintenance in the distributed memory environment is transparent. Loosely synchronous applications allocate storage for image data based on the size of the whole image. The assignment of data to specific nodes in the hypercube is transparent, as is communication between nodes

to resolve such internal conflicts in the data structure as arise.

(2) Image buffers can be of arbitrary size. IMAGES are actively allocated at run-time. Applications programs can specify the number of pixels in the vertical and horizontal directions.

(3) Pixels can have arbitrary storage class. The specific storage classes supported are those found in "C" [7].

(4) Storage allocation supports conventional array indexing into the distributed multidimensional array. Thus, row and column addresses can be specified using expressions similar to `pic[y][x]`, rather than `pic[y*wide+x]`. This feature assists program development by making dynamic storage allocation transparent. It also improves run time efficiency by eliminating multiplication for pixel accesses.

(5) Routines exist for the low-level manipulation of image buffers which are size and storage-class independent. Among the most useful are buffer-to-buffer copy with implicit type conversion, normalization, and allocation reproduction.

The present system is not yet developed well enough to completely protect applications programmers from errors. For instance, the image buffer-to-buffer copy routine casts types in the same way as "C" casts scalar types across assignments [7], so an unnormalized casting copy (e.g. from float to unsigned character) may not give the desired result. Nevertheless, endowing the distributed data

structure IMAGE with the above properties has permitted enough flexibility for most applications, and has allowed the construction of derived distributed data structures, such as distributed resolution pyramids [18].

Two principal communications algorithms are employed in the system, mostly in conjunction with IMAGES, although additional communications algorithms are invoked where necessary for expediency or efficiency. The first simply exploits the ring mapping of the image onto the hypercube. Data are exchanged between adjacent processors in the ring, e.g. in the exchange of rasters prior to neighborhood operations, or successively passed around the ring, e.g. in the case where each node must perform relatively many computations on every strip in the image.

The second algorithm, "butterfly accumulator", is employed for global calculations and conflict resolution, as in image histogramming, component labeling, and certain load balancing contexts. In this algorithm each node computes some (possibly vector valued) function of its local sub-image. Nodes then communicate along successive axes of the hypercube in  $D$  iterations, exchanging information with their neighbor along those axes, and performing (redundant) calculations to resolve the local conflict along that dimension. The communications complexity is  $O(\log P)$ .

In many cases it is easy to show that this algorithm efficiently resolves global conflicts. In this algorithm,

each node can be thought of as a general purpose accumulator for scalars, vectors, or other data structures, where the process of accumulation implements an arbitrary functional combination of the data arising from two nodes. This represents a generalization of the pseudo-binary tree or minimal spanning tree algorithm in the sense that all nodes serve as root nodes of the accumulation tree simultaneously.

The canonical example [14] of global conflict resolution on hypercube multiprocessors is the calculation of the sum of a list of  $P$  numbers (see Fig. 3). In the pseudo-binary tree algorithm communication takes place along successive axes of the hypercube (e.g. from most to least significant), and the processor on the most significant side of the current hemicube sends a partial result to its neighbor along that axis. Appropriate communications links in each axis are dropped in successive iterations. In this way a single node eventually computes the global sum (e.g. node 0). In the butterfly accumulator algorithm communication takes place in both directions, exploiting the full duplex communications between nodes, and nodes on both sides of the current axis compute partial sums.

The difference between the minimal spanning tree and the butterfly accumulator is that in the former a single node holds the final answer, whereas in the latter all nodes hold the answer. This property is useful in a wide variety of applications, since it eliminates the necessity of

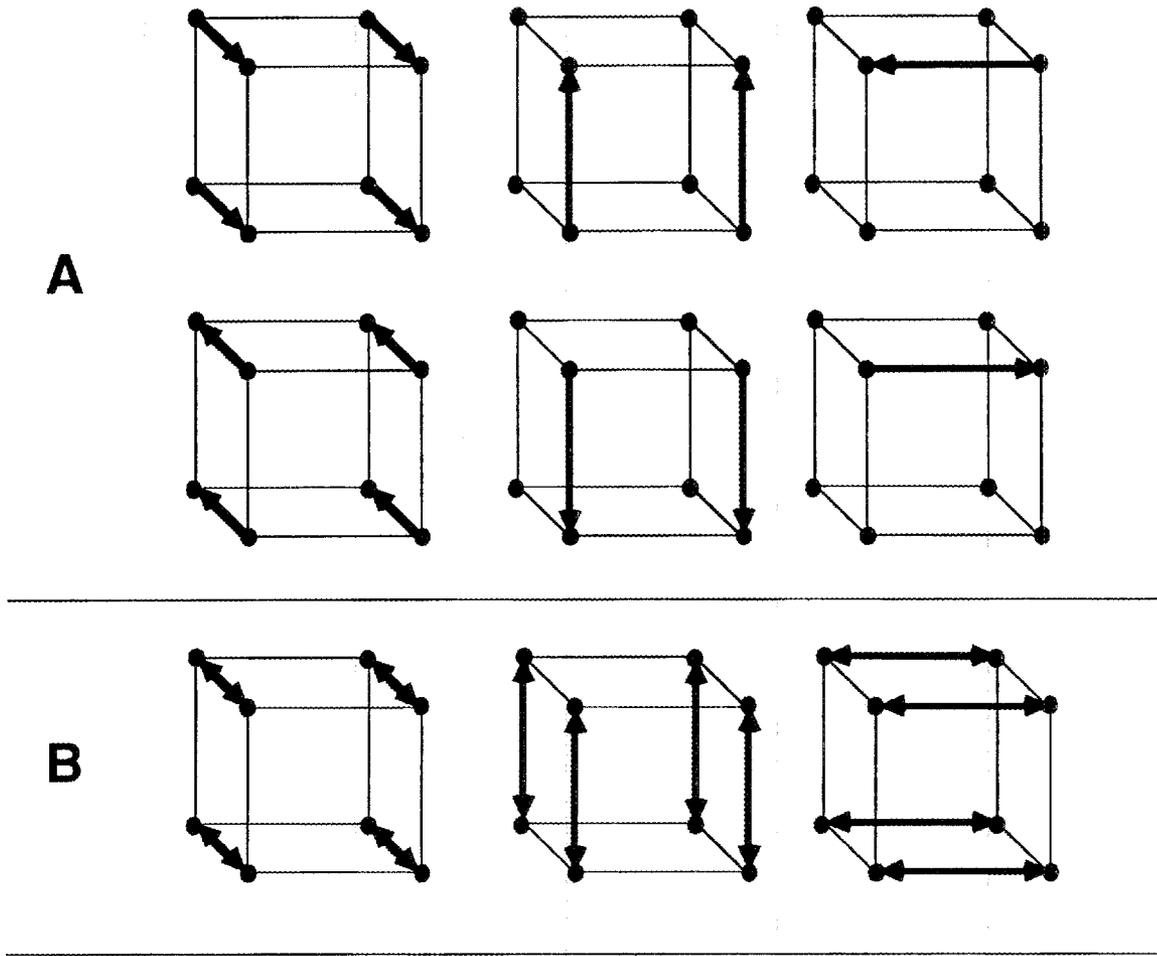


Figure 3. Two alternatives for global conflict resolution in hypercube topology multiprocessor networks. (a) In the pseudo-binary tree or minimal spanning tree algorithm unidirectional communication takes place between  $\log(N)$  hemicubes of decreasing dimension such that a single result is computed on one node (top row). This result is then broadcast to the remainder of the nodes in the hypercube, again in  $\log(N)$  steps, by following the same path in reverse. (b) In the butterfly accumulator, bidirectional communication takes place: nodes on either side of the links forming an axis exchange partial results, and redundant calculations are performed on each node to compute the next partial result. In (a) there is no opportunity to exploit full-duplex, bidirectional communication, whereas in (b) communication channels are used concurrently.

broadcasting the final result. In both algorithms communications complexity is  $O(\log P)$ .

To illustrate the utility of this algorithm, consider the following routine for image histogramming.

```
#define GRAY_RES    256

main()
{
    int  hist[GRAY_RES],temp[GRAY_RES],node,proc,host,dim;
    int  neighbor,axis,*s,*d;

    unsigned char **pic;

/* allocate image buffer and input image */

    src = imalloc(256,256,0); pic=src->p;
    imagein(src);

/* compute local histogram */

    pic = src->p;
    for( y=0; y<src->high; y++)
    for( x=0; x<src->wide; x++)
        hist[ p[y][x] ]++;

/* accumulate global histogram */

    msglen = GRAY_RES*sizeof(int);
    for( axis=0; axis<dim; axis++){
        neighbor = node^(1<<axis);
        nwrite(neighbor,hist,msglen,type,flag);
        nread (neighbor,temp,&msglen,&type,&flag);
        for(i=0;i<GRAY_RES;i++) hist[i] += temp[i];
    }
}
```

In this example the "possibly vector valued function" computed by each of the nodes is a local histogram, based on only those pixels assigned to the node. The communication occurs in the loop labeled "accumulate global histogram". The loop iterates "dim" times, where "dim" is the dimension of the hypercube. The line following "for" calculates the

neighbor along the current axis, and the following two lines exchange partially accumulated histograms. These two histograms are combined (redundantly) on each node simply by point-by-point addition. Following this communication loop, all nodes contain the same histogram.

It is interesting to note the equivalence of the butterfly accumulator to communications networks for computing generalized translation invariant representations. The most familiar example of a such a representation is the amplitude spectrum of a discrete Fourier transform with periodic boundary conditions, which may be computed with the aid of the fast Fourier transform algorithm. The pattern of communications in the fast Fourier transform algorithm is identical to that in the butterfly accumulator.

## 5. Discussion

We have developed an integrated system for image processing and analysis based on a hypercube architecture concurrent multiprocessor system. The system is easily programmable, provides rich support for applications development, and an environment for research in concurrent algorithms for computer vision. This system is based on principles which permit the applications programmer to view the machine as an SIMD machine with a very coarse grained instruction set. Input/Output and concurrency are hidden from applications programs in the the image processing and analysis functions.

To demonstrate these concepts, the system has been ported to the mobile robot HERMIES [1]. Using the I/O facilities, morphological operators, adaptive thresholding, connected components, component analysis, and the Hough transform from this system, as well as robot motion primitives, it was possible to construct a system which executed a docking maneuver to an object of a priori known geometry, and to read an analog meter. The strict segregation of input/output and computation permitted transportation of the software developed in the environment illustrated in Figure 2 to an environment with radically different I/O devices with minimal change to the I/O interface, and no change to the concurrent environment.

The system we described is evolving. Further development will of course include enhancements to the basic set of tools, specific higher level developments, e.g. model-based scene analysis and stereo vision, as well as support for additional I/O devices and effectors, as need and opportunity arise.

The hypercube concurrent architecture has proved to be both sufficiently flexible and powerful for applications in image processing and analysis to consider its use as a basic system to support R&D, and as a sensory processor in the next generation of intelligent autonomous robots. However, some of the very low-level, convolution class operations in such a system should probably be executed by special purpose

hardware, since these operations currently consume, for a period of time, the resources of a much more powerful computer. Special architectures for these kinds of operations are commercially available, and continue to evolve.

Perhaps the greatest attractiveness of these machines lies in their general purpose nature. These machines are not specialized for image processing and analysis, and in fact find use in a wide variety of applications in science and engineering [3, 5, 6]. Thus, as a tool for research and development, and as a computational resource in a system requiring great flexibility along with large computational power, hypercube architecture concurrent multiprocessors are indeed a reasonable choice.

## 6. Acknowledgements

We would like to thank G. Bilbro and W. Snyder of North Carolina State University for helpful discussions, S. Killough of the Instrumentation and Controls Division (ORNL) for constructing the VME based peripheral image acquisition system, and M. Kedl of the University of Tennessee and A. Gove of the University of Texas at Austin for enhancements of the communications software.

## 7. References

1. Burks, B.L., de Saussure, G., Weisbin, C.R., Jones, J.P., and Hamel, W.R. (1987) "Autonomous navigation, exploration, and recognition using the Hermies-IIB robot." IEEE Expert (2) 18-27.
2. Dunigan, T.H. (1987) "Performance of three hypercubes." ORNL/TM-10400, May, 1987.
3. Fox, G. (1987) "The caltech concurrent computation program." In: Hypercube Multiprocessors 1987, M.T. Heath, ed., Society of Industrial and Applied Mathematics, Philadelphia.
4. Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D. (1988) Solving problems on concurrent processors. V.1 Prentice-Hall, Englewood Cliffs, N.J.
5. Hayes, J.P., Jain, R., Martin, W.R., Mudge, T.N., Scott, L.R., Shin, K.G., Stout, Q.F. (1987) "Hypercube computer research at the University of Michigan." In: Hypercube Multiprocessors 1987, M.T. Heath, ed., Society of Industrial and Applied Mathematics, Philadelphia.
6. Heath, M.T. (1987) "Hypercube applications at Oak Ridge National Laboratory." In: Hypercube Multiprocessors 1987, M.T. Heath, ed., Society of Industrial and Applied Mathematics, Philadelphia.
7. Kernigan, B.W., Ritchie, D.M. (1978) The C Programming Language, Academic Press, New York.
8. Lee, S.-Y., Aggarwal, J.K. (1987) "Exploitation of image parallelism via the hypercube." In: Hypercube Multiprocessors 1987, M.T. Heath, ed., Society of Industrial and Applied Mathematics, Philadelphia.
9. Jones, J.P. (1988) "A concurrent on-board vision system for a mobile robot." Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, in press.
10. Jones, J.P., & Mann, R.C. (1988) "Concurrent algorithms for a mobile robot vision system.", Applications of Artificial Intelligence IV, M.M. Trivedi, ed., Proc. SPIE 937, 497-504.
11. Kushner, T.R., Rosenfeld, A. (1983) "A model of interprocessor communication for parallel image processing." IEEE Trans. Sys., Man, Cybern. (SMC-13) 600-618.

12. Nudd, G.R. (1984) "Concurrent systems for image analysis." In: VLSI for Pattern Recognition and Image Processing K.S. Fu, ed. Springer-Verlag, Berlin.
13. Palmer, J.F. (1986) "A VLSI parallel supercomputer." In: Hypercube Multiprocessors 1986 M.T. Heath, ed. SIAM, Philadelphia.
14. Quinn, M.J. (1987) Designing efficient algorithms for parallel computers. McGraw-Hill, New York.
15. Reeves, A.P. (1984) "Survey: parallel computer architectures for image processing." Computer Vision, Graphics, and Image Processing (25) 68-88.
16. Seitz, C.L. (1985) "The cosmic cube." Comm. ACM (28) 22-33.
17. Sobel, I. (1970) "Camera models and machine perception." AIM-21, Stanford AI Lab.
18. Uhr, L. (1986) "Parallel architectures for image processing, computer vision, and pattern perception." In: Handbook of Pattern Recognition and Image Processing, T.Y. Young and K.-S. Fu, ed.s, Academic Press, Orlando, 438-470.
19. Waldrop, M.M. (1988) "Hypercube breaks a programming barrier." Science (240) 286.
20. Wiley, P. (1987) "A parallel architecture comes of age at last." IEEE Spectrum (24) 46-50.
21. Yalamanchile, S., K.V. Palem, L.S. Davis, A.J. Welch, J.K. Aggarwal (1985) "Image processing architecture: A taxonomy and Survey." Progress in Pattern Recognition, V. II, P. 1-37, North Holland.

## A. APPENDIX

## A.1: CONCURRENT UTILITIES

## A.1.1 REQUIREMENTS

Image processing and analysis routines written in this system have few required components. The program must include the header file `implib.h`, and prior to any call to the processing and analysis functions, the program must call `iminit()`. This function sets up some internal variables and receives initialization data from the invoking host program. The minimal program consists of four lines:

```

#include "/usr/image/cube/implib.h"    /* 1 */
main(){
    int ctrl[NCTRL];                  /* 2 */
    iminit(ctrl);                     /* 3 */
    terminate();                       /* 4 */
}

```

Line 1 includes the requisite header. Line 2 declares an array for incoming initialization data. Line 3 acquires these data and initializes some global variables. Line 4 sends a message to the host processor informing it that the routine is finished.

The remainder of this section is subdivided into seven sections covering initializations and concurrent data structure allocation, I/O utilities, very low level utilities for various image buffer manipulations, graphics, and low level, intermediate level, and high level computer vision utilities.

## A.1.2 Initializations and concurrent data structures

ALLOCOPY --- replicate image buffer allocation

IMAGE allocopy(src)

allocopy replicates the allocation of the image buffer src and returns a pointer to the new structure. Both the image size and its storage class are replicated. This function is principally used inside the system for the creation of temporary image buffers.

IMINIT --- initialize image processing environment

iminit(ctrl)

```
int ctrl[100];
```

iminit must be called prior to invoking any of the other facilities in the library. iminit initializes internal variables and reads a list of 100 integer and floating point numbers from the host program. The first 50 numbers (ctrl[0]-ctrl[49]) are integers, the second 50 (ctrl[50]-ctrl[99]) are floating point. A popular mechanism for accessing these numbers is by declaring the array ctrl as struct { int i[50]; float f[50]; } ctrl; and referring to the Kth element of each block as ctrl.i[K] and ctrl.f[K]. These numbers are useful for parameterizing individual runs of a code. Although not as general as the argc, argv mechanism in UNIX, it is much simpler, since there is no need to parse arguments.

IMALLOC --- allocate unsigned character image buffer

IMAGE imalloc(nx,ny,nr)

```
int nx,ny,nr;
```

imalloc reserves space for an unsigned character image buffer. The arguments nx and ny specify the total width (number of pixels in each raster) and height (number of rasters) desired. These rasters are distributed over the hypercube in the standard graycode ring. The argument nr specifies the number of edge rasters to reserve on each side of the image strip in each node. For details on the distributed data structure IMAGE, see the discussion of imallocc below.

IMALLOCC --- allocate image buffer, arbitrary storage class

```
IMAGE imallocc(nx,ny,nr,string)
```

```
    int  nx,ny,nr;
    char *string;
```

imallocc reserves space for an image buffer with pixels of arbitrary storage class. The arguments nx and ny specify the total width (number of pixels in each raster) and height (number of rasters) desired. These rasters are distributed over the hypercube in the standard graycode ring. The argument nr specifies the number of edge rasters to reserve on each side of the image strip in each node. String specifies the storage class of pixels, and is one of the following: "unsigned char", "unsigned short", "unsigned int", "unsigned long", "char", "int", "short", "long", "float", or "double". In detail, the structure IMAGE contains the following:

```
struct IMAGE {
    char **p;          /* pointer to raster pointers */
    int  high,        /* number of rasters in image */
        wide,        /* number of pixels per raster */
        rast,        /* number of edge rasters per node */
        nrows,       /* number of rasters per node */
        psize,       /* number of bytes per pixel */
        class;       /* storage class code */
}
```

The image is allocated as block of memory of size

```
( ny/2dimension + 2*nr ) * nx * sizeof( storage class )
```

bytes. The image pointer p points to the nr'th element of an array of pointers to this block of memory. The memory location specified in successive array elements is the address of the first pixel on successive rasters. Thus, p[0][0] is the upper leftmost pixel of the subimage for which a node is responsible, and p[-1][0] is the pixel immediately above it. This indirect method of allocating memory promotes programming efficiency by permitting constructions such as

```
    IMAGE src; char **pic=src->p;
    val = pic[5][15];
```

This allows conventional addressing of the two-dimensional image buffer, avoiding explicit calculation of the address for each pixel. The method is independent of storage class.

IMFREE --- free image buffer

imfree(image)

IMAGE image;

imfree deallocates an image buffer previously allocated by imalloc, imallocc, or allocopy.

### A.1.3 I/O Utilities

GIMAGEOUT --- general image output

gimageout(src)

IMAGE src;

gimageout scales the image src into the range 0-255 and sends it to the output device(s). The image src can be of arbitrary storage class. Contents of the original buffer are preserved.

INPUT\_Q --- input host queued image

input\_q(dst)

IMAGE dst;

input\_q instructs the host to distribute the image in its input buffer over the hypercube. This function is used after calls to image\_q.

IMAGEIN --- input image

imagein(dst)

IMAGE dst;

imagein instructs the host to acquire an image from the currently active input device and distribute it over the hypercube. For time critical dynamic input sequences, these two operations can be overlaid in time using image\_q and input\_q.

IMAGEOUT --- output image

imageout(src)

IMAGE src;

imageout send the contents of the image buffer src to the host, which in turn forwards it to the currently active output device(s).

IMAGEIN16 --- input 128x128 image from VME system

imagein16(dst)

IMAGE dst;

instructs the host to acquire a 16 Kbyte (128x128) image from the VME system and distribute it over the hypercube. This function is principally used in time critical applications, such as time-varying image processing, where high resolution can be exchanged for speed.

IMAGEOUT16 --- output 128x128 image to VME system

imageout16(src)

IMAGE src;

imageout instructs the host to send the 16 Kbyte (128x128) image to the VME system for display. This function is used primarily to minimize communication time.

IRSMOVR --- camera position relative move

irsmovr(stage,rel)

int stage,rel;

irsmovr instructs the Newport rotation stage controller to move the rotation stage numbered "stage" to relative position loc, where loc is given in integer millidegrees.

IRSMOVA --- camera position absolute move

irsmova(stage,loc)

int stage, loc;

irsmova instructs the Newport rotation stage controller to move the rotation stage numbered "stage" to absolute position loc, where loc is given in integer millidegrees.

IMAGE\_Q --- queue image for susequent input

image\_q()

image\_q instructs the host program to acquire an image from the currently active input device, without sending the image to the hypercube. This capability is principally used in dynamic image processing.

LEFT\_CAMERA --- select left camera in stereo imaging system

left\_camera()

left\_camera instructs the host to issue a command to the VME subsystem to take further image input from the left camera.

NPRINTF --- send text to cube monitor and/or disk file

nprintf(string,[arg1,arg2,...,argN])

```
char *string;
[arbitrary arg1,...argN]
```

nprintf is similar to the UNIX printf function. The string is a format control string having the same syntax as printf, and the arguments are arbitrary in number and kind. The output string is formatted in memory and sent to the host I/O program, where it is forwarded to the terminal, a disk file, or both.

RIGHT\_CAMERA --- select right camera in stereo system

right\_camera()

right\_camera instructs the host to issue a command to the VME subsystem to take further image input from the right camera.

TERMINATE --- graceful exit

terminate();

terminate sends a message to the host program instructing it to close all files and communications channels and exit. The cube in which the calling node resides is deallocated, so the only appropriate positions for a call to terminate is immediately prior to program termination or following a fatal error.

TTYIN --- get a keystroke from the console

```
char ttyin();
```

ttyin returns an ASCII character from the console input device. (Keystrokes are broadcast by the host to all nodes simultaneously.) Processing is suspended until a character is received.

TTYFLY --- get a keystroke on the fly

```
char ttyfly();
```

ttyfly returns an ASCII character from the console input device if one is available, otherwise it returns 0. (Keystrokes are broadcast by the host to all nodes simultaneously.) This facility is useful for asynchronous keyboard interaction with a running program.

#### A.1.4 Very low level utilities

Unless otherwise noted, these utilities operate only on IMAGES of storage class unsigned character.

CAST\_COPY --- copy one image to another.

```
cast_copy(src,dst)
```

```
    IMAGE    src,dst;
```

cast\_copy copies the contents of one image buffer to another with type conversion. Any storage class image allocated by imalloc or imallocc can serve as either the source (src) or the destination (dst) image. Type conversion is performed on pixels according to the established conventions.

CUBE\_SYNC --- synchronize hypercube

```
cube_sync()
```

cube\_sync performs an approximate synchronization of all nodes in the hypercube, using blocking reads along ordered cube axes. It is used primarily for benchmarking, where it is called immediately before the routine being benchmarked.

DELAY --- delay for a time

delay(ms)

```
int ms;
```

delay waits for ms milliseconds before returning. It is principally used for debugging.

EXCHANGE --- ring raster exchange

exchange(src,nex)

```
IMAGE src;
int nex;
```

exchange performs an explicit exchange of "nex" edge rasters in the image "src" between neighboring nodes in the standard graycode ring. This function is principally used in the prelude of neighborhood operations, e.g. the morphological operations, and is usually invisible to the user. exchange works only on image buffers of type unsigned character or character, and supports only nearest neighbor communications.

GENEX --- a more general ring raster exchange

genex(src,nex)

```
IMAGE src;
int nex;
```

genex performs an explicit exchange of "nex" edge rasters in the image "src" between nodes in the standard graycode ring. This function is principally used in the same contexts as exchange. The function of this routine is identical to "exchange", with two exceptions. First, arbitrary storage class image buffers are accepted as the argument, and communications between remote (non-nearest-neighbor) nodes is executed if necessary.

IADD --- add constant to image

iadd(src,konst)

```
IMAGE src;
int konst;
```

iadd adds a constant to each pixel in the specified buffer.

ICOPY --- buffer to buffer image copy

icopy(src,dst)

IMAGE src,dst;

icopy copies the contents of one image buffer (src) to another (dst).

IKONST --- fill image buffer with a constant

ikonst(src,konst)

IMAGE src;  
int konst;

ikonst fills each pixel in the specified image buffer with a constant.

IMUL --- multiply all pixels by a constant

imul(src,konst)

IMAGE src;  
int konst;

imul multiplies each pixel in the image src by a constant.

INVERT --- invert image

invert(src)

IMAGE src;

invert replaces each pixel P in the image src with the quantity 255-P.

IZERO --- zero image buffer

izero(src)

IMAGE src;

izero fills the specified image buffer with zeros.

NGRAY --- standard graycode

```
int ngray(arg)
```

```
    int arg;
```

ngray returns the graycode i.d. of its argument. For example, the nth strip of the image is mapped onto the node ngray(n).

NORMALIZE --- normalize image buffer to a specified range.

```
normalize(src,lo,hi)
```

```
    IMAGE src;
    int  lo,hi;
```

normalize computes the global maximum and minimum in the image src, and rescales the image such that the minimum value becomes lo, and the maximum, hi. The image buffer src can be of arbitrary storage class. This function is principally used for scaling floating point images into an 8-bit range prior to display.

NPHYS --- inverse of standard graycode

```
int nphys(gray)
```

```
    int gray;
```

nphys returns the inverse graycode of its argument. For example, the strip of the image mapped onto node n is nphys(n).

NRING --- embedded ring mapping

```
nring(node,ndim,nup,ndown)
```

```
    int  node,ndim,*nup,*ndown;
```

nring returns the numbers of the nodes immediately "upwards" and "downwards" of the calling node ("node") in the standard graycode ring.

TABLE --- initialize trig lookup table

```
table(trig_table)
```

```
    int trig_table[320];
```

table sets up an integer sine/cosine lookup table (LUT) for use in later integer trigonometric LUT operations. Values in the table are computed as 256 times the value of the sine function, uniformly sampled in 320 places over the interval 0 to  $5\pi/2$ . Thus, the sine LUT begins at entry 0, and the cosine LUT at entry 64.

ZCOL --- zero range of columns

```
zcol(lo,hi,src)
```

```
    int      lo,hi;
    IMAGE    src;
```

zrow replaces the data in the columns of the image src with zero in the specified range, lo to hi inclusive.

ZROW --- zero range of image rows

```
zrow(lo,hi,src)
```

```
    int      lo,hi;
    IMAGE    src;
```

zrow replaces the data in the rows (rasters) of the image src with zero in the specified range, lo to hi inclusive.

### A.1.5 Graphics

The system supports an extremely primitive set of functions useful for creating graphic overlays on images. Graphics are drawn using an implied cursor which can be moved to an absolute position in the image, or relative to its current position.

GBUFFERON --- select active graphics buffer

```
gbufferon(dst)
```

```
    IMAGE dst;
```

gbufferon causes the results of all subsequent graphics calls to be drawn in the (unsigned character) image src.

GBUFFEROFF --- inactivate graphics buffer

gbufferoff(dst)

IMAGE dst;

gbufferoff inactivates the image src as a graphics buffer.

INTENSITY --- select drawing intensity

intensity(val)

int val;

intensity causes all subsequent graphics instructions to be written using pixels of brightness val. val must be in the range -1 to 255. if val is -1, pixels are completemeted rather than set to an absolute brightness.

AMOVE --- absolute move

amove(x,y)

int x,y;

amove moves the implicit cursor to absolute position x,y in the graphics buffer.

RMOVE --- relative move

rmove(dx,dy)

int dx,dy;

rmove moves the cursor dx,dy units relative to its current position. If the requested move is outside the limits of the current graphics buffer, it is clipped to the nearest point on the edge of the buffer.

ALINE --- absolute line

aline(x,y)

int x,y;

aline draws a line using the current intensity value from the current position of the implied cursor to the absolute coordinates x,y in the graphics buffer.

RLINE --- relative line

rline(dx,dy)

```
int dx,dy;
```

rline draws a line using the current intensity value from the current position of the implied cursor to the relative coordinates dx,dy in the graphics buffer.

#### A.1.6 Low level Utilities

ABSDIFF - absolute difference of two images

```
int absdiff(src1, src2, dst)
```

```
IMAGE src1, src2, dst;
```

absdiff subtracts src2 from src1 pixel by pixel and returns the absolute value of the difference in dst.

BINIMAGE --- grayscale to binary conversion

```
binimage(src,dst,thresh)
```

```
IMAGE src, dst;
int thresh;
```

binimage performs an absolute threshold. src is a grayscale 8-bit (0-255) image, dst is returned as a binary (0,255) image, according to whether each pixel in src is greater than or less than or equal to the integer threshold.

CONV3 --- specialized integer 3x3 convolution

```
conv3(src,dst,i1,i2,i3,i4,i5,i6,i7,i8,i9)
```

```
IMAGE src,dst;
int i1,i2,i3,i4,i5,i6,i7,i8,i9;
```

conv3 performs an integer 3x3 convolution on the image src to produce the image dst. It is frequently used for quick implementation of small, fixed structure convolution masks. Arguments i1,...,i9 are the convolution mask elements ordered from upper left to lower right.

DILATE --- binary morphological dilation

dilate( src , dst )

IMAGE src, dst;

dilate performs a binary morphological dilation of the binary image src to produce the binary image dst. Specifically, if each clear pixel has any set pixel its 8-neighborhood, it is also set.

DILATE4 --- binary morphological dilation

dilate4( src , dst )

IMAGE src, dst;

dilate performs a binary morphological dilation of the binary image src to produce the binary image dst. Specifically, if each clear pixel has any set pixel its 4-neighborhood, it is also set.

EQUAL --- histogram equalization

equal( src )

equal performs an in place histogram equalization based on the global image histogram.

ERODE --- binary erosion

erode(src,dst)

IMAGE src,dst;

erode performs a binary morphological erosion of the binary image src to produce the binary image dst. Specifically, if each set pixel has any clear pixel its 8-neighborhood, it is also cleared.

ERODE4 --- binary erosion

erode(src,dst)

IMAGE src,dst;

erode performs a binary morphological erosion of the binary image src to produce the binary image dst. Specifically, if each set pixel has any clear pixel its 4-neighborhood, it is also cleared.

GCLOSE --- grayscale morphological image closing

gclose(src,dst,nsiz)

```

    IMAGE    src,dst;
    int      nsiz;
```

gopen performs a grayscale morphological image closing on the image "src" to produce the image "dst". The structuring element is a square region of diameter "nsiz".

GLOBAL\_AVG --- global image statistics

avg = global\_avg(src,min,max)

```

    IMAGE    src;
    int      avg, *min, *max;
```

global\_avg computes simple global statistics on the image "src". The average value of all pixels in the image is returned in avg, and the maximum and minimum values in the arguments \*max and \*min respectively.

GLOBAL\_COM --- global center of mass

global\_com(src,cx,cy)

```

    IMAGE    src;
    int      *cx,*cy;
```

global\_com computes the global center of mass of all the set pixels in the binary image "src". The image coordinates of the center of mass are returned in the arguments \*cx and \*cy. This routine is used only in very simple applications, where there is only one object.

GMAX --- replace with local maximum

gmax(src,dst,nsiz)

```

    IMAGE src,dst;
    int  isiz;
```

gmax performs replaces each pixel in the image "src" with the maximum of the pixels taken over the square neighborhood of diameter "isiz", forming the output image "dst".

GMIN --- grayscale replacement with local minimum

```
gmin(src,dst, isize)
```

```
    IMAGE src,dst;
    int  isize;
```

gmin performs replaces each pixel in the image "src" with the minimum of the pixels taken over the square neighborhood of diameter "isize", forming the output image "dst".

GOPEN --- grayscale morphological image opening

```
gopen(src,dst, nsize)
```

```
    IMAGE      src,dst;
    int        nsize;
```

gopen performs a grayscale morphological image opening on the image "src" to produce the image "dst". The structuring element is a square region of diameter "nsize".

HISTO --- global histogram

```
histo(src,hist)
```

```
    IMAGE src;
    int  hist[256];
```

histo computes the global histogram of an image. The histogram is returned in the integer array hist, which must have 256 elements. The values returned in each element are the number of pixels in the image src having a gray value corresponding to the array index to hist.

LNDIFF --- difference of logarithms

```
lndiff(srcl,src2,dst)
```

```
    IMAGE      srcl,src2,dst;
```

lndiff subtracts the pixel-by-pixel natural logarithm of the image src2 from the pixel-by-pixel natural logarithm of the image srcl, and exponentiating the result to produce the output image dst.

LOCMAX --- find local maxima

locmax(src,dst,nsize)

```
IMAGE src,dst;
int nsize;
```

locmax inspects the grayscale image src, producing the binary image dst. Pixels in dst are set if they are local maxima over a neighborhood of nsize by nsize, otherwise they are cleared.

LOCMIN --- find local minima

locmin(src,dst,nsize)

```
IMAGE src,dst;
int nsize;
```

locmin inspects the grayscale image src, producing the binary image dst. Pixels in dst are set if they are local minima over a neighborhood of nsize by nsize, otherwise they are cleared.

LTHRESH --- local thresholding

lthresh(src,lti)

```
IMAGE src,lti;
```

lthresh performs a pixel-by-pixel threshold of the source image src according to the image of thresholds in the local threshold image lti. This routine is useful, for example, after determining the lti via grayscale morphology.

NAVG --- neighborhood averages

navg(src,dst,select)

```
IMAGE src,dst;
int select;
```

navg perform selected averaging operations over the 4- or 8-neighborhood, according to the operation specified by "select".

<u>select</u>	<u>operation</u>
4	average over 4 neighborhood
8	average over 8 neighborhood
0	digital 3x3 LaPlacian

SHOWL --- illustrate a single labeled region

```
showl(src,dst,label,fore,back)
```

```
    IMAGE    src,dst;
    int      label,fore,back;
```

showl extracts the pixels in the label image src (produced by connect) which are marked with the label "label", to produce the output image dst. fore and back specify the graylevel to use for marking label pixels and for marking non-label pixels in the destination image. This utility is primarily useful for inspecting the output of analysis of the labeled image, perhaps by "showling" an identified region.

SOBEL --- sobel gradient magnitude estimator

```
sobel(src,dst)
```

```
    IMAGE    src,dst;
```

sobel computes the classic sobel gradient magnitude estimation of the input image src, producing the output image dst.

THRESH --- hard threshold

```
thresh(src,thr)
```

```
    IMAGE    src;
    int      thr;
```

thresh performs, in place, a hard threshold on the image src. Each pixel in the grayscale input image is tested against the constant thr. If the pixel is greater than thr, a 255 is returned in place, otherwise a zero is returned.

#### A.1.7 Intermediate level Utilities

CONNECT --- connected component labelling

```
int connect( src, dst )
```

```
    IMAGE    src, dst;
```

connects performs connected components on the binary image src, returning the label image dst. Conflicting labels which arise from image distribution are resolved. Set pixels are connected with four-connectivity, reset pixels with eight-connectivity. Labels 1 through 127 are assigned to reset

regions, labels 129 through 255 to set regions. Labels 0 and 128 are unused. connect returns the number of set regions in the high 8 bits, and the number of reset regions in the low 8 bits.

HOUGH --- hough transform

hough(src,dst,trig\_table)

```

    IMAGE    src,dst;
    int      *trig_table

```

hough performs a hough transform on the binary image "src" to produce the grayscale image "dst". Precomputed values from an integer trigonometric lookup table are used in this calculation, so the routine "table" must be called first.

MAXSTATE --- find largest region

struct blob \*maxstate(n,seglist)

```

    int n;
    struct blob seglist[];

```

maxstate searches the list of region features seglist for the most massive region, and returns the address of its structure.

MAKE\_BLOB\_LIST --- region analysis

make\_blob\_list(src,o\_list,nblobs)

```

    IMAGE    src;
    struct blob o_list[];
    int nblobs;

```

make\_blob\_list performs a primitive region analysis of the labelled regions computed by connect. o\_list is an array of struct blob, which should be malloced prior to calling this routine. The particular quantities returned for each region are mass, center of mass, and limits of the bounding box. o\_list is indexed by the labels assigned by the routine connect.

WHOUGH --- windowed hough transform

```
whough(src,dst,trig_table,xmin,ymin,xmax,ymax)
```

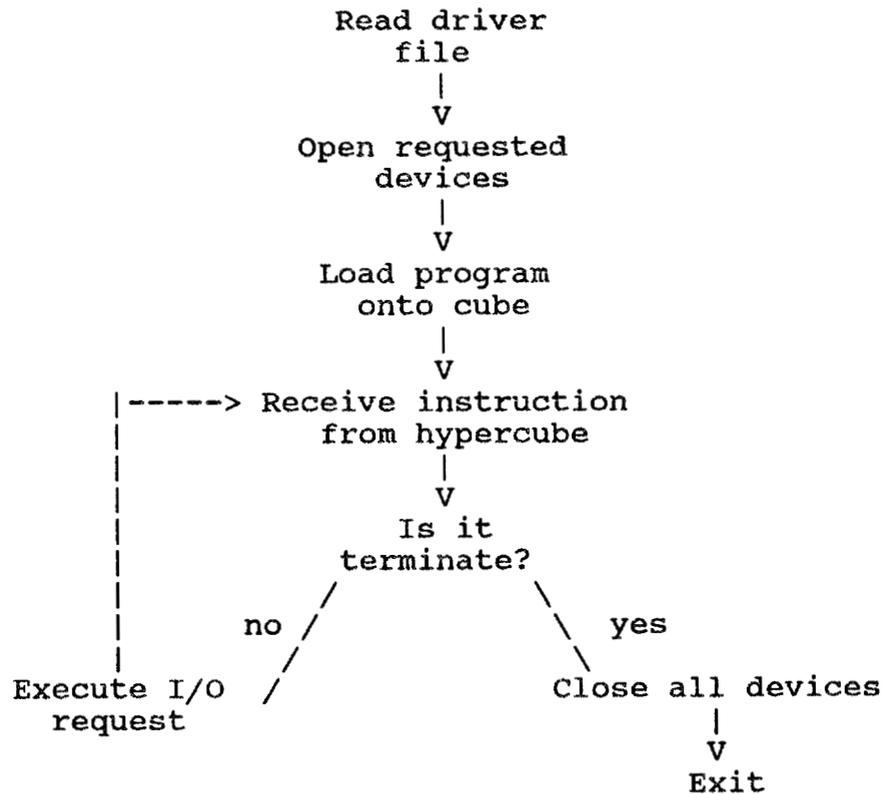
```
    IMAGE    src,dst;  
    int      *trig_table,xmin,xmax,ymin,ymax;
```

whough performs a hough transform on the image src, producing the image dst, using only those pixels in the window specified by the arguments xmin, ymin, xmax, ymax. whough is useful primarily for determining the pose of features in a segmented object, and secondarily for speeding up the global hough transform, since in general a smaller number of points will be transformed. This routine is globally balanced.

## A.2: Host I/O server and utilities

### A.2.1: The I/O server and its requirements

Almost all image processing and analysis applications employ a single program on the host. This program essentially acts as while forever switch statement, awaiting I/O commands issued from the hypercube, executing the requested I/O and awaiting new commands. Execution of the current implementation is outlined in the following flowchart.



Flowchart of Standard I/O Process

The text driver file has the following format:

```
'string'      name of node program
'in'          base input file name
'out'         base output file name
'u1'
'u2'
'u3'          u1 through u5 are unused string fields
'u4'
'u5'
4             maximum cube dimension
0             output display switch
0             output disk write switch
0             active input device
0             active graphics device
0             pause switch
0             debug switch
0             rotation stage switch
u5
u6            u5 through u8 are unused integer fields
u7
u8
nint          number of integer parameters following
i1
i2
...
nflt          number of floating point parameters following
f1
f2
...
```

The name of node program is a string up to 20 characters in length. Only the current directory is searched for this program. The input file name specifies a single file which can optionally be used as image data. This file is only read if the disc is the currently active input device. No provisions are made in the present system for disk based image sequence input. The base output file name specifies the filename portion of any images output to disc. Successive images are indexed by the extension part of the file name, beginning with '0'. Several string fields are reserved for future use. The first integer field specifies the maximum cube dimension which the program will allocate. If a cube of the maximum dimension is not available, cubes of successively smaller dimension will be requested. If no cube is available, the program terminates. The output display switch determines whether output images will be displayed (1) or not displayed (0). The output disk write switch determines whether output images will be written to disc (1) or not written to disc (0). The active graphics device switch specifies the the graphics device on which

output images will be graphed (0=NEC multisync, 1=VME subsystem). The pause switch determines whether the program will pause (1) or not pause (0) after each output image. The debug switch specifies whether the debug daughter process will be spawned (1) or not spawned (0). The rotation stage switch specifies whether communication to the Newport rotation stages is desired (1) or not desired (0). Several integer fields are reserved for future use.

The number of integer and floating point parameters to be passed to the hypercube on program initiation is specified using two integer entries in the driver file. Following each entry exactly the number of parameters specified must appear. Integer parameters are received by the node program in the first 50 variables of the array "crtl" (see iminit above); floating point parameters are received in the second 50 variables of the same array. This array can conveniently be accessed with a structure or union. No more than 50 parameters of each type are supported.

A small set of utilities have been implemented for each I/O device currently integrated into the system. In each case, the implementations support basic I/O functions, with occasional elaborations. Access to each device could arguably be greatly improved. The standard host process described in this section currently supports only some of the available utilities.

#### A.2.2 VME subsystem

IDROPEN --- open communications channel to VME subsystem

```
ichan = idropen()
```

```
integer*4 ichan
```

idropen opens a communications channel to the VME subsystem (/dev/drllw). The I/O channel number is returned.

IDRCLOSE --- close communications channel to VME subsystem

```
null = idrclose()
```

idrclose closes the communications channel to the VME subsystem.

PGET64 --- digitize image

```
null = pget64(pic)
```

```
integer*1 pic(65536)
```

pget64 instructs the VME subsystem to digitize a 64 kilobyte image and transmit the image to the NCUBE host processor, storing the image in the buffer "pic". The image is stored in a standard raster scan format, with the first byte corresponding to the pixel at the extreme upper left of the image, and the last byte corresponding to the pixel at the extreme lower right. The image format is 256 rasters, 256 pixels per raster, 8 bits per pixel.

PPUT64 --- display image

```
null = pput64(pic)
```

```
integer*1 pic(65536)
```

pput64 transmits a 64 kilobyte image from the buffer "pic" to the VME subsystem, and displays the image on the VME's monitor.

Comments: for simplicity, only 256x256x8 bit images are currently used. It is reasonably straightforward to generalize these routines, and the corresponding VME subsystem routines, to support arbitrarily dimensioned images, within certain limits established by the digitizer. An ideal mechanism would permit the definition and transmission of run-time defined image formats.

FLTPUT --- specify analog filter prior to digitization

```
null = fltput(nfilt)
```

```
integer*4 nfilt
```

fltput specifies analog prefiltering of the video signal. The digitizer supports four options for analog preprocessing of the video signal prior to digitization. These filters are principally useful for eliminating high frequency noise. Legal values for filter selection are 0 to 3: 0=3 MHz lowpass; 1=2 MHz lowpass; 2=4.5 MHz lowpass; and 3 = no filter. The default setting is no filter.

MUXPUT --- select video source

```
null = muxput(nmux)
```

```
integer*4 nmux
```

muxput selects the video source for image digitization. The digitizer supports up to 8 possible video sources. Currently there are two CCD cameras connected to multiplex channels 1 and 2. By convention, mux channel 1 corresponds to the left camera, and mux channel 2 to the right camera. Calls to pget64 digitize images using the most recently selected video source. The default selection is left camera (nmux = 1).

GANGET --- read current input gain setting

```
gain = ganget()
```

```
integer*4 gain
```

ganget returns the current setting of the analog preamplifier.

GANPUT --- set digitizer input gain

```
null = ganput(gain)
```

```
integer*4 gain
```

ganput sets the preamplifier. The digitizer features a programmable analog preamplifier which precedes digitization. Legal values for gain are 0-7, with 0 specifying low gain and 7 high gain. This facility is principally used for preconditioning the signal to make optimal use of the 8 bit dynamic range.

DCOPUT --- set digitizer D.C. offset

```
null = dcopt(dc)
```

```
integer*4 dc
```

The digitizer features a programmable analog D.C. offset which precedes digitization. Legal values for "dc" are 0-255, with 0 corresponding to blacker images, and 255 corresponding to whiter images. This feature is used principally for preconditioning the signal to make optimal use of the 8 bit dynamic range.

SEQUENCE --- regular sampling

sequence(snap\_interval)

integer\*4 snap\_interval

sequence supports the automatic digitization of images at regular sampling intervals. Images are digitized at a sampling rate which is some integer multiple of 0.01 seconds, specified by the argument snap\_interval. Images are internally circularly buffered in the VME subsystem to a depth of 16 images.

A.2.3 SBX graphics device

IGOPEN --- open communications channel to display device

ichan = igopen()

integer\*4 ichan

igopen opens /dev/matrox and returns the channel number. igopen also initializes the Hitachi ACRTC chip for 640x480 resolution.

IGCLOSE --- close communications channel to display device

null = iglclose()

igclose closes /dev/matrox

IDUMP --- display 64 Kbyte image on CRT

call idump(image,xpos,ypos)

integer\*1 image(65536)

integer\*4 xpos,ypos

idump uses the DMA facilities of the Hitachi ACRTC to transfer a 64 kilobyte image from host memory to the display memory, with the upper left hand corner of the image located at position xpos, ypos.

JDUMP --- display variable size image on CRT

call jdump(image,ixpos,iypos,ixwin,iywin)

jdump uses the DMA capability of the Hitachi ACRTC to transfer an image from host memory to display memory. The image is placed with its upper left hand corner at absolute position xpos, ypos. The image is assumed to conform to a raster scan format with ixwin bytes per raster and iywin rasters. No more than 64 Kbytes of data can be in the image buffer.

GREYSCALE --- set up grayscale LUT

call greyscale

greyscale sets the bits in the color LUT of the ACRTC to contain a mapping from the integers 0-255 into a grayscale image. 0 corresponds to black, and 255 to white.

PCOLOR --- set up pseudocolor LUT

call pcolor

pcolor sets the bits in the color LUT of the ACRTC to contain a simple pseudocolor mapping of the integers 0-255 into an RGB image. 0 corresponds to black, low integers to blue, and high integers to red.

#### A.2.4 Disk Access

IMREAD --- read image from disc

call imread(name,buffer)

```
character*40  name
integer*1    buffer(65536)
```

imread reads the 64 kilobyte image from disc file "name" into a 64 Kbyte area of memory beginning at address "buffer".

IMWRITE --- write image to disc file

call imwrite(name,buffer)

```
character*40  name
integer*1    buffer(65536)
```

imwrite writes the 64Kbyte image found at address "buffer" to the disc file "name".

### A.2.5 Hypercube image I/O

IMREADC --- read an image from a hypercube

call `imreadc(ncube,ndim,buffer)`

```
integer*4 ncube,ndim
integer*1 buffer(65536)
```

`imreadc` interacts with "imageout" to collect a 64 Kbyte image from the hypercube "ncube" having dimension "ndim" to a contiguous area of memory in the host address space, beginning with the address "buffer".

IMWRITEC --- write an image to a hypercube

call `imwritec(ncube,ndim,buffer)`

```
integer*4 ncube,ndim
integer*1 buffer(65536)
```

`imwritec` interacts with "imagein" or "input\_q" to distribute a 64 Kbyte image from a contiguous area of the host address space onto the hypercube "ncube" having dimension "ndim". The image address is specified in "buffer".

### A.2.6 Newport rotation stages

IRSOPEN --- open serial communication channel to controller

`ichan = irsopen()`

```
integer*4 ichan
```

`irsopen` opens a serial communications channel to the Newport rotation stage controller. The channel number is returned.

IRSCLOSE --- close communications channel

`null = irsclose()`

`irsclose` closes the serial communications channel to the Newport rotation stage controller.

IRSVEL --- set stage velocity

i = irsvel(stage,vel)

integer\*4 stage,vel

irsvel sets the stage velocity of the rotation stage "stage" to velocity "vel". Legal values for "stage" in this and all other calls to rotation stage routines are 1,2,3, and 4. Legal values for "vel" are in the range 0-8000. The physical units are millidegrees per second.

IRSORG --- set stage origin

i = irsorg(stage)

integer\*4 stage

irsorg resets the coordinate system of stage "stage". The current position of the stage when irsorg is called is thereafter the origin of the coordinate system.

IRSLIM --- set soft limits

i = irslim(stage,limup,limdown)

integer\*4 stage,lash

irslim sets the limits of motion for stage "stage". The limits are expressed in millidegrees relative to the current origin. limup and limdown specify the limits in the clockwise and counterclockwise directions respectively as the stage is viewed from above. Legal values for both limits are in the range +/-999999.

IRSLASH --- set stage backlash

i = irslash(stage,lash)

integer\*4 stage,lash

irslash sets the backlash parameter on stage "stage". This parameter controls overshoot on rotations in the counterclockwise direction so that the terminal position on all motions is approached from the clockwise direction. This feature minimizes hysteresis.

IRSVELQ --- interrogate current velocity

vel = irsvelq(stage)

integer\*4 stage,vel

irsvelq returns the current rotation velocity setting on stage "stage".

IRSLOCQ --- interrogate current position

loc = irslocq(stage)

integer\*4 loc,stage

irslocq returns the current position of stage "stage"

IRSLIMQ --- interrogate current limit settings

null = irslimq(stage,up,down)

integer\*4 stage,up,down

irslimq returns the current soft limit settings of stage "stage" in the variables up and down.

IRSLASHQ --- interrogate current backlash

lash = irslashq(stage)

integer\*4 stage,lash

irslashq returns the current backlash setting of stage "stage".

IRSHOME --- move to origin

null = irshome(stage)

integer\*4 stage

irshome moves stage "stage" to its current origin.

IRSMOVA --- absolute move

```
null = irsmova(stage,loc)
```

```
integer*4 stage,loc
```

irsmova performs an absolute rotation of stage "stage" to location "loc", where loc is given in millidegrees relative to the current origin. Legal values are in the range +/- 999999.

IRSMOVR --- relative move

```
null = irsmovr(stage,rel)
```

```
integer*4 stage,rel
```

irsmovr performs a relative rotation of stage "stage". The stage is rotated by the number of millidegrees given in "rel".

#### A.2.7 Console

ITTYFLY --- get a keystroke on the fly

```
i = ittyfly(char)
```

```
integer*4      i
character*1    char
```

ittyfly inspects the character ready bit on the standard input serial communications channel, returning 0 in both i and char if no character is presently available, and returning the character code if a character is available.

ITTYIN --- get a single keystroke

```
i = ittyin(char)
```

```
integer*4      i
character*1    char
```

ittyin returns the next available character in both i and char.

CURSOR --- cursor on/off

call cursor(switch)

integer\*4 switch

cursor emits the ANSI standard escape sequences for making the terminal cursor visible or invisible, according to the argument switch (1 or 0 respectively).

CLRHOM --- clear screen and home cursor

call clrhom

clrhom emits the ANSI standard escape sequences necessary for clearing the terminal screen and positioning the cursor at row 1 and column 1.

CURPOS --- absolute cursor positioning

curpos(row,col)

integer\*4 row,col

curpos emits the ANSI standard escape sequences necessary for positioning the cursor at absolute position row and col.

#### A.2.8 Miscellaneous

NGRAY --- compute standard graycode

gray = ngray(arg)

integer\*4 gray, arg

ngray returns the graycode of its argument.

NPHYS --- invert standard graycode

phys = nphys(arg)

integer\*4 phys, arg

nphys returns the inverse graycode of its argument.

NRING --- standard graycode ring

call nring(node,up,down)

integer\*4 node,up,down

nring computes the physical node numbers of the upwards and downwards nearest neighbors of the node whose number is "node" in the standard graycode ring.

SEQNAME --- compute a name string in sequence

call seqname(base\_name,target\_name,index)

character\*20 base\_name, target\_name  
integer\*4 index

seqname computes a character string in a sequence of character strings. The index is converted into a two character digit string in the range 0 to 99 and appended to the base\_name to form the target\_name. For example, the sequence of names a.0, a.1, a.2,... can be formed. This is principally used for performing disc I/O to store the results of successive image processing steps.

## INTERNAL DISTRIBUTION

- |                                 |                            |
|---------------------------------|----------------------------|
| 1. S. M. Babcock                | 20. S. M. Killough         |
| 2. D. L. Barnett                | 21-25. F. C. Maienschein   |
| 3. M. Beckerman                 | 26-30. R. C. Mann          |
| 4. B. Burks                     | 31. E. M. Oblow            |
| 5. K. M. Clinard                | 32. L. E. Parker           |
| 6. G. de Saussure               | 33. F. G. Pin              |
| 7. J. J. Dorning (Consultant)   | 34. D. B. Reister          |
| 8. J. R. Einstein               | 35. C. R. Weisbin          |
| 9. C. W. Glover                 | 36. B. A. Worley           |
| 10. E. C. Halbert               | 37. A. Zucker              |
| 11. W. R. Hamel                 | 38. Central Res. Library   |
| 12. J. Han                      | 39. Y-12 Doc. Ref. Section |
| 13. R. M. Haralick (Consultant) | 40. Lab. Records, ORNL/RC  |
| 14. J. E. Jones Jr.             | 41. ORNL Patent Office     |
| 15-19. J. P. Jones              | 42-46. EPMD Reports Office |

## EXTERNAL DISTRIBUTION

47. Office of Assistant Manager for Energy Research and Development, DOE-ORO, P. O. Box 2001, Oak Ridge, TN 37831-8600.
48. H. Alter, Advanced Technology Development, Office of Technology Support Programs, DOE, Washington, DC 20545.
49. O. P. Manley, Division of Engineering and Geosciences, Office of Basic Energy Sciences, ER-15, DOE, Washington, DC 20545.
50. R. J. Neuhold, Advanced Technology Development, Office of Technology Support Programs, DOE, Washington, DC 20545.
51. B. J. Rock, Director, Office of Technology Support Programs, DOE, Washington, DC 20545.
52. Major Frank Holly, U. S. Army Laboratory Command, Human Engineering Laboratory, Aberdeen Proving Grounds, MD 21005-5001.

ORNL/TM-10679  
CESAR-88/04

53. E. M. Simpson, University of West Florida, Pensacola, FL.
54. J. F. Palmer, NCUBE Corporation, 915 E. LaVieve Lane, Tempe, AZ 85284.
55. M. C. G. Hall, 6559 Bison Court, San Jose, CA 95119.
- 56-65. Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37830.