COMPUTERIZED MOLECULAR MODELLING USING SATELLITE GRAPHICS FACILITIES

by

STEPHEN ASHLEY WARD

S.B., Massachusetts Institute of Technology
(1966)

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January, 1967

Signature of Author _
Department of Electrical Engineering, January 14, 1967

Certified by _
Thesis Supervisor

Accepted by _
Chairman, Departmental Committee on Graduate Students

Archives

COMPUTERIZED MOLECULAR MODELLING USING SATELLITE GRAPHICS FACILITIES

by

STEPHEN ASHLEY WARD

Submitted to the Department of Electrical Engineering on January 20 1969 in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

A system of computer programs is described by means of which an investigator may observe and adjust models of large protein molecules by direct interaction. The system includes provision for generating such a model (e.g., calculation of the coordinates of each atom) from a list of the constituant amino acids and a table of values of the variable bond angles, for modifying the model according to user instructions and subject to the chemical and mechanical constraints which are known to exist, and for the display of the 2 dimensional projection of an arbitrary (user selected) aspect of the stick-figure representation of the model. This latter function is performed by a small, dedicated processor connected by a voice grade telephone line to the main (time shared) computer on which the parameters of the model are stored. Emphasis throughout centers on the implementation rather than use of the system, and the details presented fall generally into two areas of technical interest: (1) the simulation by computer of the constraints which exist on the geometry of a molecule, and (2) the real-time rotation and display of a 3-dimensional stick figure.

THESIS SUPERVISOR: Cyrus Levinthal
TITLE: Professor of Biology

II. Acknowledgement

The computerized molecular model building activities of Professor Cyrus Levinthal, who supervised this thesis, predate me by several years; the system whose implementation is described in this paper is thus simply the most recent iteration in the gradual development of a collection of programs which have already been influenced by a number of people. This paper, in order to provide useful documentation of the current system, then, necessarily describes work resulting from the efforts of several people, of which mine are only the latest.

An earlier system, which made use of a display directly connected to the time shared computer at M.I.T.'s Project MAC, was largely programmed by Hal Murray, whose innovations remain evident in the present programs. Andrew Pawlikowsky contributed a number of the essential mathematical subroutines. Dr. Martin Zwick is responsible for several routines which organize information about the chemical attributes of parts of the model, as well as much of the chemical insight which motivated this and earlier implementations. Dr. C. D. Barry provided similar insight during the more recent stages of refinement and debugging of the system. The architecture of the system has been most influenced, of course, by Professor Levinthal, who has unified and directed the efforts of all of us. I gratefully acknowledge the contributions of all of those who have made my part in this project stimulating and fruitful, and especially the loving encouragement of my wife Debbie.

Work reported herein was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number NONR-4102(01).
Reproduction in whole or in part is permitted for any purpose of the United States Government.

Table of Contents

I. Abstract 2

II. Acknowledgement 3

III. Table of Contents 4

IV. Problem Background and Description 5
- 4.1 Functions of System 6
- 4.2 Division of Functions 7

V. Overall Structure of System 10

VII. Problem-orientated Programs and Data . . . 14
- 7.1 Main Processor Data 14
- 7.2 Main Processor Programs 16
  - 7.2.1 MAKPRO - coordinate calculation from amino acid sequence 20
  - 7.2.2 COORD - coordinates from bond angles 26
  - 7.2.3 JIGL - residue coordinates from residue angles 30
  - 7.2.4 SOLVE - angle changes from atomic distance changes 34
  - 7.2.5 SETANG - angle changes 44
  - 7.2.6 LISTQQ - list interacting atoms 48
  - 7.2.7 WIRE - display list from coordinates 58

VIII. Interconnection of Processors 65

IX. Satellite Programs and Data 67
- 9.1 Supervisory Functions 67
  - 9.1.1 Device timing considerations . 68
- 9.2 Picture Handling Functions 71
  - 9.2.1 Data Structures 71
  - 9.2.2 Input Phase 73
  - 9.2.3 Picture building Phase 74
  - 9.2.4 Rotation Phases 77
  - 9.2.5 Display 82
- 9.3 Auxilliary Functions 83

X. Conclusion 85

XI. Appendices
- 11.1 References 87
- 11.2 Arithmetic Library 88

IV. Background and Description of Problem

The goal of this project has been the establishment of an interactive computer system, comprising a main processor and a remote satellite, on which the physical configuration of protein molecules may be modelled. The value of the result as a research tool rests on the hypothesis that an abstract representation, deformable by changing parameters in a machine's memory store, is a more versatile and tractable alternative to the physical "tinkertoy" models commonly used for this purpose. Certainly the potential uses of the computer model extend beyond those of the mechanical one: the availability of the parameters of the model for peripheral calculations by the machine suggests a wide range of potential functions not attainable by the older methods. The obvious reservations regarding the usefulness of the system, then, question rather the ability of the machine to perform adequately in a digital fashion those functions which are handled naturally in the physical model --- those which result, indeed, from the fact that we are ultimately modelling a physical situation. Such functions include the preservation of the 3-dimensional character of the model and the prevention of two disjoint, solid portions of the model from occupying the same space.

Before specific discussion of the features of the system, a description of the model which we are building is in order. From a topological standpoint, we may treat a molecule as a stick model consisting of a number of nodes interconnected by branches in such a way that each node is ultimately connected to each other node, and that the branches generally form a divergent "tree" structure rather than forming closed loops (trivial exceptions to this latter constraint are discussed in a later section). Furthermore, there is a degree of rotational freedom associated with each branch: the respective portions of the model connected by the branch may rotate with respect to one another about an axis parallel to that branch. An angle-measuring convention allows us to assign values to the degree of rotation associated with each branch, which we term _bond angles_ for the chemical feature we are modelling. Since, for a particular molecule, the topology as well as the lengths of the branches are fixed, the specification of the complete set of bond angles completely determines the particular configuration of the model. It is this set of bond angles which, at the lowest functional level of the system, constitutes the set of independent variables.

The topological model, however, is further constrained by chemical considerations. A protein, it happens, is built from a linear string of an arbitrary number of _amino acids_, of which there are twenty-odd topologically distinct varieties. They attach to one another in such a way as to form a regular, periodic main chain (or _backbone_) with the

remainder of each amino acid hanging from the backbone in a side chain (<u>residue</u>). The diagram of figure 1 represents schematically a typical section of a large protein comprising 5 amino acids. Since the branches of the backbone are not parallel to each other, it may fold over on itself, and in fact the gross structure of the molecule is restricted relatively little by the backbone constraints. The bond angles of the residues may be varied more or less independently so that the residues, which comprise most of the atoms in the molecule, are relatively free to move around in the vicinity of their attachment to the backbone.

We are left with a large number of degrees of freedom, and, in fact, in handling a mechanical model of a large protein, one is impressed by its obvious plasticity. Since such a model incorporates the constraints which we have described, as well as the physical constraint that several solid bodies may not occupy the same space (which, incidently, is largely applicable to molecules and is simulated by the computer system,) our contention that different instances of a protein all have the same configuration postulates further constraints on the physical system we are modelling. It is these remaining constraints which the system is intended to clarify: their appreciation is currently largely intuitive and qualitative, and it is hoped that as each constraint becomes well defined and understood it may be added to those automatically imposed by the system.

<u>4.1 Functions of System</u>

The design criteria of the computer modelling system consist of a number of top-level functions (i.e., functions apparent at the user level) which are considered essential to a useful model. The following are all required:

1) Display of a 3-dimensional figure - perhaps the most critical advantage of the mechanical models (and the one most difficult to achieve on the computer system) involves the ease with which the user can appreciate the physical configuration in 3-space which he is modelling. The process of looking at a physical model while manipulating and rotating it is unquestionably the most natural way to develop insight into its 3-dimensional character. In an attempt to simulate this interactive coupling between the user and his model, we have chosen to utilize a cathode ray tube display of a 2-dimensional projection of the 3-dimensional structure representing our model. To complete the user's understanding of the solid character of the model and to achieve something of the interactive nature of a close-hand examination of a physical

model, we allow the user to manipulate a control which has 3 mechanical degrees of freedom and which controls the rotational position of the projected figure. Thus, by a few relatively uncomplicated hand movements, a user may view the model in any aspect he wishes.

2) Configuration of molecule from amino acid sequence - the system must be able to deduce the connectivity of the model from the sequence of amino acids from which the protein is composed.

3) Coordinates of atoms from angles - given the connectivity of the model and the values of the variable bond angles, the computer system should be able to calculate the coordinates in 3-space of each of the constituent atoms. Note that the coordinates and the connectivity completely specify the picture observed by the user.

4) Collisions between atoms - the system must be able to detect and prevent manipulations of the model which cause several atoms to occupy the same physical space. In practice, this involves identifying each type of atom, and insuring that no pair of atoms is nearer than the sum of their respective van der Waals radii.

5) Manipulate specified inter-atomic distances - it seems clear that the specification of bond angles is an awkward method of manipulating the physical configuration of the model. Often a user is interested in ultimately changing the distance between some pair of atoms; to this end, we have stipulated the requirement that the system be able to calculate the changes in bond angles necessary to affect some particular change in the distance between a specified pair of atoms. One of the motivations of such a provision has been the recovery from the violations of (4) above; that is, the automatic correction of colliding pairs of atoms.

4.2 Division of Functions between Processors

The computing requirements of our problem are such that we need, on one hand, significant arithmetic power while on the other hand we require a dedicated machine for display purposes; the most viable solution appears to involve the interconnection of a large, time-shared computer with a small satellite computer to maintain the display. This use of multiple processors requires us to distinguish between functions to be performed on the main processor and those to

be performed on the satellite. The following considerations pertain to this distinction:

1) The cost of the main processor operation is proportional to the main processor time used, while the satellite cost is fixed with respect to the amount of work it performs. Thus, the economics of the situation seem optimized when the satellite is kept busy nearly all of the time.

2) Communication between the main processor and the satellite is necessarily more restricted than communication within either processor. Thus the functions should be divided in such a way as to minimize this bottleneck.

3) The satellite must maintain the display; thus, it seems reasonable to impose on the satellite other tasks relating to the display function.

4) The computing power and storage facilities of the main processor suggest that we store our problem-oriented data (parameters of the model, etc.) on the large machine where it will be available for the major computations.

5) Advanced system software on the main processor favor the development of experimental programs here rather than on the small machine; thus, we prefer to keep the program at the satellite general enough in its applicability that it is not continually being modified.

Generally, we would like to differentiate between functions which are to be performed in real time and with continual, direct user interaction and those which may be performed in batches at intervals of a few seconds. Since we might ideally like to perform <u>all</u> of the functions in real time, this distinction is largely a compromise with our limited capacity for computation at the satellite.

Our present compromise restricts the satellite to performing functions related to the display of a generalized 3-dimensional stick figure. This configuration is the result of the above considerations, as well as the desire to extend the range of usefulness of the satellite subsystem beyond the scope of our particular problem.

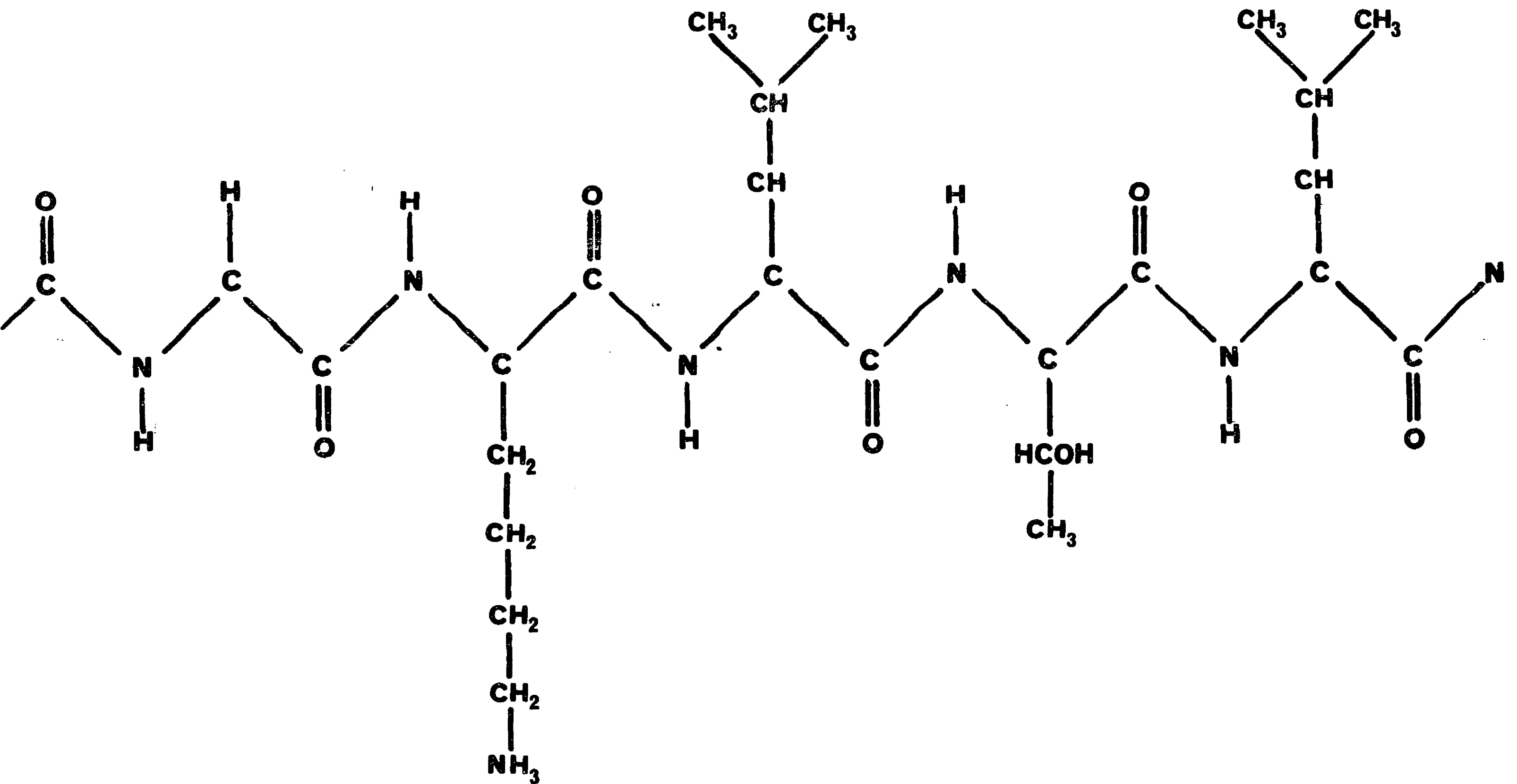

figure 1: Schematic representation of a section of typical protein molecule

V. Structure of System

The chain of control of the system, from its primary input (user commands typed at the console) to its primary output (the graphical display) encounters several levels of programming at each machine, and undergoes a number of involutions along the way. The linear, simple connectedness implied by the word "chain" is misleading: the control structure of the overall package is a multidimensional plexus of interconnected and highly recursive programming. A one-dimensional outline presents an obviously inadequate representation of such structure; although it can be structured to some extent to depict hierarchal relationships, it is constrained to neglect many of the interesting features of the architecture of the system.

We shall, however, use such an outline to introduce the reader to the skeletal structure of the system, advising him of the more subtle interconnections in later sections of the paper. We have structured the descriptions in an attempt to limn the gross hierarchy of control; however, there is much "horizontal" communication (i.e., between programs at the same level in the hierarchy) which cannot be represented. We note in the descriptions several such cases.

Outline of Control Structure

I. Main programs on main processor:

The programming of the main processor, because of limitations on the size of core memory, have been partitioned into several major segments. The partitions, or LINKs, reside on secondary storage while they are not in core memory; one link at a time is then loaded into memory (in addition to the permanently resident data structure, and a small bootstrap loader) and executed. The partitions have been arranged to minimize the amount of such loading, which is relatively time consuming: we may thus justify each link's autonomy on the basis of a comparison between the fraction of a second (on the order of one half) spent in the loading of the link to a significantly longer time spent in that link's execution.

1) LINK 1: interaction with user via teletype. The first link justifies its status by its synchronization with the relatively slow operations of teletype communication. In one respect, the main program in this link (each link has its own) may be considered to occupy the highest echelon in the control hierarchy, since it

is ultimately responsible calling each of the major subsystems into action; however, beyond the present discussion we shall find it expedient to ignore the physical boundaries of the partitions and consider each of the other main programs as extensions of that in the first link. The first link, then, contains the machinery for decoding the input language, and calling (chaining to) each of the other links as required by the requests of the user. The first link also contains some simple routines which allow the user to access the permanently resident data structure directly - i.e., to print out and enter new values in the common data storage area, as well as to write such data in permanent files on secondary storage.

2) LINK 2: calculation and modification of data structure. The second link contains the programs which perform the time consuming calculations necessary for the manipulation of individual interatomic distances in the model. Since each user request typically involves a number of iterations thru a cycle involving setting up the equations to be solved, solving them, checking that the resulting structure is valid, and setting up equations to be solved on the next iteration, it behooves us to insure that each program essential to this iteration is in link 2 to avoid building a chaining operation into the loop. Since each call to the programs of link 2 requires typically several seconds of computation time, the time spent in its loading again presents a small overhead to the system. Link 2, then, contains programs to:

a: Calculate, from a list of changes to be made in specific interatomic distances, changes to be made in bond angles to effect these distances.

b: Modify the bond angle representation in accordance with (a).

c: Calculate, from the new bond angle representation, new coordinates for each of the atoms in the molecule.

d: Determine, from the new coordinates, those pairs of atoms which are near enough one another to be in violation; a list of such pairs (and the distances they must be moved to remove the violation) is added to the list used as input to (a).

3) LINK 4: interface with the satellite display. It is link 4 which encodes the data structure of the main processor into a description of the 3-dimensional figure which the user ultimately observes on the remote display. The basic time limitation involved in this operation is the low data rate of the (telephone) line connecting the display processor with the main processor; this data rate is such that even a small picture takes several seconds to be transmitted, and a complex one may take a minute. The following functions are involved in the communication of the picture data between the common storage area of the main processor and the display hardware of the satellite:

a: A major main-processor program reconstructs the topology of the molecule in terms of connected lines in the frame of reference in which atomic coordinates are stored in the main processor. This program utilizes inherent knowledge of the topology of each amino acid type, and results, basically, in a list of "move" and "draw" commands in addition to some character information such as labels.

b: This description is further encoded, to minimize the actual number of bits being sent over the low data rate line; in the process, the vectors supplied in the last step are translated to a new frame of reference and converted to integers.

c: This data is blocked and sent over the telephone line to the remote processor.

d: At the remote processor, the data is unblocked and stored in a preliminary buffer; this buffer allows the two processors to operate asynchronously.

e: The data is eventually removed from the preliminary buffer, and the "move" and "draw" information of step (a) is reconstructed. Since the precision of the data sentover the line is much lower than the precision required in the data either at step (a) or here, some conventions are followed by the programs which encode and decode data for transmissions regarding accumulation and correction of truncation errors.

f: The "move" and "draw" representation is then rotated and translated according to internal parameters, and the result stored in a display list. The internal parameters, in particular, the rotation matrix which determines the position from which the user is to view the 3-dimensional figure, is continually being updated according to the user's instructions.

1: The "globe", an input device with three rotational degrees of freedom, is examined. As there are seven discrete states of the globe for each degree of freedom (designated as 0, +,-1, +,-2, +,-3) we allow three distinct rates of rotation about each axis in either direction, as well as no rotation.

2: For the rotations about each axis, an incremental matrix is constructed and used to update the rotation matrix. This rotation by incremental matrices allows us to accumulate a substantial rotation while ducking the question of which of the rotations applies first, since incremental rotations are commutative.

g: The display hardware, meanwhile, cycles asynchronously thru the display list (the output of step (f), above) to produce the picture.

VII. Problem-oriented Programs and Data

These paragraphs will deal with that portion of the total computer system which is most closely related to the particular problem of protein modelling. Since, as we have explained, the programs operating on the main processor largely constitute the problem-dependent portion of the system, this section becomes an overview of the main processor software. We begin with a description of the data formats (the format of the internal description of the model) and follow with a summary of the major programs for their manipulation.

7.1 Main-processor Data

The following data all resides in an area of common storage in the memory of the main processor. The symbolic names of the areas of data are included primarily for correlation of this discussion with the program listings:

ATOM(Q) is a one-dimensional array containing one element for each atom in the molecule and including information regarding each atom's type, associated amino acid, etc. The contents of the ATOM array are constant for a particular protein; thus the initialization program (MAKPRO) alone is allowed to modify it. Refer to the discussion of MAKPRO for a detailed description of the contents of ATOM.

X,Y,Z(Q) are each one-dimensional arrays containing, as floating point numbers, the coordinates of each atom in 3-space. The indexing of these arrays is, again, on an element-per-atom basis as in ATOM, above.

XP,YP,ZP(QP) are one-dimensional arrays containing the coordinates of each residue atom in a frame of reference local to that residue. The residue coordinates are, then, stored redundantly: first in the XP,YP,ZP arrays in local frames and secondly in X,Y,Z in a global frame of reference. This redundancyhas the practical advantage of requiring the residue coordinates to be generated only once; after this initialization the new residue coordinates are obtained, as bond angles are rotated, by applying each rotation to the previous coordinate values of the affected atoms. Thus, initially MAKPRO fills XP,YP,XP from tables of the zero-angle values of the coordinates of the atoms for each type of residue; these are then subsequently modified (by the subroutine JIGL) as each residue angle is varied. The coordinate

calculating program (COORD) translates the local coordinates in XP,YP,ZP to the global ones of X,Y,Z.

AA(A) is an array with an element for each amino acid, and contains integers specifying the type of each acid. The integer codes vary from 1 to 21 for the 21 permissable residues.

QCALPH(A) is an array containing, for each amino acid, the sequence number (as used by the ATOM array) of the corresponding carbon alpha atom. The carbon alpha is the atom in the periodic main chain to which each residue is attached, and it is relative to the coordinates of this atom that the residue coordinates (in XP,YP,ZP) are stored. QCALPH, again, is set up by the routine MAKPRO and remains constant subsequently.

ANG(N) is an array in which the values of the variable backbone bond angles are stored. As there are two variable bond angles in the main chain for each residue (namely, the bonds on either side of the carbon alpha are variable) there will generally be twice as many entries in ANG as in AA.

ANGP(M) is a similar array containing the values of the variable residue angles. The relation between ANG and ANGP differs from that between X,Y,Z and XP,YP,ZP in that there is no redundancy in the angle storage; a bond angle may be either in the backbone or in a residue, but not both. The angles are thus separated into two arrays simply to make each more accessable.

ANGPTR(A) is an array containing, for each amino acid, the index of the first entry in ANGP corresponding to an angle in that residue. Note that a similar array could have been provided to index XP,YP,ZP except that a simple manipulation of the QCALPH array provides the same function: since X,Y,Z contains the residue atoms plus a constant number (7) of backbone atoms for each amino acid, we observe that the first XP,YP,ZP entry for residue A is indexed by QCALPH(A)-7*A+7.

WHICH,WORK,DANG(I) are arrays containing one element for each angle currently being varied. In general, the user will have specified explicitly (by teletype or light-pen) or implicitly (by violations of van der Waals radii) that specific interatomic distances must be modified. One of the main processor programs (SOLVE) determines which of the variable angles must be rotated to

affect these modifications, and stores their indices in the array WHICH and the amount of change required to affect the desired deformation of the molecule in the array DANG. A subsequent inability to correct any of the angle values as specified in the DANG array (e.g., because of a restricting van der Waals violation) is reported back to the SOLVE program on the next iteration through the WORK array, which allows otherwise variable angles to be temporarily considered fixed.

LQ1,LQ2,LDIST(J) are the arrays in which requests for changes in specific interatomic distances are requested. LQ1 and LQ2 contain the indices of the two atoms in each pair, and LDIST contains the change in the interatomic distance required.

7.2 Main-processor Programs

Reference to the flow diagram of figure 1, which depicts the relationships between main processor subprograms and data, will enhance the following discussions of the major functions:

MAKPRO() is the program which calculates, from a list of the amino acid sequence in a protein, the internal data structure modelling the instance of that protein in which each bond angle is at its zero position. This program refers to a (constant) table of the coordinates, specified in a local frame of reference, of each of the allowable amino acid varieties. Examining the type of each amino acid in turn, MAKPRO places the coordinate values from the table for that acid into the XP,YP,ZP arrays and sets the corresponding QCALPH entry to address the element of the X,Y,Z arrays in which the coordinates of the carbon alpha atom of that amino acid will ultimately be stored. Since each amino acid contributes uniformly a nitrogen atom (with an attached hydrogen,) a carbon-alpha (to which the residue is attached) and another carbon (with an attached oxygen) to the backbone of the molecule, the coordinates of these atoms are not stored in either the constant reference table or in the XP,YP, and ZP arrays; in addition, two of the residue atoms (one a hydrogen, the other usually a carbon beta) which are connected to the carbon alpha and thus fixed with respect to it are consequently treated as backbone atoms. A subsequent call to the subprogram COORD (below)

completes the coordinate calculations.

COORD() is the program which calculates the positions of the backbone atoms from the backbone bond angles and rotates and translates the residue coordinates from the local frames of XP,YP,ZP to the global frame of X,Y,Z. This involves adjusting the frame of reference 3 times for each residue (for the 3 backbone bond angles) and placing the coordinates of the residue atoms, rotated and translated from XP,YP,ZP, into the X,Y,Z arrays at the appropriate point.

JIGL(ACID.NO,ANGL.NO,DELTA) is the program which is used to vary residue bond angles. The program COORD, which interprets the backbone angles, copies the residue coordinates from XP,YP,ZP without reference to the variable angles which may effect them; consequently, when a residue angle is changed the appropriate XP,YP,ZP values must be updated also. The call to JIGL specifies the residue containing the angle to be changed, which angle in the residue will be changed, and the amount by which the bond is to be rotated. Implicit in the program JIGL is information regarding the connectivity of the various types of residues to determine the atoms effected by a given rotation. Each call to JIGL results in modifications of both the ANGP array (to reflect the new value of the angle) and the XP,YP,ZP arrays (to reflect the new coordinates, in the local frame, of the residue atoms). A call to COORD then incorporates the modified residue angle into the X,Y,Z arrays.

WIRE() is the program which communicates to the satellite display an appropriate description of the 3-dimensional figure to be shown. Accessing the list (AA) of the amino acid sequence and (XYZ) the global coordinates of atoms, the program relies on implicit knowledge of the connectivity of the various residue types to adduce the connectivity of the entire molecule. A set of parameters in the common data area determines the regions of the molecule to be displayed, the amount of detail desired, etc., from which WIRE codes a description of the 3-dimensional stick figure which is relayed to the satellite.

LISTQQ() examines the global atomic coordinates (X,Y,Z) and determines which pairs of atoms collide. Since the number of potential collisions increases as the square of the number of atoms in the molecule, it becomes impractical to calculate the

distances (or even the squared distances) between each pair of atoms in a respectable sized protein. Rather, the global XYZ space is divided into subspaces (cubes, for obvious practical reasons) and the atoms are grouped according to their containment by each cube. It is then necessary to calculate distances between each atom and only those atoms in the same and adjacent cubes; no atom can collide with another atom in a remote cube. This process relies heavily on list processing techniques to perform the regrouping.

SOLVE() calculates the changes in bond angles necessary to effect a set of specified changes in interatomic distances. Parameters in common storage dictate whether backbone or residue angles are to be varied, and which of three possible schemes is to be used in performing the calculations. Solve refers to Q1,Q2, and D for the list of changes to be made, and ultimately changes either ANG or ANGP (although ANGP is changed through a call to JIGL).

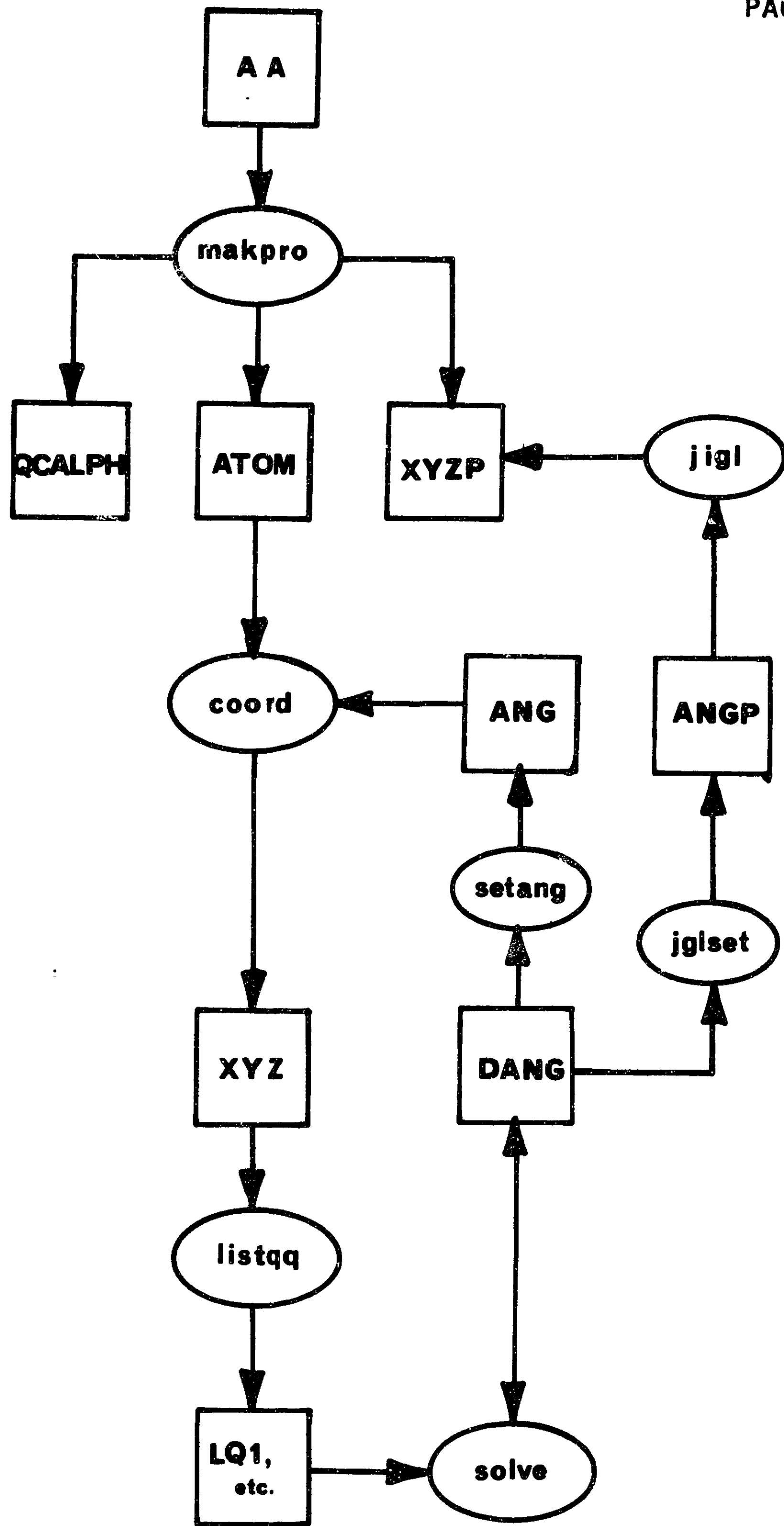

figure 2: Flow diagram of main processor programs and data

7.2.1 MAKPRO

The primary function of MAKPRO is the creation, from the amino acid sequence of a protein, a data structure corresponding to some degenerate configuration (e.g., having zero bond angles) of that protein; it performs, in some sense, the initialization of the system for work on a particular molecule. The fact that we are generally interested in only a few different molecules coupled with the evident space requirements of this program suggest that MAKPRO might be external to the system, run as a separate job once for each protein, and the resulting data structures be read from secondary storage as part of the initialization procedure of the system proper. Such thinking has been influential in the architecture of the package, and in fact the functions of MAKPRO are divided between two programs, one of which is external to the user system. The external portion is designated PROTOS, and contains the lengthy data tables (the descriptions of the zero-angle coordinates of the atoms in each type of amino acid); the output from this program is written into a file in secondary storage. The resident portion, designated MOLE, reads the file containing PROTOS output and completes the data structure representation of the degenerate (zero angle) model.

The input to PROTOS is the linear sequence of amino acids which make up the protein; specifically, PROTOS reads the contents of the AA array. The output includes the residue coordinates in local frames (the contents of XP, YP, and ZP) corresponding to residue bond angles of zero, as well as the AA array which was supplied as input and the QCALPH pointers which are a byproduct of the XP,YP,ZP generation. PROTOS makes use of a table containing the local X,Y, and Z coordinates of each om in each residue; the atoms in each residue are ordered by convention and numbered, as are the types of amino acids. An additional array, LIST, contains pointers to the first entry in the table for each sucessive residue type, and an array NRES specifies the number of atoms in each type. The coordinates of the Nth atom of a type M residue are then stored beginning at the entry indexed by LIST(M)+3*N. An alternative to the present scheme of table addressing might utilize a multiply subscripted array; our choice probably reflects historical rather than technical considerations. The operation of PROTOS is outlined below:

1) The zeroeth, first, and second elements of the XP,YP, and ZP arrays are set to the coordinates of:

    a) The hydrogen atom attaching to the carbon alpha.

    b) The carbon beta atom, which is the first residue atom in all but one type of amino acid.

c) A hydrogen atom which replaces the carbon beta in the remaining amino acid type (glycine).

These three XP,YP, and ZP slots serve the special purpose of specifying those atoms which are necessarily fixed with respect to the backbone (as a consequence of their separation from it by a single bond). They are treated as special cases to avoid storing redundantly the sets of identical local coordinates; see the discussion of COORD for notes on their application.

2) The following program variables are initialized:

An integer AA.NUMBER, specifying the current amino acid, is set to 1 for the first acid.

An integer P.POINTER, designating the next XP,YP, and ZP elements to be filled, is set to 3 (remember that the zeroeth thru second elements were filled above).

The first element of QCALPH is set to 3, as the first carbon alpha atom is always the third atom in the molecule.

3) The type of the current amino acid (that specified by AA.NUMBER) is obtained from the AA array, and an error exit is taken if the type is not between 1 and 21.

4) The number of atoms in the current residue is obtained from the element of NRES indexed by the amino acid type; the QCALPH entry for the <u>next</u> amino acid (that indexed by AA.NUMBER+1) is then set to the sum of this number and QCALPH(AA.NUMBER)+7, i.e., the next carbon alpha atom is indexed by the index of the last carbon alpha atom plus 7 (the number of backbone atoms per amino acid) plus the number of residue atoms in the current acid.

5) The table index of the coordinates corresponding to this residue type is found by LIST(AA.NUMBER); N (the number of residue atoms) coordinate triplets are copied into the N successive locations of XP, YP, and ZP specified by P.POINTER, which is incremented after each entry.

6) If the residue specified by AA.NUMBER is not the last, control returns to step (3) above. Otherwise, an integer BIGESQ (specifying the index number of the last atom in the molecule) is set to QCALPH(AA.NUMBER+1)-3.

7) A 'PROTOS' file is written onto secondary storage, consisting of:

   a) The contents of the XP, YP, and ZP arrays;

   b) The contents of the QCALPH array;

   c) The integer BIGESQ; and

   d) The AA array.

Thus, with the AA array as input, PROTOS has filled the QCALPH and XP,YP,ZP arrays.

Before proceding with our discussion of the remaining MAKPRO functions, a discussion of the details of the ATOM array is in order. ATOM is a packed array with one element corresponding to each atom in the molecule being modelled, each element having the following components:

QAA(ATOM(Q)) specifies the sequence number of the amino acid containing the atom Q.

TYPE(ATOM(Q)) is an integer between 1 and 9 encoding the type of the atom Q, e.g., hydrogen, carbon, oxygen.

ANGLO(ATOM(Q)) is the highest-indexed angle which effects the position of atom Q with respect to the low-numbered end of the molecule; ANGLO indexes the WHICH array, rather than the ANG array directly. Thus, varying ANG(WHICH(ANGLO(ATOM(Q)))) will change the relative positions of atoms Q and 1, whereas varying ANG(WHICH(ANGLO(ATOM(Q))+1) won't.

JGLNUM(ATOM(Q)), applicable only to residue atoms, specifies the highest relative sequence number corresponding to a bond angle in the residue containing Q which effects the position of Q. JGLNUM serves a function in each residue parallel to that served by ANGLO in the backbone. The local angle numbering system assigns the index 1 to the carbon alpha to carbon beta bond, with angle numbers increasing along the side chain; the numbers of the variable angles in a residue, however, are not necessarily consecutive. In order to take advantage of certain chemical similarities between residue types, the numbering system for rotatable bonds skips sequential integers in some cases. The routine RELJGN(TYPE,I) translates such a number I of the Nth variable bond angle of residue type TYPE, returning the integer N.

QBACK(ATOM(Q)) designates the highest numbered atom which will be tested against atom Q for van der Waals violations. QBACK(ATOM(Q)) contains the difference between Q and this highest numbered atom; the atoms between Q and (Q-QBACK(ATOM(Q))) are either constrained not to violate Q by the mechanics of the model, or such violations will be detected by other means (e.g., angle violations).

MOLE, the remaining portion of the MAKPRO system, sets up the arrays ANGPTR, ATOM, WHICH, ANGP, and ANG. Its operation is outlined below:

1) The 'PROTOS' file is read into common storage.

2) Each element of the ANG and ANGP arrays is set to zero.

3) The following variables are initialized:

   W.POINTER, which will point to the next element of the WHICH array, is set to 1.
   ANG.1, which will point to the ANG array at the first variable backbone angle in each amino acid (the bond between the nitrogen and carbon alpha) is set to 0.
   ANG.0, which will point to the second variable angle (carbon alpha to carbon bond) of the amino acid preceding the current one, is set to 0.
   AA.NUMBER, specifying the current amino acid, is set to 1.
   QB, which will be used to calculate the Q.BACK entries for the atoms in each amino acid, is set to 0 to point to the "zeroeth" atom.

4) WHICH(W.POINTER) is set to 2*AA.NUMBER, and W.POINTER is incremented by 1. This designates the <u>second</u> angle in the current amino acid, corresponding to the bond connecting the carbon alpha with the next carbon, as a variable angle. Note that the first variable angle ANG(1) of the first amino acid is never entered in the WHICH array; rotations about this bond do not alter interatomic distances and hence don't interest us. The variable ANG2 is set to W.POINTER to record the WHICH index corresponding to this second variable angle.

5) The routine ATTRIB is called to determine the number of variable angles in the current residue; the sum of this number and ANGPTR(AA.NUMBER) are placed in ANGPTR(AA.NUMBER+1) to allow space in the ANGP array for the variable residue angles.

6) For each of the atoms between QCALPH(AA.NUMBER)-2 and QCALPH(AA.NUMBER+1)-3 the QBACK, JGLNUM, ANGLO, QAA, and TYPE components of the packed ATOM array are filled as follows:

   a) The QAA component of each of these ATOM elements is set to AA.NUMBER, specifying the current amino acid.

   b) The types of the six backbone atoms are set explicitly: QCALPH(AA.NUMBER)-2 thru QCALPH(AA.NUMBER)+1 index the nitrogen, hydrogen, carbon alpha, and hydrogen, respectively, while QCALPH(AA.NUMBER+1)-4 and QCALPH(AA.NUMBER+1)-3 index the other carbon and the oxygen.

   c) The types of the residue atoms are found thru calls to ATTRIB, and the TYPE components of the atoms from QCALPH(AA.NUMBER)+2 thru QCALPH(AA.NUMBER+1)-5 are set accordingly.

   d) The QBACK component of each atom in the amino acid (both backbone and residue) is set to designate the atom pointed to by QB; specifically, each QBACK(ATOM(Q)) is set to Q-QB.

   e) For the first three backbone atoms, from QCALPH(AA.NUMBER)-2 to QCALPH(AA.NUMBER), the ANGLO component of the corresponding ATOM entry is set to ANG.0; this specifies that the coordinates of these atoms are effected by the second variable angle of the preceding amino acid, i.e., the carbon alpha to carbon bond, but not by the angle of the nitrogen to carbon alpha bond in the current acid.

   f) The ANGLO component for the oxygen atom, indexed by QCALPH(AA.NUMBER+1)-3, is set to ANG.2; this designates that the second variable bond angle (between carbon and oxygen) of the current amino acid is the last to effect the position of the oxygen.

   g) The ANGLO components of the ATOM entries for the remaining atoms in the residue, indexed by QCALPH(AA.NUMBER)+1 thru QCALPH(AA.NUMBER+1)-4, are set to ANG.1. This designates that their coordinates are effected by changes in the nitrogen to carbon alpha bond angle but not by the angle between the carbon alpha and carbon.

   h) The JGLNUM component of each of the residue atoms is determined by a call to ATTRIB; the JGLNUM components of the remaining atoms is left zero.

7) The pointer QB, which will specify the QBACK components for the next amino acid, is set to QCALPH(AA.NUMBER)-3; this specifies that pair checking will not occur between the atoms in the next amino acid (AA.NUMBER+1) and any atoms in the current amino acid (which start at QCALPH(AA.NUMBER)-2).

8) The type of the next amino acid is read from AA(AA.NUMBER+1), and if its nitrogen to carbon alpha bond is rotatable (one amino acid type has a fixed bond angle here) then:

   a) WHICH(W.POINTER) is set to the index of this angle (AA.NUMBER*2+1) and W.POINTER is incremented; then

   b) ANG.1 is set to W.POINTER to designate this first variable angle of the next amino acid.

   If, rather, the next amino acid has a fixed angle at this point, then:

   a) That angle, ANG(2*AA.NUMBER+1), is set to its fixed value; and

   b) ANG.1 is set to the current value of ANG.0, which points to the second variable angle of the last amino acid; this reflects the fact that any atom whose position is effected by ANG.1 is also effected by ANG.0.

9) ANG.0 is set to the current value of ANG.2, so that on the next iteration it will designate the second variable bond in the present amino acid.

10) If AA.NUMBER designates the last amino acid (indicated by the zeroeth element of the AA array, which contains the number of acids in the molecule) then the zeroeth element of the WHICH array is set to W.POINTER-1 to indicate the number of variable angles, and control returns to the calling program. Otherwise, AA.NUMBER is incremented, and control goes to step (4) above.

Note that the overall MAKPRO function, comprising PROTOS and MOLE, takes as its input the AA array, and produces as output the XP,YP, and ZP arrays, WHICH, QCALPH, ANGPTR, and ATOM; the degenerate values (zero, with the exceptional case noted above) it places in the ANG and ANGP arrays specify completely an instance of the protein molecule. The actual global-frame atomic coordinates, which are, at this level, dependant variables, may now be obtained by a call to the subprogram COORD.

7.2.2 COORD

The main algorithm of COORD proceeds sequentially down the backbone, adding atoms while translating them from a local frame to the global one. This procedure utilizes a rotation matrix ROT and a translational XYZ triplet to mechanize the switching from local to global coordinate frames, and these parameters are updated periodically to compensate for the successive local systems. For the purpose of adding backbone atoms, the matrix ROT is maintained so that the bonds lie perpendicular to the Z-axis; at each time when a bond between backbone atoms violates this condition, ROT is adjusted. An internal subroutine, SIDE(X,Y) is used to position atoms within the current XY plane. A second internal subroutine, PUTRES(), copies residue atoms from XP,YP,ZP into X,Y,Z after rotating them by ROT and translating them to the frame of the current carbon alpha. The general algorithm is outlined below:

1) By way of initialization, the following variables are set:

   The matrix ROT is set to unity.
   An integer BASE.ATOM, specifying the sequence number of the atom whose coordinates specify the translational reference, is set to 0. Since the coordinates of the first atom in the molecule are stored in X(1), Y(1), and Z(1), the zeroeth elements of these arrays will specify an initial translation of the entire molecule.
   XYZP.NUMBER, an integer denoting the next element of XP,YP,ZP to be placed into the X,Y,Z arrays, is set to point to the first XP,YP,ZP entry; this involves setting XYZP.NUMBER to 3 rather than 1, to allow for the special entries in XP, YP, and ZP mentioned below.
   The integer ACID.NUMBER is set to 1, denoting the first amino acid in the sequence.
   The integer ATOM.NUMBER, which will be used to indicate the next X,Y,Z elements to be filled, is set to 1.
   The integer ANGLE.NUMBER, which will point to the next backbone angle (ANG element) to be used, is set to 1.

2) The coordinates of the nitrogen, hydrogen, and carbon alpha are added to X,Y,Z by calls to SIDE. No manipulation of the matrix ROT is necessary since the carbon and oxygen of the previous amino acid lie in the same plane as these three atoms. The bond between the nitrogen and the carbon alpha is variable, but account of this rotation need not be taken until after the carbon alpha is treated, since rotation about a bond obviously does not change the relative position

of the atoms connected by that bond.

3) The matrix ROT is updated by two increments:

a) A rotation R1 (in the XY plane) which aligns the nitrogen-carbon alpha bond with the X-axis. The matrix representation of this rotation component is:

COS R1	-SIN R1	0
SIN R1	COS R1	0
0	0	1

b) A rotation in the YZ plane corresponding to the rotation specified about the variable N-CA bond. This matrix has the form:

1	0	0
0	COS Ø1	-SIN Ø1
0	SIN Ø1	COS Ø1

The total increment is thus specified by the product of these matrices, or:

COS R1	-SIN R1 COS Ø1	SIN Ø1 SIN R1
SIN R1	COS R1 COS Ø1	-SIN Ø1 SIN R1
0	SIN Ø1	COS Ø1

where Ø1 is the angle value in the ANG element pointed to by ANGLE.NUMBER. The previous matrix ROT is replaced by the product of ROT and this increment. The pointer ANGLE.NUMBER is incremented by 1.

4) The residue is attached in two steps:

a) The two residue atoms which are fixed with respect to the carbon alpha (by virtue of their separation from it by a single bond) are handled as special cases. Since they are not in the XY plane currently specified by ROT, it is not convenient to treat them as backbone atoms (i.e., to attach them by calls to SIDE). Since their position (in the local frame whose origin is at the carbon alpha) is constant from one residue to the next, it does not make sense to store their coordinates once for each acid. On the other hand,the routine PUTRES, which addresses the XP, YP, and ZP arrays directly, contains the mechanism necessary to attach these atoms, since it locates atoms in 3 dimensions in the local frame specified by the carbon alpha coordinates and the matrix ROT. The solution arrived at involves the reservation of the first three XP, YP, and ZP elements to store, during the MAKPRO operation,

the local coordinates of the three such atoms; then, at the proper point in the COORD algorithm, they may be attached by temporarily setting XYZP.NUMBER to 1, 2, or 3 and calling PUTRES. The first such atom, stored in XP,YP,ZP(0), is a lone hydrogen which always attaches to the carbon alpha. The second, which is really the first atom in the side chain (the rest of the residue attaches to it) is usually another carbon (designated the _carbon beta_); in the case of one amino acid type, it is a hydrogen. The carbon beta coordinates are stored in XP,YP,ZP(1), while the coordinates of the alternative hydrogen are in XP,YP,ZP(2). Now, the COORD algorithm does the following:

1: The current value of XYZP.NUMBER is saved in a temporary location. XYZP.NUMBER is set to 0, and PUTRES is called; this attaches the hydrogen.

2: The type of the current amino acid AA(ACID.NUMBER) is examined; if it is glycine, XYZP.NUMBER is set to 2 (to attach the second hydrogen); otherwise, it is set to 1 to attach the carbon beta. PUTRES is called to attach this second special atom.

3: The previous (saved) value of XYZP.NUMBER is restored.

b) The remainder of the residue is added by repeated calls to PUTRES, each call attaching one residue atom and incrementing XYZP.NUMBER as well as ATOM.NUMBER. PUTRES bears the responsibility of translating the coordinates of each atom by the coordinates of the atom specified by the pointer BASE.ATOM, in addition to the rotation by the matrix ROT from the local frame of XP,YP,ZP to the global one of X,Y,Z. The number of atoms in each residue, and hence the number of calls to PUTRES necessary to attach it, is available in a 21-element array NRES.

5) The matrix ROT is again updated by two increments:

a) A rotation R2 in the XY plane, which aligns the Carbon alpha-carbon bond with the X-axis. This matrix has the same form as the above matrix specifying the rotation R1 in the XY plane.

b)A rotation Ø2 in the YZ plane, corresponding to the rotation about the variable CA-C bond.

The product of these matrices becomes:

```
COS R2      -SIN R2 COS Ø2    SIN Ø2 SIN R2
SIN R2       COS R2 COS Ø2   -SIN Ø2 COS R2
0              SIN Ø2          COS Ø2
```

where Ø2 is the value in the ANG element pointed to by ANGLE.NUMBER, and the product of this matrix and ROT replaces ROT. ANGLE.NUMBER is incremented by 1.

6) The carbon and its attached oxygen atoms are added to X,Y,Z by calls to SIDE. ACID.NUMBER is incremented; if any amino acids remain to be attached, control returns to step 2) above, otherwise control is returned to the calling program.

The procedure SIDE(XC,YC), which is used to attach backbone atoms in the XY plane of the current carbon alpha, operates as follows:

1) The vector (X, Y, 0) is multiplied by ROT and the product is added to the coordinates of the atom pointed to by BASE.ATOM; i.e., the specified 3-element vector (whose Z-component is zero) is mapped into the frame of the current carbon alpha.

2) This result is added to the X,Y,Z arrays in the element pointed to by ATOM.NUMBER. ATOM.NUMBER is incremented, and control returns.

The function PUTRES(), which copies atoms from XP,YP, and ZP to X,Y,Z executes a similar algorithm:

1) The XP,YP,ZP triplet pointed to by XYZP.NUMBER is multiplied by ROT and the product is added to the X,Y,Z elements indicated by BASE.ATOM.

2) This result is placed in X,Y,Z in the position specified by ATOM.NUMBER. ATOM.NUMBER and XYZP.NUMBER are incremented, and control returns.

7.2.3 JIGL

The subprogram JIGL is used to make changes in atomic coordinates corresponding to specified changes in residue bond angles. These changes are made to the coordinate representations in both the local frame of XP, YP, and ZP and the global X, Y, Z frame; most of the calculations of JIGL involve rotating and translating vectors between the global frame of X, Y, and Z (which we shall refer to as the XYZ frame), the frame of XP, YP, and ZP (designated the XYZP frame) and a frame local to the bond being adjusted (the 'local' frame). The following mathmatical identities are widely used by JIGL and merit a quick review:

1) Given a frame of reference F1 and three mutually perpendicular unit vectors VX, VY, VZ in F1 designating the axes of a new frame F2, a rotation matrix R12 may be constructed as follows:

```
VX(1)     VX(2)     VX(3)
VY(1)     VY(2)     VY(3)
VZ(1)     VZ(2)     VZ(3)
```

which has the property that the product of R12 and an arbitrary vector V1 in F1 yields the corresponding vector V2 in F2.

2) Similarly, the matrix R21:

```
VX(1)     VY(1)     VZ(1)
VX(2)     VY(2)     VZ(2)
VX(3)     VY(3)     VZ(3)
```

has the property that the product of R21 and V2 (in F2) yields V1 (in F1). Thus, we note that the inverse of a rotation matrix may be obtained by transposing that matrix along its main diagonal.

The ultimate objective of the JIGL algorithm is simple: the coordinates of the residue atoms are translated and rotated to the local frame from the XYZ frame, the rotation is performed about the indicated bond, and the result is mapped back into both the XYZ and the XYZP frames. The program, which is largely devoted to initializing the matrices necessary for these mappings to and from the local frame, is outlined below:

1) By referring to two tables internal to JIGL, the two atoms adjacent to the bond being adjusted are singled out. The integer variables QA and QB are set to the sequence numbers of these atoms (i.e., their indices in the X, Y, Z arrays); in addition, QC is set to designate the next higher numbered residue atom (QB+1).

2) The indices of the same three atoms in the XP, YP, and ZP arrays are determined; they are designated QAP, QBP, and QCP.

3) The vector TRA1 is set to the coordinates (in XYZ) of atom QB; similarly, TRA2 is set to the coordinates (in XYZP) of QBP. The significance of these coordinates is that the atom designated as QB (in the XYZ frame) and QBP (in the XYZP frame) will be located at the origin of the local frame; thus the vectors TRA1 and TRA2 specify the relative translations between the XYZ and XYZP frames and the local frames.

4) The vectors VX, VY, and VZ are set to designate, in the XYZ frame, the axes of the local frame. They are arranged so that the bond being adjusted lies parallel to the local X axis (or parallel to VX in XYZ) and that the next bond (that connecting QB with QC) lies in the XY plane (or perpendicular to VZ). This is done as follows:

   a) VX is set to the vector connecting QB with QA, normalized so that its magnitude is 1:

$$VX=(QB-QA)/|QB-QA|$$

   b) VZ is set to the cross product between VX and the vector connecting atoms QB and QC; VZ is then normalized. Thus VZ is perpendicular to the plane containing the QA-QB bond and the QB-QC bond:

$$VZ=(VX \times (QB-QC))/|VX \times (QB-QC)|$$

   c) VY is set to the cross product between VZ and VX; thus the axes are mutually perpendicular:

$$VY=VZ \times VX$$

5) The rotation matrix R1 is set (by the elements of VX, VY, and VZ) to rotate from the XYZ frame to the local frame:

6) In a manner identical to that outlined above, the vectors VX, VY, and VZ are now set to specify the axes of the local frame relative to the XYZP frame. The procedure of step (4) above is followed, using the atoms QAP, QBP, and QCP to relate the axes to XYZP; the matrix R3, however, is adjusted from these vectors to rotate to the XYZP frame from the local frame.

7) The matrix R4 is set up to represent the requested rotation, in the local frame, about the designated bond. Since that bond is parallel to the local X axis, the rotation is in the YZ plane; the matrix becomes:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & COS(\emptyset) & -SIN(\emptyset) \\ 0 & SIN(\emptyset) & COS(\emptyset) \end{matrix}$$

where Ø is the rotation to be applied about the designated bond.

8) The matrix R6 is set to the transposed matrix R1. Thus R6 now rotates from the local frame to the XYZ frame:

R6=R1'

9) The matrix R5 is set to the product of R1 and R4. Thus R5 now incorporates two rotations: that between XYZ and the local frame, and that between the old local frame and the new one resulting from the rotation of the old about the designated bond:

R5=R1*R4

10) Now the actual coordinates are modified. For each atom in the residue the following steps ensue:

a) The X, Y, Z elements corresponding to the atom are translated by the negative of the coordinates in TRA1; the coordinates of the atom QB, for example, become zero (since this atom is at the origin of the local frame).

b) The resulting XYZ coordinates are rotated by R5. The resulting values are, then, the local frame coordinates of the atom after having undergone rotation about the designated bond.

c) The XYZP frame representation of the coordinates is obtained by rotating by R3 and translating by TRA2; the resulting values are placed into the appropriate XP, YP, and ZP entries.

d) Finally, the new X, Y, and Z values are obtained by rotating the local frame coordinates by R6 and translating by TRA1.

XYZP(I)=((XYZ(I)-TRA1)*R5)*R3+TRA2

```
XYZ(I)=((XYZ(I)-TRA1)*R5)*R6+TRA1
```

7.2.4 SOLVE

The subroutine SOLVE which is called in to perform calculations necessary for the manipulation of inter-atomic distances actually represents the entry point to a system of subprograms which selects one of several possible algorithms pursuant to these manipulations and calls the appropriate lower-level functions for its execution. Since the completely analytic solution of the system of equations relating bond angle values to even a small number of interatomic distances is impractically complex and time consuming, the SOLVE system necessarily resorts to heuristic measures. The simplest and most obvious of these involves a "linear approximation" in which it is assumed that changes in the bond angles resulting from any calculation step will be small; this allows us to approximate trigonometric relations by linear ones and to solve the equations by the traditional methods of linear algebra. The limitation on angle changes imposes the requirement that we effect gross structural changes by a series of more modest increments, but in fact the resulting quasi-continuous deformation of the model probably lends itself to our model building requirements particularly well; our treatment of disallowed configurations of the model, for example, suggests gradual hill climbing techniques rather than the intractable alternative of analytic solution.

SOLVE chooses, on the basis of a user-supplied parameter, one of three algorithms for calculating the bond angle changes necessary to implement a given set of interatomic distance changes:

1) The method of "steepest descent" in which the increment applied to each bond angle is the sum of the partial derivatives of distance between each atom pair with respect to that angle, weighted by the change required in the distance between that pair. This method assumes, in addition to the linear approximation mentioned above, that calculations regarding each atom pair may be performed independently. It thus avoids the time consuming operations of matrix inversion and multiplication entailed in the other two methods, and in a large number of situations it seems to perform adequately.

2) The solution of the system of linear equations relating differential changes in bond angles to differential changes in interatomic distances. The coefficients in these equations each represent the partial derivative of an atomic pair distance with respect to an angle.

3) The solution of equations similar to the above, except relating each <u>component</u> of the interatomic distances

to each angle. Since this method results in three times as many equations (matrix rows) as above, it bases the angle changes on more information about the structure, and is useful for resolving deadlocks in which the previous methods may find themselves. However, it results in a large system of equations which present substantial time and storage demands to the system.

In addition, the user may specify whether he wants the changes effected by modifying backbone or residue angles; no means is available for varying both simultaneously.

Each of the above algorithms makes use of partial derivatives of interatomic distances with respect to angle changes, and these derivatives are computed by subroutines whose arguments include the atom pair in question and an angle number, and return the differential change in relative position of the two atoms as a result of a differential change in the bond angle. For cases 1 and 2 above, the result is a scaler distance, whereas in the third case a vector is returned. The derivatives involving changes in backbone angles are computed by the routines DERIV and DERIV3 (depending on whether method 2 or 3, above, is being used) which function as follows:

1) The unit vector V1, parallel to the axis of rotation, is computed by scaling the vector connecting the two backbone atoms on either side of the bond in question.

2) The vector V2 between one of the atom pair and an atom connected by the rotating bond is computed. We may consider that the remaining atom in the pair is fixed to the lab frame, and thus the change in the position of the atom chosen here is the change in the relative positions of the two atoms.

3) The amount by which the above atom moves as a result of a differential angle change is then computed by taking the cross product of V1 and V2. In the case of DERIV3, the resulting 3-element vector is returned; DERIV returns the dot product of this vector with a unit vector parallel to the line connecting the two atoms in the pair, representing the differential change in distance.

The routines RDRV and RDRV3 perform the same function for rotation about residue bonds, calling the routine JGLABL to insure that the angle in question does in fact effect the distance between the atom pair. JGLABL incorporates structural information about the various residue types, and specifies whether a given angle in a particular type of residue will effect the position of a specific atom relative to the backbone. Thus, if one of the pair of atoms is fixed relative to the backbone while the other is not, then the

distance between the pair is a function of the angle in question.

Since the user may specify that either residue angles or backbone angles are to be varied, nearly all of the routines in the SOLVE subsystem are duplicated for each of these cases. There are several basic differences between the residue angle calculations and the backbone angle calculations:

1) In order for a residue angle to effect the distance between a pair of atoms, at least one of these atoms must belong to the residue containing that angle. Thus, each pair in LQ1 and LQ2 is effected by the angles of at most 2 residues.

2) There are a large number of potentially variable residue angles, and hence we are reluctant to provide a second pair of DANG and WHICH arrays with which to manipulate the residue angles. Also, since at any given time we are varying <u>either</u> residue angles or backbone angles, we don't ever need the DANG array while we are solving for residue angle changes.

As a consequence of (1), there are, in general, relatively few residue angles relevent to a set of LQ1, LQ2, LDIST specifications. We take advantage of this situation by dividing up the DANG array, during residue angle SOLVE operations, into 3 parts: one which specifies the angle changes required (the usual function of DANG) and the other two to indicate the angle being varied. The names used two refer to these latter 2 sub-arrays are CODE and KEY; the name DANG is used to refer to the first.

This organization requires the KEY and CODE arrays to be initialized, before any residue SOLVE operation, to indicate the relevent variable residue angles; subsequent programs may then use them in a manner similar to the way the WHICH array is used for backbone closes. This initialization operation is performed by the subroutine SETKEY, which operations as follows:

1) The zeroeth element of the KEY array is set to 0. This element is used to indicate the number of entries in the KEY, CODE system; each time a new angle is entered, KEY will be indexed. It is noteworthy that each angle is represented at most once: at each point where a new KEY, CODE entry is to be made, the entire array is first searched to insure that an identical entry has not already been made.

2) An integer variable LQ.NUMBER, which will be used to specify the LQ1, LQ2 entries currently being considered (and hence the pair of atoms for which we

are finding relevent angles) is set to 1.

3) Since, as we have noted, the residue angles effecting the distance between atoms Q1 and Q2 belong to one of the two amino acids containing these atoms, we perform the following steps once for each (LQ=LQ1, LQ2):

   a) The number of the atom LQ relative to the carbon alpha of its containing residue is calculated by RQ=LQ-QCALPH(LQ).

   b) The JGLNUM component of the ATOM entry corresponding to atom LQ is extracted. This number indicates the highest numbered residue angle shich effects the position of LQ with respect to the backbone, and hence the highest numbered angle effecting the distance between this pair of atoms. Then, for each integer I between 1 and this JGLNUM component:

      1: The subroutine JGLABL is called to determine whether angle I effects the position of LQ;

      2: If JGLABL returns boolean true, then the number of the amino acid containing LQ is put into CODE and the number RQ is put into KEY.

4) If LQ.NUMBER points to the last LQ1, LQ2 entries, control is returned to the calling program. Otherwise, LQ.NUMBER is incremented and control passes to step (3) above.

The method of steepest descent involves a minimum of arithmetic manipulation, and will be treated first. The routine SOLVE serves simply to call, in succession, two relevant subprograms; the first derives the angle changes required by manipulation of the LQ1, LQ2, and LDIST arrays, while the second actually changes the angles. If backbone angles are to be varied, these two subprograms are STEPES and SETANG, respectively; if residue angles are beign varied then their counterpart JGLSTP and JGLSET are called.

The routine STEPES refers to the LQ1, LQ2, and LDIST arrays and fills the DANG array. STEPES functions as follows:

1) The DANG array is set to zero.

2) For each I between 1 and LQ1(0) (used to indicate the number of LQ1, LQ2, and LDIST entries) the following occurs:

a) For each J between 1 and WHICH(0) (the number of entries in the WHICH and DANG arrays), DANG(I) is set to DANG(I)+LDIST(I)*DERIV(WHICH(J),I).

Thus, each DANG(I) is set to the sum of the derivative of the distances between each atom pair and the distance by which that pair is to be moved.

3) The magnitude of the DANG array is determined by a call to NMAG; each element is scaled by a factor of STEP/MAG where MAG is the vector's magnitude and STEP is a user supplied step size.

The residue-angle varying counterpart, JGLSTP, performs in an identical manner except for the following differences:

1) The routine SETKEY is called before LQ1 and LQ2 are examined; this sets up KEY and CODE.

2) RDRV is called to calculate the partial derivatives.

The operation is completed by a call to the second function (JGLSET or SETANG) which makes the actual changes in the angles. JGLSET simply calls the routine JIGL (described elsewhere) once to change each angle in the KEY, CODE, and DANG arrays. The routine SETANG, used to vary backbone angles, performs an auxiliary function, and is described in a later paragraph.

In each of the other two SOLVE algorithms, a pair of subroutines is called by SOLVE to perform functions approximately analogous to those of STEPES and SETANG. The first such function bears the responsibility for compiling the matrix of partial derivatives (of interatomic distances with respect to variable angles) and the vector representing the necessary changes in interatomic distances. The output of this program, then, is a set of simultaneous equations which relate differential changes in interatomic distances to corresponding differential changes in bond angles. The remainder of the task, which is carried out in the routine SOLVE itself, is the solution of these equations (in which the angle increments are the independent variables) for the angle changes in terms of the distance changes. This process involves, basically, the inversion of the matrix of partial derivatives; the product of this inverted matrix and the vector of distance changes (set up by the first subprogram) results in the necessary angle changes (the contents of the DANG array). The second function may then be called to make the actual angle changes; this second function is the same as that used in the method of steepest descent, and mentioned above: SETANG for backbone angles, JGLSET for residue angles.

The first function mentioned above (that which sets up the matrix and array of independent variables) is, in the case of backbone angle manipulations, either of the routines SETONE or SETTHR depending upon whether the 1-dimensional or the 3-dimensional calculations are being made; for residue angles the corresponding programs are JGLONE and JGLTHR.

The operation of the routine SETONE, which initializes the matrix of partial derivatives of distance with respect to backbone angles, is outlined below:

1) For each LQ1, LQ2, and LDIST entry (remember that the number of such entries is signified by the zeroeth LQ1 entry) the following steps are performed:
   a) QA and QB are set to the LQ1 and LQ2 entries being examined on the current iteration. ANGTO is set to the maximum of ANGTO and ANGLO(ATOM(QB)), or the highest numbered angle which will effect the position of the atom QB with respect to lower-numbered atoms.

   b) For the atom QB, necessarily the lower numbered atom of the pair (the constraint is imposed upon programs which make LQ1, LQ2 entries that the low sequence number be entered in LQ1) we need the reverse of the ANGLO parameter; that is, we must know the lowest numbered backbone angle which effects the position of QA relative to the higher numbered atoms. We calculate this index as follows:

      1: If QA is the carbon alpha (i.e., if QA=QCALPH(QAA(ATOM(QA)))) then the index is ANGLO(ATOM(QA))+3.

      2: If QA is the nitrogen or the other backbone carbon, then the index is ANGLO(ATOM(QA))+2.

      3: Otherwise, the index is ANGLO(ATOM(QA)))+1.

   The variable ANGFRO is set to the minimum between ANGFRO and this index.

   This iteration has, then, established the range of backbone angles which relate to the set of LQ1, LQ2, and LDIST entries. ANGTO-ANGFRO is the number ofvariables in the equations and hence the number of columns in the matrix to be constructed; the number of rows is the number of LQ1 entries, or LQ1(0).

2) The actual matrix of partial derivatives, A, is constructed. The element in the Ith row, Jth column, is set to DERIV(WHICH(I),J) or the partial derivative of the distance between the Jth LQ1, LQ2 atom pair with respect to the WHICH(I)th element of the ANG

array.

3) The vector B, containing the values of the independent variables, is set up so that B(I)=LDIST(I).

4) Control returns to the calling program.

The program SETTHR, which creates a matrix with three times as many rows (and, of course, a vector with three times as many values of independent variables) operates in an identical manner except that:

1) The three matrix elements A(3*I+1,J), A(3*I+2,J) and A(3*I+3,J) are set to the three elements of the vector returned by DERIV3; and

2) The three vector elements B(3*I+1), B(3*I+2), and B(3*I+3) are set to LDIST(I) times the X, Y, and Z components of the unit vector pointing from atom LQ1(I) to atom LQ2(I). Thus B(3*I+1)=LDIST(I)*(X(LQ1(I))-X(LQ2(I)))/DIST(LQ1(I),LQ2(I)), etc.

The subroutine SETKEY, described in the preceding discussion of the steepest descent approach, is used to perform much of the corresponding bookkeeping in the case of varying residue angles. SETKEY initializes the KEY array, a WHICH type list of variable angles; the KEY array, unlike WHICH, contains however only the indices of those residue angels relating to the pairs in the current LQ1, LQ2 arrays. Thus, the first iterations of the SETONE and SETTHR programs are not necessary in their residue angle counterparts: the function of these iterations (namely, to find the range of relevent angles) is performed by SETKEY. The number of columns in the matrix becomes, then, the contents of the zeroeth eleent of the KEY array.

The remainder of the residue-angle matrix initialization routines is identical to their backbone-angle equivalents, with the obvious exceptions that RDRV and RDRV3 are called rather than DERIV and DERIV3, and that the arrays KEY and CODE are referred to rather than WHICH. These two subroutines are designated JGLONE and JGLTHR in their respective 1-component and 3-component instances.

The completion of the SOLVE operation involves finding solutions to the system of linear equations represented by the matrix A and the vector B; in particular, a vector X must be found such that AX=B. This solution vector, X, becomes the set of angle changes required to implement the set of changes in interatomic distances specified by the vector B. In general, if the matrix A is square (corresponding to an equal number of equations and unknowns) and invertable, straightforward methods of linear algebra

may be used to find the matrix C (the inverse of A) such that CAX=X=CB. More commonly, however, the matrix A is nonsquare, and its inverse is undefined. The two categories of this situation, representing more equations than unknowns in the first instance and more unknowns than equations in the second, are treated in the following manner:

1) If the matrix A has n columns and m rows, where m>n, then the equation AX=B has no consistent solution (assuming the rows of A are linearly independent). If, however, we relax our condition that the solution be exact, we can look for a reasonable least-squares approximation between AX and B. In particular, we seek an X (dimension n) such that the vector AX-B (dimension m) dotted into itself is at a minimum. But (AX-B) (AX-B) is at a minimum when its first derivative is zero, or when 2A'(AX-B)=0 where A' is the transpose of A (A(I,J)=A'(J,I) hence A' is dimensioned n by m). Thus A'AX=A'B, giving us a new matrix equation for the (approximate) matrix X; but, we note that A'A is a square, symmetric matrix having a well defined inverse. Hence A'AX=A'B may be solved for X, yielding X=CA'B where C is the inverse of the n by n matrix A'A. Two subprograms in our arithmetic library aid in this process: SYMAK, which calculates the symmetric matrix A'A from the matrix A, and SYMINV which finds the inverse of A'A.

2) If, on the other hand, the matrix A has more columns than rows, then there is an infinity of solutions to the equation AX=B. This situation appears less serious than the last, since we may here insist on an exact solution and still retain several degrees of freedom. Rather than selecting a solution at random, however, we find that it costs us little to find the solution which, in some sense, has the slightest effect on the structure of our model. To this end, we minimize the sum of the squares of the angle changes required by the soultion to our equation. Using the method of undetermined multipliers (due to Lagrange) we finda vector L (of dimension m, the number of rows of A) such that the quantity X X/2-L (AX-Y) is minimized. The derivative of this function with respect to the vector X is zero when X-LA'=0, where A' is again the transpose of the matrix A. Hence X=A'L and AA'L=B, from which we deduce that since AA' is a square, invertable, symmetric matrix of dimension m then L=CB where B is the inverse of AA'. But X=A'L=A'CB, whence our least-squares solution of the equation AX=B. SYMAK is again called upon to find the symmetric matrix AA', and SYMINV is used to invert it. Consequently, very little additional machinery must be introduced to handle this case.

We are now in a position to outline the details of the SOLVE program:

1) The parameter specifying which of the three permissable algorithms is to be used (named COMPO) is examined:

   a) If the method of steepest descent is to be used, the appropriate subprograms (described earlier) are called. Specifically, STEPES and SETANG are called for backbone angles, or JGLSTP and JGLSET are called for residue angles.

   b) If the 1-dimensional case of the matrix inversion method is to be used, SETONE or JGLONE is called (for backbone and residue angles, respectively).

   c) If the 3-dimensional matrix method is to be used, SETTHR or JGLTHR is called.

   In the steepest descent case, control returns to the caller (as the operation is completed). In the other cases, the matrix A of partial derivatives of interatomic distances with respect to bond angles has at this point been set; furthermore, the vector B of required distance changes has been set. The implied equation AX=B where X is the vector of angle changes (i.e., DANG) remains to be solved.

2) The dimensions m and n of the matrix A are compared:

   a) If m=n, the routine MTXINV is called to find the inverse of A; DANG is set (by the routine MATMUL) to the product of this inverse and the vector B:

   DANG=A B

   b) If m<n, the routine SYMAK is called to calculate AA'. SYMINV finds the matrix C which is the m by m inverse of AA', and MATMUL is called to find the product of C and B. Finally, the routine TRNPX (which finds the product of a vector and the transpose of a matrix) is called to set DANG to A'CB:

   DANG=A'((AA') B)

   c) If m>n, the routine SYMAK is used to find A'A, and SYMINV finds the n by n inverse of A'A, called C. Then TRNPX is used to find A'B, and finally MATMUL sets DANG to CA'B:

DANG=(A'A) (A'B)

3) Control returns to the calling program.

7.2.5 SETANG

From the standpoint of the calling program, the routines JIGL and SETANG perform the same function, namely, the variation of values assigned to variable bond angles in the molecule. The analog between SETANG for varying backbone angles and JIGL for avrying residue angles is, however, misleading; the motivation for each program bases itself on independent and unique technical requirements.

The organization of the package is such that the routine JIGL must be called to vary residue angles; if the user attempts to modify, for example, such an angle by simply changing an entry in the ANGP array, the rest of the system will remain unaware of the change. In particular, the atomic coordinates of atoms which should be moved by this alteration will be uneffected. The routine JIGL, on the other hand, contains the machinery required to modify the coordinates of effected atoms, and consequently will give the user the results he expects.

The programs which involve changes in backbone angles, however, are organized in a fashion basically differing from those relating to residue angles.Since the subprogram COORD calculates coordinates of atoms based on the current values of backbone angles (as represented in the elements of the ANG array) an explicit change in an ANG element will ultimately result in a corresponding alteration of the coordinates of the proper subset of atoms. Consequently, we would expect the SOLVE package, for example, to give us the proper results by chinging the ANG array directly.

There was, however, a separate motivation for isolating the rest of the system from direct access to the ANG array. It happens that large numbers of illegal interations between adjacent atoms can be avoided by placing constraints on the values that backbone angles are allowed to assume. The violations which we seek to avoid, of course, could be treated by the more laborious methods of LISTQQ; but our checking the angle values for violations allows us to substantially decrease the load on LISTQQ by stipulating that the atoms Q1 and Q2 need not be tested for interaction if they are sufficiently near one another in the topology of the molecule (see the discussion of QBACK in the section on MAKPRO).

The violations which we seek to prevent by these methods may be detected by examining each pair of angles (consisting of the two bond angles adjacent to the carbon alpha) for a disallowed combination. This test is implemented by the subroutine ANGVL, which makes use of a 72 by 72 matrix whose elements each comprise 1 binary bit; the allowed values of the angle pair is coded into these 1-bit elements. ANGVL, which has an argument specifying the type

of amino acid (since the configurations allowed vary with the acid type) and 2 arguments for the values of the indicated angles, performs a very fast table lookup and returns a boolean value specifying whether the values indicated are in violation.

Another routine subordinate to SETANG is CHANG, which is used to determine thefraction of a requested change in a pair of ANG values is consistent with an allowed angle configuration. CHANG, whose arguments comprise the pair of angles to be changed, the amount by which each is to be changed, and the type of the related amino acid, first determines whether the changes result in an allowed configuration. If they do, the function returns the value 1 (the fraction of the requested change allowed); otherwise, an exponential search is performed to find the maximum such fraction which still causes no angle violations; the following outlines this procedure, where DANG1 and DANG2 are the increments that the calling program would like to apply to the angles ANG1 and ANG2:

1) A test is performed to insure that the original ANG1 and ANG2 values are valid; an error return is taken if they are not. Obviously, an exponential search for the transition between allowed and disallowed states makes little sense if there is no such transition.

2) DANG1 and DANG2 are halved, and the resulting increments are applied to the angles (i.e., ANG1=ANG1+DANG1, etc).

3) If the new ANG1 ANG2 pair represents a disallowed configuration, the signs of DANG1 and DANG2 are changed (i.e., DANG1=-DANG1, etc).

3) Control returns to step (2) until the iteration has been executed a fixed number of times; note that this fixed number is exponentially related to the maximum difference between the resulting DANG1, DANG2 pair and the actual transition value between allowed and disallowed regions in the ANG1, ANG2 space. A ratio between the original DANG1 and the amount by which ANG1 was actually changed is returned as the value of CHANG. The original elements of the ANG array are not changed by CHANG: ANG1 and ANG2 are copies of these values.

We may envision the domain of the CHANG arguments as an ANG1 ANG2 plane, with general regions corresponding to valid configurations as well as other regions which are not allowed. The current ANG1, ANG2 values may be represented as a point (presumably in a valid region) on the plane, and the desired new values correspond to a second point; CHANG determines a point on the line connecting these extremes

which corresponds to a valid configuration. The restriction that CHANG only considers ANG1, ANG2 value pairs which are on this line results from one heuristic argument; our apparent ignorance of the possibility that the line intersect two disjoint valid regions reveals another. The first stems largely from the economics of performing a search in two dimensions, rather than one; the second is a compromise with our temptation to attack a classic problem of artificial intelligence. The fact that CHANG first attemts to make 100% of the requested change and the widely varying size of the steps which it takes permits it to skip over disallowed regions in some instances, so that there is at least some hope that it may "escape" from one allowed region into another disjoint one. This capability leads us to prefer the present CHANG algorithm to a continuous hill climbing approach, since in fact our actual map of allowed ANG1, ANG2 regions does contain several completely disjoint valid areas. The size of the step in ANG1 and ANG2 is, however, related in complex ways to the size of the resulting steps in the X,Y, and Z coordinates of atoms, hence it is unclear in any given SOLVE operation whether such an escape is feasible. A future iteration of the present package might consider these complications in more detail.

There are two major functions which SETANG performs. The first involves finding the maximum requested angle change (the element of the DANG array whose absolute value is the greatest) and comparing it with a fixed maximum. This is to check the assumption of SOLVE that angle changes resulting from any given step are small; this assumption is made, as mentioned in the discussion of SOLVE, to linearize the calculations. If the maximum requested DANG is, in fact, greater than our parameter allows, then a message is printed and all of the angle changes are scaled down by a constant factor such that the maximum DANG becomes the parametric maximum. The argument for scaling down the DANG elements relates again to our linear assumption, and the message is printed so that the user will know to decrease the step (in X, Y, Z) requested from the SOLVE system.

The second function of SETANG is to recover from requested angle changes which place any pair of backbone angles in violation of the angular constraints mentioned earlier in this section. The current SETANG algorithm moves each angle through as much of the requested change as is possible without causing angle violations. A previous algorithm, discarded in favor of the present one, found the angle which was the most restricting and then scaled all DANG elements by a constant factor calculated to render this one corrigible. The net result was that all DANGs were continually being scaled to nearly zero, and progress was consequently very slow. The empirical success of the present scheme suggests that the value of each angle change

is to a significant extent independent of each other: that if A and B are useful as a combination then each will probably be useful alone.

The present algorithm performs the further step of noting, in the WORK array, those angles whose variation was restricted; this causes them to be considered fixed by the SOLVE routines in their next iteration. This heuristic measure has several obvious nuances, some of which have been explored: the number of iterations for which restricting angles are held fixed may be varied or made a function of the degree of restriction, an algorithm may be executed which attempts to "jump" over the restricting disallowed region, etc.

7.2.6 LISTQQ

The problem of preventing pairs of atoms in the model from approaching each other within some critical radius is probably the most time consuming function of the computer system and consequently the one for which heuristic approaches have the most potential. The brute force checking of each atom against each other results in a total of N squared comparisons for N atoms, clearly an impractical procedure. The first obvious step of checking only in one direction (since the distance between Q1 and Q2 is the same as that between Q2 and Q1) gains us only a factor of 2.

The scheme we have adopted to speed up this pair checking procedure involves dividing the global X,Y,Z coordinate space up into cubes of a fixed size, and listing the subset of atoms contained in each cube. Then the coordinates of each atom Q need only be tested against other atoms the cube containing it, as well as those in immediately adjacent cubes. The fact that each atom is tested against those in a fixed number (specifically, 27) of cubes changes the basic N squared dependence of the computation time. The exact functional dependence depends upon the way in which the atoms distribute themselves among the cubes; the two extremes are:

1) The density of atoms might remain constant; i.e., the total space occupied by the molecule might be proportional to the number of atoms, the average number of atoms per cube remaining constant. In this case, since there is a fixed average number of comparisons per atom (27 times the average number of atoms per cube) then the total computation time should be very nearly proportional to N. Thus, in this limit, the LISTQQ function is _real time computable_.

2) The total volume occupied by the molecule might remain constant, the atomic density increasing as N. In this case, the number of comparisons would still increase as the square of N, the cubing scheme effecting only a constant factor.

In fact, we can choose our cube size such that the LISTQQ operation is nearly approximated by the first limit; the size is adjusted so that very few atoms are likely to be in one cube, and the number of empty cubes is high. The opposing consideration of the overhead involved in creating and accessing a cube with very few atoms in it curbs our temptation to make the cubes infinitesimal; there is an empirically derived optimum cube size (about 5 angstroms).

The variable number of cubes and the undetermined number of atoms in each cube makes a static allocation of cube storage impractical. The techniques of list

processing, however, allow us to substitute a fixed storage overhead of about 50% (taken by pointers) for the several orders of magnitude of inefficiency anticipated due to space allocated empty cubes, etc. Space for a particular cube is not required until an atom is placed into that cube, and the total space alloted need only accomodate the actual number of atoms. Since each cube is not indexed by a fixed address, a search must be performed to locate any specific cube; however, the list structure is arranged such that each coordinate of the cube may be located as a separate operation by searching along a separate linear list. The time spent searching, then, increases as the cube root of the total number of atoms, and in actual operation it appears negligible in comparison to the time spent testing atom pairs.

The actual list structure is handled by a package of subprograms, written in assembly language, called CUBE. The structure, a typical segment of which is diagrammed in figure 3, is composed of two types of list cells:

1) A cell occupying two 36-bit words, and with three components:

   IJK.VALUE, occupying the decrement portion of the first word, used to specify the I, J, or K coordinates corresponding to each level of list.

   NEXT, occupying the address portion of the first word, contains a pointer to the next cell on the same level of list structure; this component contains zero in the last cell at any level.

   SUBLIST, occupying the second word of the cell,specifies a list substructure attached to the current cell. As we shall see, the SUBLIST component of a cell actually functions as the first cell in the next lower level of list structure, and contains NEXT and IJK.VALUE pointers required by that function.

2) A cell occupying one 36-bit word, with two components:

   Q.NUMBER, specifying the index of the atom corresponding to this cube; and

   NEXT, pointing to the next cell on this level of list structure.

Since this type of cell is only used on the lowest structural level, there is no provision for sublist pointers.

The overall structure has four levels. The first level has one 2-word cell for each I value represented; they are ordered in their linear sequence in ascending values of I. The value of I corresponding to each cell on this level occupies the IJK.VALUE component of that cell. The NEXT component points to the next cell on the same level, i.e., the cell containing the next higher represented I value. Associated with each cell, and hence with each I value, is a list substructure; a pointer to this sublist is contained, for reasons which will become apparent in the ensuing discussion, in the SUBLIST component of the _next_ sequential cell. The second level of the structure, or the top level of the substructure pointed to by the SUBLIST components of the first level, has an identical format with each cell corresponding to one of the represented J values at the I value whose sublist it occupies. Thus, there is one such cell somewhere in the overall structure for each represented (I,J) pair. The SUBLIST components of this J level point, in turn, to a third or K level of lists whose format is identical to that of the first two levels. There exists, then, one 2-word cell at this third level for each represented (I,J,K) triplet, and hence for each non-empty cube. The SUBLIST components of each K level cell point to the list of atoms associated with the (I, J,K) triplet corresponding to the previous cell; thus, the SUBLIST components in each of the top 3 levels are deferred to the next cell in an identical manner. The fourth level, comprising the actual lists of atoms in each cube, is unstructured and hence requires only a 1-word cell. The Q.NUMBER of each such cell contains the index of an atom occupying the cube associated with its host list, and the NEXT component points to the next cell in this list.

Since, as we have noted, the SUBLIST component (and hence the second word information) associated with each cell in the top three list levels is contained in the _next_ sequential cell in the list, the first cell in each of these levels need not comprise 2 36-bit words. As a consequence, the SUBLIST component, which has been allocated an entire 36-bit word, may contain the first cell in the sublist itself rather than the pointer to the first cell preferred by conventional list processors. The reason that this technique is practical for our purposes but not for more general-purpose list processing involves the complexity inherent in any storage collection system which allows varying cell size. Since our application does not require erasure of list substructure and recovery of the space used by it, we can allow ourselves the luxury of making odd-sized cells out of free storage in a cavalier fashion.

For searching list structure, CUBE makes use of an internal subroutine FIND(VALUE). This program is entered with a global variable INDEX pointing to the start of a level (I,J, or K) of list structure; it takes either a

success exit, with INDEX pointing to the start of the next lower level corresponding to the specified VALUE, or a failure exit if VALUE is not represented in the structure. It functions as follows:

1) The word pointed to by INDEX is examined: since this is the first word of a 2-word cell, it contains components NEXT and IJK.VALUE; and NEXT.

2) IJK.VALUE is campared with VALUE:

   a) if IJK.VALUE is greater than VALUE, the failure exit is taken - since the I,J, and K values are arranged in ascending order, we have passed the appropriateplace for a cell corresponding to VALUE.

   b) If VALUE is equal to IJK.VALUE, NEXT+1 (a pointer to the second word of the next cell) replaces INDEX and control returns. This is the success exit: INDEX now points to the first word in the next list level.

   c) Otherwise, IJK.VALUE is less than VALUE, and the search must be continued. If the NEXT component is zero, we have reached the end of the level, and the failure exit is taken. Otherwise, the NEXT component replaces index, and control returns to 1) above.

An entry point to the \UBE package, INTEST(I,J,K,ARRAY) uses the routine FIND to return, in the array ARRAY, a list of atoms in the cube specified by I,J, and K:

1) the global variable INDEX (which is, in our implementation, an active register) is made to point to the first word in the overall list structure, i.e., to the first word of the I level.

2) FIND is called with the argument I. If the failure exit of FIND is taken, the cube does not exist, and a value of zero is returned to the calling program. If the success exit is taken:

3) FIND is called with the argument J, causing the J level of the structure to be searched. Again, the failure exit causes INTEST to return zero.

4) FIND is again called with the argument K. A failure exit causes the return of zero; the success exit leaves INDEX pointing to the first cell in the fourth level.

5) An integer ARRAY.INDEX is initialized to zero, and the list of atoms is scanned:

6) ARRAY(ARRAY.INDEX) is set to the Q.NUMBER component of the cell pointed to by INDEX; ARRAY.INDEX is incremented.

7) If the NEXT component of the cell pointed to by INDEX is zero, the current value of ARRAY.INDEX, specifying the number of atoms in the cube (and the number of entries in ARRAY) is returned to the caller. Otherwise:

8) The NEXT component of the word pointed to by INDEX replaces INDEX, and control returns to step (6) above.

A similar function, BIGTST(I,J,K,ARRAY) returns a list of atoms in the larger cube consisting of a 3 by 3 by 3 volume of smaller cubes and centered on the cube at I,J , and K. BIGTST functions in a manner identical to INTEST, repeating the search 27 times.

There are two entries which relate to the building of the list structure: NUCUBE, which resets certain variables which effectively "erase" any previously existing structure, and MOCUBE(QTO) which adds all atoms not previously included into the list structure thru atom QTO. The fact that we can specify that the list structure is to include only those atoms below a particular index allows us to add atoms incrementally to the lists, checking each atom only against those others which are lower in Q number. This avoids the redundancy of checking first Q1 versus Q2 and then Q2 versus Q1, and reduces the total number of comparisons by a factor of 2. MOCUBE uses an internal subroutine, PLACEU(VALUE), which performs a function analogous to that of FIND except that instead of taking a failure exit if an appropriate cell is not found, one is inserted:

1) The word pointed to by INDEX is examined, as in FIND. If VALUE is the same as IJK.VALUE, INDEX is adjusted in an identical manner, and control is returned; if VALUE is greater than IJK.POINTER, then:

    a) If the NEXT component of the cell pointed to by INDEX is non-zero, it replaces INDEX and control returns to (1).

    b) Otherwise, a new cell must be spliced onto the end of this level. Control goes to (2a) below:

2) If, however, VALUE is less than IJK.VALUE, the following steps are performed to splice a new cell into this level:

a) A pointer to the contiguous block of free storage is incremented by 2, and the two words to which it previously pointed are thus allotted to the new cell.

b) The entire first word of the cell pointed to by INDEX replaces the first word of the new cell.

c) The second word of the new cell is set to zero, specifying that the sublist is as yet empty.

d) The NEXT pointer of the cell pointed to by INDEX is set to point to the new cell.

e) The IJK.VALUE component of the cell pointed to by INDEX is adjusted to the value VALUE. Thus, the new cell has replaced the one pointed to by INDEX; this cell, in turn, now corresponds to our new level of lists.

f) INDEX is made to point to the second word of the new cell, which will become (on the next call to PLACEU) the first cell in a new level of list. Control returns to the calling program.

A parameter internal to the CUBE package, QFROM, keeps track of the last atom to be included in the list structure. Thus, in a call to MOCUBE, atoms between QFROM+1 thru QTO are added to the structure, each by 3 successive calls to PLACEU:

1) The index QFROM is incremented. The I,J, and K coordinates of the cube containing the atom indexed by QFROM are determined by a call to the routine GRID, which converts between the floating point X(Q), Y(Q), and Z(Q) values to the integers I, J, and K. GRID, which is also an external entry point, scales each coordinate by dividing by the cube size and truncates the results into integers. Large constants are added to these to insure that they are non-negative, and the results become the I, J, and K values corresponding to this atom.

2) INDEX is set to point to the first word of the first level of list structure.

3) Three successive calls, PLACEU(I), PLACEU(J), and PLACEU(K) adjust INDEX to point to the first word of the proper fourth-level list. Then:

4) A new 1-word cell containing the current atom Q is spliced into the <u>beginning</u> of this list of atoms:

a) If the Q.NUMBER component of the word pointed to by INDEX is zero, then this level must be empty. The Q.NUMBER component of this word is set to Q, and control goes to step 5).

b) Otherwise, a new 1-word cell is made by incrementing the pointer to free storage. The contents of the word pointed to by INDEX replace the contents of this cell.

c) The Q.NUMBER component of the cell pointed to by INDEX is set to Q; its NEXT component is made to point to the new cell. Our new Q now occupies the first cell on this level; thus, the Q numbers along a given fourth level list will be in <u>descending</u> order.

5) If QFROM is equal to QTO, all of the indicated atoms have been included; control returns to the calling program. Otherwise, control goes to step (1) for the next atom.

In addition to the CUBE package, LISTQQ calls another function (supplied as a LISTQQ argument) to handle the violating pairs once they have been discovered. The only such function presently available treats such pairs by making appropriate entries into the LQ1, LQ2, and LDIST arrays, thereby requesting that these atoms be moved a respectable distance apart. This function is designated PAIRVL, and its arguments are SEPSQ, the squared distance separating the atoms, and Q1 and Q2, the indices of the interacting atoms. PAIRVL first makes a test to determine precisely whether SEPSQ is less than the prescribed minimum squared distance for the particular types of atoms involved; a table of such minima (with an entry for each unique combination of two atoms) speeds up this test. If they are, in fact, in violation, a request is made to SOLVE (via the LQ1, LQ2, and LDIST entry) to have them move so that they will be the minimum distance from each other.

LISTQQ uses the QBACK entries in the ATOM array (see the discussion of MAKPRO) to determine in which stage of construction of the total list structure each atom should be tested for violations. Certainly, one could successively test each atom, and then add it to the list structure; this would insure that no atom was tested for violations with itself or with higher numbered atoms. Each Q1, Q2 pair would thus be tested only once, a desirable situation; however, to cut down on the time spent checking pairs, we try to take advantage of other physical constraints on the model which prevent atoms which are very close to each other in the connective plexus from approaching each other. We thus assert that each atom Q will not interact with other atoms in a surrounding topological region of the molecule,

and we don't bother testing such pairs. Although we have provision for specifying such a region for each atom in the molecule (there is one ATOM element, hence one QBACK component, for each atom) our current use of this provision specifies identical QBACK regions for each atom in a given residue. Our current initialization of these components prevents pair checking in adjacent residues (again, see discussion of MAKPRO, esp. the section describing MOLE).

The operation of LISTQQ is summarized below:

1) NUCUBE is called to initialize the CUBE package. An integer AA.POINTER, which will point to the amino acid currently being added, is set to 3 (since the first two amino acids, like any two adjacent amino acids, are not tested against each other by the argument given above).

2) A variable Q.NUMBER, specifying the atom currently being tested, is set to indicate the first atom in amino acid AA.POINTER, namely, QCALPH(AA.POINTER)-2.

3) A call to MOCUBE(QBACK(ATOM(Q.NUMBER))) adds to the list structure all atoms contained in amino acids before AA.POINTER-1.

4) A call to GRID finds the I, J, and K coordinates of the cube containing Q.NUMBER. BIGTST is called to make a list of atoms in the 27 cubes surrounding atom Q.NUMBER.

5) Each of the atoms QLIST returned by BIGTST is checked against atom Q.NUMBER for possible violation:

   a) The square of the distance separating atom QLIST and atom Q.NUMBER is calculated (via subroutine DISQ(Q1,Q2)) by summing the squares of the differences between each of the coordinate values

   b) A preliminary rough check is made by comparing this distance against a parameter MAXDSQ. If the squared distance exceeds MAXDSQ, no further tests are made between this pair. Otherwise, PAIRVL is called to make precise tests and correct any violation.

6) If Q.NUMBER does not point to the last atom in amino acid AA.POINTER (i.e., if it is less than QCALPH(AA.POINTER+1)-3) then it is incremented and control returns to step (4) above.

7) If AA.POINTER points to the last amino acid in the molecule (as specified by the zeroeth element of the AA array) then control returns to the calling program.

Otherwise, AA.POINTER is incremented and control goes back to step 2) above.

Thus, the input to the overall LISTQQ function is the X, Y, and Z atomic coordinates, and its output is a list of changes to be made in interatomic distances.

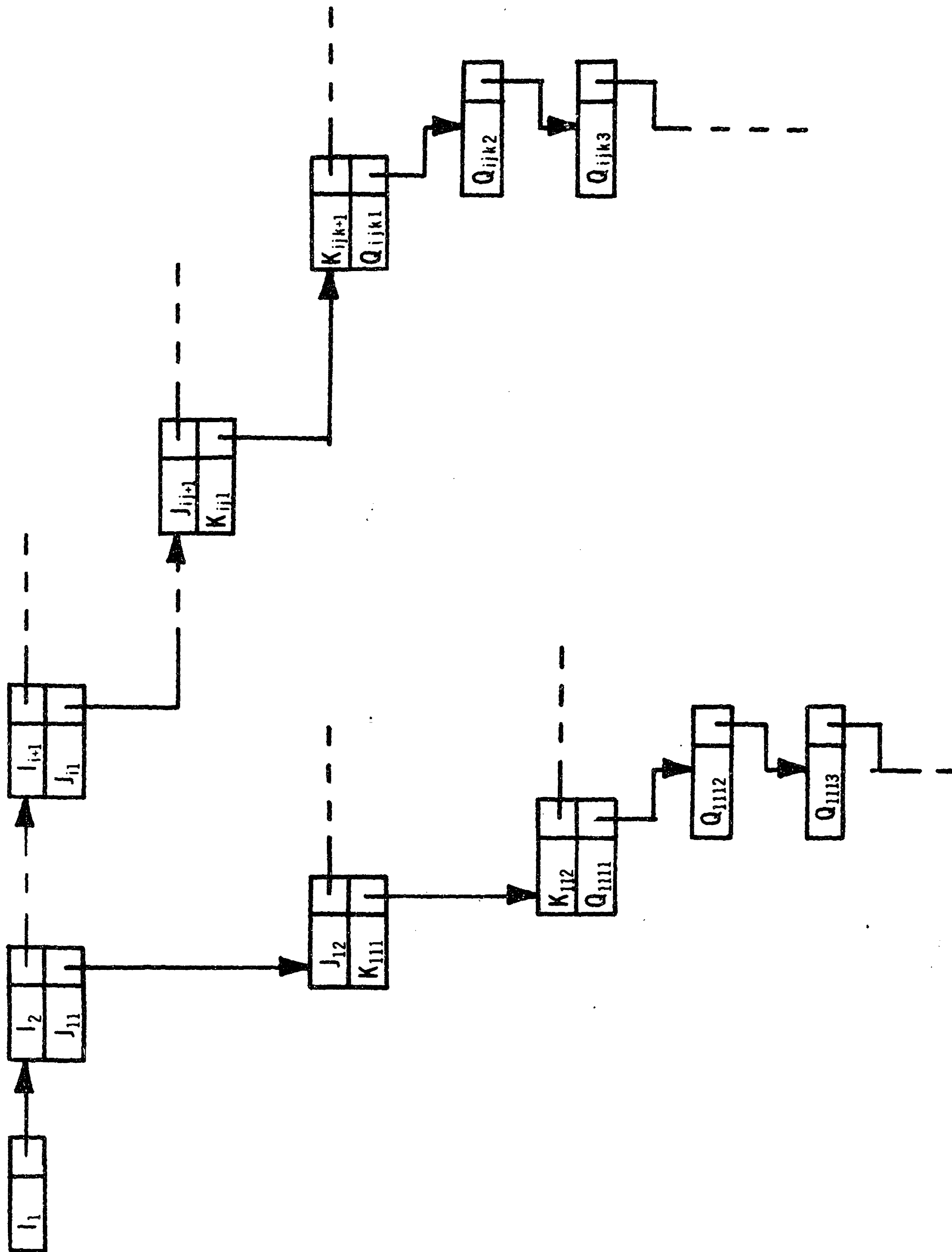

figure 3: Typical LISTQQ list structure

7.2.7 WIRE

WIRE is the name of the entry to a system of programs responsible for transmitting picture data to the satellite. The top level of WIRE deduces, from user supplied parameters, the nature of the required display, and constructs the "stick figure" representation of a relevent portion of the molecule. In this process, calls are made to a lower level system of subprograms (e.g., VVEC, IVEC) which are responsible for reformatting the data and interfacing to the telephone line. The top WIRE level, then, contains information relating to the structural properties of the various amino acids. The user-specified parameters which influence the operation of WIRE include the following:

1) The parameters BEG and END allow the user to specify the region of the molecule to be viewed; they are set to the first and last amino acid sequence numbers, respectively, to be included in the display.

2) The parameter SCALE is a floating point quantity relating the distance between atoms in the 3-dimensional display space to their actual interatomic distance; hence, increasing SCALE proportionately increases the size of the displayed figure.

3) The parameter START specifies the sequence number of the atom on which the displayed figure is to be centered. The coordinates of the atom thus specified map into the fixed point at the center of the display space.

4) the boolean parameters LTR and NUM determine the type of labels to appear on each residue. NUM specifies whether the sequence number is to appear; LTR specifies whether the 3-letter abbreviation of the amino acid name is to appear.

5) The BACK array, containing an integer element for each amino acid, specifies the amount of detail in which the segment of backbone associated with each amino acid is to be drawn; its values range from 0 to 3, 0 specifying the least detail.

6) The RES array, containing an integer for each amino acid, specifies the amount of detail to be included in the display of each residue. Its elements range from 0 to 2.

7) The boolean parameter BLOWUP may be set to request a special, detailed display of a particular region of the molecule. The mechanism triggered by this request makes use of the cubing programs of LISTQQ to

partition the space occupied by the molecule into cubes. BLOWUP requires the specification of the following additional parameters, each of which is ignored when the BLOWUP display is nor requested:

a) QCTR is an integer specifying the sequence number of an atom whose cube is to be at the center of the display space.

b) NBOXES is an integer specifying the number of cubes in each direction for which the display is to extend beyond that containing QCTR. Thus, the display space will include all atoms in the large cube, each of whose sides extends over 2*NBOXES+1 small cubes.

c) SIDAA is an integer specifying the length of the amino acid chain which is to be included in the display beyond those amino acids within the large cube (above). This parameter allows the user to include in his BLOWUP display a topological region of the molecule, independent of its actual position.

In its normal mode of operation (i.e., not BLOWUP) the top level of WIRE consists primarily of a loop in which the display corresponding to each successive amino acid is constructed. This major loop is outlined below:

1) The iteration variable, I, is set to the first amino acid to be displayed (that specified by BEG). A call to the subroutine NEWPIX signifies, to the display subprograms (and hence to the satellite) that a new picture is being built.

2) The Ith BACK element is examined, triggering one of several degrees of backbone detail:

   a) BACK(I)=0: The backbone segment of this amino acid is not drawn. An invisible line is, however, drawn to the carbon alpha (by the routine IVEC) in order to update the beam position for labels and residue displays.

   b) BACK(I)=1: A single visible line is drawn between the carbon alpha of the I-1th amino acid and the carbon alpha of the Ith residue. This rough schematic representation is useful to convey the gross shape of large sections of the molecule without cluttering the display with unnecessary detail.

   c) BACK(I)=2: visible lines are drawn between the I-1th carbon alpha and the subsequent carbon,

between the carbon and the nitrogen, and finally between the nitrogen and the Ith carbon alpha. This display preserves the connectedness of the backbone, ignoring only the superfluous hydrogens and oxygen.

d) BACK(I)=3: as in (c) above, but the hydrogens attaching to the carbon alpha and the nitrogen as well as the oxygen attaching to the carbon are drawn. Each of these atoms require a visible line in addition to an invisible line to preserve the connectivity of the backbone.

3) Each of the backbone displays is arranged so as to leave the beam position at the coordinates of the Ith carbon alpha. At this point, an internal subroutine RESD is called to attach the Ith residue in accordance with the value of the Ith element of the RES array:

a) If RES(I) is zero, no lines are drawn to represent the residue; control returns from the subroutine RESD.

b) If RES(I) is 1, an internal table is consulted which selects, on the basis of the type of the Ith residue, the single atom at the end of the side chain furthest from the backbone. A visible line is drawn to this point. This type of residue display affords a rough idea of the space occupied by a residue without complicating the display to a great extent.

c) If RES(I) is 2, all of the residue atoms are to be included in the display. In this case, RESD makes use of an internal table of instructions necessary to draw each type of residue. For each residue, the section of the table (named CON) which applies to that amino acid type is obtained from an array of pointers (PTR); successive elements of that section of CON are then interpreted as primitive move and draw commands. Throughout this interpretive loop, a variable J is updated to keep track of the residue atom last added to the picture; J is set at the start of the loop to the sequence number of the Ith carbon alpha. The legal CON elements and their effect on the picture are summarized below:

0 causes J to be set to the Ith carbon alpha and an invisible line to be drawn to this carbon alpha.

1 causes a visible line to be drawn to the J+1st atom, and J to be set to J+1.

2 causes visible lines to be drawn between the Jth and J+1 atom and between the Jth and J+2nd atom; J is set to J+2.

3 causes visible lines to be drawn between the Jth and J+1st atoms, between the Jth and J+2nd atoms, and between the Jth and J+3rd atoms; J is set to J+3.

4 causes visible lines to be drawn between the Jth and J+1st atoms, between the J+1st and J+2nd atoms, and between the Jth and J+3rd atoms; J is set to J+3.

8 causes an exit from the routine RESD, and control to be returned to the calling program.

-n, where n is an integer, causes a line to be drawn between the Jth atom and the J-nth atom, visible or invisible depending on the <u>next</u> element of the CON array. The latter CON element is skipped; i.e., it is not interpreted as a move or draw command. J is set to J-n.

99 causes a line to be drawn between the Jth atom and the nth atom in the residue (relative to the carbon alpha), where n is the next element in the CON array. The visibility of this line is specified by the CON element following that containing n; these two CON elements are skipped by the interpretive loop. J is set to the sequence number of the nth atom in the residue.

100 causes the chain of commands being interpreted to be interrupted, and commands to be interpreted starting with the nth CON element, where n is the contents of the next successive CON element. Thus, CON(I)=100 is analogous to a transfer instruction.

+n, where n is an integer other than those specified above, causes a visible line to be drawn between the Jth atom and the nth atom in the residue (relative to the carbon alpha). J is set to the sequence number of this atom.

4) The parameters LTR and NUM are examined. If NUM is set, a label is added to the display (by the routine SHGL) specifying the 3-digit sequence number of the current amino acid. If LTR is set, the 3-character

abbreviation (obtained by a call to ATTRIB) of the amino acid name is added to the label.

5) If I specifies the last amino acid to be included in the display (i.e., if I=END) then the subroutine PLOT is called (to specify that the picture building is completed) and control returns to the calling program. Otherwise, I is incremented, and control returns to 2) above.

The lines drawn in the above algorithm are implemented thru the internal subroutine LIN, which draws a line (visible or invisible) from the current beam position to the position of a atom whose sequence number is specified as an argument. LIN, in turn, calls upon the display programs VVEC and IVEC, described in a later paragraph in this section, to draw the actual vectors. LIN is responsible for translating the picture by the coordinates of the atom specified by START (or, in the case of a BLOWUP display, by QCTR) and for scaling the vectors by SCALE.

In the case of a BLOWUP display, the machinery described above is used in the same manner; however, its use is prefaced by a call to the internal subprogram QPAND which presets the BACK and RES arrays so as to suppress the display of those amino acids not within the blowup region. In particular, QPAND does the following:

1) The BACK and RES arrays are zeroed. The cubing subroutines of the LISTQQ package (see 7.2.6) are called upon to find the I,J, and K coordinates of the cube containing the atom QCTR.

2) For each i,j,k such that I-NBOXES≤i≤I+NBOXES, J-NBOXES≤j≤J+NBOXES, and K-NBOXES≤k≤K+NBOXES the followin steps are performed:

   a) The list of atoms in the cube (i,j,k) is obtained thru a call to INTEST.

   b) For each atom on this list, a non-zero value (specified by the user set parameters ALLBAK and ALLRES) is placed in the RES and BACK elements corresponding to the amino acid containing the atom.

3) Finally, a loop determines the first and last amino acid (by sequence number) having non-zero BACK elements; then the BACK elements of all acids from the first-SIDAA thru the last+SIDAA are set to the non-zero value. Similarly, the RES elements of the same residues are made non-zero.

The second level of display programs, those called by WIRE, include the subroutines NEWPIX, VVEC, IVEC, PLOT, and SHGL. These programs provide the interface to the satellite display, translating data between the floating point internal representation of the main processor and the truncated integer representation which is used in the transmission of the data over the telephone lines. One major function of these programs is to decrease the volume of data to be transmitted to the satellite, thus minimizing the time spent in picture transmission. To this end, the components of vectors are sent over the line to only 6 bits of accuracy. To avoid the accumulation of errors along an additive chain of vectors thus truncated, the display subprograms simulate the process by which they are ultimately added at the satellite. The programs are thus aware, at any point during the transmission of a picture, of the beam position both in terms of the floating point components which the calling programs have supplied and in terms of the truncated integers which have been sent to the satellite. Consequently, when the routines VVEC (to draw a visible vector) and IVEC (for an invisible one) are called, their floating point arguments (the vector components) are added to the floating point program variables FX, FY, and FZ. These variables now contain the coordinates of what the calling program 'assumes' the beam position to be, based on an accumulation of vector components. These values are then compared with the integers IX, IY, and IZ which represent the beam position according to the information which has been sent to the satellite, and the calculation of the vector sent to the satellite is based on the difference between these sets of coordinates. In particular, the vector (FX-IX,FY-IY,FZ-IZ) with components rounded to the nearest integer is sent to the satellite. After each vector is sent, its components are added to IX, IY, and IZ to simulate the change in beam position resulting at the satellite.

The display hardware of the satellite, however, constrains us to limit the magnitude of each component of the 2-dimensional vectors which we ask it to draw; longer lines consequently must be built up from short vectors. Since the 2-dimensional lines which appear on the CRT face are projections of arbitrary rotations of the 3-dimensional figure, we must limit the magnitude of the 3-dimensional vectors so that their longest projection is within the maximum allowable magnitude for 2-dimensional vector components. This amounts to a limitation on the magnitude of the vectors sent over the line by the VVEC-IVEC system; and in order to take full advantage of the 5 magnitude bits transmitted for each vector component, it makes sense to scale the transmitted vectors so that the maximum magnitude is that of the vector with 2 zero components and one containing a sign bit followed by 5 ones.

Before the vector (FX-IX,FY-IY,FZ-IZ) is sent to the satellite, then, its magnitude is compared with the maximum. If it is too long, it is scaled so that it retains the same direction but is less than the maximum magnitude, the result being sent to the satellite; IX, IY, and IZ are then updated by this scaled vector. The difference vector (FX-IX, etc) is again calculated, and the process repeats until the difference vector is within the allowed magnitude.

The other functions of the display subprogram are straightforward, generally involving the sending to the satellite of control characters; these characters and the format of transmitted data is discussed in more detail in section 9.2.5. NEWPIX zeros the program variables (FX, IX, etc) and sends a control character; PLOT sends a control character, insures that the main processors output buffer is emptied (i.e., that all characters belonging to the picture have been sent) and causes the special trap character (discussed in 9.2.4) to be sent thru the message channel (section VIII). The entry SHGL, called to add labels to the display, sends a control character followed by the characters in the label (converted to the character set used by the display hardware).

In addition to the normal mode of operation, in which the data transmitted to the satellite consists of a mixture of control characters and vector components, facilities are provided by the display subprograms for the transmission of the vector components alone. During such operation, the satellite assumes that the control information is identical to that transmitted for the previous picture; hence, this mode of transmission is useful only for the transmission of a revised instance of a previous configuration. Only the values of the component vectors may change from each picture to the next. Such transmission is initiated by a call to REPIX instead of the NEWPIX call; subsequent calls to VVEC and IVEC then result in the transmission of just the component values, and subsequent calls to SHGL are ignored. This technicque results in considerable reduction of the transmission time, depending upon the nature of the picture; displays with many labels frequently are transmitted in less than half the normal time. WIRE calls REPIX (rather than NEWPIX) automatically each time it is called with values of the display parameters identical to those at the time of the last call.

VIII. Interconnection of Processors

The actual medium by which the main processor and satellite communicate is a voice-grade telephone line, multiplexed so as to simulate full duplex operation. The nominal rate of information flow, in either direction, is 1200 bits per second, although as we shall see, a substantial fraction of this total data flow is used for control information. Transmission in each direction is character synchronous; consequently messages must be padded with null characters so that the line is never idle in either direction.

At the most basic level, then, there are two independent information paths between the machines, one in either direction. Data thus transmitted is organized into messages, each containing a maximum of about 60 characters of real data in addition to several characters of control information; each character, in turn, consists of 10 bits of which 6 are data. The actual transmission of a message from one machine to the other involves several preliminary exchanges to establish the fact that the sender has a message and that the reciever wants it, as well as a final exchange to determine whether the message was recieved error free. These auxilliary exchanges utilize an elaborate lexicon of control characters and conventions for their use, as well as unwieldy programs in either machine for their implementation. In the light of modern information theory, this scheme has no very impressive defense; indeed, it doesn't hold up very well under the scrutiny of a modicum of common sense. Yet it does perform, however inefficiently, the function required by out programs: it provides for bilateral communications between our processors on an interactive basis. This last qualification (that the communications are interactive) allows us freedom in the organization of the programs running on each processor; it allows that the functions of each are asynchronous, and that the process won't be irretrivably boggled by one machines arrival at the point in its algorithm where it is ready to send data before the other arrives at the corresponding point where it is ready to accept data. In the current scheme if one processor is ready to send before the other is ready to recieve, the first processor goes into a waiting state, and no data appears on the line until the second is ready. Such an arrangement is critical to an organization of the software (on either machine) which relegates the servicing of input and output devices and the buffering of their data to an independent supervisor.

The archetecture of the main processor's supervisory program is such that messages arriving at the satellite are distinguishable as to their source; the format of messages, intended normally for printing on the remote terminal' printer, is slightly different from that of data explicitly

transmitted by the user programs. Thus the information path from the main machine to the satellite is multiplexed, allowing in theory two logically independent channels of communication: one for messages directly to the user (e.g., via teleprinter) and another for binary data to be decoded by the satellite programming. In fact the logical independence of these channels is subject to several restrictions; the most striking among these is that the filling of the satellite buffers corresponding to one channel effectively blocks transmission via the other. Since a finite (indeed, small) amount of the satellite memory is allocated to the preliminary buffering of input data and messages, it is possible for the satellite to be removing data, for example, from one buffer while the other is filling; when the latter buffer becomes full, the satellite cannot accept from the main processor any further messages (of either variety) until the satellite programs empty the full buffer. The main processor program, then, cannot mix indescriminately messages and data unless the satellite programs are prepared to reorganize the mix. Since it is impractical to impose any new restrictions on, for example, the library functions of the main processor, the burden falls largely on the satellite programming. Thus, the satellite programmer must avoid the temptation to enter, at the start of the transmission of a picture, a loop which does nothing but read data and build the picture and which is left only when the picture is completed: since messages may be mixed with the picture data, the message buffer might overflow before the completion of the picture, leaving both channels blocked and the satellite hopelessly hung up.

IX. Satellite Programs and Data

The satellite programs are divided, in vague analogy to the similar division of the main processor programs, between those operating in a "supervisory" capacity and those those running as "user" programs. This distinction was not emphasized in the case of main processor programs, since supervisory programs on large (especially time-shared) processors are taken for granted as a part of the operating environment of the user programs, since no features of the main supervisor uniquely or subtly effected the basic operation of our system, and since the main processor supervisor is the subject of excellent documentation by its archetects. A firm dichotomy between supervisory and user functions is less conventional, however, on small machines, and its operation is substantially and deeply imbedded in the function of our small machine as a satellite.

9.1 Supervisory Functions

The principal justification of a supervisor at the remote satellite is that it provides a uniform means for controlling I/O devices from program to program. Since the major I/O devices are tied together through the interrupt facility, the programs for handling each device are not entirely independent; hence the I/O programming constitutes an intricate and complex package which is not easily separable into independently functioning modules. It is convenient to have the entire I/O package operating as an integrated program, with standardized calls to perform the various input and output functions; this relieves the satellite programmer of the burden of worrying about details of I/O management each time he writes a new program, and tends to standardize the use of each device. Furthermore, such a system isolates the user program from hardware changes: when the configuration of the I/O hardware is modified, a corresponding change need be made in only the programs comprising the supervisor.

The handling of the various devices by the supervisor varies somewhat depending on the particular characteristics of each device. One class of devices communicates with the user program by data transfers directly to and from the user core region, while another transfers data thru an intermediate buffer in the supervisor. The distinction between these classes of devices is largely a function of their speed: those in the first class are characterized by high data rates (e.g., magnetic tape, display) while the latter class comprises lower speed devices (e.g., teletype, dataphone connection to main processor)

The philosophy underlying the buffering of low speed devices involves the substitution of data transfers between

supervisory buffers and the user program (at electronic speeds) for transfers between the user program and physical devices (at mechanical speeds). Consequently, the user program is not "hung up" waiting for a device to complete operation. A user program performing an output function, for example, will communicate to the supervisor the data to be transferred; the supervisor will put this data in a buffer associated with the indicated output device. When the device is free to accept further data, it causes an interrupt to the supervisor, and the actual transfer is made from the buffer to the device; meanwhile, the user program has continued with its algorithm, ignorant of the timing of the actual transfer to the output device. Thus, to the user program the output device is apparently very fast (since this program wha held up for only the few instructions necessary for the supervisor to stack the data into the appropriate buffer). In the case where the buffer associated with an output device is full, however, when the user program requests further output, the supervisor has no recourse but to hang up the user program until there is space in the buffer for the new piece of data. Thus the amount of time which the user program spends waiting for I/O operations is a function of both the buffer size and the rate at which the user program generates (or requests) data to be transmitted. In addition, this idle time depends upon the actual distribution of I/O demands of the user system over time. It is obvious that if the average data rate of the user program exceeds that of the device, the program will eventually hang up (assuming it continues long enough). On the other hand, if the user program has a data rate lower than that of the device, a sufficently large buffer will reduce the idle time of the program to nearly zero.

9.1.1 Device Timing Considerations

There are several potential pitfalls involved in programing a machine with a single interrupt level to service many I/O devices concurrently. Such a machine is characterized by an interrupt structure comprising two levels of programming: one which operates with the interrupt mechanism enabled, the other running with it disabled. In the usual configuration, the user program operates normally with the interrupt enabled; a signal from any of the several operating devices at any time causes the user program to be interrupted and the appropriate device-handling routine to assume control. This routine services the device (sends data to an output device or gets data from an input device) while the interrupt is disabled, preventing another interrupt from occurring while control is in the service routine. Upon completion, the service routine restores control to the interrupted user program and reinstates the former (enabled) interrupt status.

We immediately note that, since the interrupt mechanism is disabled while control is in a service routine, no other devices may be serviced until any currently operating service routine is completed. If an interrupt signal is recieved from a device while another device is being serviced, the actual interrupt is deferred until control is restored to the user program and the interrupt mechanism is enabled. If the device whose service is thus deferred is sufficiently timing sensitive as to be intolerant of the resulting delay, an error condition will result.To further complicate this situation, we must consider the possibility that more than one additional interrupt might occur while control is in a particular service routine; in this case, some device will have to wait for its service while the other devices are being serviced. We have the freedom to choose, in such a situation, which of the interrupting devices will be serviced first, but it is necessary to ensure that no device will be intolerably hung up by the maximum number of stacked interrupt services which it must endure. To this end, we specify for each device values for the following parameters:

1) The maximum <u>latency</u> of the device, or the greatest delay it will tolerate in the service of an interrupt request.

2) The <u>service time</u> for the device, which is the maximum time the interrupt mechanism will be disabled as a result of an interrupt request from this device.

3) The <u>load factor</u> of the device, a parameter pertaining only to those devices connected to the data break facility of the computer. This connection allows the device to access the computer's core memory directly on a "cycle stealing" basis: the currently operating program is periodically delayed for a single cycle while the data transfer is made. This arrangement effectively slows down the progress of the program, and the load factor is defined as the fraction of real time thus lost to the main processor program.

In designing the interrupt structure of the supervisor, it is necessary to assign to each I/O device a relative priority to resolve situations in which more than one device is requesting interrupt service.Normally, the more timing sensitive (least latency) devices are assigned to the high priority end of the queue, while slow and insensitive devices recieve lower priorities. In case of simultaneous interrupt requests, then, the device whose priority is highest will be serviced first. Now, we may generalize our timing constraints as follows:

1) The total load on I/O system must not exceed unity. The total load is defined as (1-F1-F2-...-Fn)(S1R1+S2R2+...+SnRn) where each Fi is the load factor of device i, Si is its service time, and Ri is the maximum rate at which device i can request interrupt services.

2) The maximum latency time of each device k must exceed the larger of:

   a: The sum of Si for each device i<k (i.e., for each device having greater priority than device k) plus the maximum service time of any device j>k;

   b: The maximum service time Sj of any device j<k plus the sum of the service times Si over those devices i<k of higher priority than device i.

9.2 Picture Handling Functions

The programs responsible for the generation, from the data transmitted over the telephone line, of the ultimate 3-dimensional figure generally divide themselves into three major classes. An input program is used to decode the binary picture description from its input format and produce, for example, a series of 3 element vectors whose components are in a representation tractable by the satellite. Secondly, a display list compiler program takes this move-and-draw data as its input, producing in core memory a description of a 2-dimensional projection of the figure in the format acceptable to the display hardware. A third program supervises the cycling of the display logic thru the display list, insuring that the display is refreshed at a tolerable rate. Finally, a fourth program periodically examines the status of the globe (an input device by which the operator specifies a requested rate and direction of rotation) and modifies the particular projection of the figure represented by the display list so as to simulate the rotation of the 3-dimensional structure being viewed.

9.2.1 Data Structures

The format of the parametric 6-bit move-and-draw representation of the picture which is transmitted over the telephone line is described in the section of this paper on the subprogram WIRE. This data, organized at the lowest level in 6-bit characters, is read into a preliminary buffer of about 60 characters (within the supervisor); this 60 character buffer length corresponds to the maximum length of a message, into which the characters are organized for their transmission over the line. Message organization is ignored by the picture programs, which consider the data transmitted to be a pure character stream; nevertheless, this level of organization is required for the error checking and communications conventions to which we are constrained. The characters which constitute a complete message, immediately upon the verification of the correct reception of that message, are moved from the preliminary buffer to a substantially larger secondary buffer (also in the supervisor). This process involves no interpretation or decoding of the data, each character being moved without examination.

As the input data is subsequently interpreted, several internal data structures are created, collectively providing a useful local representation of the figure. The display list, which is interpreted directly by the display hardware, provides the only storage of 'fixed' attributes of the picture, e.g. character strings (labeling portions of the figure). The display list itself provides sequential

instructions for the actual movement of the electron beam on the CRT face; typical display list commands request the movement of the beam by a certain increment (containing horizontal and vertical components), the drawing of a character, etc. In a single display list command, a vector may be drawn which extends one eigth of the distance across the screen, or 1 1/4 inches, and each component of this vector is represented in the command word by a sign bit and 7 magnitude bits. Hence, we deduce that there are 1024 addressable points along each axis. The limitation imposed by the hardware that vector magnitudes be 7-bit quantities stems from a desire to allow one vector to be represented in the display list by a single 18-bit word; this, in turn, implies either a restriction on the length of a single vector or a limitation in the precision with which its end points are located on the CRT. In fact, there are provisions in the hardware for increasing the maximum length of single vectors at the expense of an effective reduction in the number of addressable points along each axis (i.e., a reduction in resolution); since, by their nature, are displays are composed of vectors of limited length, we make no use of this option. Hence on those relatively rare occasions when a long vector is required in our figure, on is constructed by stringing short ones (see the notes on VVEC and IVEC in the WIRE description).

Since the display list contains only enough information to characterize a 2-dimensional projection of our figure, we obviously require more information regarding the 3-dimensional character of the vectors to be stored elsewhere. As the format in which the 2 components are stored in the display list is awkward in terms of retrieval by the program, the 3 vector components are, in fact, stored redundantly as full word, 18-bit signed ones complement numbers in additional tables, obviating the need for the recovery of vector information from the display list. In addition to the 3 arrays which are used to hold the components of each vector, a fourth array is used to establish a correspondance between the vector components and the storage of their 2-dimensional projection in the display list. Since, in general, consecutive words in the display list do not represent consecutive vectors (there are other words containing control information, characters, etc) a pointer is required for each vector to the word in the display list in which it is stored.

We may notice, at this point, that the storage space required for a figure containing n vectors (both visible and invisible) is more than 5n 18-bit words, ignoring the buffering of the parametric information and the display list storage of control data, etc. We have obviously allowed a degree of redundancy and a consequent loss of storage efficiency by choosing this representation; this choice results from consideration of the timing requirements of

real time rotation of the displayed figure.

9.2.2 Input Phase

The size of the supervisor's data buffers is a compromise between our desire to store the parametricrepresentation of an entire picture in a preliminary buffer and the storage requirements of the final internal picture representation. Ideally, we would like to be able to store the representation of a new picture in a large buffer while displaying (and rotating) an earlier version until we are sure that all of the new picture data has been recieved; in this way, the use of the display functions of the satellite are not interrupted for a significant interval. However, the storage of representations of two pictures simultaneously, which this organization would entail, imposes intolerable restrictions on the length of either. The compromise which we have arrived at involves the use of an input data buffer large enough to store simple figures in their entirety, but substantially smaller than the space allocated to the final representation of the figure; furthermore, the satellite system is arranged to make optimum use of the buffer space by maintaining the previous picture's representation until the input buffer is full.

The input buffer is thus handled by an expedient if inelegant scheme which requires that the lowest level of display programming on the main processor be aware of the actual size of the input buffer in the satellite. The main-processor subprogram PLOT, called at the completion of a picture, signifies that an entire picture has been sent to the satellite by means of a message (in a special format, distinguishable from teletype messages) over the information path (see section VIII) normally reserved for communication between the user and the main processor programs. At the satellite, this message triggers the mechanism by which the input buffer is emptied and the new picture is built (i.e., the display list and vector component arrays are constructed). The old picture is lost at the beginning of this picture building process (since the space occupied by it must now be used for the new picture) and the new one appears at its completion; however, if the representation of the entire new picture is in the input buffer at the time that the picture building process is begun, the interval for which the display is inactive (i.e., while the new picture is being built) is small.

If, on the other hand, the representation of the entire new picture will not fit in the input buffer, we might envision the satellite becoming hopelessly hung: the overflow of the data buffer implies that communication over both paths is blocked (se section VIII) and no messages,

including the one which triggers the emptying of the data buffer, can be recieved. We avoid such an impasse in the current scheme by requiring the main processor programs to anticipate the overflow of the input buffer (since it can keep track of the number of characters which it has sent) and, if necessary, send the message triggering the building of the new picture and the concurrent emptying of the input buffer before the catastrophe occurs. The requirement that the main processor system be aware of the size of a satellite buffer is unfortunate, and indeed there is no logical reason why the satellite might not avoid the overflow of its buffers independently; the choice of the former system was primarily due to the expedience of making use of an existing mechanism rather than the construction of a new one.

Once the picture building operation has started, the input program is repeatedly called upon to extract input characters from the input buffer, and to decode those characters corresponding to vector components into full word, ones complement fixed point values; in particular, the 6-bit sign-magnitude representation is loaded into the high orderbits of an 18 bit word (the others being zero) and if the high order bit is set, the low order 17 bits are complemented. This allows subsequent computations (e.g., the matrix rotation) to be carried out to the full 18-bit precision allowed by the arithmetic circuits of the machine.

9.2.3 Picture Building Phase

The analogy between the translation of a move-and-draw picture representation to display list code and the translation of a higher level programing language to machine instructions is revealing. In each case, the translator must map a single source language operation into several object language operations, anticipating and correcting for changes in the state of the object machine (e.g., the contents of active registers) with each object instruction. In each case the translation process may be properly termed compilation.

The display hardware has a number of state variables, specifying the intensity of the display, the format of subsequent display list words, etc. In general, the display list contains a mixture of control words (changing the values of state variables) and data words (specifying that the beam position be changed). Examples of the former are instructions that the following display list words are to be interpreted as vectors, or that subsequent lines are to be drawn at a particular intensity; words of characters or vector components typify the latter class of display list commands.

The actual compilation of the display list requires that program variables be maintained which reflect the state of the display at each successive display list command, so the the appropriate control words may be inserted as the various display functions are encountered. While the specific (machine dependent) bookkeeping implicit in this process will not be discussed in detail, the reader should be aware that the addition of data words to the display list frequently referred to in the following discussion involves, in general, the prior addition of control words. The following paragraphs, then, largely ignore this corollary function of the compiler, following rather the transfer of data to the display list as each request is processed in the input stream.

The format of the input picture representation is such that actual data characters (e.g., components of vectors) are preceded by control characters identifying them. It is designed so that no 'look ahead' is required - that is, the processing of each character is not dependent upon characters which are to be encountered subsequently in the input stream. Consequently, the display compiler functions by reading successive control characters and dispatching to the appropriate algorithm depending upon their values:

1) This character signifies the start of a new picture. It is the result of a main processor call to the subprogram NEWPIX, and results in the initialization of the program variables of the display compiler. The arrays containing the X,Y,Z components of the vectors and the pointer to the corresponding display list words are emptied; the display list is emptied and refilled with the commands generating a single invisible setpoint at the center of the CRT. This will be the only setpoint in the display list, and specifies the point about which the projection is to rotate.

2) There are two characters which specify that a vector is to be added to the display, one of which stipulates that the vector is to be invisible. Each causes the retrieval (via the input program) of the 3 next characters from the input stream, and their conversion to the full word vector components; these components are placed directly into the next successive elements of the appropriate arrays. The components are next converted into the 7-bit sign-magnitude form required by the display list format, and an appropriate data word is added to the display list. The address of this display word is placed in the pointer array so that it becomes associated with the component values placed in the vector component arrays, and the index to these arrays is incremented. The algorithms for visible and invisible vectors differ only in that a

bit in the data word corresponding to the visible vector in the display list is set to inform the display that a line is to be drawn.

3) There is a character specifying that a label is to be constructed. The convention is that the very next character in the input stream specifies the number of characters in the label, and this character count is read immediately from the input buffer. Since characters change the beam position, the display compiler takes special pains to correct for the offset produced by a label by adding an invisible, fixed, 2-dimensional vector to the display list which corrects for the beam displacement. Thus, a label can be added between any two vectors without affecting their relative positions; furthermore, although the label will rotate as a rigid structure flexibly attached to the rotating figure, the characters themselves will not rotate. Thus, the compiler reads a number of characters (in BCD) from the input data stream equal to the character count, converting them to the character set used by the display hardware and packing them into successive data words, 3 to a word. Finally, the commands necessary to generate an invisible vector back to the beam position at the start of the label are added to the display list. Since the latter vector is not to rotate, no entries are made in the component and pointer arrays.

4) A final character, sent by the main processor routine PLOT, signifies the end of the picture. This triggers the display program (section 9.2.5) which causes the picture to appear.

5) A character sent by the main processor subroutine REPIX performs a function analogous to that of the chanacter sent by NEWPIX, above. The REPIX call, however, establishes a special mode of picture building in which the structural character of the figure is the same as that of the previously sent picture; that is, the number and order of vectors, labels, etc are the same. The only features of the picture which are allowed to change are the actual values of the vector components. In this mode, successive triplets of 6-bit characters in the input stream are taken to be sets of vector components until each of the 3-dimensional vectors in the picture have been revised. The machinery of step (2), above, is used in this process: the component arrays are updated, an element at a time, and the display list word pointed to by the pointer array element corresponding to each set of component values is revised to reflect the new values. The bit specifying the visibility of each vector, as well as the other

display list control information, remains unchanged. This mode of operation is useful whenever the main processor system requires that the existing picture be deformed rapidly.

9.2.4 Rotation Phases

The effective rotation of the displayed figure being observed constitutes a critical link in the man-computer system, providing the sole means by which the user may appreciate, in a qualitative sense, the 3-dimensional character of his model. It is consequently essential that the visual effect produced by the rotation mechanism simulate, in a natural way, actual rotation of a figure in 3-space. Smooth motion in a real time display requires the frequent (say, 20 times per second) revision of the display list. Since each revision of the display list involves the rotation of the 3-dimensional representation and the projection onto the 2-dimensional CRT, this process makes heavy computational demands on the satellite. It was, in fact, unclear at the start of the project that the computational power of the satellite computer was adequate to perform satisfactorily the rotation function; our current assertion that it is remains provisional. Earlier experiments with the rotation algorithm significantly sacrificed the quality of the display for rotational speed (i.e., for increased frequency of display list revision,) but ultimately a compromise was reached which results in a high quality display with only a moderate reduction in the smoothness of the rotation. Since the earlier algorithm may be of some technical interest to users of machines with limited computational facilities, its salient features are briefly discussed in addition to the description of the program finally arrived at.

Incremental Rotation

A principal virtue of the rotation algorithm first used is that it does not make use of multiply or divide instructions, allowing it to be implemented on a machine without the corresponding hardware. To appreciate the 2-dimensional instance of the algorithm, we observe that the points whose X and Y coordinates in the cartesian plane are:

$$X = \cos \emptyset$$
$$Y = \sin \emptyset$$

lie on the unit circle centered at the origin. Incrementing the angle Ø by the differential dØ, the point (X,Y) traces the unit circle; the corresponding X and Y increments are:

$$dX = -\sin \phi \, d\phi = -Y \, d\phi$$
$$dY = \cos \phi \, d\phi = X \, d\phi$$

We are led, quite naturally, to expect that the motion of our point along the unit circle to be approximated by the difference equations:

$$X' = X-kY$$
$$Y' = Y+kX$$

for sufficiently small k. Furthermore, we expect that the motion of the point might be reversed by a change in sign of k (analogous to a change in sign of dØ,) or equivalently by a reversal of the roles of X and Y in the equations. Finally, we note that the constant k may be a negative power of 2, so that the multiplications kY and kX might be replaced by arithmetic shifts, thus minimizing the amount of computation necessary to update X and Y. In fact, the locus traced by X and Y for shifts of 6 or 7 places (in an 18 bit computer word) reproduces a circle quite acceptably within visual tolerances.

As the radius of the circle approximated by the locus of X and Y is determined solely by the initial X and Y values, the application of the above difference equations to the rotation of strings of vectors is immediately suggested. If our 2-dimensional figure is composed of a number of vectors (some of which may be considered invisible) strung end to end in such a way that the coordinates of the origin of each vector is the vector sum of those vectors preceding it in the string, then we presumably may rotate the entire system about the origin of the initial vector by applying the above difference equations simultaneously to the X and Y components of each component vector. In practice, we ensure that the rotation is centered about a fixed point on the CRT screen by the placement of an invisible setpoint at this position at the start of the display list. It is convenient that the hardware configuration of our display facility allows for the specification of relative vectors, although little additional effort would presumably be required to calculate the absolute coordinates, say, of vector end points from the lists of their relative components.

Several further observations regarding the bahavior of these equations in a digital field led to further refinement of the algorithm. First, we note that the matrix R used to increment each vector is:

$$\begin{matrix} 1 & -k \\ k & 1 \end{matrix}$$

and the corresponding matrix R' used to rotate incrementally in the opposite direction (obtained by reversing X and Y in the equations) is:

1 k
-k 1

Now, taking a step first in the positive direction and then another in the negative direction corresponds to updating each vector by the matrix RR', or:

1+k**2 0
0 1+k**2

so that the process is not, in fact, absolutely reversible. Since we expect that in practice a user will want to rotate his figure repeatedly in either direction as he is studying it, the fact that it grows in size might prove annoying. However, if we modify our equations slightly:

X'=X-kY
Y'=Y+kX'

we find that R and R' are, respectively

1 -k
k 1-k**2 and
1-k**2 k
-k 1

so that their product becomes

1 0
0 1

and the incremental rotation is reversible. This minor change in the algorithm actually simplifies the program, eliminating the necessity for a temporary storage and retrieval, hence providing a most satisfactory solution to the reversibility problem.

Somewhat more startling is the observation that the above system of equations, when applied iteratively through a full 360 degree rotation and when implemented using ones complement arithmetic, yields <u>exactly</u> the initial X and Y component values. The number of complete 360 degree rotations necessary for this exact closure varies with the actual numbers involved, but the fact that it will eventually return to its initial values guarantees us that the scale is niether vanising nor becoming infinite. That the closure is dependent on the use of ones complement arithmetic reflects the symmetry of the bit patterns encountered during a complete cycle; in particular, since there is a one to one correspondance between the positive values which each component assumes during a rotation and its negative values, the truncation errors accumulated during each half cycle may cancel each other. In ones complement arithmetic, the number -n is represented as the

complement of the representation of n, so that the symmetry is preserved and the errors do, in fact, cancel. In twos complement arithmetic, for example, such symmetry does not exist between +n and -n in their respecive binary representations, and consequently error can accumulate over a cycle.

Our adaptation of the algorithm to the 3-dimensional case involves the application of the 2-dimensional rotation to each pair of coordinates. Thus, for rotation about the Z axis, we apply the above equations to the X and Y components of each vector. Of course the closure of the 2-dimensional case does not imply that we may apply rotational increments about the three coordinates arbitrarily and expect to accumulate no error; but since continuous rotation about a fixed axis through an integral number of cycles causes no error, we may expect to introduce errors only when the axis of rotation is changed. As this occurs only when the operator physically manipulates the input device (globe) we might be optimistic regarding the rate at which the vector representation of our figure deteriorates, and in fact a picture thus rotated typically retains a tolerable semblance of its origional configuration through 20 or 30 minutes of active manipulation.

The display resulting from this rotation scheme was objectionable, then, not primarily because of the long term deterioration of the figure; rather, it was the errors between one rotationalincrement and the next which motivated us to abandon this algorithm. Since the display consisted typically of several hundred (a maximum of about 700) vectors strung end to end, the coordinates toward the end of the string were very prone to an accumulation of roundoff (and other) errors along the chain. This situation is compounded by the tendency of the roundoff errors of each vector along the chain to be of the same sign, and consequently not canceling each other to the extent which one would expect if they were occurring randomly. The net result was a somewhat objectionable jumping around of the vectors toward the end of the chain as the figure rotated. In fact, however, this problem occurred (to a lesser extent) in the algorithm currently being used, and it is not clear that the steps taken to cure the problem in the later program cannot be applied to the incremental rotation algorithm with comparable success. The other drawback of this algorithm, however, is that it involves the actual modification of the parametric representation of the picture; i.e., the components of the constituent vectors are actually modified in the process of the rotation. Consequently, recovery from the situation in which the picture has deteriorated due to accumulated errors involves the storage of an extra copy of the original 3-dimensional vector components, an impractical approach in a machine of very limited memory size.

Matrix Rotation

The obvious solution to the latter problem is to modify a rotation matrix rather thatn the vectors themselves, thus preserving the canonical description of each vector in its unrotated form. This technique obviously involves a matrix-vector multiplication for each component vector at each time that the display is updated, but in fact the computational load thus imposed on our particular satillite processor is about comparable to that due to the earlier incremental algorithm.

The use of an intermediate rotation matrix allows us to preserve our original vector representation, but in itself merely defers the actual rotation process. We are still faced with the problem of updating the rotation matrix, which we may attack in a variety of ways. Certainly, we may use the incremental approach discussed above to vary the matrix elements corresponding to rotation about each axis, by applying the difference equations to pairs of matrix columns; however, since the principal advantage of the incremental method is its execution speed, the argument for its use here seems less compelling. The matrix must be updated only once for each display list revision, so that we may justify spending more time changing the matrix than we were able to devote to updating each vector.

We might be tempted to save the rotational attitude of the figure in terms of 3 rotational angles, and calculate the new rotation matrix at each incremental step by incrementing the appropriate angles (depending upon the rotation requested by the user) and using their new values in a trigonometric formula for the new matrix; this, presumably, would avoid the problem of accumulated error in the matrix. The rejection of any such scheme is due to the following considerations:

1) It seems crucial to the effective coupling between the display system and the user that the rotation be specified in the laboratory frame, that is, the frame of reference attached rigidly to the user. It is necessary that the manipulation of the globe, in the lab frame, be easily and naturally related to the rotation of the figure: when the user twists the globe to the right, he should see the figure rotating to the right.

2) Although differential rotations are additive, finite rotations are in general not. Thus, the order in which two rotational increments are applied effects the resultant position.

Thus, if we attempt to save the position of the figure in terms of 3 rotational angles, we must establish some

convention regarding the order of their application. Since the angles are finite, the cumulative rotation is different depending upon the order in which the rotational increments are performed, and in fact the axes of the latter rotations will be a function of the former rotations, so that the 3 angles will not in general represent independent rotations about axes fixed in the lab frame. We thus conclude that we cannot keep track of the rotations independently. It is necessary, at each rotational increment, to apply the increment to the composite position which the figure currently assumes, i.e., to the current composite rotation matrix.

Since, then, no very compelling alternatives have presented themselves, the scheme in present use updates the rotation matrix at each increment by a matrix-matrix multiplication. At each update operation, a matrix (3 by 3) representing the required incremental rotation is calculated, and the old matrix is replaced by its product with the incremental matrix. A loop is then entered in which the X and Y components of the product of each vector with the matrix are inserted in the corresponding positions in the display list.

9.2.5 Display

At the completion of the compilation of the display list, a program is called which causes the display hardware to cycle thru the display list at regular intervals. The operator may select one of several modes of display refreshment by the manipulation of console controls:

1) The normal mode of operation causes the display to be refreshed 20 times per second, if possible. This timing function is performed by a program within the supervisor, which sets a clock each time the display is restarted; the display is not restarted again until the clock registers 1/20 second. In the event that the allotted 1/20 second expires and the display has not completed the interpretation of the display list, then a flag is set and the display is restarted immediately when it is finished. In this mode, the display is not synchronized with the rotation phase; consequently, each pass thru the display list will generally encounter a portion of one rotational frame as well as a portion of the next

2) A mode of operation is provided in which the operation of the hardware is synchronized with the rotational phase, guaranteeing that no display frame includes portions of two rotational frames. This mode

substantially reduces the rate of display refreshment.

3) A mode of operation is provided in which both the rotation and the display hardware are synchronized with a movie camera. This mode insures that each film frame include exactly one rotational and display frame.

The timing of the display is complicated by the fact that the time spent in one cycle thru the display list by the hardware is very nearly that spent in the computation of a new rotational frame. To make matters worse, each of these times is subject to wide variation, effected by the nature of the specific data, as well as the particular I/O load of the machine during their execution. Consequently, the synchronization of the two requires that one phase (e.g., the rotation) be substantially completed before the other (e.g., the display) is started; the expense in terms of rate of display refreshment is therefore high. On the other hand, for most applications the distortion introduced by the asynchronous operation of the display and rotation is not objectionable. At slow rates of rotation, the consequent jumpiness of the ends of the figure is barely discernable.

The synchronization of the display with the rotation is accomplished by restarting the display only at the point in the rotation loop when exectly half of the vectors have been updated. In the synchronized mode, the rotation program is interrupted at this point, and a test is made to determine whether the display is running; if it is, then the rotation algorithm pauses until the display stops. When the display is stopped, it is restarted (at the beginning of the display list) and the rotation algorithm continues. When the rotation phase completes the computation of one rotational frame, it immediately goes on to compute another, assuming that it will not cross paths with the display before it reaches the halfway point. The success of this scheme is based on the assumption that neither phase will go twice as fast as the other, which seems safe.

9.3 Auxilliary Functions

Apart from the picture building functions of the satellite programs, their primary function is to simulate the operation of a conventional (teletype) terminal. This is largely a matter of translating messages between the BCD character set used by the large machine to the ASCII character set of the teletype, and in the process moving characters from the dataphone buffer to the teletype buffer and vice versa. However, provision is made for the user to type commands directly to the local (satellite) program; in particular, lines typed at the teletype beginning with the character # are interpreted as input to the local system and

cause control to be dispatched to a local command interpreter. This program recognizes the following command lines:

#SAVE n - where n is an integer between 1 and 34, causes the current 3-dimensional picture to be saved on microtape.

#SHOW n - causes a picture to be retrieved from microtape and displayed.

#EXIT - causes a local monitor system to be read from microtape and entered. This command is typed when the user is done with that satellite subsystem which is used for 3-dimensional displays, and wishes to engage another.

The SAVE and SHOW functions take an integer argument, specifying a microtape address where the picture is to be stored or read from. The microtape is divided into 34 equal fixed-length blocks, each of a size large enough to contain the data representing one picture, and which are numbered from 1 to 34. The data stored in each block includes the display list, the component and pointer arrays, and an integer specifying the number of 3-dimensional vectors in the picture; hence it contains all data relevent to the rotation and display of the figure. The current rotation matrix is <u>not</u> thus stored and retrieved, so that the SHOW command will not, in general, result in a projection of the figure which is exactly the same as that current at the time of the SAVE. This feature has some useful applications in cases where two different pictures are to be viewed in the same orientation.

X. Conclusion

In abstract terms, the ultimate goal of this project is to couple a man and a machine together in order to solve a problem whose solution is evidently not tractable by either alone. Clearly, there is a class of problems which may be solved more easily by present-day machines than by people; similarly, there is another class of problems which are obviously much better suited to humans than to any machines we can presently build. Our apparent hypothesis is that we may solve problems beyond these two classes by coupling the two methods. In particular, we attempt to take advantage of the chemical insight of an investigator by allowing him to interact freely with a model, while requiring the machine to attend to as many technical details as possible, thereby allowing the man greater freedom in his chemical pursuit. In the framework of our particular problem, we of course cannot expect the specific capabilities of the component problem solvers (i.e., the man and machine) to complement one another perfectly. In terms of molecular model building, for example, it is probably true that neither the man nor the machine is very well adapted to recognising and correcting collisions between space-filling parts of the model; as a result, this function provides a serious bottleneck, limiting somewhat the usefulness of the system. The acceptance of this and other such limitations is the price we pay in order to apply the power and flexibility of the machine to those aspects of the problem which it handles best.

Major differences between the implementation of the current system and that of earlier versions include the overall reorganization of the main processor programs at the top level, as well as some changes in program details. The segmentation of the system, motivated by limited core memory in the large machine, simultaneously relieves a serious restriction and forces us to think carefully about the relation between the various subsystems. In addition, the program segmentation probably provides a good background for the eventual operation of the package on the next generation of time sharing systems, which promise easier and more efficient solutions to segmentation problems.

Most of the technical aspects of the satellite display depart from previous implementations. The use of a remote processor, with limited computing power of its own, and the restricted data rate capacity of the interconnecting telephone line stimulate us to organize the system so as to minimize communications between processors. The use of a software scheme to perform the 3-dimensional rotation of vectors in the display list imposes further organizational restrictions at the satellite.

It seems unlikely that the whole range of technical detail presented in this paper will ever be of significant

interest to anyone, except as a reference document for the maintenance of the system which it describes. Such an interest would suggest the duplication of the current system, a project which is certainly not the most practical except in circumstances identical to those leading to this system's development, and even then would be hard to justify in the light of technical advances being made in both hardware and software. Rather, each of the technical discussions would seem most valuable when abstracted to its individually much wider area of applicability.

Appendix 1: References

Levinthal, Cyrus: Molecular Model-building by Computer; Scientific American 214,6 (June 1966)

Crisman, P.A., ed: The Compatible Time-Sharing System, M.I.T. Press, 1965

Appendix 2: Arithmetic Library

The following conventions will be used in the ensuing brief descriptions of mathmatical routines called by the package:

A,B,C will represent two dimensional floating point matrices. If not otherwise specified, the dimensions of each matrix are taken to be 3 by 3. The 3 by 3 matrix M is represented in MAD by:

M(1)	M(2)	M(3)
M(4)	M(5)	M(6)
M(7)	M(8)	M(9)

The zeroeth element of the vector representing a matrix is ignored.

V1,V2,V3 will represent vectors of floating point numbers. If not otherwise specified, the dimension of each vector will be 3. Again, the zeroeth element of vectors are ignored.

N will be an integer specifying the dimensions of vectors or matrices supplied as arguments.

X,Y will be floating point variables.

The following functions are all coded in FAP and are contained in PKG FAP as well as PKG or LIBE BSS:

MOVE.(V1,V2,N) - moves the first thru Nth elements of V2 to V1. The zeroeth elements are ignored.

X=NMAG.(V1,N) setx X to the magnitude of V1.

X=VMAG.(V1) same as X=NMAG.(V1,3)

NVADD.(V1,V2,V3,N) vector addition: sum of V2 and V3 replaces V1.

VVADD.(V1,V2,V3) same as NVADD.(V1,V2,V3,3)

NVSUB.(V1,V2,V3,N) vector subtraction: V1=V2-V3

VVSUB.(V1,V2,V3) same as NVSUB.(V1,V2,V3,3)

MATMUL.(A,B,C) matrix multiplication: A=B*C. Each matrix is assumed to be 3x3.

Y=NVDOT.(X,V1,V2,N) dot product of V1 and V2 is stored in X and Y.

Y=VVDOT.(X,V1,V2) same as Y=NVDOT.(X,V1,V2,3).

X=MDET.(A) sets X to determinant of 3x3 matrix A.

MVMLT.(V1,A,V2) matrix vector multiplication: V1=A*V2

UPDAT.(A,B) update a rotation matrix: A=A*B

MMMLT.(A,B,C) matrix matrix multiplication: A=B*C

NSCA.(V1,X,V2,N) scales a matrix or vector; scaler multiplication: V1=X*V2.

VSCA.(V1,X,V2) same as NSCA.(V1,X,V2,3)

NNRM.(V1,V2,N) normalize vector: NSCA.(V1,NMAG.(V2,N),V2,N).

VNRM.(V1,V2) same as NNRM.(V1,V2,3)

VVCRS.(V1,V2,V3) cross product: V1=V2*V2

TRNPX.(V1,A,V2,NROW,NCOL) where NROW is the dimension of V2 and the number of rows in the matrix A, and NCOL is the dimension of V1 as well as the number of columns of A: multiplies the vector V2 by a transposed version of the matrix A and puts the result in V1. A and V2 are not changed.

I=MTXINV.(A,N,EPS,DIRT) where N is the dimension of the square matrix A, DIRT is an array for temporary storage (dimensioned at least 3*N+1) and EPS is a small floating point number, specified by the calling program, to be used in the singularity test, inverts the matrix A and stores 0 in the integer I if the operation is successful, otherwise 1.

I=SYMINV.(A,N,EPS) inverts the symmetric matrix A. I, N, and EPS are as in MTXINV, above; the original configuration of A need not contain relevent values below the main diagonal.

SYMAK.(SYMA,A,NROW,NCOL) calculates, from the nonsquare matrix A, the square symmetric matrix AA' (for NCOL > NROW) or A'A (for NROW > NCOL) where A' is the transpose of A; the symmetric matrix is stored in SYMA, excepting those below the main diagonal.

The following functions take, as arguments, MAD lists of arbitrary length. Such argument lists may contain dot notation (e.g., A...B,A(5)... (33),B...7) as well as single

variables and constants. The argument lists of this type are specified as "LIST":

ZERO.(LIST) sets arguments to zero.

SPRAY.(X,LIST) sets arguments to X (which, of course, may be either fixed or floating but must be same as arguments in LIST).

X=MIN.(LIST) sets X to minimum of LIST. X and LIST may be fixed or floating, but not mixed.

X=MAX.(LIST) sets X to maximum value in LIST. X,LIST are fixed or floating, but not mixed.

X=MINABS.(LIST) sets X to minimum of absolute value of LIST. X and LIST fixed or floating but not mixed.

X=MAXABS.(LIST) sets X to maximum of absolute values of LIST. X and LIST fixed or floating, but not mixed.

The following functions relate specifically to molecular modelling, and most of them contain the problem-dependent program common declarations; the BSS decks are largely in NPKG BSS:

X=DIST.(Q1,Q2) where Q1 and Q2 are the indices (integers) of atoms (i.e., they point to entries in the X, Y, and Z arrays,) sets X to the distance between the atoms.

X=DISQ.(Q1,Q2) sets X to the squared distance between Q1 and Q2; this is much faster than DIST since it need not find the square root.

V1,Q1) translates the X, Y, and Z entries corresponding to atom Q1 by the vector V1: X(Q1)=X(Q1)+V1(1), etc.

VECROT.(A,Q1) rotates the X, Y, and Z entries corresponding to Q1 by the rotation matrix A.

VECTOR.(V1,Q1,Q2) sets the vector V1 to the difference between the coordinates of Q1 and those of Q2: V1(1)=X(Q1)-X(Q2), etc.

X=NRMANG.(Y) sets X to the normalized representation (between plus and minus pi) of the angle Y.

QBACK, JGLNUM, ANGLO, QAA, and TYPE(Q1) each unpack the corresponding component of the Q1 element of the ATOM array and return it as their value (an integer).

I=ATTRIB.(PROPERTY, A. ACID,RELATIVE.Q) is the primary source of chemical in the system. ATTRIB contains a number of internal tables, returning a value corresponding to a particular property (coded into PROPERTY) of a specific type of amino acid (coded into A.ACID) if the optional third argument is missing, or of a specific atom (RELATIVE.Q) of the amino acid type in cases which require the third argument. The property codes and the resulting values are summerized below:

ATTRIB.(0,*,I) returns the BCD name of the atom type whose code is I. The second argument is ignored.

ATTRIB.(1,A.ACID) returns the BCD name of the amino acid whose type code is A.ACID.

ATTRIB.(2,A.ACID) returns an integer specifying to what extent the amino acid type A.ACID is hydrophobic.

ATTRIB.(3,A.ACID) returns zero, +1, or -1 reflecting the charge on the amino acid whose type code is A.ACID.

ATTRIB.(8,A.ACID) returns the number of atoms in the residue of amino acid type A.ACID.

ATTRIB.(9,A.ACID) returns the number of variable residue angles in amino acid type A.ACID.

ATTRIB.(10,A.ACID,Q1) returns the code for the type of the Q1th atom in the residue of an amino acid type A.ACID.

ATTRIB.(11,A.ACID,Q1) returns the van der Waals radius of the atom Q1 in amino acid A.ACID.

ATTRIB.(13,A.ACID,Q1) returns the number of the last residue angle which effects the position of Q1 relative to the backbone in the amino acid A.ACID.

I=ANGVL.(X,Y,TYPE) returns zero (boolean false) if the angles X and Y are not a valid angle configuration for an amino acid of type TYPE. ANGVL uses an internal table (72 by 72 entries,

each of 1 bit) which specifies the regions of violation.

I=ANGCHK.(Q1) returns zero (boolean false) if the atom Q1 is a member of an angle pair assuming illegal values.

I=JGLABL.(TYPE,RELATIVE.ANGLE,RELATIVE.Q) returns boolean false if changing residue angle RELATIVE.ANGLE moves atom RELATIVE.Q with respect to the backbone.

RECOV.() is the uniform recovery procedure from fatal errors within the package. The default RECOV. which resides in NPKG BSS simply prints a message and returns to the error entry of the first link.