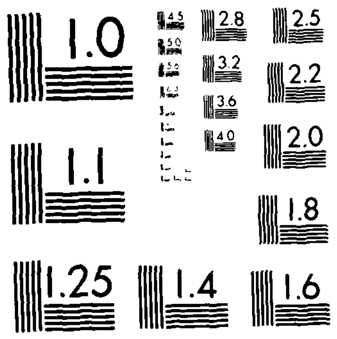
AD-A156 955   THE DATABASE UNIFORMIZATION PROBLEM: A SEED/DAVID   1/2
INTERFACE(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON
AFB OH J AULINO 1985 AFIT/CI/NR-85-44T

UNCLASSIFIED   F/G 9/2   NL


1.0
4.5
5.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

THE DATABASE UNIFORMIZATION PROBLEM: A SEED/DAVID INTERFACE

by
Joseph Aulino

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1985

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 85-44T	2. GOVT ACCESSION NO. AD A156 455	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Database Uniformization Problem: A Seed/David Interface		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Joseph Aulino		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: University of Maryland		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433		12. REPORT DATE 1985
		13. NUMBER OF PAGES 158
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17

[illegible]

LYNN E. WOLAVER
Dean for Research and
Professional Developmen
[illegible] AFIT, Wright-Patterson AFB O

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT

Title of Thesis: The Database Uniformization Problem: A SEED/DAVID Interface

Joseph Aulino, Master of Science, 1985

Thesis directed by: Dr. A. K. Agrawala
Professor
Department of Computer Science

Abstract: The database uniformization problem deals with creating a common user interface for a collection of heterogeneous databases. The DAVID System is a database management system which is being implemented by the National Aeronautics and Space Administration to create such an interface. This paper deals with the implementation of certain routines used by the DAVID System to interface with SEED, a Codasyl database management system.

DEDICATION:

**This is dedicated to my wife Janet and my son Andrew.
With all my love...**

ACKNOWLEDGEMENT:

This thesis would not have been possible without the continual support and guidance of Dr. Barry E. Jacobs, NASA GSFC, Dr. A. K. Agrawala, University of Maryland, Dr. John Welch, Catholic University, and Dr. Isadore Brodsky, University of Puerto Rico at San Juan.

Table of Contents

LIST OF FIGURES

LIST OF FIGURES (cont)

CHAPTER 0. INTRODUCTION

0.0 THE UNIFORMIZATION PROBLEM

The database uniformization problem is: given a collection of distributed, heterogeneous databases, build a common user interface facility that will support uniform access to the set of databases [12]. Solving this problem would allow a user to access all of the information in the databases without learning the different data manipulation languages. Further, it should leave intact component databases and their existing software [10].

0.1 SURVEY OF THE LITERATURE

Many projects have focused on the database uniformization problem. Most of these deal mainly with the global data manager. At UCLA, projects by Cardenas [2] and Chu [3] use the Entity–Relationship (E–R) model as the global model. Two other projects using the E–R model are Nihon–Systemix's Distributed Multi–Data Base [14] and INRIA's Sirius–Delta [6].

There have also been projects dealing with remote, heterogeneous, database management systems (DBMS). Two of these, CSIN [4] and COSYS [1] may have been implemented. Designed at the University of Grenoble, COSYS (COoperation SYStem) is not extensively documented. On the other hand, CSIN (Chemical Subetances Information Network) is a fairly well described system. One feature of CSIN is that some of its component systems are not general DBMSs. This causes some data to be accessible only through direct contact with the local DBMS. It also resulted in a global model which is less than the union of the component models. Finally, [7] deals with eleven projects on uniform access to distributed DBMSs. Six of these deal with interconnecting heterogeneous systems. The remaining five cover the simpler problem of interconnecting homogeneous systems based on the relational model.

0.2 THE DAVID SYSTEM

DAVID (Distributed Access View Integrated Database) is a system allowing users to uniformly access distributed heterogeneous databases [9]. Its development is funded by the National Aeronautics and Space Administration. Basically, DAVID is a database management system (DBMS) built atop existing DBMSs and file management systems. Its front end is Database Logic (DBL).

Simply put, DBL can be thought of as a uniform layer placed on heterogeneous databases. In this way, it is used as a framework for solving the database uniformization problem. DBL is used to construct a "Global Data Manager" (GDM). It is the GDM which allows users the uniform access to the distributed heterogeneous databases [12].

To accomplish its tasks, the GDM is divided into 8 basic processes. These processes are:

1. The Global Model;
2. External to Conceptual Mapping;
3. Global Dictionary;
4. Report Generation;
5. External to Conceptual Translation;
6. Query Execution, Assembly, and Loading;
7. Process Control and Communication;
8. Local Data Manipulation Language (DML) Dependent Processes.

The exact purpose of the different parts is explained in [10]. This thesis deals with Local DML Dependent Processes. In particular, interfacing SEED (a Codasyl DBMS) with DAVID.

The Local DML Dependent Processes are programs written in local DMLs. These processes interface the GDM and local databases. The main functions of these processes is to scan the databases, extract information from the databases, enter data into DAVID's own DBMS, test currently scanned records to see if they satisfy some boolean condition, and transaction processing.

0.3 SEED INTERFACE FOR DAVID

This thesis deals with implementing certain basic routines needed to create the Local DML Dependent Processes for interfacing DAVID with a Codasyl DBMS. The specific DBMS selected was SEED [8]. SEED is a commercially available Codasyl DBMS used by the National Aeronautics and Space Administration (NASA).

To create the interface, it is only necessary to identify databases and create certain basic routines. These routines are described in section 1.3. The routines created here are the basis from which the basic routines for interfacing are made [10]. They allow (in a simple manner) creation of routines to satisfy the requirements of Local DML Dependent Processes.

The work presented here deals with three basic areas essential to the task of creating the interface. The first area is "Schema Conversion". This requires creating a SEED schema given a DBL schema (these terms are explained in the next chapter). The next area is "Query Translation". This refers to the creation of certain basic routines which will make it possible to convert from a DAVID query to a SEED query. Lastly, is "Transaction Translation". This is the creation of routines needed to process transactions on a SEED database given a transaction on a DAVID database.

0.4 AN OUTLINE OF THE THESIS

Chapter 1 gives the particulars on this thesis. This includes examples of a DBL Data Description Language (DDL) Definition and a DBL Schema. This chapter defines certain terms used through the thesis, explains in greater detail the primitive operations needed to implement the DAVID/SEED interface, and describes certain programming tools used. The remainder of the thesis deals with the specifics of the implementation. Chapter 2 details the implementation of the "Schema" conversion. Chapter 3 details the implementation of the automatic generation of code used in translating queries on the database. Chapter 4 details the implementation of the automatic generation of code used in translating transactions on the database.

record owns a recursive occurrence of another record. In this way, special actions which need to take place can do so.

After this, control returns to the "relate" routine. Here, the correct value of the "typr" field for each record is finally established. This is done by counting the number of "hook" structures pointing to a "table" structure and putting this number in the "typr" field of that "table" structure. The "areset" routine is then called which performs its function as explained above. The numbers put in the "rcnt" field of the table are used to differentiate variables of recursive occurrences of a record from the record itself. By using increasing numbers, a record may have several recursive occurrences, but variables for each will be unique using the naming system described in section 1.2.

This completes the creation of the schema data structure. A graphic representation of the data structure for the recursive network of figure 1.1 appears in figure 1.7. The relationships established here model the DBL and SEED schemas. Having completed the formation of the schema data structure, control returns to the calling program and the process of schema conversion can begin.

"relat" field indicating its selection will be made through one of its owners (it is a child). In this system, once a record has been marked for selection as a "parent" it cannot be changed (thus selecting things the most efficient way). On the other hand, something marked with a "C" may be changed often and eventually marked with a "P" if possible. There is also the possibility that a table structure will have no pointer in its "for_write" field and no value in its "relat" field. This indicates that the record is owned by "SYSTEM" and selection will come via "SYSTEM".

While this is being done, the typr field of each table structure with a typ field of "R" is incremented. Once this is done, recursive relationships are established and counted. To do this, non–zero typr fields of all table structures are set to 1, and the "recurfind" (for "recursive find") routine is called. In recurfind, the schema structure is again traversed. It looks for table structures which have a "typr" field of 1. It then looks for another table structure meeting the following criteria. First, the second table structure is pointed to by one of the "hook" structures in the list of hook structures pointed to by the "for_sets" field of the current "table" structure. Second, the first table structure is pointed to by one of the "hook" structures in the list of "hook" structures pointed to by the "for_sets" field of the second "table" structure. If these two criteria are met, the second "table" structure must represent the record used to link the record and its recursive occurrence. So in this case, a "hook" structure is allocated. Its "atab" field is set to point to the first "table" structure. The hook record itself is then made part of a list of hook records pointed to by the "rhook" field of the second record. Later, when the schema structure is being traversed for the creation of the schema and various of the Fortran programs, the fact that the "rhook" field =/= 0 will indicate that this

The records section of the DBL schema could actually be written in several ways, each of which would change the way records were selected. Suppose the sets were listed in the DBL schema as "A", "B", "C". The "A" record would be selected via "SYSTEM_A", "B" would be selected via "SYSTEM_B", and "C" would be selected through "B_C". The point being that records are going to be selected in the order they appear in the DBL schema. So if they are listed in the order "A", "C", "B". "A" will be selected as before, "C" will now be selected via the set "A_C", and "B" will be selected via "B_C" where it will be selected as a parent of "C". Selecting records as parents is advisable (in SEED) whenever possible as each record can have only one of each type of parent. Selecting in this way can then save search time. This is the general plan for how selects will be done. What the relate routine does is establish the relationships necessary to implement this plan.

The relate routine accomplishes the above by keeping track of where records occur in table structures in relation to where they appear in item structures. If a record name appears in an item structure, and the table name that record appears in occurs before the table structure this item structure is attached to, in that case, and that case only, the record will be selected through one of its children. Hence, a pointer is established from the "for_write" field of the current table structure, to the table structure containing the name of the record found in the item structure. Also, the table structure pointing to the item structure will have its "relat" field marked as "P". This indicates it is the parent of the record through which selection will be made. If a record name appears as a field of another record, and the table structure containing that name appears after the current table structure, the later table structure is marked with a "C" in its routines are related in that "relate" calls "recurfind" to make some of the needed relationships. These relationships include determining which records are members and which are owners of sets, which records own recursive occurrences of other records, and the number of recursive relationships a record is a member of. The basic formula for doing this is to traverse the part of the schema structure already built and create these relationships.

To do this, the schema is traversed table structure by table structure and item structure by item structure. If an item structure has type "T" or "R", the schema structure is re-searched until the table structure corresponding to it is found. The original table structure is then added to the list of structures which owns the current table structure. This is done by allocating a hook structure whose "atab" field points to the original table structure. If this is the first set to which the current table structure is a member, the "for_write" field of the table structure then points to the "hook" structure just allocated. Otherwise, the hook structure is pointed at by the "nhook" field of the last "hook" structure in the list of "hook" structures pointed at by the "for_sets" field of the current "table" structure. By doing this, all set relationships are established.

As the structure is traversed, it must also be determined through which set the query on any particular record will be made. The single path traversed to any record (though many may exist) will always be found in the same way. This is derived from the DBL schema. A record will always be selected through some set, it may participate in this set as a member or owner. Most commonly selection will be through an owner. Selection can be made through a member in the case of a "Y" network as in figure 1.6.

Here, the sets are "SYSTEM_A", "SYSTEM_B", "A_C" and "B_C".

certain of the table structures to represent particular relationships between records. The particular fields of the table structures represent particular things. The different representations are found in figure 1.5.

The "parser" routine reads in the DBL schema. As it does so, it stores the database name in the "dname" variable and the view name in the "schem" variable. Then it begins parsing the remainder of the schema. As it comes to each record name, it allocates a new "table" structure and puts the record name into the structure. Next it reads the field names for the particular record. It allocates an item structure filling the appropriate elements of the structure as it does so. Each table structure is linked to the list of the previously allocated table structures via the "ntable" field of the last table structure. The "item" structures are linked to the list of item structures pointed to by the the "to_items" field of the appropriate table structure. Also, in the "parser" routine, if a field has a key entry of "1", a special pointer is assigned to it so that when the SEED schema is created later this field can easily be found. It is pointed to by the "to_key" field of its "table" structure or by the "nkey" field of the last key field read in for this record. When the entire DBL schema has been input, the above has been completed for all records of this database.

The "parser" routine then returns control to the main program, returning as it does so a pointer to the first "table" structure allocated. By following the pointers emanating from this structure all other "table" and "item" structures can be reached. It is this combination of "table" and "item" structures that (along with the "hook" structures to be assigned in the "relate" and "recurfind" routines) make up the "schema" structure.

The relationships which make up the remainder of the schema structure are created in the "relate" and "recurfind" routines. These an "A". Here too, the same rules hold for forming variables for recursive occurrences of records.

Having explained the naming conventions of variables, explanation of the "declarer" and "commons" routines follows. "declarer" creates the Fortran declaration of the counter variables used in the Fortran subroutines. These variables are created according to the formula described above. The routine keeps track of the length of the variables it creates and generates continuation lines where necessary.

The "commons" routine is used to declare the common statement used by every Fortran routine created. As in the "declarer" routine, this routine keeps track of variable and line lengths and generates continuation lines where necessary. Every record name is in this statement, every set name is in it, every logical pointer is in it, and all currency pointers are in it. The routine creates its variables by traversing the schema data structure and outputting the appropriate declarations as it progresses.

This leaves only the "parser", "relate" and "recurfind" routines to be explained here. The "parser" routine takes a DBL schema from standard input and puts it into the schema data structure, building the basics of the structure as it goes along. To explain how this is to done, requires understanding the schema data structure.

The schema data structure is made up of three separate structures; the "table" structure, the "item" structure, and the "hook" structure. The purpose of the entire structure is to represent a DBL schema. Each of the separate structures also has a particular purpose. The individual structures are as declared in figure 1.4. The table structure represents one record of the database, the "item" structure represents a field of a record (either a table field or an entry field), and the "hook" structure is to "connect" set name. These variables are formed by appending the name of the member record to the name of the owner record and separating the two by a "_". As an example, in the HDBP database, the "DOCTOR" record is owned by the "HOSPITAL" record. These then form the set variable "HOSPITAL_DOCTOR". For those records owned by "SYSTEM", the word "SYSTEM" replaces the set name of the owner. One of these types of variables is created for each record, non_record field name, and for each set defined in the database. The SEED processor automatically defines these variables in the UWA when processing the SEED sub-schema.

There are some other variables which also have to be declared. The first of these are the counter variables. These are used to count the of number records queried before selecting one. Their final value can then be used as a logical pointer to a record. For those records which have no recursive occurrences, these variables are named by appending to the letter "C" (for counter) the name of the record type they are to count. So for the "HOSPITAL" record, the counter variable is "CHOSPITAL". If a record has a recursive occurrence, then a number is also appended to the end of the variable name. The number comes from the "rcnt" field which would then be reduced. Hence, if another recursive occurrence has a counter variable, it would then get the next lower number. If "rcnt" = 0, no number is appended to the end. The way the "typr" field is assigned values guarantees there will be as many numbers as occurrences of a record. As an example, were there a recursive occurrence of the "HOSPITAL" record its counter might be named "CHOSPITAL1".

The variables to hold the currency pointers are formed in a similar manner, the only difference being the first letter. Here the first letter is an "N". Logical pointers are also formed in the same way with the first letter append a number to the end of a variable in the Fortran routine. This is done due to the way certain variable names are created (explained in detail below). The "transf" routine generates code for switching the value of two variables. In certain of the automatically generated Fortran routines, there is the chance that the value of certain variables will be changed by "accident". This routine is used to generate code to specifically prevent this.

Two other important routines are the "gotowrite" routine and the "gorite" routine. These routines generate the line numbers used to create Fortran computed "go to" statements. "gorite" generates the line numbers only. This is used by several routines. When used alone, the calling routine must generate the words "GO TO" and the ending of the statement (ie ")" and the variable controlling the statement). This routine needs starting values in all variables, however, once started it will update all values as needed. The routine automatically generates the correct continuation lines where necessary.

The "gotowrite" routine is used to generate a complete computed go to statement for the "outpt" routine (discussed later). It traverses the schema structure and generates one line number for every field in the database, every record in the database, every currency pointer in the database, and every logical pointer in the database. This is done because any and all possible information in the database may have to be output. For more complete information, see sections 3.3 and 3.4.

There are certain variables needed to ensure all parts of the database can be referenced correctly. First there are the variables representing records and fields of those records (non–table or recursive occurrence fields). These variables are the names of the records and fields as given in the DBL schema. The next most basic type of variable is the variable representing a these basic "building block" routines. For more information on the primitive processes and how they are used see [10].

1.4 SOME COMMON TOOLS

The main routines used to implement this part of the DAVID project are detailed in the appropriate sections of this thesis. However, there are some routines basic to the other routines. These are detailed here and referred to throughout the thesis.

Two basic routines for inputting data are the "namer" routine and the "getint" routine. The namer routine takes in a pointer to a string and as a "side-effect" enters a string in the location "pointed to" by the pointer. This string comes from standard input. The routine returns the first character from standard input after the string. The routine recognizes certain string terminators in the standard input, these are: " ",";", and "#". The blank is the normal string terminator. The ";" terminates one record's entry in the DBL schema. The "#" is used to terminate the entire DBL schema. The "getint" routine reads an integer from standard input and returns it to the calling program. This routine changes the standard input to its integer representation.

Another function which returns an integer value is "how_big". It takes a pointer to a string as input and returns the length of the string. The "eqname" routine is another string oriented routine. It takes as input, pointers to two strings. It returns a "1" if the strings are identical; "0" otherwise.

The next two routines are the "areset" and the "transf" routines. "areset" sets the "rcnt" field of all table structures in the schema structure (explained below) equal to their "typr" field. The "rcnt" field is used to schema, it is possible to express any of the three types of databases considered here. From this then, certain Fortran routines are made. These routines make it possible to query the database, insert, modify, and delete information, and generate the SEED schema and sub–schema.

1.3 PRIMITIVE PROCESSES

In implementing this interface, only certain operations need to be implemented. These are called "primitive processes" [10]. Anything which can be done in a DBMS can be built up from these processes. The processes applicable to SEED are:

1. Project query;
2. Selection query;
3. Selection–Projection query;
4. Store–To–Database query;
5. Semijoin and Join query.

The project query is having certain specific values "output" (in some way) from each type of record in the database. For some records the output may be null. The select query outputs every value from each record of the database meeting certain criteria. Select–project queries output certain specific values from each type of record in the database satisfying certain criteria. Numbers 4 and 5 above, don't have to be implemented for SEED, but the output from numbers 1, 2, and 3 must be in a form which can be used by DAVID.

These routines can be implemented using more basic tools. This thesis deals with the creation of these more basic tools. These include procedures for sequentially traversing a database, retrieving records from a database based on currency pointers for those records, outputting information in a way which facilitates interfacing with DAVID, and procedures for transaction processing. The remainder of this thesis deals with the implementation of

In addition to the SEED schema, there is the DBL schema. The DBL schema is the construct which was derived from the DBL DDL Definition [10] (see figure 1.0 and figure 1.1). A DBL schema is not normally used by a user of DAVID. DAVID uses a different construct called the DEFINE CLUSTER [11]. The DBL schema is a special construct used to convey the definition of the database to the DBMS. A DBL schema contains the name of the database, a view name for the database, and the actual database definition.

Note how the ownership of records is expressed. The "DOCTOR" and "BEDNR" records are expressed as fields of the "HOSPITAL" record. Also the "PATIENT" record is expressed as a field of both the "DOCTOR" and "BEDNR" tables. This indicates that these tables are "owned" by the records in which they appear as fields. The first two names are the database and view names respectively. The first name in each row is the table name. This is followed by the field names and their descriptions. The description of any field starts with the field name. This is followed by one of a "T", an "R", a "C", an "I", or an "F". If followed by a "T" or an "R", the field represents a table owned by this particular record. The "R" means the table has a recursive occurrence somewhere in the database. If the field is a table, then there is no more to the entry. The "C" indicates a character field, the "I" indicates an integer field, and the "F" indicates a real field. Whenever any of these three characters appear, they are followed by 2 integers. The first is the size of the field and the second is always a "1" or "0". If a "1", it indicates the field is to be a key field, if a "0", it indicates the field is not a key field. (Key field here refers a record being sorted on this field.).

It is the DBL schema on which everything else is based. Using this

DBMS, this type of record is said to be "system" owned. This is usually represented by having the word "SYSTEM" somehow associated with the set.

Another type of structure here is the recursive record. A recursive record is one which is owned by a record of the same type as itself. In most if not all Codasyl DBMS, this ownership must be indirect [9]. That is, a record of type "A" will have to own a record of type "B" which in turn can then own a different record of type "A". Figure 1.1 is an example of this. Here, a record of type COURSES owns a record of type PREREQ. The record of type PREREQ then owns a record of type COURSES. The first record of type COURSES is referred to as the "original" occurrence and the record of type COURSES owned by the record of type PREREQ is referred to as the "recursive" occurrence.

Another important aspect of the network DBMS is the schema. The schema defines the database for the DBMS. It allows the DBMS to set–up the data structures and the storage facilities necessary to construct the database [8]. It can be thought of as a template for the Database. Figure 1.2 is the SEED schema defining the HDBP database of Figure 1.0.

In addition to the schema, there is the sub–schema. The sub–schema determines exactly what part of the database will be available to the user at any one time. The sub–schema corresponds with the external view of the database [8]. It can be changed each time the database is referenced. Any part or all of the database can be made available for referencing. In figure 1.3, there is an example sub–schema used to reference all records and sets of our example database.

In SEED, the schema and sub–schema are processed by a pre–processor. From the sub–schema, the pre–processor creates the United Working Area (UWA) [8].

various points. The differences between a DBL DDL Definition and a DBL Schema are illustrated in the next section.

1.2 DEFINITIONS

The most basic term here is database. A database is nothing more than a collection of data [5]. A database management system (DBMS) is an automated method for referencing this data. In the course of this thesis, reference will be made to 3 distinct type of DBMS. These are the relational DBMS, the hierarchical DBMS, and the network DBMS.

A relational DBMS can be thought of as a table whose rows are referred to as "tuples" and columns as "attributes" [15]. In the hierarchical approach, data is represented as a simple tree where the "root" and each "leaf" are different record types. Each "leaf" record type is "owned" by some other record type. Owning can be thought of as a path from one record to another. While a record type may own any number of other record types, each record type can have only one owner. Lastly, comes the network approach. In this approach, data is represented as records and links. Each record type is linked to some other record type in a set relationship. As in the hierarchical approach, one record can be thought of as owning another set. In the network approach, any record can have any number of owners and/or members. Each single record may own many other records of one type, hence "linking" those records together [9].

This thesis deals with the network case. There are several terms and concepts which are unique to the network case. For instance, the "singular set". A singular set is one owned by the DBMS itself. The purpose for this is to allow a database to have only one type of record and to link different instances of this type of record together [15]. In most network

CHAPTER 1. PRELIMINARIES

1.0 INTRODUCTION

This chapter covers preliminary information needed for the remainder of the thesis. Section 1.1 gives an example of a DBL DDL Definition and DBL schema derived from this definition. This is because the DBL schema serves as the basic input in this implementation. In section 1.2, certain basic definitions are given. Some of these definitions are basic database definitions, some are definitions for terms unique to this thesis. There are certain primitive operations from which the entire interface can be constructed. Those operations, a brief explanation of them, and how they relate to this work are given in section 1.3. Lastly, section 1.4 deals with some basic C routines and data structures referred to throughout this thesis.

1.1 EXAMPLES

This section gives examples of DBL DDL Definitions [9] and the DBL schemas derived from them. It is the DBL schema which is the input to this implementation and which is "operated" on to get the required output. There are two examples given here. These will be referenced throughout the work as basic examples. The first example is of a "Y" network. Figure 1.0a gives the DBL DDL Definition and figure 1.0b gives the DBL schema. The DBL schema is a stylized form of the Language and View Definitions of the DBL DDL Definition.

The second example is another DBL DDL Definition and its DBL schema. In this case, the example is for a network with a recursively owned record. The example appears in figure 1.1a and figure 1.1b.

These two examples will be used throughout the thesis to illustrate

```
V(HDBP): VIEW DEFINITION
 S(HDBP):  SCHEMA DEFINITION
           TABLE HOSPITAL = (NAME,CITY,ZIP,DOCTOR,BEDNR)
           TABLE DOCTOR = (DRNAME,DRNR,PATIENT)
           TABLE BEDNR = (WARD,PATIENT)
           TABLE PATIENT = (PNAME,DISEASE,AGE,SEX,VITS)

 L(HDBP):  LANGUAGE DEFINITION
    T(HDBP):  TYPING DEFINITION
           TYPE NAMTPE = {NAME,DRNAME,PNAME} EBCDIC CHAR(30)
           TYPE CITYP  = {CITY} EBCDIC CHAR(30)
           TYPE ZP#TPE = {ZIP} EBCDIC DEC(9)
           TYPE DR#TPE = {DRNR} EBCDIC DEC(5)
           TYPE BED#TPE = {BEDNUM} EBCDIC DEC(1)
           TYPE WARDTPE = {WARD} EBDIC CHAR(20)
           TYPE DISTPE  = {DISEASE} EBCDIC CHAR(20)
           TYPE AGETPE  = {AGE} EBCDIC DEC(3)
           TYPE SEXTPE = {SEX} EBCDIC CHAR(1)
           TYPE VITPE  = {VITS} EBCDIC CHAR(80)


    NONLOGICAL SYMBOLS DEFINITION
       PREDICATE SYMBOLS

       HOSPITAL-DOCTOR-BEDNR-PATIENT:
          (NAMTPE,CITYP,ZP#TPE,DOCTOR,NAMTPE,DR#TPE,
           BEDNR,BED#TPE,WARDTPE,PATIENT,NAMTPE,DISTPE,
           AGETPE,SEXTPE,VIIPE) - FULL CLUSTER
                                PREDICATE SYMBOL

       FUNCTION SYMBOLS
          NONE

 C(HDBP):  CONSTRAINTS DEFINITIONS
    C(HDBP,1): CONSTRAINT HOSPITAL:
          ZIP-->CITY
    C(HDBP,2): CONSTRAINT DOCTOR:
          DRNR-->DRNAME
    C(HDBP,3): CONSTRAINT BEDNR:
          WARD,BEDNUM-->PATIENT
    C(HDBP,4): CONSTRAINT PATIENT:
          PNAME-->DISEASE,AGE,SEX VITS
```

Figure 1.0a A DBL DDL Definition for Database HDBP.

```
HDBP ONE
HOSPITAL NAME C 30 1 CITY C 30 0 ZIP C 5 0 DOCTOR T BEDNR T;
DOCTOR DRNAME C 30 1 DRNR I 5 0 PATIENT T;
BEDNR BEDNUM I 10 1 WARD C 20 0 PATIENT T;
PATIENT PNAME C 30 1 DISEASE C 20 0 AGE I 3 0 SEX C 1 0
                                              VITS C 80 0;#
```

Figure 1.0b A DBL Schema for Database HDBP.

```
V(CATALOGUE): VIEW DEFINITION
 S(CATALOGUE):  SCHEMA DEFINITION
          TABLE DEPT = (DNAME,DNR,COURSES)
          TABLE COURSES = (CNAME,CNR,CREDS,PREREQ)
          TABLE PREREQ = (COURSES,FILLER)

 L(CATALOGUE):  LANGUAGE DEFINITION
     T(CATALOGUE):  TYPING DEFINITION
          TYPE DCNTPE = {DNAME,CNAME} EBCDIC CHAR(10)
          TYPE DC#TPE = {DNR,CNR} EBCDIC DEC(3)
          TYPE CRDTPE = {CREDS} EBCDIC DEC(1)
          TYPE FILTPE = {FILLER} EBCDIC CHAR(1)

     NONLOGICAL SYMBOLS DEFINITION
         PREDICATE SYMBOLS
         DEPT-COURSES-PREREQ:
             (DCNTPE,DC#TPE,COURSES,DCNTPE,DC#TPE,
              CRDTPE,PREREQ,COURSES,FILTPE)-FULL CLUSTER
                                          PREDICATE SYMBOL

         FUNCTION SYMBOLS
            NONE

 C(CATALOGUE):  CONSTRAINTS DEFINITIONS
     C(CATALOGUE,1): CONSTRAINT DEPT:
            DEPT-->CATALOGUE
            DNAME-->DNR
     C(CATALOGUE,2): CONSTRAINT COURSES:
            CNR-->CNAME,CREDS
```

Figure 1.1a A DBL DDL Definition for a Database with a Recursive Occurrence.

```
CATALOGUE SCHED
DEPT DNAME C 3 1 DNR I 3 0 COURSES R;
COURSES CNAME C 10 1 CNR I 3 0 CREDS I 1 1 PREREQ T;
PREREQ COURSES R FILLER C 1 0;#
```

Figure 1.1b The DBL Schema for the DBL DDL Definition of Figure 1.1a.

```
SCHEMA NAME IS HDBP.
AREA NAME IS BASIC_DATA_AREA.
AREA NAME IS POINTER_DATA_AREA.

RECORD NAME IS HOSPITAL
   LOCATION MODE IS VIA SYSTEM_HOSPITAL
   WITHIN BASIC_DATA_AREA.
      NAME TYPE IS CHARACTER 30.
      CITY TYPE IS CHARACTER 30.
      ZIP TYPE IS CHARACTER 5.

RECORD NAME IS DOCTOR
   LOCATION MODE IS VIA HOSPITAL_DOCTOR
   WITHIN BASIC_DATA_AREA.
      DRNAME TYPE IS CHARACTER 30.
      DRNR TYPE IS INTEGER.

RECORD NAME IS BEDNR
   LOCATION MODE IS VIA HOSPITAL_BEDNR
   WITHIN BASIC_DATA_AREA.
      BEDNUM TYPE IS INTEGER.
      WARD TYPE IS CHARACTER 20.

RECORD NAME IS PATIENT
   LOCATION MODE IS VIA DOCTOR_PATIENT
   WITHIN BASIC_DATA_AREA.
      PNAME TYPE IS CHARACTER 30.
      DISEASE TYPE IS CHARACTER 20.
      AGE TYPE IS INTEGER.
      SEX TYPE IS CHARACTER 1.
      VITS TYPE IS CHARACTER 80.

RECORD NAME IS DNHDBP
   LOCATION MODE IS VIA SYSTEM_DNHDBP
   WITHIN POINTER_DATA_AREA.
      DHDBP TYPE IS INTEGER.
      NHOSPITAL TYPE IS INTEGER.
      AHOSPITAL TYPE IS INTEGER.
      NDOCTOR TYPE IS INTEGER.
      ADOCTOR TYPE IS INTEGER.
```

Figure 1.2 Page 1 of 3.

```
        NBEDNR TYPE IS INTEGER.
        ABEDNR TYPE IS INTEGER.
        NPATIENT TYPE IS INTEGER.
        APATIENT TYPE IS INTEGER.

SET NAME IS SYSTEM_HOSPITAL
    MODE IS CHAIN LINKED PRIOR
    ORDER IS SORTED
    OWNER IS SYSTEM
    MEMBER IS HOSPITAL MANDATORY AUTOMATIC
        LINKED TO OWNER
        ASCENDING KEY IS NAME
        SET SELECTION IS THRU CURRENT OF SET.

SET NAME IS HOSPITAL_DOCTOR
    MODE IS CHAIN LINKED PRIOR
    ORDER IS SORTED
    OWNER IS HOSPITAL
    MEMBER IS DOCTOR MANDATORY AUTOMATIC
        LINKED TO OWNER
        ASCENDING KEY IS DRNAME
        SET SELECTION IS THRU CURRENT OF SET.

SET NAME IS HOSPITAL_BEDNR
    MODE IS CHAIN LINKED PRIOR
    ORDER IS SORTED
    OWNER IS HOSPITAL
    MEMBER IS BEDNR MANDATORY AUTOMATIC
        LINKED TO OWNER
        ASCENDING KEY IS BEDNUM
        SET SELECTION IS THRU CURRENT OF SET.

SET NAME IS DOCTOR_PATIENT
    MODE IS CHAIN LINKED PRIOR
    ORDER IS SORTED
    OWNER IS DOCTOR
    MEMBER IS PATIENT MANDATORY AUTOMATIC
        LINKED TO OWNER
        ASCENDING KEY IS PNAME
        SET SELECTION IS THRU CURRENT OF SET.
```

Figure 1.2 Page 2 of 3.

```
SET NAME IS SYSTEM_DNHDBP
    MODE IS CHAIN LINKED PRIOR
    ORDER IS NEXT
    OWNER IS SYSTEM
    MEMBER IS DNHDBP MANDATORY AUTOMATIC
        LINKED TO OWNER
        SET SELECTION IS THRU CURRENT OF SET.
```

Figure 1.2 Page 3 of 3.

Figure 1.2 An Example SEED Schema.

```
SUB-SCHEMA NAME IS ONE FOR FORTRAN OF SCHEMA HDBP.
RECORD SECTION.
   COPY ALL RECORDS.
SET SECTION.
   COPY ALL SETS.
```

Figure 1.3 An Example SEED Sub–Schema.

```
struct hook {
    struct hook *nhook;
    struct table *atab;
};


struct table {
    char aname[30];
    char relat;
    int typr;
    int rcnt;
    int nr;
    struct table *ntable;
    struct table *for_write;
    struct hook *for_sets;
    struct item *to_items;
    struct item *to_key;
    struct hook *rhook;
};


struct item {
    char iname[30];
    char typ;
    int size;
    struct item *nitem;
    struct item *nkey;
};
```

Figure 1.4 Declarations of the C structures.

Explanation of the General Table Structure:

1.	**aname**	– A PARTICULAR TABLE'S NAME.
2.	**relat**	– THE SET RELATIONSHIP TO THE TABLE IN THE SAME SET USED TO FETCH THIS TABLE. VALUES MAY BE: a) "P" – PARENT. b) "C" – CHILD.
3.	**typr**	– VALUE = "0" : A NON-RECURSIVE TABLE; VALUE > "0" : A RECURSIVE TABLE WITH "TYPR" RECURSIVE RELATIONSHIPS.
4.	**rcnt**	– USED AS A TEMPORARY STORAGE FOR TYPR THE VALUE HERE IS DECREASED EACH TIME THE TABLE IS USED IN ONE OF ITS RECURSIVE RELATIONSHIPS AND IT IS RESET TO TYPR WHEN IT REACHES 0; SEE THE SUBROUTINES "**firstwhere**" AND "**areset**" TO SEE PRECISELY HOW IT'S USED AND RESET.
5.	**nr**	– KEEPS TRACK OF A RECORDS POSITION IN RELATION TO OTHER RECORDS. THIS IS USED TO HELP DETERMINE IF A RECORD OWNS OR IS OWNED BY ANOTHER RECORD.
6.	**ntable**	– A LINK TO THE NEXT TABLE IN A LIST OF TABLES IN THIS PARTICULAR CLUSTER.
7.	**for_write**	– USED TO LINK THE TABLE TO A TABLE IT HAS A SET RELATIONSHIP WITH. IT WILL BE FROM THIS SET THAT ALL RECORDS OF THIS TABLE WILL BE SELECTED.
8.	**for_sets**	– USED TO POINT TO A "HOOK" STRUCTURE TO HOOK A TABLE TO EVERY OTHER TABLE IT IS INVOLVED IN A SET WITH.

Figure1.5 Page 1 of 3.

9. **to_items** – POINTS TO A LIST OF ITEM STRUCTURES WHICH WILL REPRESENT THE FIELDS OF THIS TABLE.

10. **to_key** – POINTS TO THE KEY FIELD FOR THIS TABLE IN THE LIST OF ITEM STRUCTURES. THIS FIELD MAY IN–TURN POINT TO A SECONDARY KEY FIELD WHICH IN TURN MAY POINT TO A THIRD KEY FIELD ETC. IF NO KEY FIELD IS GIVEN, THE FIRST FIELD IS MADE THE KEY FIELD.

11. **rhook** – POINTS TO A "HOOK" RECORD TO LINK TOGETHER ALL THE TABLES WHICH HAVE A RECURSIVE RELATIONSHIP WITH THIS ONE.

Explanation of the General Item Structure:

1. **iname** – THE NAME OF A PARTICULAR FIELD.

2. **typ** – THE TYPE OF THE FIELD. POSSIBLE VALUES ARE:
   a) "I" – AN INTEGER FIELD.
   b) "F" – A REAL VALUE FIELD.
   c) "C" – A CHARACTER FIELD.
   d) "T" – THE FIELD IS AN EMBEDDED TABLE.
   e) "R" – THE FIELD IS AN EMBEDDED RECURSIVE TABLE.

3. **size** – THE SIZE OF A GIVEN FIELD. THIS IS NOT APPLICABLE TO FIELDS WHOSE TYP VALUES ARE "T" OR "R".

Figure 1.5 Page 2 of 3.

4. **nitem** – A POINTER TO THE NEXT ITEM STRUCTURE IN A LIST OF ITEM STRUCTURES.

5. **nkey** – A POINTER TO THE NEXT KEY IN THE LIST OF KEYS FOR A GIVEN TABLE. SEE TABLE ABOVE FOR FURTHER INFORMATION.

Explanation of the General Hook Structure:

1. **nhook** – USED TO "LINK LISTS" OF HOOK STRUCTURES.

2. **atab** – USED TO "POINT" AT TABLE STRUCTURES.

Figure 1.5 Page 3 of 3.

Figure 1.5 Explanations of the C Data Structures Comprising the Schema Structure.

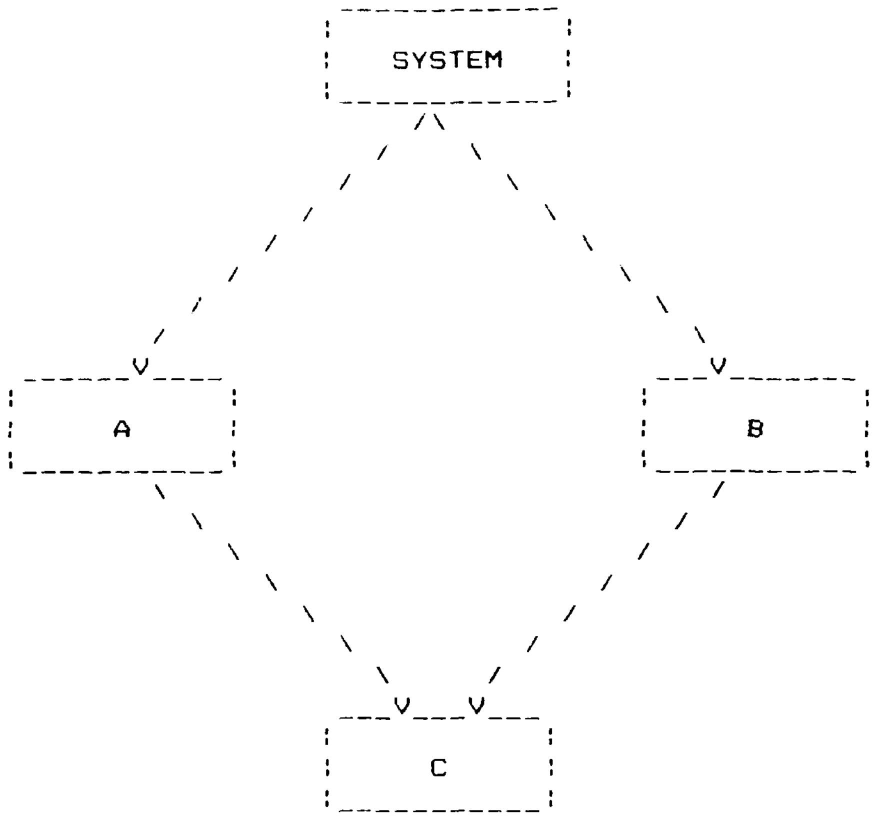


Figure 1.6 A Typical "Y" Network.

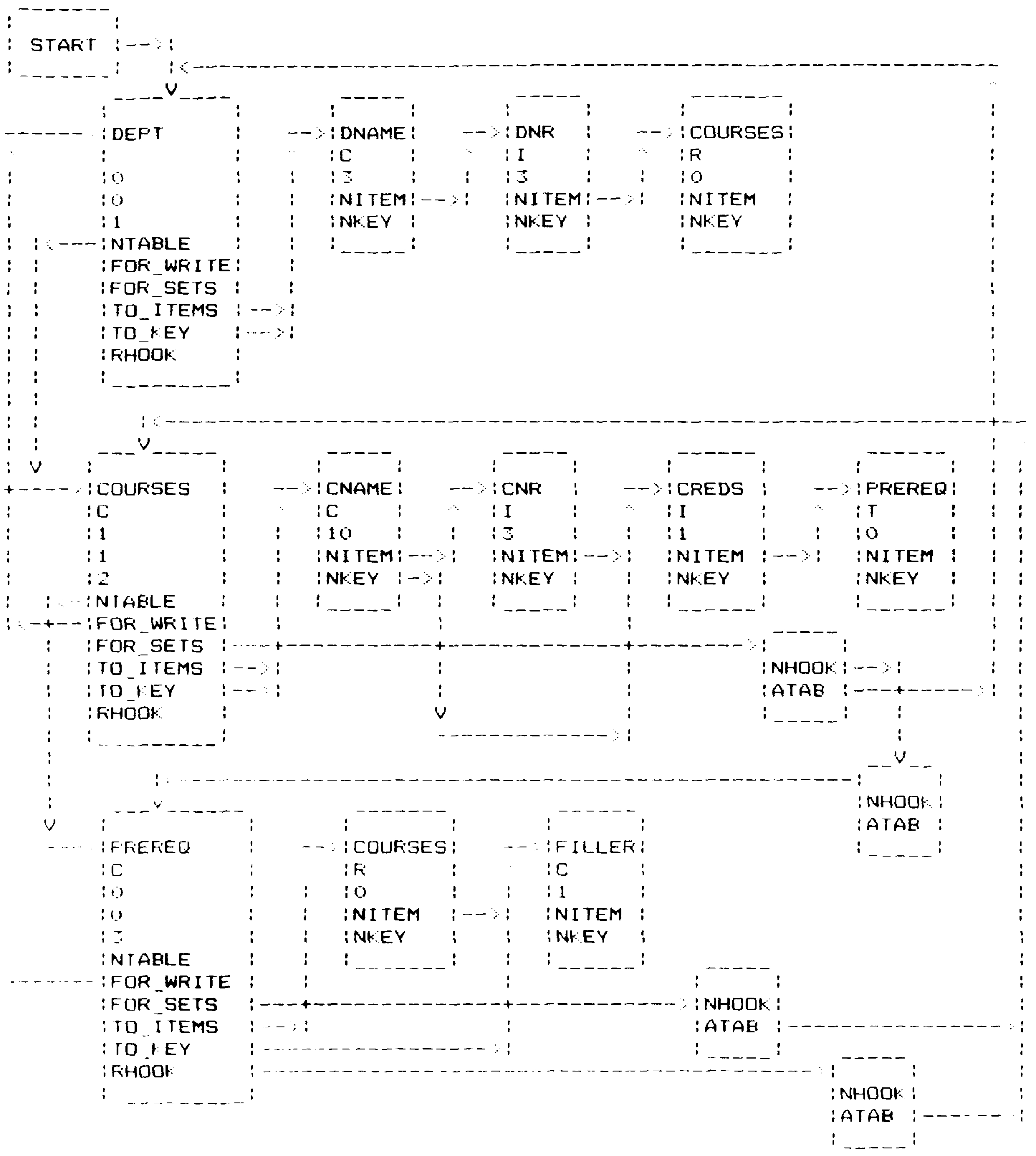


Figure 1.7 A Representation of An Example Schema Data Structure.

CHAPTER 2. CONVERSION OF SCHEMA

2.0 INTRODUCTION

The purpose of this section is to explain my method of schema conversion and to show how any database logic schema can be represented in Codasyl by means of this conversion. In section 2.1, an example SEED schema and sub–schema are presented. Certain constraints on the schema and sub–schema and the general methodology of the conversion are given in section 2.2. A detailed description of the C code is given in section 2.3. Lastly, section 2.4 has the main C routines which implement the schema and sub–schema conversion.

2.1 AN EXAMPLE

Figure 1.3 is an example of a SEED schema derived from the "Y" network of figure 1.1. This schema is the desired output given the DBL Schema for this example.

As mentioned in the last section, a sub–schema must also be produced. The output sub–schema for the same figure is in figure 1.4.

2.2 THE GENERAL METHODOLOGY

This section of the thesis gives certain constraints on the schema and sub–schema and the block description of the code used in schema conversion. The next section will give the specific technical description of the code. Flow charts for the code in these two sections can be found in figures 2.0 and 2.1. There is no flow chart for the subscm routine because of its simplicity.

The schema conversion takes place within certain constraints. First, it

```
   :              :
   V  :           :
   :  :  _________V_________
   :  :  :                  :
   :  :  : HOOKY =          :
   :  :  : (*HOOKY).NHOOK   :
   :  :  :__________________:
   :  :           :
   :  :           V
   :  :<-----------
 __V______________
/                /
 OUTPUT SPECIAL /
POINTER SET    /
______________/
   :
 __V____
:       :
: CLOSE :
: FILE  :
:_______:
   :
 __V__
:     :
: END :
:_____:
```

Figure 2.1 Page 2 of 2.

Figure 2.1 The Flow Chart for the sets Routine.

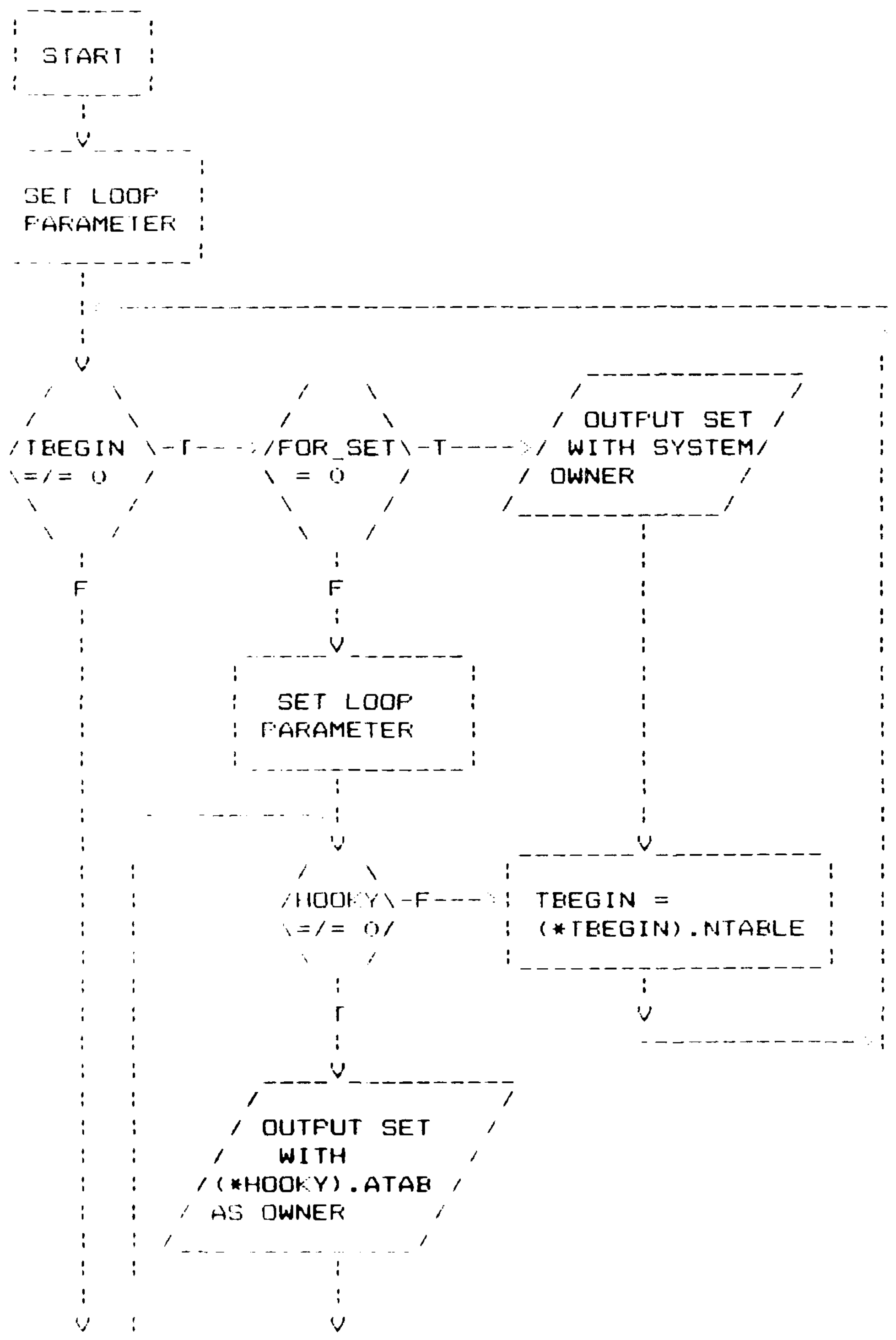

START
SET LOOP PARAMETER
TBEGIN =/= 0
F
T
FOR_SET = 0
T
OUTPUT SET WITH SYSTEM OWNER
F
SET LOOP PARAMETER
HOOKY =/= 0
F
TBEGIN = (*TBEGIN).NTABLE
T
OUTPUT SET WITH (*HOOKY).ATAB AS OWNER

Figure 2.1 Page 1 of 2.

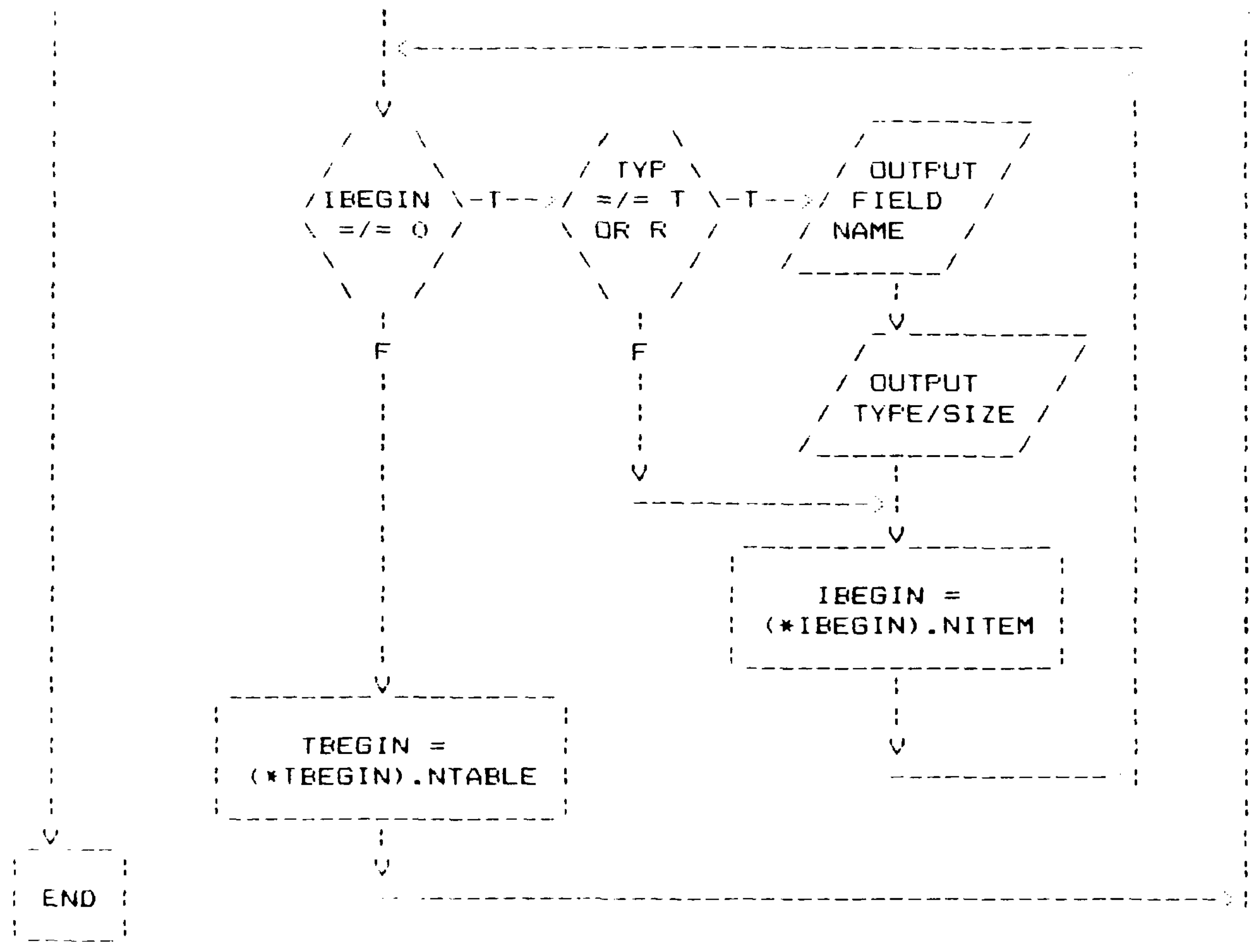
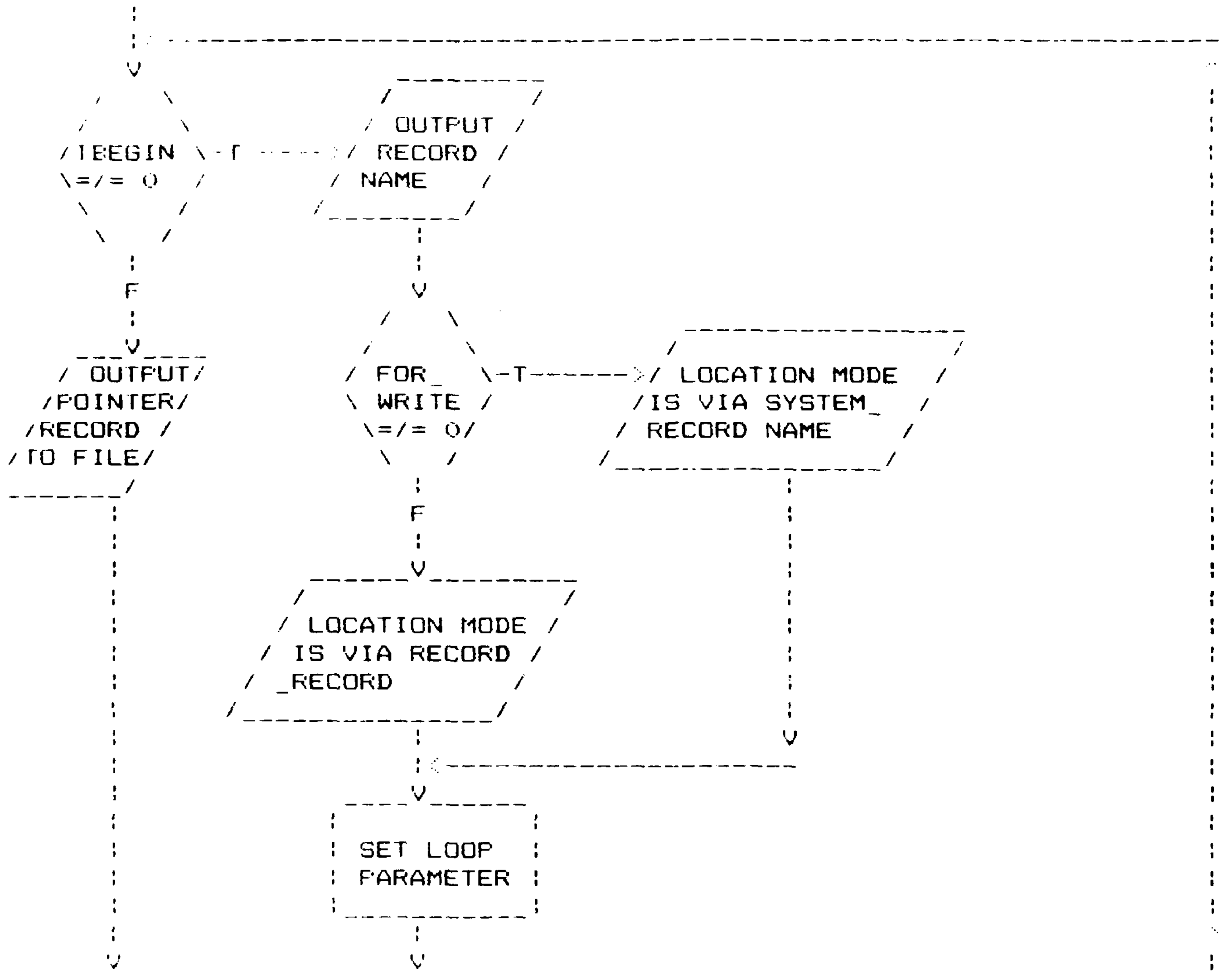


Figure 2.0 Page 3 of 3.

Figure 2.0 The Flow Chart for the records Routine.



IBEGIN =/= 0
T
F
OUTPUT RECORD NAME
OUTPUT POINTER RECORD TO FILE
FOR_WRITE =/= 0
T
F
LOCATION MODE IS VIA SYSTEM_ RECORD NAME
LOCATION MODE IS VIA RECORD _RECORD
SET LOOP PARAMETER

Figure 2.0 Page 2 of 3.

```
       ________
      :        :
      : START  :
      :________:
          :
          V
   ______V______
  :             :
  : OPEN SCHEMA :
  :    FILE     :
  :_____________:
          :
          :
          V
       ___V_____________
      /               /
     / OUTPUT OVERHEAD/
    / INFORMATION TO /
   / THE FILE       /
  /________________/
          :
          :
          V
    _____V_____
   : SET LOOP  :
   : PARAMETER :
   :___________:
          :
          :
          V
```

Figure 2.0 Page 1 of 3.

```
"fprintf(fp,"\n        COPY ALL SETS. \n"); "
```

Control then returns to the calling program where the "SUBSCHM" file is closed. Now the conversion of the DBL schema to the Codasyl schema and sub–schema is complete. An example of such a sub–schema can be found in figure 1.4.

To summarize this section, I have detailed how the Codasyl schema and sub–schema are created from the schema structure (which was in turn created from the DBL schema). This comprises the schema conversion. Next, the Fortran programs must be generated.

2.4 THE CODE

This section gives the main routines used in schema conversion. They, as all code for the DAVID project, are written in the language "C". Here, the "record", "sets", and "subscm" routines are given. The first two routines create the SEED schema. These routines can be found in figure 2.2a and 2.2b, respectively. The "subscm" routine creates the general SEED sub–schema. This routine can be found in figure 2.3.

the other hand, for_sets is not equal to zero, the variable "hooky" is assigned the value of the for_sets field in the structure pointed to by begin. Hooky is now used to traverse hook structures linked together and "originating" from the record pointed to by begin. For each record pointed to by "hooky", a set will be created. The set will have the record pointed to by "hooky" as the owner and to the record pointed to by "begin" as the member. So by traversing this list of hook structures, all sets which have the record pointed to by "begin" as a member will be created.

Finally, the singular set whose owner is "SYSTEM" (by definition) and whose member is the pointer record is created. Upon creation of this set, the Codasyl schema is complete. Now control returns to the calling program, the file which is accepting the schema is closed, and the next procedure is started. An example of a schema created in this manner is in figure 1.3.

The next procedure requires creation of a sub_schema. Again the "fopen" command is used to create a file for writing. In this case, the file is named "SUBSCHM". Once the file is created, the "subscm" function is called to create the sub_schema. This function also takes three arguments. The first is the name of the schema in the "schem" variable, the second is the view name in the "dname" variable, and the last is the pointer to the "SUBSCHM" file in the variable "fp". The schema and view names are used expressly for the purpose of giving the schema name and the view name. The sub-schema is always for Fortran since the automatically generated programs are in Fortran. The record section is then created by the code:

```
"fprintf(fp,"\n      COPY ALL RECORDS. "); "
```

Similarly, the set section is created with the statement:

table structure. This process is repeated until all table structures and all item structures have been visited. In this way, all records contained in the DBL schema are converted to records in the Codasyl schema.

This does not quite complete the "records" function. The function next creates the pointer record. This record is named DN[schema name]. Its location mode is always as a singular set (that is via "SYSTEM_DN[schema name]"). This is the only record within the "POINTER_DATA_AREA". After these statements are entered into the "SCHEMA" file by means of appropriate "fprintf" commands, two fields are created for each table structure in the schema structure. These fields will contain the logical and currency pointers. In SEED these must be of type integer. The actual creation of these fields is done by traversing the table structures in the schema structure and producing the fields as it goes along. Control then returns to the main program and the "sets" function can be called to create the necessary sets.

The "sets" routine takes three inputs. The pointer to the start of the schema structure is passed in the "start" variable, the schema name is passed in the "schem" variable and a pointer to the SCHEMA file is passed in the "fp" variable. Here, the schema structure is traversed to create the necessary sets. The schema name will be used to create the singular set which has the pointer record. The file pointer will be used to reference the output file.

This program begins by assigning to the "begin" variable the value of the "start" variable. The table structures in the schema structure will again be traversed. If the "for_sets" field of the structure pointed to by begin is 0, then the set has one owner – "SYSTEM". In this case, a set whose owner is "SYSTEM" is created and given the applicable attributes. If, on information about size of pages and other specific information but this is not done. The reason is that it is not required (the system will do it automatically) and it keeps the schema as simple as possible. This can and might be changed later if necessary [13].

When this has been completed, the variable "tbegin" is assigned the value of the variable "start". One at a time, the records and fields will be created. First, the record is named. The name coming from the "aname" field of the structure pointed to by tbegin. Next the location mode is created. Location mode is via a set which has the record pointed to by tbegin as a member. If there is no such set, the location mode is via "SYSTEM_record name". For instance, if the record were called "DOCTOR" and it was not owned by any other record then its location mode would be via "SYSTEM_DOCTOR". On the other hand, suppose the record is a member of one or more sets. Then the location mode via the set whose owner is pointed to by the "for_write" field of the record pointed to by the "tbegin" pointer. Since this is a data record (as opposed to a pointer record), it will be "...WITHIN BASIC_DATA_AREA" and this statement is inserted in the file.

Having accomplished this, the fields of this record must be created. This is done by assigning the pointer in (*tbegin).to_items to the variable "ibegin". Then for each item structure (whose "typ" field =/= "T" or "R") associated with this record, a field is created. The type of the field is as designated in the typ field of the item structure pointed to by the "ibegin" pointer. Once the field is printed, the next one is reached by following the pointer in the "nitem" field of the current item structure. Once the last item structure of a record has been thus "visited", the "tbegin" variable is updated to the value in the "ntable" field of the current schema data structure described in section 1.4. It will also cover specifically how the schema and sub–schema are created.

First, in the main program a file to hold the schema is created. In the example program listed this file is always called "SCHEMA". By substituting a string variable (containing a different name), for the word "SCHEMA" this file could be made to accept a general file name. This file is created by use of a C "fopen" command and is opened for writing. This function returns a pointer to the opened file which is assigned to the file pointer variable "fp".

Once this has been accomplished, the "records" function is called. This function will create the static and overhead information needed for the schema. It will also create all the codasyl records based on the information given in the schema structure. The original source of this information being the DBL schema. The records function takes as input the schema and view names. These are supplied in the variables "schem" and "dname". It takes a pointer to the file to write to. This is supplied in the "fp" variable. And, it takes a pointer to the schema structure which is passed in the variable "start". The routine does not explicitly return anything but when it is completed the file pointed to by "fp" has the required overhead and records information.

The first thing done in the program is the creation of the schema name and data areas as follows:

```
"fprintf(fp,"SCHEMA NAME IS %s",schem,);
 fprintf(fp,"\nAREA NAME IS BASIC_DATA_AREA \n");
 fprintf(fp,"AREA NAME IS POINTER_DATA_AREA \n");"
```

This names the schema with the name of the DBL schema name. Next the two data areas are named. When this is complete, the overhead/static information of the schema is complete. It would also be possible to include sets function. The sets function takes the same input as the records function and creates the set section of the schema. The process here again entails traversing the schema structure. Here, as each table structure is "visited", all the sets which have this table as a member record are created. One set is created for each set which has that particular table structure as a member record. If the for_sets pointer is null, the record is a member of only one set and it is owned by "SYSTEM".

Once the structure has been traversed, one more set is created. This is a singular set whose member is the pointer record. This set has special attributes described in the next section. These attributes will be identical for all pointer records/sets in all SEED databases. This then completes the schema conversion, control passes back to the main program, the file being used in closed and creation of the sub–schema can begin.

The sub–schema creation also begins with the opening of a file for writing. This file along with the schema name, and the view name is passed to the subscm subroutine. Here, the most basic sub–schema is created. The sub–schema name and the view name are printed and then all records and sets are copied. This is all that will be required whenever the database is to be opened. When this is done, conversion of the sub–schema is complete, control returns to the main program, the file is closed, and the next section of the program is entered.

2.3 A DETAILED DESCRIPTION

In this section, the code needed to perform the schema conversion is discussed in detail. The section will cover storage of the schema for later use by the SEED processor, specifically how the file for this is created. It will cover how the new schema is produced from the old in terms of the be filtered out. This filtering process can take place through either the filter subroutine or in the output of data after the query. In either case, a simple schema and sub–schema form has thus been adopted and it is this form which gives the freedom to mirror the DBL schema in the SEED schema while still meeting the restrictions mentioned above.

Having explained the constraints under which the schema and sub–schema conversion is accomplished, the actual conversion is now explained. The first process in schema conversion is to open a file for writing. This file will hold the completed schema. Once completely written to this file, the schema can then be "fed" into the SEED processor. A pointer to this file is passed to a subprogram along with a pointer to the schema structure to begin the schema conversion.

In the "records" subroutine, the conversion begins with output of static/overhead information. This includes the schema name and two working areas. Both the schema name and the working areas are required by SEED. The working areas are used by "SEED" as buffers to handle data during processing.

In this subroutine, the next thing to happen is the creation of all records. These are created with the attributes described in the last section. This is done by traversing the schema structure and creating one record for each table structure within this larger structure. The fields of this record are created from the information in the item structures.

Once the list has been traversed, one more record is created. This is the record containing the appropriate pointer fields. This record contains two fields for each table structure in the schema structure. One of these fields is for a logical pointer and the other is for a currency pointer.

To create the sets needed in the schema, the main program calls the selected through one of its pointers, this will not hinder the operation.

Fields of records are limited to one of three types: integer, real, or character. Only the character fields are permitted formatting descriptors. This allows any sort of data to be represented and still keeps the data as straightforward as possible. In particular, the fields of records which will contain pointers are represented as integers. The size information for real, integer, and character types will be used when outputting the result of queries.

Ordering is another attribute which is limited. All records must have a sorted order or be ordered as "NEXT". Non–pointer records will have a sorted order. This will be on a field specified in the DBL schema. Pointer records will be sorted as NEXT. All records will be "CHAIN LINKED PRIOR" and "LINKED TO OWNER". This simplifies the conversion process and facilitates traversing the database sequentially.

The last important attribute is how a member record is inserted into the database. There are many permutations of this attribute possible. I have chosen two possibilities here. For any set member, the record must be either "MANDATORY MANUAL" or "MANDATORY AUTOMATIC". All data records are "MANDATORY MANUAL". For these records to exist, they must be connected to a set and the connection must be made by means of an explicit command. This is done to allow a record to be inserted into one set in which it is a member while not being inserted into another set where it can have membership. The pointer record will be MANDATORY AUTOMATIC since each time one is created it can only become a member of the singular set for pointers.

In the same way the sub–schema is a "vanilla" sub–schema. It allows access to all records and sets. It assumes that "unwanted" information will must mirror the database logic schema as closely as possible. Next, the facilities of the particular database used will be a subset of the total facilities provided by SEED. This is done for several reasons:

1. Not all of the facilities will be needed.
2. It will simplify the conversion process.
3. It will keep schemas as uniform as possible greatly simplifying the automatic program generation.

Also certain facilities available in DAVID are not available in SEED. However, these facilities, if requested, will have to somehow be "handled" by SEED.

In order to satisfy the above constraints, the "vanilla" database was developed [10]. This is a database containing just enough elements of the SEED DBMS to create a schema equivalent to the DBL schema. There has to be enough elements to allow the schema to model any "legal" database. The features which were selected for elimination were those unique to SEED, those not utilized by the automatically generated programs, and those features which were for any reason not needed.

The first attributes are the available areas. There are always only two areas allowed. The first is the "basic_data_area" where all data will be interfaced. The second is the "pointer_data_area" where the currency and logical pointers will be interfaced. Currently, the area and page sizes for these areas are the default sizes for the machine. The default sizes are relatively small. With experience in using these routines, an efficient manner of declaring sufficient area and page sizes should become evident [13].

Location mode is through current of set or location mode of owner. This will be sufficient for sequentially traversing the database. It does mean, however, that care must be taken to ensure there is a current of a set before any operation is attempted on it. In the case where a record must be

```
 records(start,schem,fp)

/* JOSEPH AULINO */
/* DAVID PROJECT */
/* MARCH 1984    */
/*               */
/* THIS PROCEDURE WRITES ALL THE RECORDS NEEDED FOR ANY DATABASE */
/* WHICH DAVID WILL STORE.  IT DOES NOT ATTEMPT TO IMPLEMENT ALL */
/* OF THE OPTIONS AVAILABLE TO RECORDS IN "SEED". HOWEVER, EACH  */
/* RECORD WILL "INHERIT" THE OPTIONS OF THE DATABASE FROM WHICH  */
/* ITS FIELDS CAME.  FOR INSTANCE, IF A FIELD CAME FROM A RECORD */
/* IN WHICH DUPLICATES WERE NOT ALLOWED, THEN THIS FIELD CAN NOT */
/* BE DUPLICATED SINCE THE SOURCE OF THE DATA WILL CONTAIN NO    */
/* DUPLICATES.                                                   */
/*                                                               */
/*                                                               */
/* LOCAL VARIABLES:                                              */
/*                                                               */
/*      1.  tbegin - USED TO TRAVERSE THE LIST OF TABLES.        */
/*      2.  ibegin - USED TO TRAVERSE ALL LISTS OF ITEMS.        */


struct table *start;
char schem[];
FILE *fp;
{
struct table *tbegin;
struct item *ibegin;


/*******************************************/
/*                                         */
/* FIRST ALL OF THE STATIC DATA IS OUTPUT. */
/*                                         */
/*******************************************/


fprintf(fp,"SCHEMA NAME IS %s",schem);
fprintf(fp,"\nAREA NAME IS BASIC_DATA_AREA.\n");
fprintf(fp,"AREA NAME IS POINTER_DATA_AREA.\n");


/**********************************************/
/*                                            */
/* NOW THE RECORDS ARE PRODUCED ONE AT A TIME. */
/*                                            */
/**********************************************/
```

Figure2.2a Page 1 of 3.

```
tbegin = start;
while (tbegin != 0)
{

   /*************************************************************/
   /*                                                           */
   /* HERE RECORDS, LOCATION MODES, AND AREAS ARE PRODUCED.    */
   /*                                                           */
   /*************************************************************/


    fprintf(fp,"\nRECORD NAME IS %s",(*tbegin).aname);
    fprintf(fp,"\n    LOCATION MODE IS VIA");
    if ((*tbegin).for_write == 0)
       fprintf(fp," SYSTEM_%s",(*tbegin).aname);
    else
       fprintf(fp," %s_%s",(*(*tbegin).for_write).aname,(*tbegin).aname);
    fprintf(fp,"\n    WITHIN BASIC_DATA_AREA.");


    /*********************************************************/
    /*                                                       */
    /* THE NEXT SECTION WRITES THE ITEMS FOR EACH RECORD.   */
    /*                                                       */
    /*********************************************************/


    ibegin = (*tbegin).to_items;
    while (ibegin != 0)
    {
       if (((*ibegin).typ != 'T') && ((*ibegin).typ != 'R'))
       {
          fprintf(fp,"\n       %s TYPE IS ",(*ibegin).iname);
          if ((*ibegin).typ == 'C')
             fprintf(fp,"CHARACTER %d.",(*ibegin).size);
          else
             if((*ibegin).typ == 'I')
                fprintf(fp,"INTEGER.");
             else
                fprintf(fp,"REAL.");
       }
       ibegin = (*ibegin).nitem;
    }
    tbegin = (*tbegin).ntable;
    fprintf(fp,"\n");
}
```

Figure 2.2a Page 2 of 3.

```
/***************************************************************************/
/*                                                                         */
/* THE NEXT SECTION PRINTS THE POINTER RECORD. THIS RECORD WILL ALWAYS */
/* CONTAIN TWO FIELDS FOR EACH TABLE IN THE DATABASE.  ONE FIELD WILL  */
/* BE AVAILABLE FOR A GIVEN RECORD'S CURRENCY POINTER, THE OTHER FIELD */
/* WILL BE AVAILABLE FOR A LOGICAL POINTER FOR A GIVEN RECORD.         */
/*                                                                         */
/***************************************************************************/


fprintf(fp,"\nRECORD NAME IS DN%s",schem);
fprintf(fp,"\n    LOCATION MODE IS VIA SYSTEM_DN%s",schem);
fprintf(fp,"\n    WITHIN POINTER_DATA_AREA.");
fprintf(fp,"\n        D%s TYPE IS INTEGER.",schem);

tbegin = start;
while (tbegin != 0 )
{
   fprintf(fp,"\n        N%s TYPE IS INTEGER.",(*tbegin).aname);
   fprintf(fp,"\n        A%s TYPE IS INTEGER.",(*tbegin).aname);
   tbegin = (*tbegin).ntable;
}
return;

}       /*****      END OF THE RECORDS ROUTINE      *****/
```

Figure 2.2a Page 3 of 3.

Figure 2.2a The records Routine C Code.

```
 sets(start,schem,fp)

/* JOSEPH AULINO */
/* DAVID PROJECT */
/* MAY 1985      */
/*               */
/* THIS PROCEDURE WRITES ALL OF THE SETS NEEDED BY A SCHEMA.   */
/* IT DOES THIS BY FOLLOWING THE LIST POINTED TO BY START.     */
/* THE RELATIONSHIP OF RECORDS IN THIS LIST WAS DEVELOPED BY   */
/* THE ROUTINE RELATE.  IT WAS IN THAT ROUTINE THAT CORRECT    */
/* SET MEMBERSHIP WAS ESTABLISHED.                             */
/*                                                             */
/*                                                             */
/* LOCAL VARIABLES:                                            */
/*                                                             */
/*      1.  begin - USED TO TRAVERSE THE TABLE STRUCTURES.     */
/*      2.  follow - USED TO TRAVERSE THE TABLE STRUCTURES.    */
/*      3.  hooky - USED TO TRAVERSE A HOOK STRUCTURE LIST.    */

struct table *start;
char schem[];
FILE *fp;
{
struct table *begin, *follow;
struct hook *hooky;

begin = start;

/*****************************************************************/
/*                                                               */
/* NOW FOR EACH MEMBER OF THE LIST THE CORRECT SET IS CREATED.  */
/*                                                               */
/*****************************************************************/


fprintf(fp,"\n");
while (begin != 0)
{
   if ((*begin).for_sets == 0)


    /*******************************************/
    /*                                         */
    /* IN THIS CASE THE SET IS "SYSTEM" OWNED. */
    /*                                         */
    /*******************************************/
```

__Figure 2.2b__ __Page 1 of 3.__

```
 {
    fprintf(fp,"\nSET NAME IS SYSTEM_%s",(*begin).aname);
    fprintf(fp,"\n   MODE IS CHAIN LINKED PRIOR");
    fprintf(fp,"\n   ORDER IS SORTED");
    fprintf(fp,"\n   OWNER IS SYSTEM");
    fprintf(fp,"\n   MEMBER IS %s MANDATORY AUTOMATIC",(*begin).aname);
    fprintf(fp,"\n      LINKED TO OWNER");
    fprintf(fp,"\n      ASCENDING KEY IS ");
    if ((*begin).to_key == 0)
       fprintf(fp,"%s",(*(*begin).to_items).iname);
    else
       fprintf(fp,"%s",(*(*begin).to_key).iname);
    fprintf(fp,"\n      SET SELECTION IS THRU CURRENT OF SET.\n");
}
else
{
   hooky = (*begin).for_sets;


   /************************************************************/
   /*                                                          */
   /* IN THIS CASE, A SET WILL BE CREATED FOR EACH SET OF WHICH */
   /* THIS RECORD IS A MEMBER.                                 */
   /*                                                          */
   /************************************************************/


   while (hooky != 0)
   {
      follow = (*hooky).atab;
      fprintf(fp,"\nSET NAME IS ");
      fprintf(fp,"%s_%s",(*follow).aname,(*begin).aname);
      fprintf(fp,"\n   MODE IS CHAIN LINKED PRIOR");
      fprintf(fp,"\n   ORDER IS SORTED");
      fprintf(fp,"\n   OWNER IS %s",(*follow).aname);
      fprintf(fp,"\n   MEMBER IS %s MANDATORY AUTOMATIC",(*begin).aname);
      fprintf(fp,"\n      LINKED TO OWNER");
      fprintf(fp,"\n      ASCENDING KEY IS ");
      if ((*begin).to_key == 0)
         fprintf(fp,"%s",(*(*begin).to_items).iname);
      else
         fprintf(fp,"%s",(*(*begin).to_key).iname);
      fprintf(fp,"\n      SET SELECTION IS THRU CURRENT OF SET.\n");
      hooky = (*hooky).nhook;
   }
 }
```

Figure 2.2b Page 2 of 3.

```
   begin = (*begin).ntable;
}
/* END OF THE OUTSIDE WHILE LOOP */


/*********************************************************************/
/*                                                                   */
/* THE LAST SECTION OF CODE PRINTS OUT ONE SPECIAL SET.  THIS SET */
/* WILL BE USED TO HOLD POINTER INFORMATION ABOUT EACH RECORD AND */
/* SET.                                                              */
/*                                                                   */
/*********************************************************************/


fprintf(fp,"\nSET NAME IS SYSTEM_DN%s",schem);
fprintf(fp,"\n    MODE IS CHAIN LINKED PRIOR");
fprintf(fp,"\n    ORDER IS NEXT");
fprintf(fp,"\n    OWNER IS SYSTEM");
fprintf(fp,"\n    MEMBER IS DN%s MANDATORY AUTOMATIC", schem);
fprintf(fp,"\n       LINKED TO OWNER");
fprintf(fp,"\n       SET SELECTION IS THRU CURRENT OF SET.\n");

return;

}      /*****     END OF THE SETS PROCEDURE.     *****/
```

Figure 2.2b Page 3 of 3.

Figure 2.2b The sets Routine C Code.

```
subscm(schem,dname,fp)

/* JOSEPH AULINO */
/* DAVID PROJECT */
/* JUNE 1984     */
/* THIS PROCEDURE CREATES A SUB-SCHEMA FOR ANY DATABASE DEFINED. */
/* THIS SUB-SCHEMA WILL BE VERY SIMPLE AND MERELY OPEN ALL SETS  */
/* AND RECORDS FOR ACCESS.  IT TAKES IN THE SCHEMA NAME (schem), */
/* THE SUB-SCHEMA NAME (dname), AND THE FILE (fp), TO PUT THE    */
/* OUTPUT INTO.                                                  */


FILE *fp;
char schem[], dname[];
{
   fprintf(fp,"\nSUB-SCHEMA NAME IS %s FOR FORTRAN", dname);
   fprintf(fp," OF SCHEMA %s.",schem);
   fprintf(fp,"\nRECORD SECTION.");
   fprintf(fp,"\n   COPY ALL RECORDS.");
   fprintf(fp,"\nSET SECTION.");
   fprintf(fp,"\n   COPY ALL SETS.\n");

   return;

}     /*****     END OF THE SUBSCM ROUTINE     *****/
```

Figure 2.3 The subscm Routine C Code.

CHAPTER 3. QUERY TRANSLATION

3.0 INTRODUCTION

This chapter deals with the automatically generation of programs used to translate the basic queries. Fortran programs are used to interface with SEED. C routines are used to generate this Fortran. There are 4 such routines:

1. FIRSTWHERE;
2. NEXTWHERE;
3. GRASP;
4. FILER;

First, section 3.1 gives examples of each of 4 Fortran routines which might be created. Section 3.2 contains the block descriptions of the C code and Fortran code it creates. In section 3.3, the technical description of the C and Fortran code is given. Lastly, section 3.4 gives the actual C routines used to generate the Fortran code.

3.1 AN EXAMPLE

This section contains one example of each of the automatically generated programs produced for the purpose of query translation. The examples are the routine which would be created for the non-recursive DBL schema example of section 1.1. The first such program is the "FIRSTW" routine. As stated previously, its purpose is to get the first instance of a database meeting certain criteria. An example of this is in figure 3.0a.

The next example is the NEXTW routine. This finds the next occurrence of a set of records in a database meeting certain requirements. The corresponding example of this routine is in figure 3.0b.

The next example is the GRASP routine which would be generated for the same database. This routine is illustrated in figure 3.1. The Grasp routine, you will notice, has a different control structure than the FIRSTW and NEXTW routines.

The last example is for the FILLER routine found in figure 3.2. This routine uses a control structure similar to that of the GRASP routine. Note, however, the difference in the number of items of interest here.

3.2 THE GENERAL METHODOLOGY

This section gives the block descriptions of code to query the data base. This includes descriptions of both the C and Fortran code.

First, is a block description of the Firstwhere and Nextwhere code. These two routines are described together because of their similarities. Because of the similarities in these two routines, only a flow chart for firstwhere has been included. It is in figure 3.3.

The first thing done is a file is opened (either FIRSTW.F or NEXTW.F as appropriate). Next the overhead information is output to the file. This information includes local variable declaration, parameter declaration, subroutine header information, and a common statement. In FIRSTW, the main part of the program is now created. However, in NEXTW, the next thing made is a "GO TO" statement to the line where execution should begin. At this point, the main body of both programs can begin.

The programs are created by traversing the schema structure. At each table structure, a fetch is generated (either through child or parent as appropriate). Also, if the particular record owns one or more recursive records, a fetch is created for each of the recursive records through the record in question. After each fetch has been generated, a call to FILTER is generated and the appropriate loops are generated. These loops are explained more fully in the next section and can be understood by examining the appropriate flow charts. Also, if a fetch is generated on any recursive records, the appropriate variables are saved in the proper locations. Next, the return and end statements are generated. Lastly, the file (FIRSTW.F or NEXTW.F) is closed.

The GRASP routine creation also requires the traversal of the Schema structure. A flow chart for the grasper routine (which created GRASP) is in figure 3.4. To begin this routine, a file is opened (GRASP.F) and overhead information is written. After the overhead information, comes the terminating test for the loop. This is followed by creation of the computed "GO TO" statement. Now the Schema structure can be traversed and the subroutine written. At each table structure of the subroutine, a fetch (via currency pointer) is generated. Once all of the table structures have been visited and the proper fetches generated, a go to statement is created sending control back to the termination condition line. Finally, the return/end statements can be generated and the file closed.

The FILER code creation is very similar to the GRASP code creation. It is created by the outpt and filwrite routines whose flow charts are in figures 3.5a and 3.5b respectively. Again a file is opened and overhead information output. This is also followed by the creation of the loop termination statement and the computed "GO TO" statement. In this case, there is more of the main body of code to generate. The main body again is created as the schema structure is traversed. In this case, however, the item structures must also be traversed. At each table structure, output statement are created for two fields (the currency and logical pointers). At each item structure, output statements are created for every non–table field of that record. Statements are also generated to output field of recursive records without any additional calls to the routine. Once this has been accomplished for the entire schema structure, a "GO TO" statement is generated to return control to the loop termination line of code, the return/end statements are created and the "FILER.F" file is closed.

The FIRSTW and NEXTW routines are the first Fortran routines discussed. Flow charts for example FIRSTW and NEXTW routines are in figures 3.6a and 3.6b respectively. These two routines are discussed together because of their aforementioned similarity. The first record selected is the first record listed in the DBL schema. A check is immediately made to see if an error was made in selecting this record (ERRSTA is checked). If an error was made, control returns to the code for the last record selected (to the calling program if it's the first record). If no error was made, FILTER is called and the record is checked to see if it meets the required criterion. If not, the next record of this type is selected and the process is repeated until a record is found or ERRSTA =/= 0 (end of file). If a record is found, the next type of record is selected. The process is repeated until one type of each record has been selected or it is not possible to satisfy the selection criterion in the database. Control then reverts to the calling program.

The grasp routine is the next routine to be discussed. Its accompanying flow chart is in figure 3.7. The grasp routine is controlled by a while loop implemented by standard "GO TO" statements and a computed go to statement. It takes in a vector of integers corresponding to the records to be selected. These records can then be selected through their currency pointers. It is assumed that the currency pointer for a record to be selected is in the appropriate variable which is then used as a parameter for the call to the subroutine. The routine terminates when a zero element is found in the vector.

The FILER routine is quite similar to the GRASP routine as is evident from its flow chart (figure 3.8). This routine is controlled in the same manner as the grasp routine. It does, however, have more possibilities in its computed go to statement. The fields selected for output are output to the same line and a line feed is output after the last record has been written. Fields which may be output include all fields of all records and the pointers (either currency or logical) for each record.

3.3 A DETAILED DESCRIPTION

This section is logically divided into two parts. First is the technical description of the C code and then comes the technical description of the Fortran code. The description of the C code will cover the main routines used in producing the Fortran routines. The remainder covers the Fortran routines produced by the C code.

The FIRSTW routine is created first. Creation begins with the opening of a file to contain the created Fortran code. A pointer to this file along with a pointer to the Schema structure is then passed to the firstwhere subroutine. This is where the actual production of Fortran begins. It starts with the output of the subroutine name and the parameters passed in. In this case "SUBROUTINE FIRSTW(N,TEST)". This is followed by the declaration of the variables... "INTEGER TEST,N". This is followed by creation of the "counter" variables for this program. This done by the "declarer" subroutine. This is followed by a call to the "commons" subroutine creating the necessary commons statement.

Upon return from generating the commons statement, the counter variables are initialized. There is one counter variable for each record in the table (including one for each recursive record). They are generated by traversing the schema structure and generating them as you go from table structure to table structure. These were declared earlier by declarer (see section 1.4 For details). These statements are written to initialize the counters to either the first logical pointer (in FIRSTWHERE) or the current logical pointer (in NEXTWHERE). The counters are named here as they are in "declarer" (see section 1.4 for details). The basic initialization statement is output as:

```
"fprintf(fp,"\n        C%s = 1",(*begin).aname)"
```

In the case of the recursive occurrence:

```
"fprintf(fp,"\n        C%s%d = 1",(*begin).aname,
                                  (*begin).rcnt);"
```

generates the correct statement. In this case, rcnt is decremented and the statement is repeated until rcnt = 0. When rcnt reaches zero, the loop terminates and rcnt is reset with the statement "(*begin).rcnt = (*begin).typr;"

At this point, the generation of the go to statement of the NEXTW routine is explained. This statement is necessary to ensure execution begins in the correct spot. Back in firstwhere, a variable called "count" keeps track of the current line number being written. This would include counting lines for those records selected through their parents but not for those selected through their children. When the "GO TO" statement for beginning the execution of the Nextwhere routine is to be created, a subroutine called "findgo" is called. This counts all table structures whose relat field is not "C". This count * 4 – 3 is the exact amount count would be too great if there are any fields selected through their children. This figure is then subtracted from the "count" amount (which is returned by the firstwhere routine) and the go to statement can be created. The only other difference between the two programs is that firstwhere returns the value count and nextwhere does not.

Next, the statements to fetch data can be generated. To generate the fetch statements, the schema structure is again traversed. At each table structure, the correct statement is generated based on one of three possibilities [10]:

1. The record is being fetched as a child of some other record.
2. The record is being fetched from one of its children.
3. The record is being fetched is recursive and being fetched from one of its parents.

The first possibility occurs when the table structure has relat field of "C". In this case, the code for the proper select is created by the "own" and "ownwriter" routines. The "own" routine generates the find statements and determines the sets owner (for selection purposes). The "own" routine takes the following arguments:

1. fp – a pointer to the outout file;
2. begin – a pointer to a particular table structure in the schema structure.
3. follow – the value of the (*begin).for_write field.
4. adder, count, last – used in numbering lines.
5. swap – used with recursive records:
   "1" – if some variables in the Fortran code need to be saved.
   "0" – otherwise.

Either the owner is pointed to by (*begin).for_write or the owner is "SYSTEM". The value of (*begin).for_write is passed into the routine in the "follow" variable. Following this, the ownwriter routine is called and the remainder of the needed code is generated.

The "ownwriter" routine takes the same input arguments as the "own"

```
ST .NE. 1) GO TO 9999
9990
I = I + 1
GO TO 1

RETURN

END
```

<u>Figure 3.1 Page 2 of 2.</u>

Figure 3.1 An Example GRASP Routine.

```
SUBROUTINE GRASP(NRS,N,TEST)

INTEGER NRS,N,I,TEST

COMMON/BANK/NAME,CITY,ZIP,NHOSPITAL,AHOSPITAL,HOSPITAL,
          SYSTEM_HOSPITAL,DRNAME,DRNR,NDOCTOR,
          ADOCTOR,DOCTOR,HOSPITAL_DOCTOR,BEDNUM,WARD,
          NBEDNR,ABEDNR,BEDNR,HOSPITAL_BEDNR,PNAME,
          DISEASE,AGE,SEX,VITS,NPATIENT,APATIENT,PATIENT,
          DOCTOR_PATIENT,
          ERRSTA,CRRECT,CRRUNU,CRCALC,DB


TEST = 1

I = 1

IF (NRS(I) .EQ. 0) GO TO 9999

GO TO(2,3,4,5),
     NRS(I)

CALL FINDD(NHOSPITAL)
IF (ERRSTA .NE. 0) GO TO 9999
CALL GET(HOSPITAL)
TEST = FILTER(1)
IF (TEST .NE. 1) GO TO 9999
GO TO 9990

CALL FINDD(NDOCTOR)
IF (ERRSTA .NE. 0) GO TO 9999
CALL GET(DOCTOR)
TEST = FILTER(2)
IF (TEST .NE. 1) GO TO 9999
GO TO 9990

CALL FINDD(NBEDNR)
IF (ERRSTA .NE. 0) GO TO 9999
CALL GET(BEDNR)
 EST = FILTER(3)
 F (TEST .NE. 1) GO TO 9999
 O TO 9990

 ALL FINDD(NPATIENT)
 F (ERRSTA .NE. 0) GO TO 9999
 ALL GET(PATIENT)
 EST = FILTER(4)
```

Figure 3.1 Page 1 of 2.

```
 CALL FINDV(HOSPITAL_BEDNR,NEXT)
CBEDNR = CBEDNR + 1
GO TO 9
  NBEDNR = CBEDNR
ABEDNR = CURRNT(HOSPITAL_BEDNR)

  CALL FINDV(DOCTOR_PATIENT,FIRST)
 IF (ERRSTA .NE. 0) GO TO 10
CALL GET(PATIENT)
TEST = FILTER(4)
IF (TEST .EQ. 1) GO TO 15
 CALL FINDV(DOCTOR_PATIENT,NEXT)
CPATIENT = CPATIENT + 1
GO TO 13
  NPATIENT = CPATIENT
APATIENT = CURRNT(DOCTOR_PATIENT)

RETURN

END
```

Figure3.0b Page 2 of 2.

Figure 3.0b An Example NEXTW Routine.

```
      SUBROUTINE NEXTW(N,TEST)

      INTEGER TEST,N

      INTEGER CHOSPITAL,CDOCTOR,CBEDNR,CPATIEN

      COMMON/BANK/NAME,CITY,ZIP,NHOSPITAL,AHOSPITAL,HOSPITAL,
     *             SYSTEM_HOSPITAL,DRNAME,DRNR,NDOCTOR,
     *             ADOCTOR,DOCTOR,HOSPITAL_DOCTOR,BEDNUM,WARD,
     *             NBEDNR,ABEDNR,BEDNR,HOSPITAL_BEDNR,PNAME,
     *             DISEASE,AGE,SEX,VITS,NPATIENT,APATIENT,PATIENT,
     *             DOCTOR_PATIENT,
     *             ERRSTA,CRRECT,CRRUNU,CRCALC,DB

      CHOSPITAL = NHOSPITAL
      CDOCTOR = NDOCTOR
      CBEDNR = NBEDNR
      CPATIENT = NPATIENT

      GO TO 10

        CALL FINDV(SYSTEM HOSPITAL,FIRST)
      IF (ERRSTA .NE. 0) GO TO 9999
      CALL GET(HOSPITAL)
      TEST = FILTER(1)
      IF (TEST .EQ. 1) GO TO 3
      CALL FINDV(SYSTEM_HOSPITAL,NEXT)
      CHOSPITAL = CHOSPITAL + 1
      GO TO 1
       NHOSPITAL = CHOSPITAL
      AHOSPITAL = CURRNT(SYSTEM_HOSPITAL)

        CALL FINDV(HOSPITAL_DOCTOR,FIRST)
      IF (ERRSTA .NE. 0) GO TO 2
      CALL GET(DOCTOR)
      TEST = FILTER(2
      IF (TEST .EQ. 1) GO TO 7
      CALL FINDV(HOSPITAL_DOCTOR,NEXT)
      CDOCTOR = CDOCTOR + 1
      GO TO 5
       NDOCTOR = CDOCTOR
      ADOCTOR = CURRNT(HOSPITAL_DOCTOR)

        CALL FINDV(HOSPITAL BEDNR,FIRST)
      IF  ERRSTA .NE. 0) GO TO 6
      CALL GET(BEDNR)
      TEST = FILTER(3)
      IF (TEST .EQ. 1) GO TO 11
```

Figure 3.0b Page 1 of 2.

```
)      CALL FINDV(HOSPITAL_BEDNR,NEXT)
      CBEDNR = CBEDNR + 1
      GO TO 9
.       NBEDNR = CBEDNR
      ABEDNR = CURRNT(HOSPITAL_BEDNR)

        CALL FINDV(DOCTOR_PATIENT,FIRST)
3      IF (ERRSTA .NE. 0) GO TO 10
      CALL GET(PATIENT)
      TEST = FILTER(4)
      IF (TEST .EQ. 1) GO TO 15
4      CALL FINDV(DOCTOR_PATIENT,NEXT)
      CPATIENT = CPATIENT + 1
      GO TO 13
5       NPATIENT = CPATIENT
      APATIENT = CURRNT(DOCTOR_PATIENT)

999   RETURN

      END
```

Figure 3.0a Page 2 of 2.

Figure 3.0a An Example FIRSTW Routine.

```
      SUBROUTINE FIRSTW(N,TEST)

      INTEGER TEST,N

      INTEGER CHOSPITAL,CDOCTOR,CBEDNR,CPATIENT

      COMMON/BANK/NAME,CITY,ZIP,NHOSPITAL,AHOSPITAL,HOSPITAL,
     *           SYSTEM HOSPITAL,DRNAME,DRNR,NDOCTOR
     *           ADOCTOR,DOCTOR,HOSPITAL_DOCTOR,BEDNUM,WARD,
     *           NBEDNR,ABEDNR,BEDNR,HOSPITAL_BEDNR,PNAME,
     *           DISEASE,AGE,SEX,VITS,NPATIENT,APATIENT,PATIENT,
     *           DOCTOR_PATIENT,
     *           ERRSTA,CRRECT,CRRUNU,CRCALC,DB


      CHOSPITAL = 1
      CDOCTOR = 1
      CBEDNR = 1
      CPATIENT = 1

        CALL FINDV(SYSTEM HOSPITAL,FIRST)
1     IF (ERRSTA .NE. 0) GO TO 9999
      CALL GET(HOSPITAL)
      TEST = FILTER(1)
      IF (TEST .EQ. 1) GO TO 3
2     CALL FINDV(SYSTEM_HOSPITAL,NEXT)
      CHOSPITAL = CHOSPITAL + 1
      GO TO 1
3      NHOSPITAL = CHOSPITAL
      AHOSPITAL = CURRNT(SYSTEM_HOSPITAL)

        CALL FINDV(HOSPITAL_DOCTOR,FIRST)
5     IF (ERRSTA .NE. 0) GO TO 2
      CALL GET(DOCTOR)
      TEST = FILTER(2)
      IF (TEST .EQ. 1) GO TO 7
6     CALL FINDV(HOSPITAL_DOCTOR,NEXT)
      CDOCTOR = CDOCTOR + 1
      GO TO 5
7      NDOCTOR = CDOCTOR
      ADOCTOR = CURRNT(HOSPITAL DOCTOR)

        CALL FINDV(HOSPITAL BEDNR,FIRST)
9     IF (ERRSTA .NE. 0) GO TO 6
      CALL GET(BEDNR)
      TEST = FILTER(3)
      IF (TEST .EQ. 1) GO TO 11
```

Figure 3.0a Page 1 of 2.

The last and major difference with this routine is the number correspondence scheme. There are many more "cases" here than there are in the GRASP routine, consequently, the numbering scheme is more complicated. "1" corresponds to the currency pointer of the first record in the DBL schema. "2" corresponds to the logical pointer of the first record. Assuming this particular record has n fields, the next n numbers will correspond to the n fields of the record. The number n+1 corresponds to the currency pointer of the next record listed in the DBL schema. The number n+2 corresponds to this record's logical pointer, etc. An exception occurs if a record owns a recursive occurrence of another record. In this case, before the next record gets assigned a number correspondence, the recursive occurrence gets its numbering assigned. Then by loading the NRS array with the appropriate numbers, the desired variables will be printed. Again the termination condition is a 0 in the NRS array with the looping structure the same as for the Grasp routine. Before returning to the main program however, a "line feed" is output so any additional calls to this routine will have output on the following "lines".

3.4 THE CODE

This section contains the code for the main "C" routines described in this chapter. The specific routines contained here are the "firstwhere", "nextwhere", "grasper", "outpt", and "filwrite" routines. These do the tasks explained in the first four sections of this chapter. The firstwhere and nextwhere routines appear in figures 3.9a and 3.9b respectively. The grasper routine appears in figure 3.10. and the outpt and filwrite routines appear in figures 3.11a and 3.11b respectively.

parameters and has the same control logic as the GRASP routine. The same variables must be checked upon return to the calling program. The differences with this program are what this routine actually does, the number correspondence for the computed go to statement, and the lack of any error checking. I will discuss each of these features separately below.

The first difference is what is done. Here appropriate variables are output one set per line. By set here, I mean, the entire result of one call to this routine. An appropriate variable could be any field of the database, any currency pointer, or any logical pointer. These variables must be assigned reasonable values if reasonable results are to be expected. The output is controlled with format statements. The format descriptor is as given in the DBL schema with appropriate conversion to match the Fortran requirements. The only punctuation here is that variables are output with one space between each. An typical example for a currency pointer might be:

```
" 2      WRITE(6,3) NHOSPITAL
  3      FORMAT('+',1X,I10)
         GO TO 9990"
```

An example for a character field might be:

```
"4       WRITE(6,5) NAME
 5       FORMAT('+',1X,A30)
         GO TO 9990"
```

These illustrate precisely what the typical statements look like and how the output is controlled.

As you can see from above, there is no error checking for these statements as there is in the FIRSTW/NEXTW and GRASP routines. This is because these statements cannot really fail. The output can be "garbled" if the user does not ensure variables to be output have reasonable values before using this routine. It is suggested that this routine be used only after using FIRSTW, NEXTW, GRASP, or the initial loading of the database.

The next line is a computed go to statement. It is controlled by the value of the NRS(I) location. The numbers in this array must correspond to the records in the database as follows: The first record in the DBL schema corresponds with the number 1. If this record owns any recursive occurrences of any records, the recursive occurrences get the next numbers in the order they occur. The next record in the schema then gets the next available number with any owned recursive occurrences being numbered as before. This process is repeated until all records have been numbered. In this way, a unique integer can be assigned to each record. Then the number corresponding to each record can be loaded into the NRS array and when NRS(I) = that number the computed go tu will direct control to the correct statement. The NRS array must contain 0 as the last entry to terminate the loop.

This is followed by the CALL FINDD statement for each record type (including recursive occurrences). The argument to the FINDD subroutine is the currency pointer established in the FIRSTW/NEXTW routines. This does not imply, these routines have to be run to use this routine. The only requirement is that the N type variable have the correct currency pointer. Both ERRSTA and TEST are tested for success. If they fail control goes to statement labeled 9999. This is the return statement. If they both succeed, control goes to statement 9990. In that case, the following is executed:

```
9990  I = I + 1
      GO TO 1
```

This increments "I" and control passes back to the termination test. When one of the termination conditions is finally reached, control goes to line 9999 which is the return statement. Control then returns to the calling program.

Finally, the FILER routine is detailed. This routine takes the same record there is no successful selection. Records may be selected by their parent or child. If being selected from their parent, a FINDV is used with the set name and either the word FIRST or NEXT (whichever is appropriate). The same is true for selection through children except the call is made to FINDO.

After a selection has been made completely successfully or has been shown to be a complete failure, control returns to the calling program. Here it is essential the user checks the value of the "TEST" and "ERRSTA" variables to determine if the selection was successful or not.

The GRASP routine takes in three arguments: "NRS", "N", and "TEST". "NRS" is a one dimensional array of integers, "N" is the size of this array, and "TEST" is as in FIRSTW. NRS can have integers between 0 and the number of records in the database. The numbers in the different positions of the NRS array correspond with the records of the database. The exact numbering scheme is explained below.

As in FIRSTW, the arguments are declared, however here there is no need for the counter variables. The same common statement appearing in all the subroutines again appears here. These statements are followed by initializing the local variable I to 1. This will be used to index into the NRS array. After this the main part of the code can begin.

The next line is an IF statement controlling the program. It implements a while loop. The statement which follows it will be executed until NRS(I) = 0. This is the termination condition. When this occurs, the return statement is executed and control returns to the calling program. Control will also return to the calling program if "ERRSTA" ever becomes non–zero or if TEST ever becomes zero. So again, the calling program must carefully check these variables when control finally does return.

have been assigned and there may not be a currency value for some record.

Beyond the arguments, there are the local variables formed by appending a "C" to the front of the appropriate record name and an integer to the back of the name in the case of recursive records. These will be used to count for the logical pointers. The common statement contains all variables referenced in the UWA. This includes the currency pointer variables and the logical pointer variables. Immediately after the common statement, comes the initialization of the counter variables. In the FIRSTW routine, they are initialized to zero. In the NEXTW routine, they are initialized to the current value of the logical pointer. This value of course comes in through the logical pointer in the common statement.

In the NEXTW routine, once this has been done a go to statement is executed. This statement takes control to the last record selected by the last call to either FIRSTW or NEXTW. The next record of this type is then called on with a CALL FINDV. If this call is successfully executed (by success conditions on both ERRSTA and TEST) the program then returns successfully. If not, control passes to the CALL FINDV of the next to last record selected and the next of this type of record is called upon for selection. This process continues until a successful selection has been made or until all records (up to the first one) have been unsuccessfully selected and then the main program gets a return of failure.

In the FIRSTW routine, it is just the opposite. Control starts with the outer most record. If this fails, a return of failure occurs. If it is successful, the next record down is attempted. If this record fetch fails, control goes back to the first record but now the NEXTW one of this type is called upon. This process repeated over and over until one complete instance of the cluster has been successfully selected or for every instance of one type of

Filer, like grasp, is created by one pass through the schema structure. Here, however, the output is longer. There must sufficient output statements to output both the pointers for each record (currency and logical), plus the value of each field of a record. This is accomplished through the traversal and simple creation of the necessary statements. The only detailed point here is that the output is done using "WRITE" commands and "FORMAT" statements. The size and type information for the format statement exists in the appropriate "ITEM" structure.

Having given the technical description of each of the C routines applicable to this section, the equivalent description of the created Fortran routines follows. This second part of this section, begins with a description of the FIRSTW and NEXTW routines.

These routines takes as arguments the variables N and TEST. "N" is the number of records (including the recursive occurrences) in the database. "TEST" will be used to return a value. Upon return from the FILTER routine, "TEST" will contain a 0 or 1 depending on whether or not the query was successful. On return to the calling program both "TEST" "ERRSTA" must be checked ("ERRSTA" for zero and "TEST" for one). A fail on "ERRSTA" indicates that the DBMS had some "problem" doing the requested select. The precise reason can be found by printing out the error message associated with the number or looking up the number in a SEED manual. A fail on TEST indicates that the FILTER criterion failed. If either fail, then the query was unsuccessful and the reason for its lack of success can be determined by the value of the failing variable. Note here, that if FIRSTW returns a failing condition for any reason, NEXTW should not be called. Most likely, no harm would be done by calling NEXTW however the result is indeterminate since pointer values may or may not recursive record. As the schema structure is traversed, the above two cases are checked as each table structure is reached and the appropriate code is generated. After that is accomplished, the "rhook" field of the particular table structure is checked. If this field is not zero, the record it represents is the owner of one or more recursive records. If this is the case, the list of "hook" structures pointed at by the rhook field is traversed. For each of these, code is generated to swap the necessary variables and own is called to generate the find and other needed statements. After this, the return and end statements are generated, areset is called to reset the rcnt fields and control returns to the main program. Here the file in question is closed and the next subroutine is called. In the case of Firstwhere, the value of "count" is returned for later use in Nextwhere. In nextwhere, nothing is returned.

The "grasp" and "outpt" routines have a technical description almost identical with those of the C routines described in the next chapter. The only real difference is the precise wording of the output. In this section then, only a slightly more precise description of these routines is given. For more detail, the next section and the actual code should make this routine clear.

For the grasp program, the schema structure is traversed and one select (using the currency pointer) is created for each record. There is no need here to worry about most relationships. The currency pointer indicates precisely which record is to be selected. In the event of a recursive occurrence, the correct record (the original or recursive) will be selected as this is uniquely determined by the currency pointer. However, if you wish to select an original record and a recursive occurrence then two calls must be made to grasp. One to select each record.

routine. It consists mainly of the print statements to create the actual "find" statement needed.

As is evidenced here, when the execution gets here most of the decisions have already been made. The only thing which really needs to be done is to output to the file the necessary Fortran code. Two decisions which are not obvious which are made here deal with the parameters swap and last. If last is =/= 0, then it contains the line number for control of the Fortran to go to in the event the "FINDV" returns an errsta =/=0. On the other hand if last == 0, then this is the first record being selected and by convention here, the "go to" will be to line number 9999 the last line number in the program. The second decision deals with swap == 1 or swap == 0. If swap == 1, the record being selected is a recursive record. In this case when the Fortran code is executed, the variables filled by fetching the original record will be overwritten. To prevent this, code is generated to swap the values in these variables to other variables. The variables the values are swapped to are named with their rcnt number as described in section 1.4. A test for "ERRSTA == 0" is also created here. This is done so that the swap will take place only when absolutely necessary.

The second possibility is that the record we want to write the "FIND" for is being "selected" from one of its children. In this case, the subroutine "ownee" is called. "Ownee" takes almost the same arguments as "own" (except swap). As in the "ownwriter" routine, the code here is mostly print statements. Here swap is not needed because as the schema structure is built in relate, it is not possible for a recursive record to be selected via anything except one of its parents. This being true if a record is selected through one of its children, it cannot be a recursive occurrence.

This leaves only the possibility that code is being created for a

```
          SUBROUTINE FILER(NRS,N)

          INTEGER N,NRS(N),I

          COMMON/BANK/NAME,CITY,ZIP,NHOSPITAL,AHOSPITAL,HOSPITAL,
     *                SYSTEM_HOSPITAL,DRNAME,DRNR,NDOCTOR,
     *                ADOCTOR,DOCTOR,HOSPITAL_DOCTOR,BEDNUM,WARD,
     *                NBEDNR,ABEDNR,BEDNR,HOSPITAL_BEDNR,FNAME,
     *                DISEASE,AGE,SEX,VITS,NPATIENT,APATIENT,PATIENT,
     *                DOCTOR_PATIENT,
     *                ERRSTA,CRRECT,CRRUNU,CRCALC,DB

          I = 1

1          IF (NRS(I) .EQ. 0) GO TO 9998

          GO TO (2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,
     *                40),
     *           NRS(I)


2         WRITE(6,3) NHOSPITAL
3         FORMAT('+',1X,I10)
          GO TO 9990

4         WRITE(6,5) AHOSPITAL
5         FORMAT('+',1X,I10)
          GO TO 9990

6         WRITE(6,7) NAME
7         FORMAT('+',1X,A30)
          GO TO 9990

8         WRITE(6,9) CITY
9         FORMAT('+',1X,A30)
          GO TO 9990

10         WRITE(6,11) ZIP
11         FORMAT('+',1X,A5)
          GO TO 9990

12         WRITE(6,13) NDOCTOR
13         FORMAT('+',1X,I10)
          GO TO 9990

14         WRITE(6,15) ADOCTOR
15         FORMAT('+',1X,I10)
          GO TO 9990
```

Figure 3.2 Page 1 of 3.

```
16      WRITE(6,17) DRNAME
17      FORMAT('+',1X,A30)
       GO TO 9990

18      WRITE(6,19) DRNR
19      FORMAT('+',1X,I5)
       GO TO 9990

20      WRITE(6,21) NBEDNR
21      FORMAT('+',1X,I10)
       GO TO 9990

22      WRITE(6,23) ABEDNR
23      FORMAT('+',1X,I10)
       GO TO 9990

24      WRITE(6,25) BEDNUM
25      FORMAT('+',1X,I10)
       GO TO 9990

26      WRITE(6,27) WARD
27      FORMAT('+',1X,A20)
       GO TO 9990

28      WRITE(6,29) NPATIENT
29      FORMAT('+',1X,I10)
       GO TO 9990

30      WRITE(6,31) APATIENT
31      FORMAT('+',1X,I10)
       GO TO 9990
```

Figure 3.2 Page 2 of 3.

```
32      WRITE(6,33) PNAME
33      FORMAT('+',1X,A30)
       GO TO 9990

34      WRITE(6,35) DISEASE
35      FORMAT('+',1X,A20)
       GO TO 9990

36      WRITE(6,37) AGE
37      FORMAT('+',1X,I3)
       GO TO 9990

38      WRITE(6,39) SEX
39      FORMAT('+',1X,A1)
       GO TO 9990

40      WRITE(6,41) VITS
41      FORMAT('+',1X,A80)
       GO TO 9990

9990   I = I + 1
       GO TO 1

9998   WRITE(6,9999)
9999   FORMAT('/')

       RETURN

       END
```

Figure 3.2 Page 3 of 3.

Figure 3.2 An Example FILER Routine.

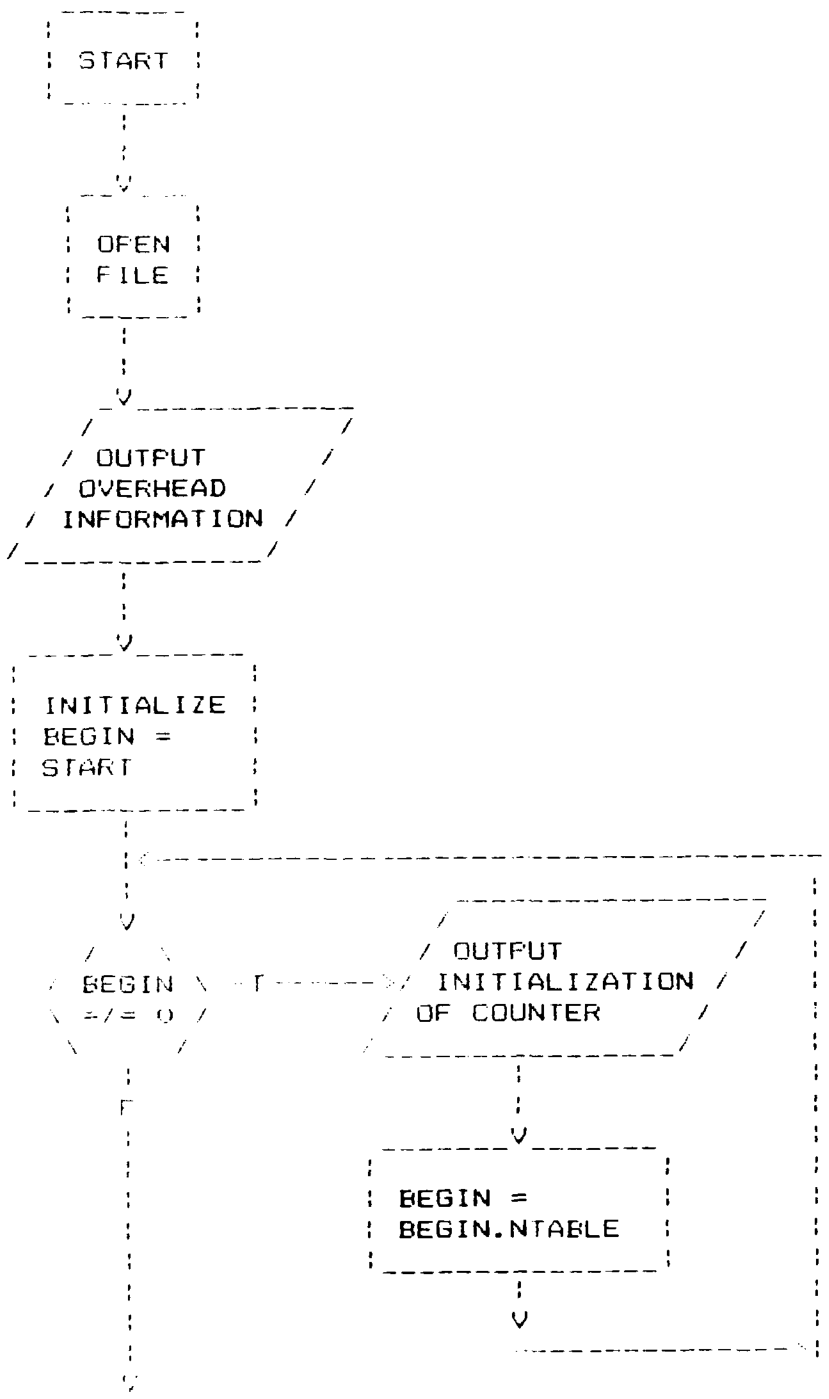


Figure 3.3 Page 1 of 3.

```
    :
 ___V_____
:         :
: BEGIN = :
: START   :
:_________:
    :
    :<--------------------------------------------
    :
    V          /  \         /        /           :
  /   \       /    \       / OUTPUT /            :
 /BEGIN\-T-->/BEGIN.\-T-->/ SELECT /             :
 \=/= 0/     \RELAT /    / VIA    /              :
  \   /       \="C"/    / PARENT /               :
    :           :      /________/                :
    F           F          :                     :
    :           :          :                     :
    :       ____V________  :                     :
    :       /           /  :                     :
    :      / OUTPUT    /   :                     :
    :     / SELECT    /    :                     :
    :    / VIA CHILD /     :                     :
    :   /___________/      :                     :
    :           :          V                     :
    :           :<----------                     :
    :           :
    V           V                                :
```

Figure 3.3 Page 2 of 3.

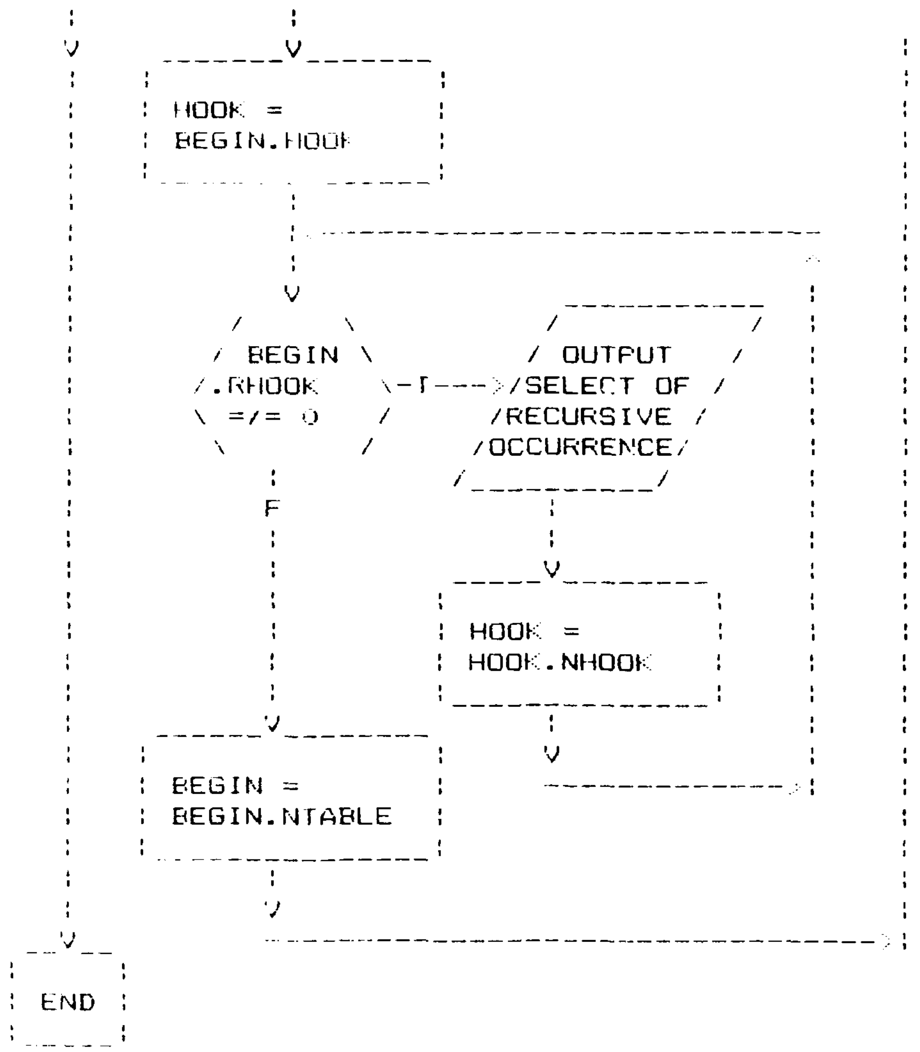


Figure 3.3 Page 3 of 3.

Figure 3.3 Flow Chart for firstwhere Routine.

```
        --------
       :        :
       : START  :
       :________:
           :
           :
        ___V__
       :      :
       : OPEN :
       : FILE :
       :______:
           :
           :
        ___V____________
       /               /
      / OUTPUT        /
     / OVERHEAD      /
    / INFORMATION   /
   /_______________/
           :
           :
        ___V________________
       /                   /
      / OUTPUT START OF   /
     / COMPUTED GOTO     /
    / STATEMENT         /
   /___________________/
           :
           :
     ______V_______
    :              :
    : BEGIN =      :
    : START        :
    :______________:
           :
           V
```

Figure 3.4 Page 1 of 4.

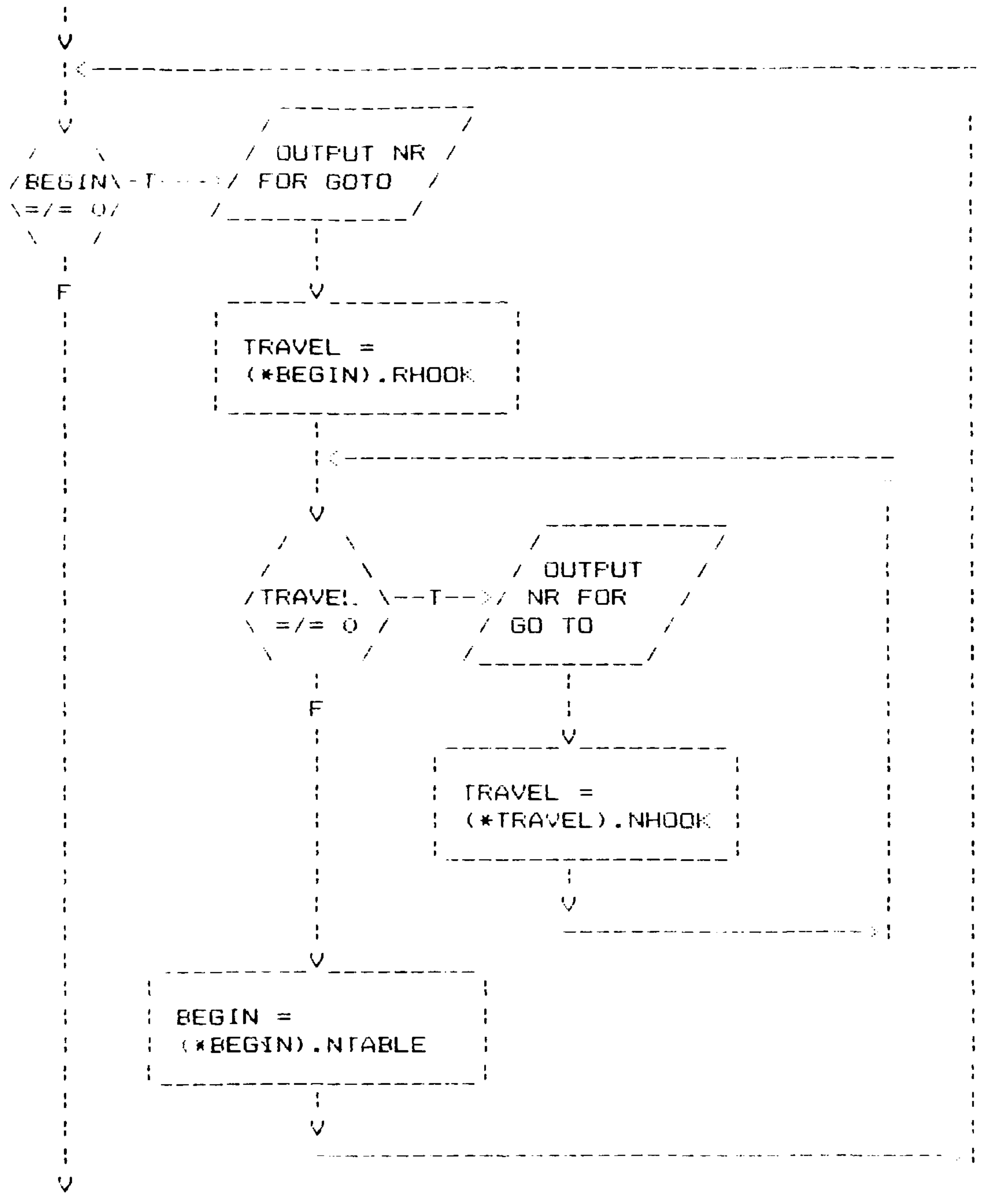


Figure 3.4 Page 2 of 4.

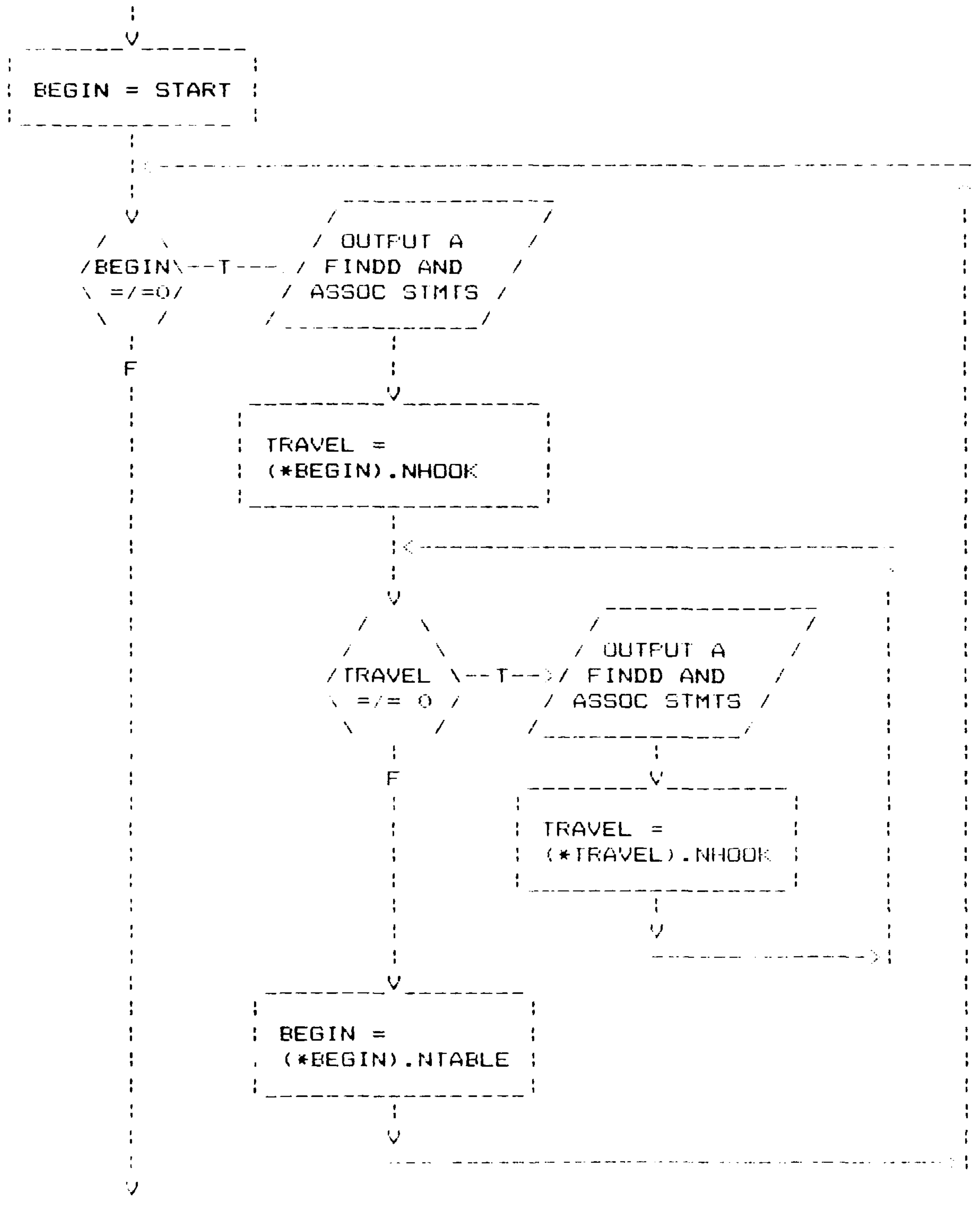


Figure 3.4 Page 3 of 4.

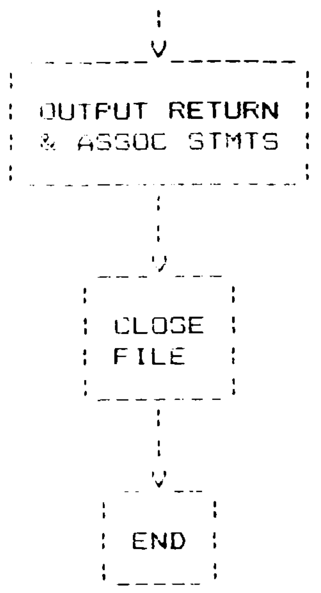
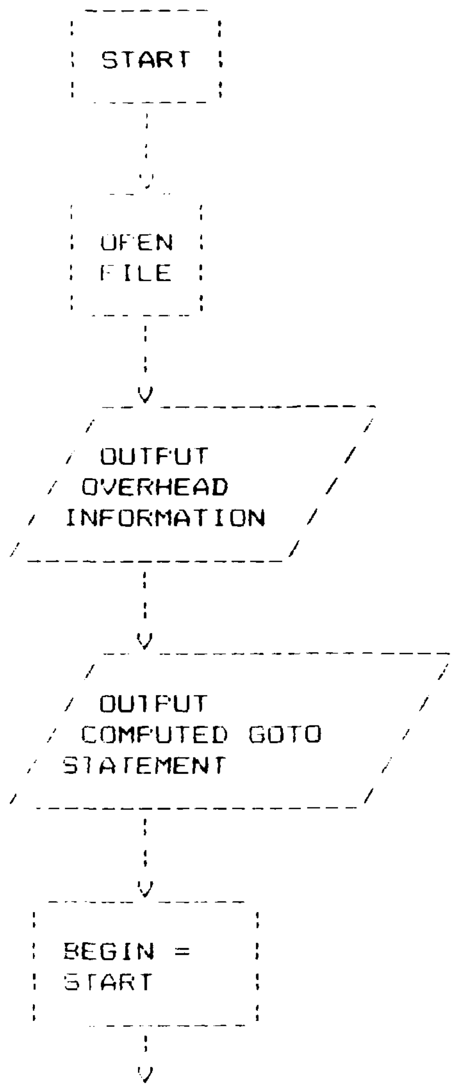


Figure 3.4 Page 4 of 4.

Figure 3.4 Flow Chart for grasper Routine.



<u>Figure 3.5a</u> <u>Page 1 of 3.</u>

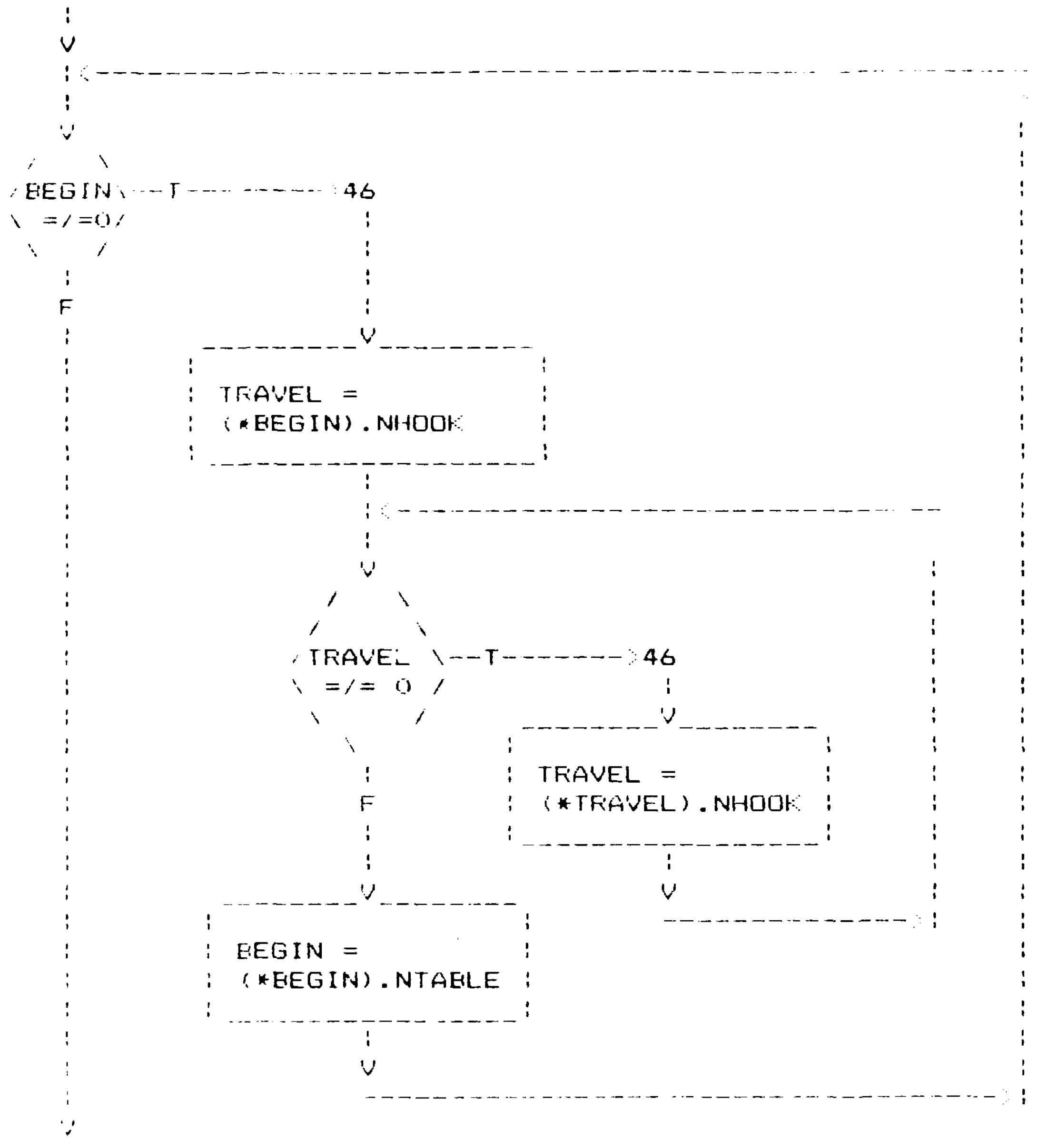


Figure 3.5a Page 2 of 3.

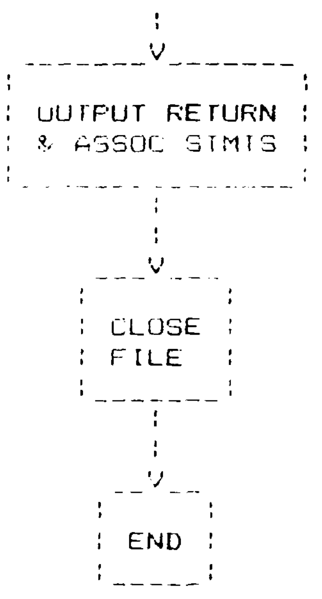


Figure 3.5a Page 3 of 3.

Figure 3.5a Flow Chart for the outpt Routine.

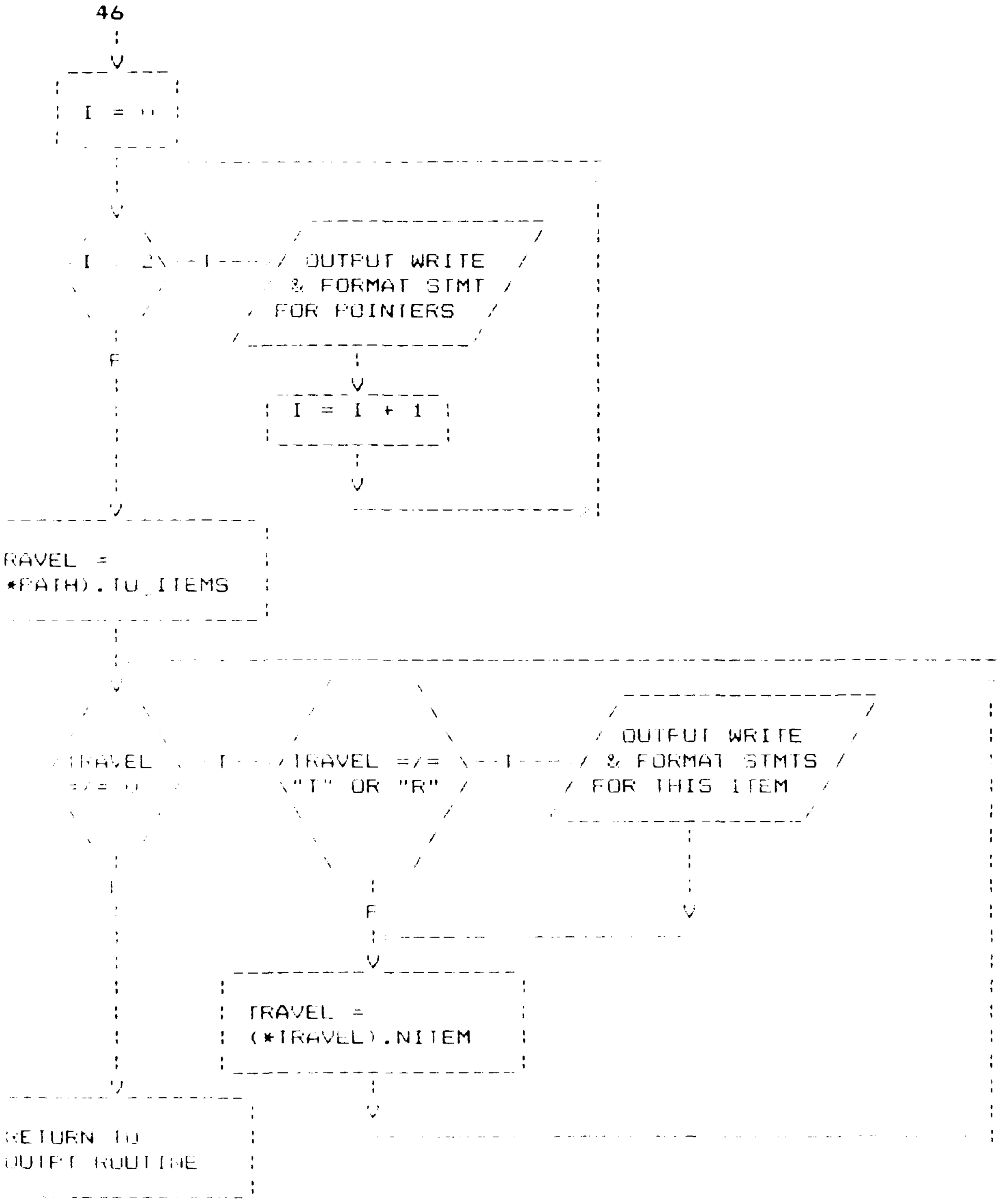
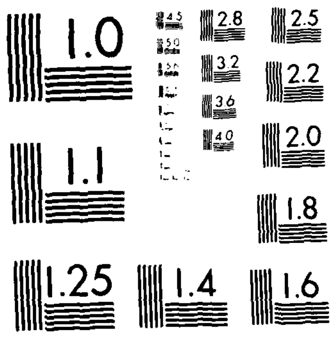


Figure 3.5b Flow Chart for filwrite Routine.

AD-A156 955 THE DATABASE UNIFORMIZATION PROBLEM: A SEED/DAVID INTERFACE(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH J AULINO 1985 AFIT/CI/NR-85-44T 2/2

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

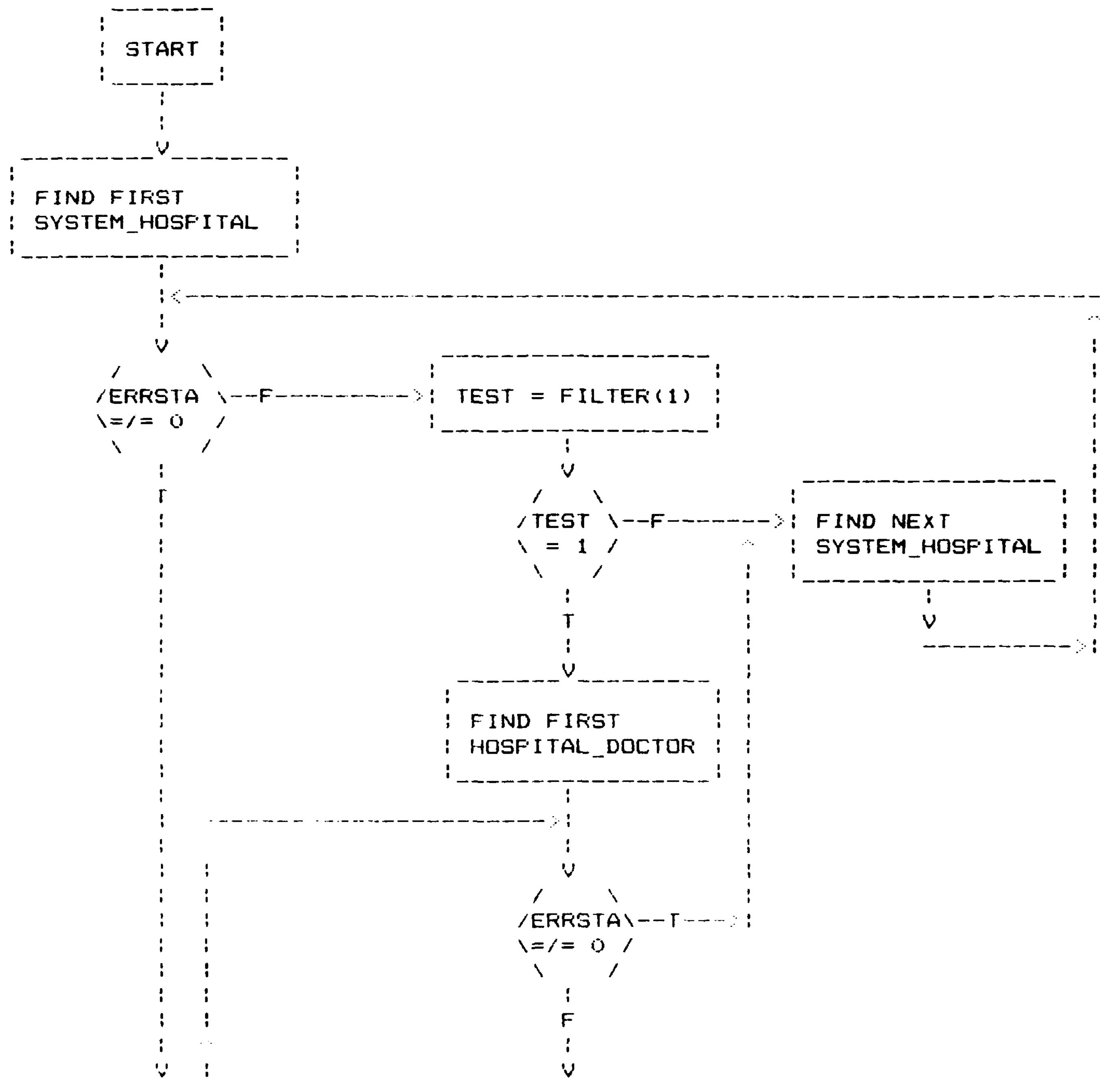

START
FIND FIRST SYSTEM_HOSPITAL
ERRSTA =/= 0
F
T
TEST = FILTER(1)
TEST = 1
F
FIND NEXT SYSTEM_HOSPITAL
T
FIND FIRST HOSPITAL_DOCTOR
ERRSTA =/= 0
T
F

Figure 3.6a Page 1 of 3.

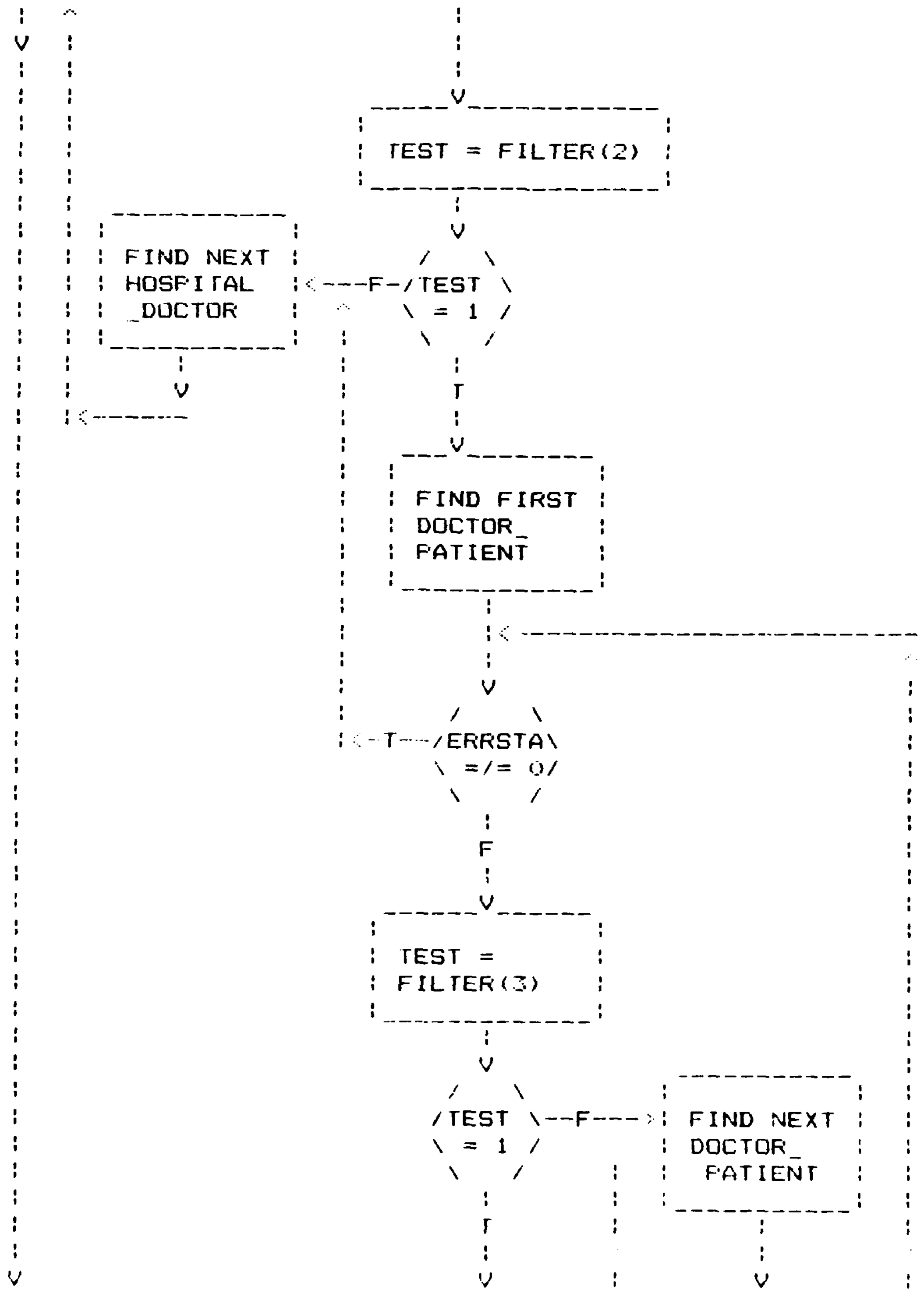
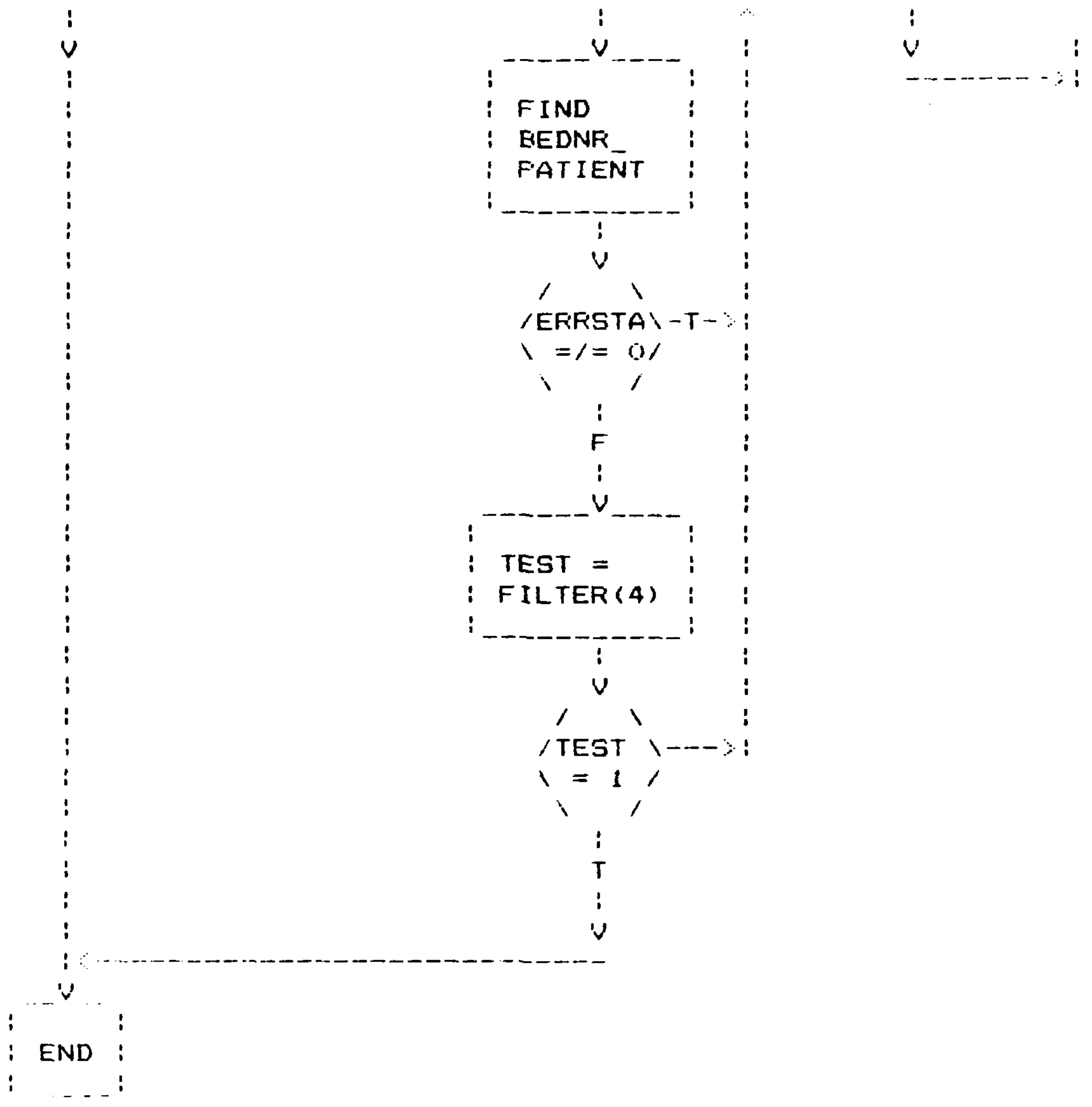

TEST = FILTER(2)
TEST = 1
F
T
FIND NEXT HOSPITAL _DOCTOR
FIND FIRST DOCTOR_ PATIENT
ERRSTA =/= 0
T
F
TEST = FILTER(3)
TEST = 1
F
FIND NEXT DOCTOR_ PATIENT
T

Figure 3.6a Page 2 of 3.



<u>Figure 3.6a</u> <u>Page 3 of 3.</u>

Figure 3.6a An Example Flow Chart for a FIRSTW Routine.

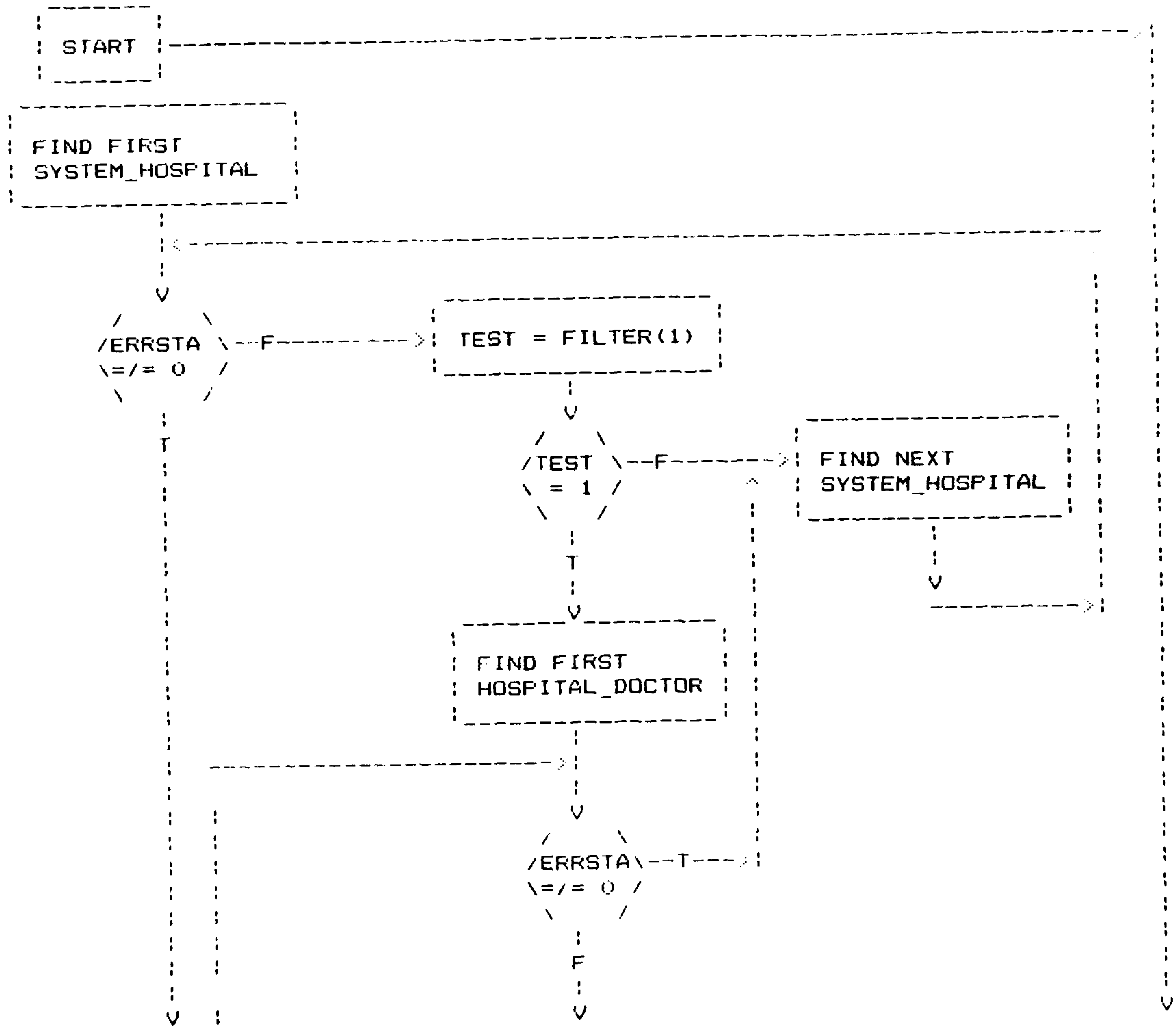
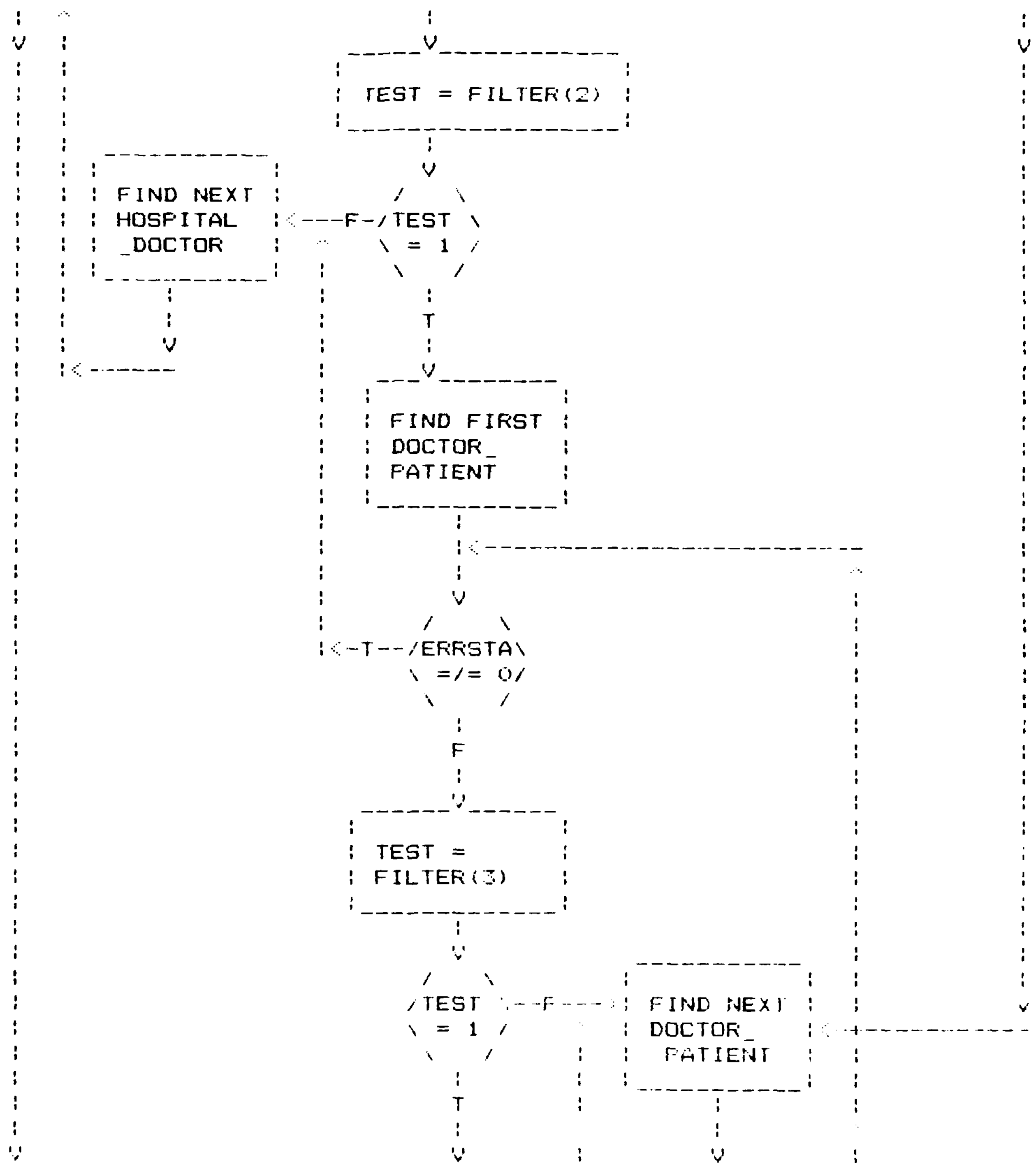

START
FIND FIRST SYSTEM_HOSPITAL
ERRSTA =/= 0
F
T
TEST = FILTER(1)
TEST = 1
F
T
FIND NEXT SYSTEM_HOSPITAL
FIND FIRST HOSPITAL_DOCTOR
ERRSTA =/= 0
T
F

Figure 3.6b Page 1 of 3.



TEST = FILTER(2)
FIND NEXT HOSPITAL _DOCTOR
F
TEST = 1
T
FIND FIRST DOCTOR_ PATIENT
T
ERRSTA =/= 0
F
TEST = FILTER(3)
TEST = 1
F
FIND NEXT DOCTOR_ PATIENT
T

Figure 3.6b Page 2 of 3.

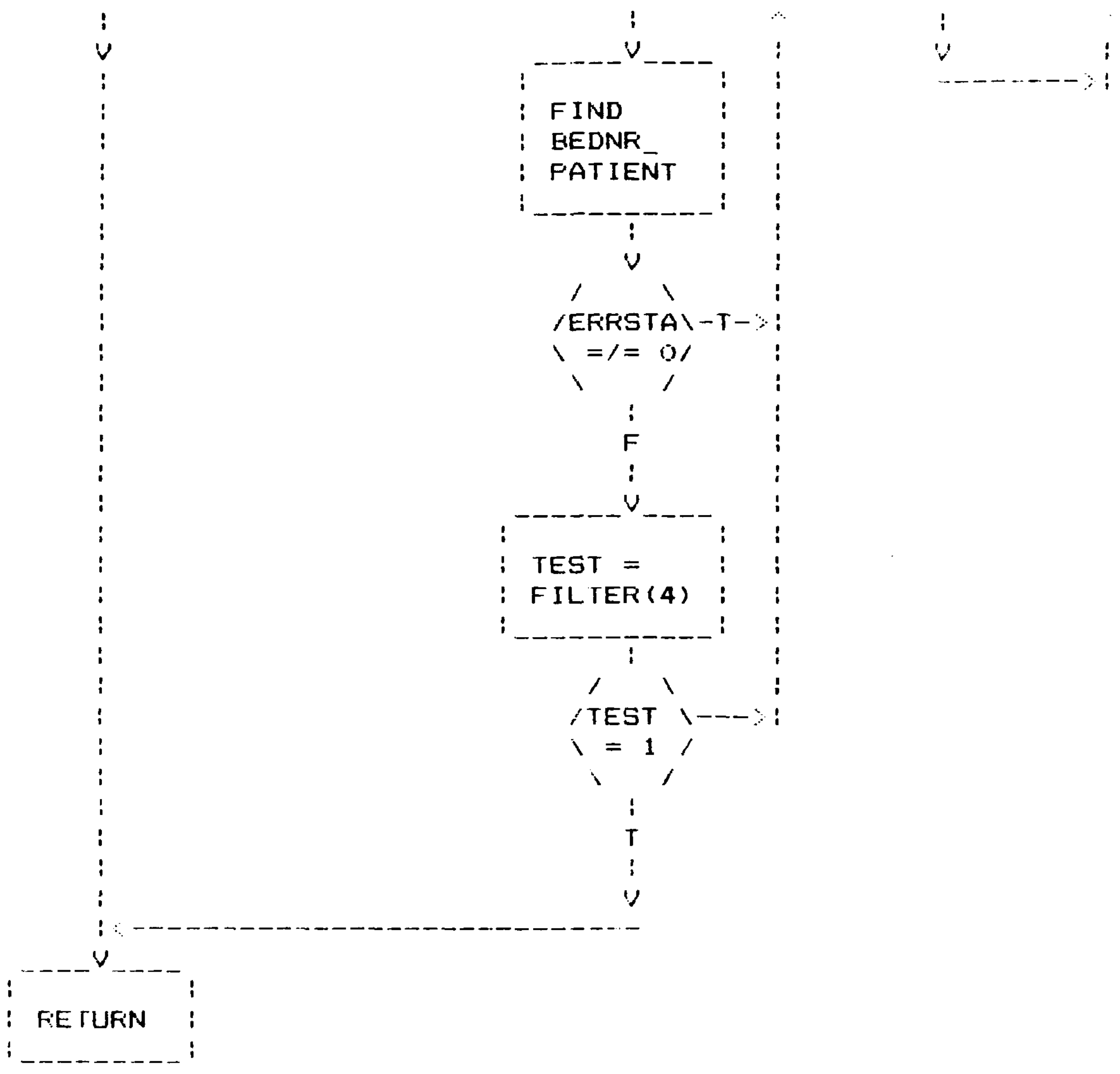


Figure 3.6b Page 3 of 3.

Figure 3.6b An Example Flow Chart for a NEXTW Routine.

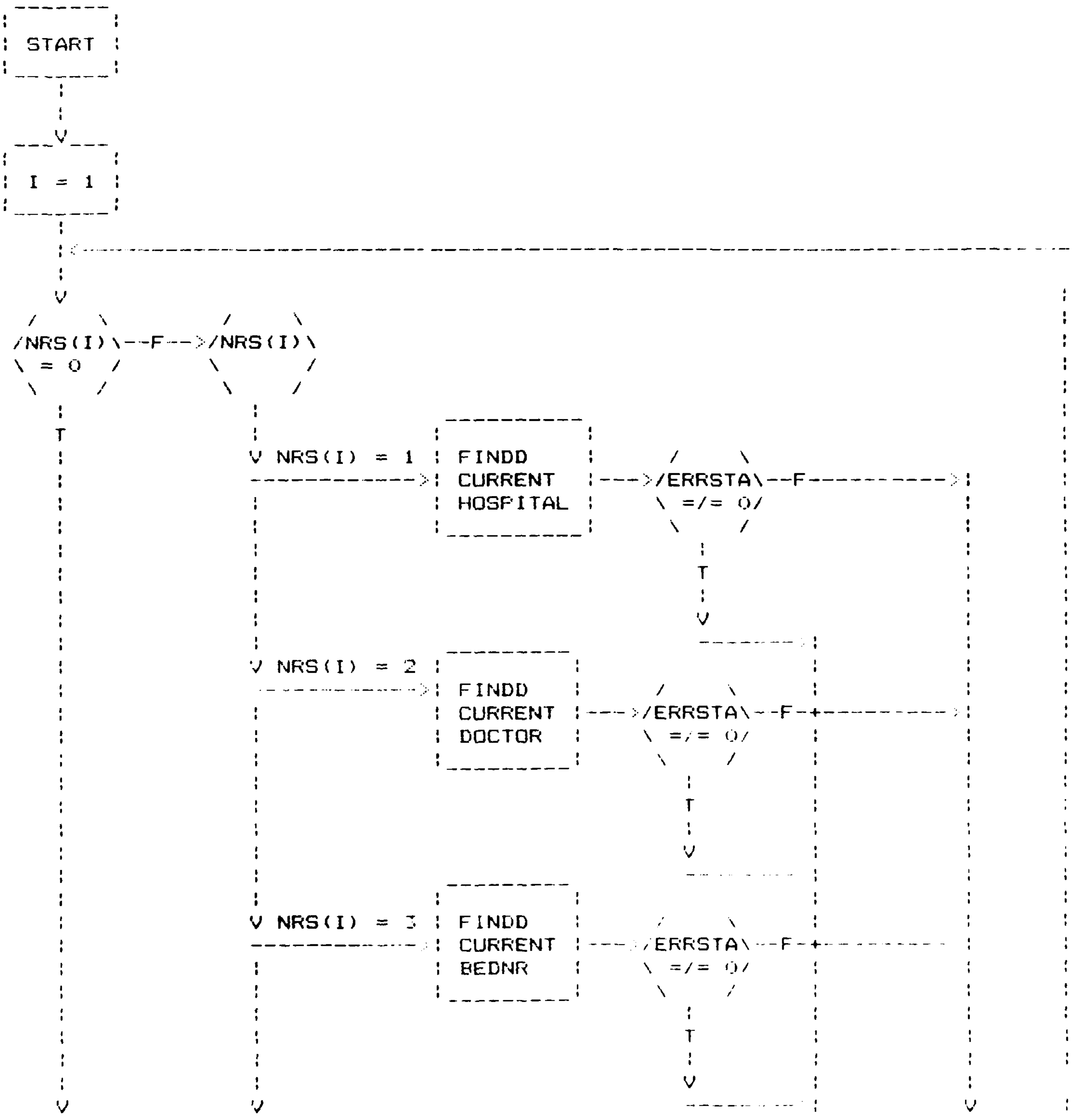


Figure 3.7 Page 1 of 2.

```
                               ----------
 :          :                 :          :                          :            :
 V          V NRS(I)  =  4    : FINDD    :     /      \             V            V          :
 :          ----------------->: CURRENT  :-->/ERRSTA\-F------+---------------->:          :
 :                            : PATIENT  :    \ =/= 0/       :                   :          :
 :                            :__________:     \      /      :                   :          :
 :                                                :          :                   :          :
 :                                                T          :                   :          :
 :                                                :          :                   :          :
 :                                                V          V                   :          :
 :<-------------------------------------------------------------                 :          :
 :                                                                               V          :
 :                                                                       --------V---       :
 :                                                                      :            :      :
 V                                                                      : I = I + 1  :      :
----V---                                                                :____________:      :
:        :                                                                      :           :
: RETURN :                                                                      :           :
:________:                                                                      V           :
                                                                                ------->:
```

Figure 3.7 Page 2 of 2.

Figure 3.7 A Flow Chart for an Example GRASP Routine.

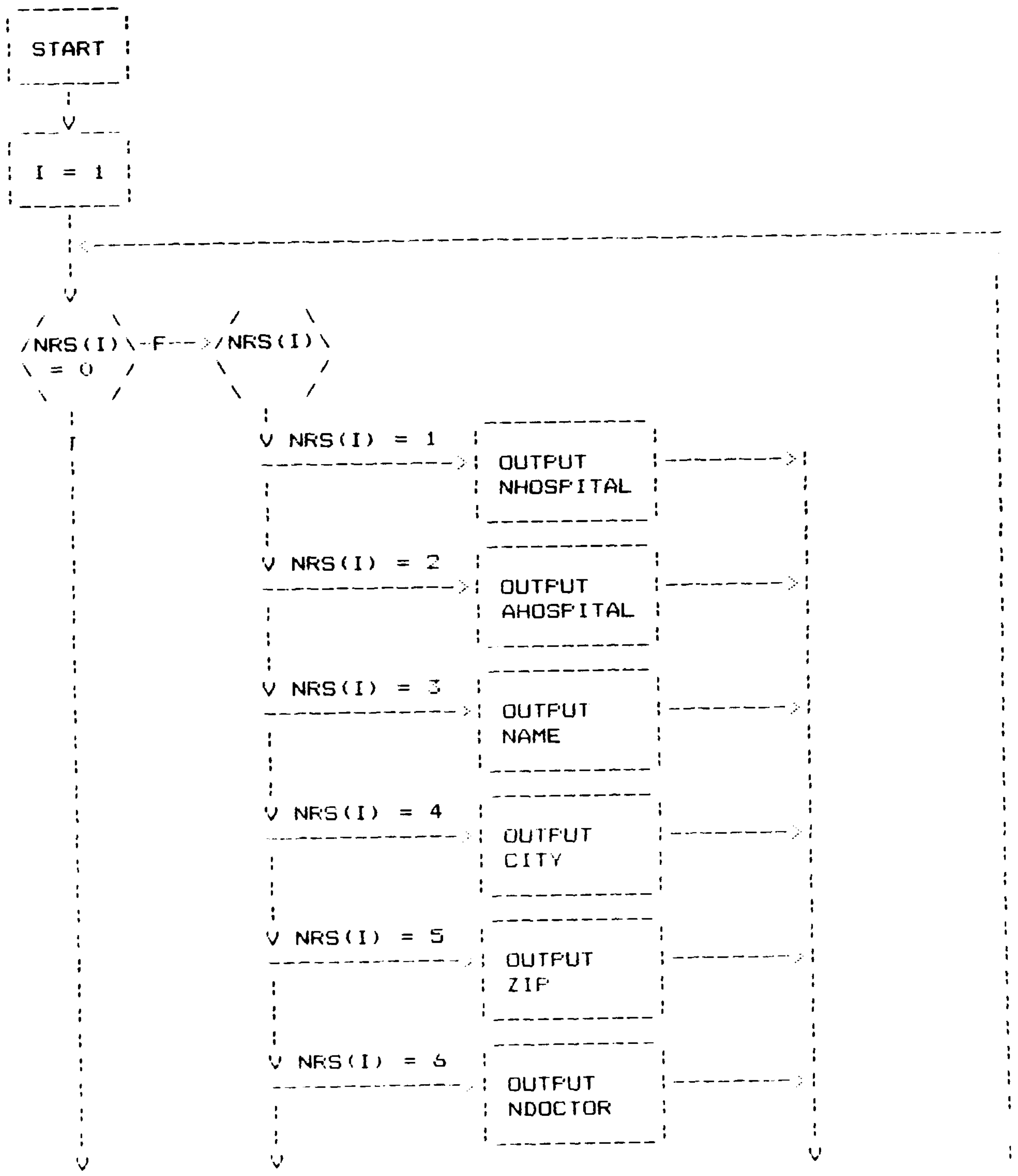


Figure 3.8 Page 1 of 3.

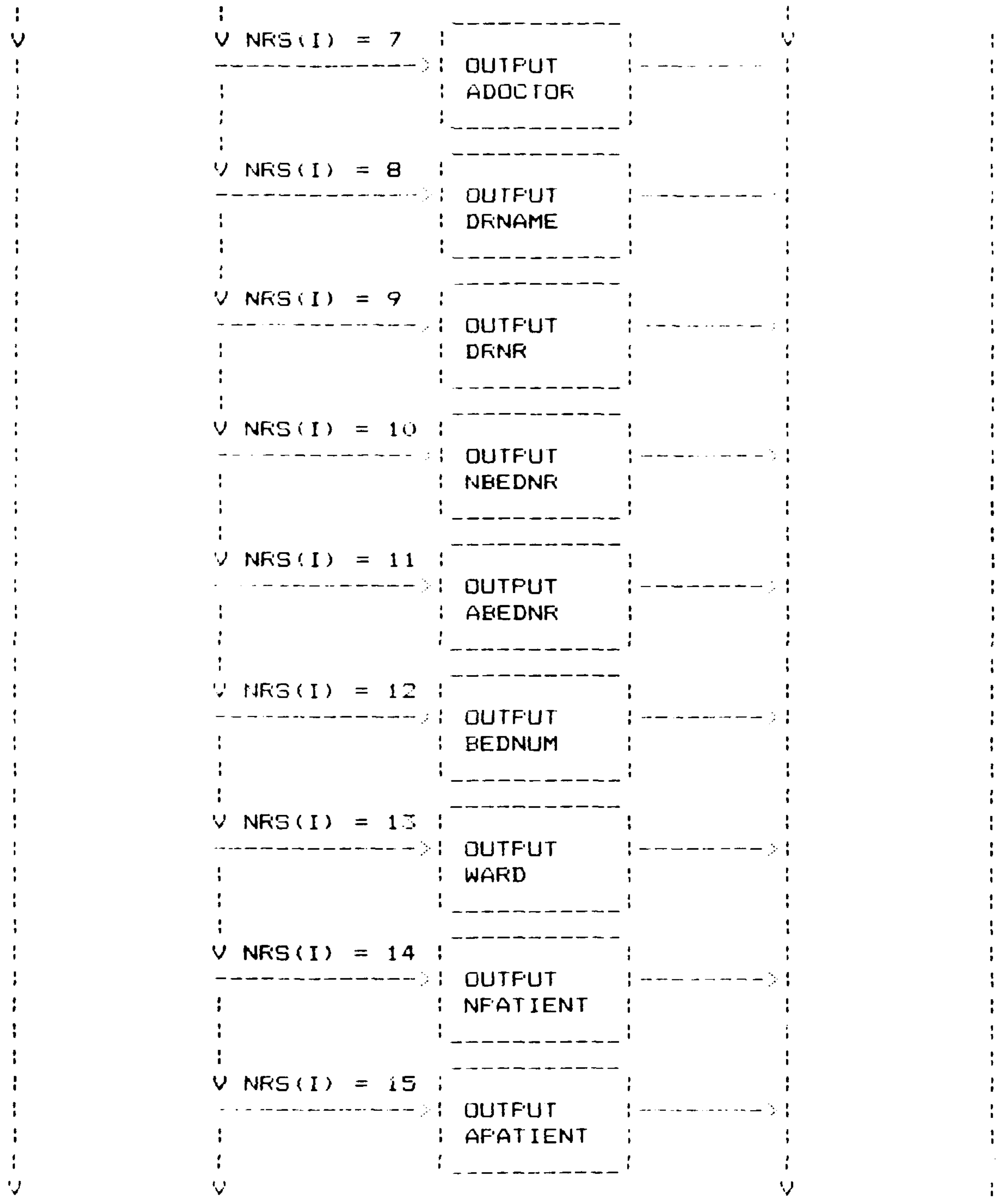
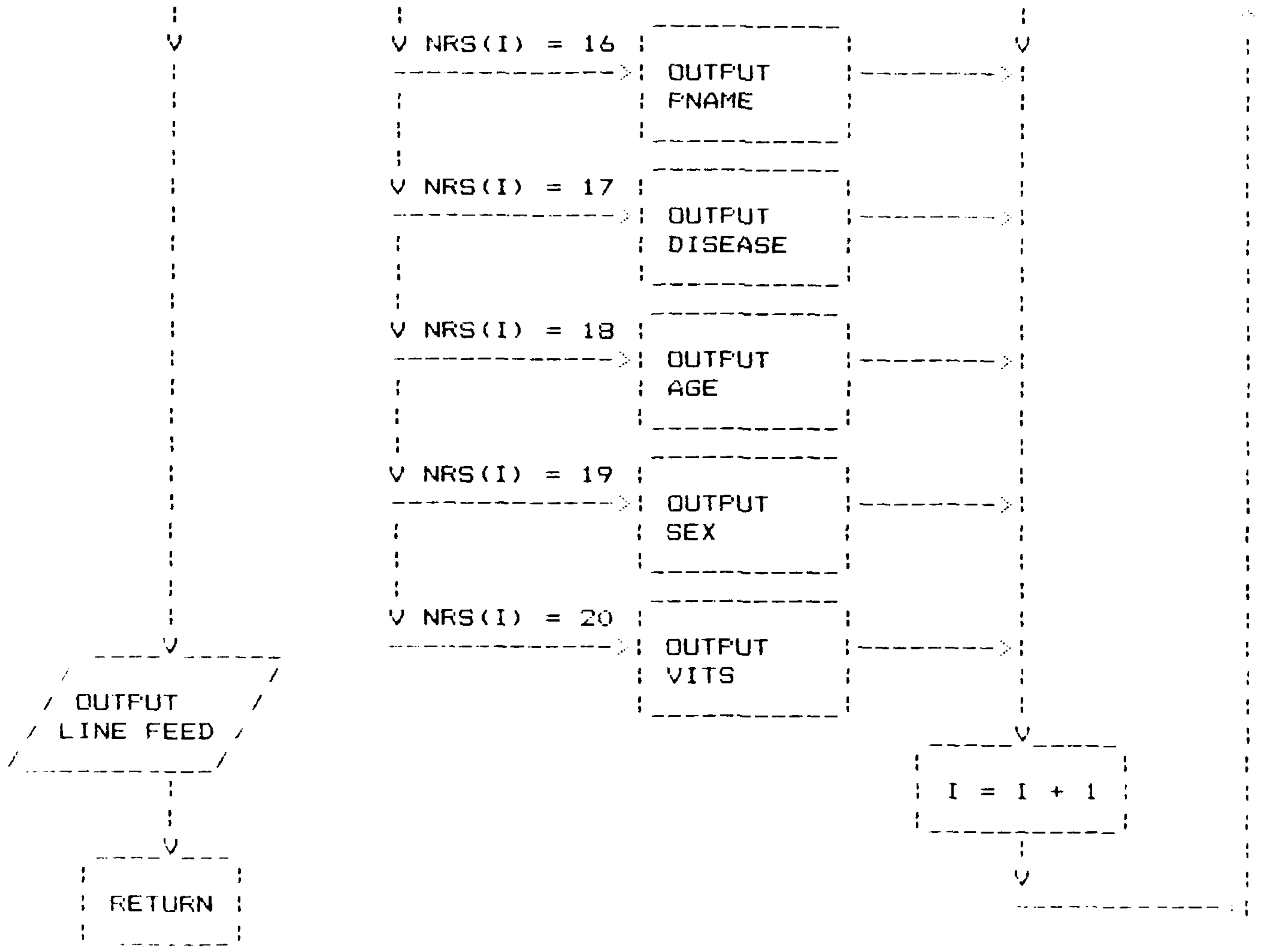


Figure 3.8 Page 2 of 3.



<u>Figure 3.8</u> <u>Page 3 of 3.</u>

Figure 3.8 A Flow Chart for an Example FILER Routine.

```
ıtwhere(start,fp)

OSEPH AULINO */
ıAVID PROJECT */
IAY 1984      */
              */
'HIS PROCEDURE CREATES FIRSTWHERE (CALLED FIRSTW) SUBROUTINES. */
'HE FIRSTW ROUTINES CREATED ARE USED TO EXTRACT THE FIRST      */
ICCURRENCE OF EACH RECORD IN A DATABASE WHICH MEETS CERTAIN     */
:RITERIA.  THE CRITERIA IS JUDGED IN THE FILTER SUBROUTINE     */
WHICH IS CALLED BUT NOT WRITTEN HERE).  THE ROUTINE TAKES AS   */
NPUT A POINTER THE MAIN DATA STRUCTURE AND A POINTER TO THE    */
'ILE USED FOR OUTPUT.  IT RETURNS A VALUE CALLED "SET" USED BY */
:OUTINE NEXTWHERE TO DETERMINE WHERE EXECUTION OF THE FORTRAN  */
'ROGRAM CREATED BY NEXTWHERE SHOULD BEGIN.                     */
                                                               */
.OCAL VARIABLES:                                               */
                                                               */
   1.  begin - USED TO TRAVERSE A LIST OF TABLE STRUCTURES.    */
   2.  follow - USED TO TRAVERSE A LIST OF TABLE STRUCTURES.   */
   3.  contin - USED TO TRAVERSE A LIST OF TABLE STRUCTURES.   */
   4.  travel - USED TO TRAVERSE A LIST OF HOOK STRUCTURES.    */
   5.  adder - HELPS GENERATE THE PROPER LINE NUMBERS BY OWN   */
               AND OWNEE.                                      */
   6.  count - HELPS GENERATE LINE NUMBERS AS IN ADDER.        */
   7.  last - HELPS GENERATE LINE NUMBERS AS IN ADDER.         */
   8.  swap - INDICATES WHEN FORTRAN VARIABLES MUST BE         */
              PROTECTED FROM BEING OVERWRITTEN.                */
                                                               */
ION-STANDARD SUBROUTINES USED:                                 */
                                                               */
   1.  commons - FOR CREATING THE NECESSARY COMMON STATEMENT.  */
   2.  own    FOR CREATING SECTIONS OF CODE FOR "GETTING" A    */
              RECORD GIVEN ITS PARENT.                         */
   3.  ownee   FOR CREATING SECTIONS OF CODE FOR "GETTING" A   */
               RECORD GIVEN ONE OF ITS "CHILDREN".             */
   4.  declarer   USED TO DECLARE CERTAIN COUNTER VARIABLES.   */
   5.  areset   USED TO RESET THE VALUE OF THE "TYPR" FIELD.   */

ıct table *start;
: *fp;

ıct table *begin, *follow;
ıct item *contin;
ıct hook *travel;
 adder, count, last;
 swap;
```

Figure 3.9a Page 1 of 4.

```
*************************************************************/
                                                            */
IE NEXT SECTION OF CODE CREATES THE FINAL GO TO STATEMENTS */
ND THE. END OF THE FORTRAN SUBROUTINE.                      */
                                                            */
*************************************************************/


ntf(fp,"\n\n 9990  I = I + 1");
ntf(fp,"\n        GO TO 1");
ntf(fp,"\n\n 9993  WRITE(6,9999)");
ntf(fp,"\n 9999  FORMAT('/')");
ntf(fp,"\n        RETURN\n\n        END\n\n");

et(start);

rn;

 /*****     END OF THE OUTPT ROUTINE     *****/
```

<u>Figure</u> <u>3.11a</u> <u>Page</u> <u>3</u> <u>of</u> <u>3.</u>

Figure 3.11a The outpt Routine C Code.

```
uct table *begin;
uct hook *travel;
 numb;


****************************************************
                                                   */
FIRST, THE STATIC INFORMATION, COMMON STATEMENT, AND */
OTHER OVERHEAD INFORMATION IS WRITTEN.              */
                                                   */
****************************************************/


p = 2;
intf(fp,"\n\n        SUBROUTINE FILER(NRS,N)");
intf(fp,"\n\n        INTEGER N,NRS(N),I");
mons(start,fp);
intf(fp,"\n        I = 1");
intf(fp,"\n\n 1       IF (NRS(I) .EQ. 0) GO TO 9998");
owrite(start,fp);


*********************************************************/
                                                        */
NEXT, FOR EACH TABLE AND ALL FIELDS OF THAT TABLE A WRITE */
STATEMENT IS PRODUCED.  IF A TABLE IS A LINK FOR A       */
"RECURSIVE" TABLE, THE FIELDS OF THAT NODE ARE ALSO      */
OUTPUT.                                                  */
                                                        */
*********************************************************/

in = start;

le (begin != 0)

filwrite(fp,begin,&numb);
if ((*begin).rhook != 0)
{
   travel = (*begin).rhook;
   while (travel != 0)
   {
      filwrite(fp,(*travel).atab,&numb);
      travel = (*travel).nhook;
   }
}
begin = (*begin).ntable;
```

Figure 3.11a Page 2 of 3.

```
tpt(start,fp)

 JOSEPH AULINO */
 DAVID PROJECT */
 JUNE 1984     */
               */
 THIS PROGRAM CREATES A FORTRAN OUTPUT PROGRAM FOR EACH DATA */
 BASE CREATED FOR THE "SEED" DBMS.  THE PROGRAM IS DESIGNED  */
 (CURRENTLY) TO RUN IN A "UNIX" ENVIRONMENT.  IT OUTPUTS THE */
 REQUIRED  FIELDS FROM THE DBMS (INCLUDING POINTERS IF       */
 DESIRED).  THE FORTRAN PROGRAM TAKES AS INPUT A VECTOR      */
 "NRS" OF SIZE "N".  THE NUMBERS IN THIS VECTOR CORRESPOND   */
 TO FIELDS/TABLES OF THE DATABASE.  FOR INSTANCE, 1 EQUALS   */
 THE CURRENCY POINTER OF THE FIRST TABLE, 2 EQUALS THE       */
 LOGICAL POINTER FOR THE FIRST TABLE, 3 EQUALS THE FIRST     */
 FIELD OF THE FIRST TABLE,...N EQUALS THE Nth FIELD OF THE   */
 FIRST TABLE, N+1 EQUALS THE CURRENCY POINTER OF THE 2nd     */
 TABLE, N+2 EQUALS THE LOGICAL POINTER OF THE 2nd TABLE ETC. */
 SHOULD THE APPLICABLE NUMBER APPEAR ANYWHERE IN THE "NRS"   */
 VECTOR, THE CORRESPONDING ENTRY WILL BE PRINTED AT THAT     */
 POINT.  ENTRIES ARE PRINTED ON ONE ROW PER CALL TO THE      */
 FORTRAN PROGRAM.  THE TERMINATION CONDITION FOR THE FORTRAN */
 PROCEDURE IS A "0" ENTRY IN THE "NRS" VECTOR.               */
                                                             */
                                                             */
 LOCAL VARIABLES:                                            */
                                                             */
      1.  begin - TRAVERSES A LIST OF TABLE STRUCTURES.      */
      2.  travel - TRAVERSES A LIST OF HOOK STRUCTURES.      */
      3.  numb - USED TO DENOTE A LINE NUMBER.               */
                                                             */
                                                             */
 NON-STANDARD SUBROUTINES USED:                              */
                                                             */
      1.  commons - WRITES THE NECESSARY COMMON STATEMENT.   */
      2.  gotowrite - WRITES THE NUMBERS OF THE COMPUTED     */
                      GOTO STATEMENT USED IN THE FORTRAN     */
                      PROGRAM.                               */
      3.  filwrite - WRITES ALL THE OUTPUT STATEMENTS USED   */
                     BY THE FORTRAN PROGRAM.                 */
      4.  areset - RESETS THE "RCNT" FIELD OF EACH TABLE     */
                   STRUCTURE AS THE LAST THING DONE IN THE   */
                   PROGRAM.                                  */


ruct table *start;
LE *fp;
```

Figure 3.11a Page 1 of 3.

```
        if ((*(*travel).atab).rcnt != 0)
           transf((*travel).atab,fp);
        fprintf(fp,"\n        GO TO 9990\n");

        if ((*(*travel).atab).rcnt != 0)
           (*(*travel).atab).rcnt--;

        travel = (*travel).nhook;

     } /* END OF THE WHILE TRAVEL != 0 LOOP */
  } /* END OF THE IF BEGIN... */

  begin = (*begin).ntable;


*************************************************************/
*                                                           */
* LASTLY, THE LAST LINES OF THE FORTRAN PROGRAM ARE OUTPUT */
* TO INCLUDE THE FINISHING OF THE "WHILE" LOOP AND THE     */
* RETURN STATEMENT.                                        */
*                                                           */
*************************************************************/


printf(fp,"\n\n 9990  I = I + 1");
printf(fp,"\n        GO TO 1");

printf(fp,"\n\n 9999  RETURN");
printf(fp,"\n\n        END\n\n");

reset(start);

eturn;

     /*****     END OF THE GRASPER ROUTINE     *****/
```

<u>Figure 3.10</u> <u>Page 4 of 4.</u>

Figure 3.10 The grasper Routine C Code.

```
/*********************************************************/
/*                                                       */
/* THE NEXT SECTION OF CODE GENERATES THE ACTUAL FETCHES */
/* AND "GETS" AS NEEDED. IT IS ALSO HERE THAT TRANSF IS  */
/* CALLED IF NEEDED.                                     */
/*                                                       */
/*********************************************************/

begin = start;
addon = 1;
numb = 2;

while (begin != 0)
{
   fprintf(fp,"\n %d      CALL FINDD(N%s",numb,(*begin).aname);
   if ((*begin).rcnt != 0)
      fprintf(fp,"%d",(*begin).rcnt);
   fprintf(fp,")");
   fprintf(fp,"\n         IF (ERRSTA .NE. 0) GO TO 9999");
   fprintf(fp,"\n         CALL GET(%s)",(*begin).aname);
   fprintf(fp,"\n         TEST = FILTER(%d)",addon);
   fprintf(fp,"\n         IF (TEST .NE. 1) GO TO 9999");
   numb++;
   addon++;

   if ((*begin).rcnt != 0)
      transf(begin,fp);
   fprintf(fp,"\n         GO TO 9990\n");
   if ((*begin).rcnt != 0)
      (*begin).rcnt--;

   if ((*begin).rhook != 0)
   {
      travel = (*begin).rhook;

      while (travel != 0)
      {
         fprintf(fp,"\n %d      CALL FINDD(N%s",numb,(*(*travel).atab).aname);
         if ((*(*travel).atab).rcnt != 0)
            fprintf(fp,"%d",(*(*travel).atab).rcnt);
         fprintf(fp,")");
         fprintf(fp,"\n         IF (ERRSTA .NE. 0) GO TO 9999");
         fprintf(fp,"\n         CALL GET(%s)", (*(*travel).atab).aname);
         fprintf(fp,"\n         TEST = FILTER(%d)",addon);
         fprintf(fp,"\n         IF (TEST .NE. 1) GO TO 9999");
         numb++;
         addon++;
```

Figure3.10 Page 3 of 4.

```
/***************************************************************/
/*                                                             */
/* FIRST THE HEADING AND OTHER STATIC INFORMATION IS WRITTEN.  */
/*                                                             */
/***************************************************************/


fprintf(fp,"\n\n       SUBROUTINE GRASP(NRS,N,TEST)");
fprintf(fp,"\n\n       INTEGER NRS,N,I,TEST");

commons(start,fp);
fprintf(fp,"\n\n       TEST = 1");

fprintf(fp,"\n\n       I = 1");
fprintf(fp,"\n\n 1     IF (NRS(I) .EQ. 0) GO TO 9999");
fprintf(fp,"\n\n       GO TO(");

/***********************************************************/
/*                                                         */
/* NEXT BY TRAVERSING THE DATA STRUCTURE AND USING GORITE */
/* THE COMPUTED GO TO STATEMENT IS CREATED.               */
/*                                                         */
/***********************************************************/


begin = start;
numb = 2;
incre = 1;
adder = 3;
sum = 12;

while (begin != 0)
{
   gorite(fp,&sum,&adder,&numb,incre);

   if ((*begin).rhook != 0)
   {
      travel = (*begin).rhook;
      while (travel != 0)
      {
         gorite(fp,&sum,&adder,&numb,incre);
         travel  = (*travel).nhook;
      }
   }
   begin = (*begin).ntable;
}
fprintf(fp,"),\n     *       NRS(I)\n");
```

Figure 3.10 Page 2 cf 4.

```
grasper(start,fp)

/* JOSEPH AULINO */
/* DAVID PROJECT */
/* MARCH 1984    */
/*               */
/* THIS PROCEDURE WRITES THE FORTRAN CODE TO RETRIEVE ANY RECORD */
/* FROM THE DATABASE GIVEN ITS CURRENCY POINTER.  TO DO THIS THE */
/* FORTRAN CODE WILL TAKE AS INPUT THE VECTOR "NRS" OF SIZE "N". */
/* THIS VECTOR WILL CONTAIN NUMBERS CORRESPONDING TO THE RECORDS */
/* TO BE FETCHED.  TO RETRIEVE THE FIRST RECORD YOU WOULD        */
/* INCLUDE THE NUMBER 1 IN THIS VECTOR, THE SECOND RECORD WOULD  */
/* BE NUMBER 2 AND SO ON.  IT ALSO PRESUPPOSES THAT THE CURRENCY */
/* POINTER FOR ANY RECORD IS IN ITS APPROPRIATE VARIABLE - THAT  */
/* BEING THE VARIABLE WITH THE SAME NAME AS THE RECORD BUT WITH  */
/* AN "N" IN FRONT OF IT.                                        */
/*                                                               */
/*                                                               */
/* LOCAL VARIABLES:                                              */
/*                                                               */
/*      1.  begin - TRAVERSES A LIST OF TABLE STRUCTURES.        */
/*      2.  numb - USED BY GORITE TO HELP PRODUCE LINE NUMBERS.  */
/*      3.  incre - USED BY GORITE TO HELP PRODUCE LINE NUMBERS. */
/*      4.  adder - USED BY GORITE TO HELP PRODUCE LINE NUMBERS. */
/*      5.  sum - USED BY GORITE TO HELP PRODUCE LINE NUMBERS.   */
/*      6.  addon - SPECIFIES WHAT RECORD IS BEING SENT TO THE   */
/*                  FILTER.                                      */
/*                                                               */
/*                                                               */
/* NON-STANDARD SUBROUTINES USED:                                */
/*                                                               */
/*      1.  commons   PRODUCES THE NECESSARY COMMON STATEMENT.   */
/*      2.  gorite - PRODUCES THE NUMBERS IN THE COMPUTED GO TO  */
/*                   STATEMENT.                                  */
/*      3.  transf   TRANSFERS THE VALUES OF CERTAIN VARIABLES TO */
/*                   TEMPORARY STORAGE (IN THE FORTRAN CODE) TO  */
/*                   PREVENT THEM FROM BEING OVERWRITTEN.        */
/*      4.  areset   RESETS THE TYPR FIELD OF TABLE STRUCTURES.  */


struct table *start;
FILE *fp;
{
struct table *begin;
int numb, incre, adder, sum, addon;
struct hook *travel;
```

Figure 3.10 Page 1 of 4.

```
    if ((*begin).rhook != 0)
    {  swap = 1;
       travel = (*begin).rhook;
       while (travel != 0)
       {  contin = (*(*travel).atab).to_items;
          while (contin != 0)
          {
             if (((*contin).typ != 'T')&&((*contin).typ != 'R'))
             {
               fprintf(fp,"\n      %s%d",(*contin).iname,(*(*travel).atab).rcn);
               fprintf(fp," = %s",(*contin).iname);
             }
             contin = (*contin).nitem;
          }
          fprintf(fp,"\n        A");
          fprintf(fp,"%s%d =A",(*(*travel).atab).aname,(*(*travel).atab).rcnt);
          fprintf(fp,"%s",(*(*travel).atab).aname);
          fprintf(fp,"\n        N");
          fprintf(fp,"%s%d = N",(*(*travel).atab).aname,(*(*travel).atab).rcnt);
          fprintf(fp,"%s",(*(*travel).atab).aname);
          fprintf(fp,"\n        ITEMP = C%s",(*(*travel).atab).aname);
          fprintf(fp,"\n        C%s",(*(*travel).atab).aname);
          fprintf(fp," = C%s%d",(*(*travel).atab).aname,(*(*travel).atab).rcnt);
          fprintf(fp,"\n        C%s",(*(*travel).atab).aname);
          fprintf(fp,"%d = ITEMP",(*(*travel).atab).rcnt);
          follow = (*travel).atab;
          own(fp,follow,begin,adder,count,last,swap);
          adder++;
          last = count + 1;
          count = count + 4;
          (*(*travel).atab).rcnt--;
          travel = (*travel).nhook;
    } }
    swap = 0;
    begin = (*begin).ntable;
}
fprintf(fp,"\n 9999  RETURN\n");
fprintf(fp,"\n       END\n");
areset(start);
return;
}     /*****     END OF THE NEXTWHERE ROUTINE      *****/
```

<u>**Figure 3.9b**</u> <u>**Page 4 of 4.**</u>

Figure 3.9b The nextwhere Routine C Code.

```
      begin = (*begin).ntable;
   }
   fprintf(fp,"\n");
   findgo(start,set);


   /***********************************************************/
   /*                                                         */
   /* NOW THE REMAINDER OF THE CODE IS GENERATED JUST AS IT WAS */
   /* IN THE "firstwhere" ROUTINE.                            */
   /*                                                         */
   /***********************************************************

   begin = start;
   adder = 1;
   count = 1;
   last = 0;
   swap = 0;

   while (begin != 0)
   {
      if (((*begin).relat = 'C') || ((*begin).relat = ' '))
      {
         follow = (*begin).for_write;
         own(fp,begin,follow,adder,count,last,swap);
         adder++;
         last = count + 1;
         count = count + 4;
      }
      else
      {
         follow = (*begin).for_write;
         ownee(fp,begin,follow,adder,count,last);
         adder++;
         count = count + 4;
      }


   /*******************************************************/
   /*                                                     */
   /* THE LAST SECTION OF CODE GENERATES THE CODE NEEDED TO */
   /* FETCH A RECURSIVE RECORD.  THIS WILL INCLUDE        */
   /* PROTECTING ANY VARIABLES ALREADY HOLDING VALUES.    */
   /*                                                     */
   /*******************************************************/
```

Figure 3.9b Page3 of 4.

```
struct hook *travel;
int adder, count, last;
int swap;

/**********************************************************/
/*                                                        */
/* FIRST THE STATIC INFORMATION IS CREATED. THIS INCLUDES */
/* (BUT IS NOT LIMITED TO) THE INFORMATION CREATED BY     */
/* "commons" AND "declarer".                              */
/*                                                        */
/**********************************************************/


fprintf(fp,"\n      SUBROUTINE NEXTW(N,TEST)\n");
fprintf(fp,"\n      INTEGER TEST,N");
fprintf(fp,"\n");
declarer(start,fp);
fprintf(fp,"\n");
commons(start,fp);
fprintf(fp,"\n");


/***************************************************************/
/*                                                             */
/* NEXT THE COUNTERS ARE INITIALIZED TO THEIR VALUES AFTER THE */
/* LAST CALL TO "nextwhere" or "firstwhere".  ALSO, "findgo"   */
/* IS CALLED TO CREATE THE "GO TO" STATEMENT TO DETERMINE      */
/* WHERE EXECUTION SHOULD BEGIN.                               */
/*                                                             */
/***************************************************************/


begin = start;
while (begin != 0)
{
   fprintf(fp,"\n      C%s = N%s",(*begin).aname,(*begin).aname);
   if ((*begin).typr != 0)
   {
      while ((*begin).rcnt != 0)
      {
         fprintf(fp,"\n      C%s",(*begin).aname);
         fprintf(fp,"%d = N%s%d",(*begin).rcnt,(*begin).aname,(*begin).rcnt);
         (*begin).rcnt--;
      }
      (*begin).rcnt = (*begin).typr;
   }
```

Figure 3.9b Page 2 of 4.

```
nextwhere(start,set,fp)

/* JOSEPH AULINO */
/* DAVID PROJECT */
/* JUNE 1984     */
/*               */
/* THIS PROCEDURE GENERATES THE FORTRAN TO GET THE NEXT */
/* RECORD FROM ANY DATABASE FOR WHICH A SCHEMA AND A    */
/* FIRSTWHERE ROUTINE HAVE BEEN GENERATED.  IT IS       */
/* AFTER "firstwhere" IS CALLED.  THE CODE GENERATED IS */
/* NEARLY IDENTICAL FOR THE CODE FOR "firstwhere".  THE */
/* MAJOR DIFFERENCE IS WHERE IN THE CODE THE EXECUTION  */
/* ACTUALLY BEGINS. IN "nextwhere" THE EXECUTION BEGINS */
/* AT THE LOWEST "LEVEL" POSSIBLE AND WORKS BACK UP TO  */
/* THE HIGHEST LEVEL AS NEEDED.                         */
/*                                                      */
/* LOCAL VARIABLES:                                     */
/*                                                      */
/*      1.  begin - USED TO TRAVERSE A LIST OF TABLE    */
/*                  STRUCTURES.                         */
/*      2.  follow - USED TO TRAVERSE A LIST OF TABLE   */
/*                   STRUCTURES.                        */
/*      3.  contin - USED TO TRAVERSE A LIST OF TABLE   */
/*                   STRUCTURES.                        */
/*      4.  adder - USED IN SETTING LINE NUMBERS.       */
/*      5.  count - USED IN SETTING LINE NUMBERS.       */
/*      6.  last - USED IN SETTING LINE NUMBERS.        */
/*      7.  swap - INDICATES WHEN A FORTRAN VARIABLE    */
/*                 MUST BE PROTECTED FROM BEING         */
/*                 OVERWRITTEN.                         */
/*                                                      */
/* NON-STANDARD SUBROUTINES USED:                       */
/*                                                      */
/*      1.  findgo - DETERMINES THE LINE NUMBER WHERE   */
/*                   THE FORTRAN PROGRAM SHOULD BEGIN.  */
/*      2.  areset   USED TO RESET THE "TYPR" FIELD.    */
/*      3.  declarer - DECLARES COUNTER VARIABLES.      */
/*      4.  commons - CREATES THE NECESSARY COMMON      */
/*                    STATEMENT.                        */

struct table *start;
int set;
FILE *fp;
{
struct table *begin, *follow;
struct item *contin;
```

Figure 3.9b Page 1 of 4.

```
            fprintf(fp,"\n        A");
            fprintf(fp,"%s%d = A",(*(*travel).atab).aname,(*(*travel).atab).rcnt);
            fprintf(fp,"%s",(*(*travel).atab).aname);
            fprintf(fp,"\n        N");
            fprintf(fp,"%s%d = N",(*(*travel).atab).aname,(*(*travel).atab).rcnt);
            fprintf(fp,"%s",(*(*travel).atab).aname);
            fprintf(fp,"\n        ITEMP = C%s",(*(*travel).atab).aname);
            fprintf(fp,"\n        C");
            fprintf(fp,"%s = C%s",(*(*travel).atab).aname,(*(*travel).atab).aname);
            fprintf(fp,"%d",(*(*travel).atab).rcnt);
            fprintf(fp,"\n        C%s",(*(*travel).atab).aname);
            fprintf(fp,"%d = ITEMP",(*(*travel).atab).rcnt);
            follow = (*travel).atab;
            own(fp,follow,begin,adder,count,last,swap);
            adder++;
            last = count + 1;
            count = count + 4;
            (*(*travel).atab).rcnt--;
            travel = (*travel).nhook;
          }
      }
      swap = 0;
      begin = (*begin).ntable;
   } /* END OF THE WHILE LOOP */

   fprintf(fp,"\n 9999  RETURN\n");
   fprintf(fp,"\n        END\n")

   areset(start);  /* THE VALUE OF THE "TYPR" FIELD IS RESET. */

   return(count);

   }     /*****     END OF THE FIRSTWHERE ROUTINE     *****/
```

Figure 3.9a Page 4 of 4.

Figure 3.9a The firstwhere Routine C Code.

```
begin = start;
adder = 1;
count = 1;
last = 0;
swap = 0;

while (begin != 0)
{
   if (((*begin).relat == 'C') || ((*begin).relat == ' '))
   {
      follow = (*begin).for_write;
      own(fp,begin,follow,adder,count,last,swap);
      adder++;
      last = count + 1;
      count = count + 4;
   }
   else
   {
      follow = (*begin).for_write;
      ownee(fp,begin,follow,adder,count,last);
      adder++;
      count = count + 4;
   }

/***************************************************************/
/*                                                             */
/* THE NEXT SECTION OF CODE IS USED TO GENERATE THE SPECIAL   */
/* CODE NEEDED IF A "LINK" NODE IS REACHED.  THIS IS NEEDED   */
/* TO SAVE ANY VARIABLE WHICH MAY BE "OVERWRITTEN" WHEN THE   */
/* "RECURSIVE" RECORD IS "FETCHED".                            */
/*                                                             */
/***************************************************************/

   if ((*begin).rhook != 0)
   {
      swap = 1;
      travel = (*begin).rhook;
      while (travel != 0)
      {
         contin = (*(*travel).atab).to_items;
         while (contin != 0)
         {
            if (((*contin).typ != 'T') && ((*contin).typ != 'R'))
            {
               fprintf(fp,"\n        %s",(*contin).iname);
               fprintf(fp,"%d = %s",(*(*travel).atab).rcnt,(*contin).iname);
            }
            contin = (*contin).nitem;
         }
```

Figure 3.9a Page 3 of 4.

```
/************************************************************************/
/*                                                                      */
/* FIRST THE STATIC INFORMATION COMMON TO ALL FIRSTW SUBROUTINE IS WRITTEN. */
/*                                                                      */
/************************************************************************/

fprintf(fp,"\n       SUBROUTINE FIRSTW(N,TEST)\n");
fprintf(fp,"\n       INTEGER TEST,N");
fprintf(fp,"\n");

/*********************************************************************/
/*                                                                   */
/* NEXT COMMONS IS CALLED TO WRITE OUT THE REQUIRED COMMON STATEMENT */
/* AND DECLARER IS CALLED TO WRITE OUT THE NECESSARY VARIABLES.      */
/*                                                                   */
/*********************************************************************/

declarer(start,fp);
fprintf(fp,"\n");
commons(start,fp);
fprintf(fp,"\n");

/************************************************************************/
/*                                                                      */
/* NOW USING OWN AND OWNEE AND THE RELATIONSHIPS DEVELOPED IN "RELATE" */
/* THE REMAINDER OF THE SUBROUTINE IS GENERATED.                        */
/*                                                                      */
/************************************************************************/

begin = start;

while (begin != 0)
{
   fprintf(fp,"\n       C%s = 1",(*begin).aname);
   if ((*begin).typr != 0)
   {
      while ((*begin).rcnt != 0)
      {
         fprintf(fp,"\n       C%s%d = 1",(*begin).aname,(*begin).rcnt);
         (*begin).rcnt--;
      }
      (*begin).rcnt = (*begin).typr;
   }
   begin = (*begin).ntable;
}
fptintf(fp,"\n");
```

Figure 3.9a Page 2 of 4.

```
filwrite(fp,path,numb)

/* JOSEPH AULINO */
/* DAVID PROJECT */
/* JUNE 1984     */
/*               */
/* THIS PROCEDURE WRITES THE CODE TO "WRITE" ONE RECORD AND ALL ITS */
/* FIELDS TO A FILE.  IT IS USED BY THE PROCEDURE OUTPT.  OUTPT     */
/* CALL FILWRITE ONE FOR EACH RECORD AND ONCE FOR EACH OCCURRENCE   */
/* OR A RECORD IN A RECURSIVE RELATIONSHIP.                         */
/*                                                                  */
/*                                                                  */
/* LOCAL VARIABLES:                                                 */
/*                                                                  */
/*      1.  i - A LOOP COUNTER.                                     */
/*      2.  temp - HELPS SET A FIELD SIZE.                          */
/*      3.  travel - TRAVERSES A LIST OF ITEM STRUCTURES.           */
/*                                                                  */
/*                                                                  */
/* NOTE -                                                           */
/*                                                                  */
/*      1.  path - POINTS TO A PARTICULAR TABLE STRUCTURE.          */
/*      2.  numb - DESIGNATES A LINE NUMBER.                        */


FILE *fp;
struct table *path;
int *numb;
{
int i, temp;
struct item *travel;


/*****************************************************************/
/*                                                               */
/* FIRST THE WRITE STATEMENTS FOR THE 2 POINTERS FOR EACH RECORD */
/* ARE PRODUCED.                                                 */
/*                                                               */
/*****************************************************************/


for (i = 0; i < 2; i++)
{
   fprintf(fp,"\n\n %d      WRITE(6,%d) ",*numb,*numb+1);
   if (i == 0)
      fprintf(fp,"N");
   else
      fprintf(fp,"A");
```

Figure 3.11b Page 1 of 3.

```
    fprintf(fp,"%s",(*path).aname);

    if((*path).rcnt != 0)
       fprintf(fp,"%d",(*path).rcnt);

    fprintf(fp,"\n %d      FORMAT('+',1X,I10)",*numb+1);
    fprintf(fp,"\n        GO TO 9990");
    *numb = *numb + 2;
}


/*******************************************************/
/*                                                     */
/* NEXT OUTPUT STATEMENTS ARE CREATED FOR EACH NON-TABLE */
/* FIELD OF A RECORD.                                  */
/*                                                     */
/*******************************************************/


travel = (*path).to_items;
while (travel != 0)
{
    if (((*travel).typ != 'T') && ((*travel).typ != 'R'))
    {
       fprintf(fp,"\n\n %d      WRITE(6,%d) ",*numb,*numb+1);
       fprintf(fp,"%s",(*travel).iname);

       if ((*path).rcnt != 0)
          fprintf(fp,"%d",(*path).rcnt);

       fprintf(fp,"\n %d      FORMAT('+',1X,",*numb+1);

       if ((*travel).typ == 'C')
          fprintf(fp,"A");
       if ((*travel).typ == 'I')
          fprintf(fp,"I");
       if ((*travel).typ == 'F')
          fprintf(fp,"F");

       fprintf(fp,"%d",(*travel).size);
       if ((*travel).typ == 'F')
       {
          temp = ((*travel).size)/2;
          fprintf(fp,".%d",temp);
       }
      fprintf(fp,")\n        GO TO 9990");
```

Figure 3.11b Page 2 of 3.

```
         *numb = *numb + 2;

      } /* END IF (*TRAVEL).TYP... */

      travel = (*travel).nitem;
   }

   if ((*path).rcnt > 0)
      (*path).rcnt--;

   return;


}     /*****     END OF THE FILWRITE ROUTINE     *****/
```

Figure 3.11b Page 3 of 3.

Figure 3.11b The filwrite Routine C Code.

CHAPTER 4. TRANSACTION TRANSLATION

4.0 INTRODUCTION

This chapter deals with the purpose of the C code used to create those Fortran routines which can effect the database. The main C routines which handle this are idu, indelup, and inserter. The general purpose of these routines is to create Fortran programs which can modify the database. These C routines make this possible by creating Fortran code to call the proper SEED defined subroutines to perform these operations.

In section 4.1, examples of each of the Fortran routines created are given. Section 4.2 discusses the general methodology for this creation and for the operation of the created routines. Next, section 4.3 gives the technical description of the C code and the Fortran code it creates. Lastly, section 4.4 gives the mainn C routines discussed.

4.1 AN EXAMPLE

This section contains one example of each of the Fortran routines created to oallow implementation of transaction translation. These routines are based on the non-recursive network DBL schema of section 1.1. The first of them is the basic routine to modify information in the database. This routine is named MODITE and an example of it is in figure 4.0.

The next example is for the basic routine to delete records (figure 4.1). This is also based on the non-recursive DBL schema in section 1.1. Note the similarities between this routine and the modify routine above.

The last example is for inserting new records into the database (figure 4.2). This routine is based on the same DBL schema as the other two. The differences between this routine and the two above are discussed in section 4.3 of this thesis.

4.2 THE GENERAL METHODOLOGY

The indelup routine acts as a "router" to idu. It "tells" idu whether to produce the delete routine or the modify routine. The routine opens the files for the delete and modify routines respectively. It then calls the idu routine to create the Fortran to delete or modify records. Then the current open file is closed. The difference between results is that idu is passed the name of the SEED subroutine. If passed "DELALL", it creates the delete routine. If passed "MODIFY", it creates the modify routine.

The idu routine traverses the schema structure and creates the necessary subroutine and error checking statements. The control mechanism created for the Fortran routines is similar to grasp's. The subroutine names created for the FORTRAN routines are derived from the SEED subroutine calls used in the respective routines. The control structure and the numbering correspondence for records is the same for both the delete and update routines.

The insert routine is more complex. Here recursive records must be considered and the actual inserts must be done in terms of sets. This entails ensuring that a record is inserted into all appropriate sets. It is simplified some by the "vanilla" nature of the database. This routine creates the necessary overhead statements and then the remainder of the code. The code is created by traversing the schema structure. For each table structure a store is created. Then by traversing the list of hook records pointed to by the for_sets field, inserts into all appropriate sets are created. If the rhook field is not zero, this is followed by creation of stores and inserts to recursive records. In this way, the created Fortran program can insert any record of any type into the database.

This is all that the C routines do in this process. The Fortran routines which they create will be what actually effects the database. These routines are described next.

MODITE is the routine to modify records in an existing database. It takes a NRS array and a dimension argument as input. NRS is used to control a computed goto statement and while loop with "0" as the loop terminator. A successful update can be tested by checking the value of the ERRSTA variable. The user must ensure the main program has the correct currency of the record or records to be modified. To modify recursive occurrences of a record, the routine may have to be called twice; once to modify the "original" record and once to modify the recursive occurrence.

DELITE takes the same arguments as the MODITE and has the same control structure and number correspondences. The above precautions also apply. Here the cautions are stronger, since it is possible to lose records and any records owned by them. For recursive records to be deleted, a separate call to this routine is needed.

INSRTN is the routine for inserting new records into the database. This has the same control structure as the other two routines but the number correspondence scheme is more complex. The precautions hold here also but are not as serious since if a mistake is made most likely an error will result and no information will be lost. There are limitations however. These are imposed by the vanilla nature of the database. The most important one is that when any record is inserted it is inserted into all sets in which it is a member (excluding sets where it has a recursive occurrence). Hence, "selective" insertions are not possible. However, in later versions this will most likely be changed.

This ends the block description of the code. In the last two sections, I give a technical description of the code. This description includes discussions of the problems mentioned above and requirements of the main program relating to these subroutines. Also, input arguments and output of these two routines is discussed.

4.3 A DETAILED DESCRIPTION

Indelup is the first routine discussed here. It is called from the main program. It's sole argument is a pointer to the schema structure. This is a simple program used as a router to enable idu to create the two Fortran routines DELITE and MODITE. The routine consists of opening the appropriate files, calling idu with the proper parameters, and closing the appropriate files. Control then returns to the main program. Upon return to the main program, two files have been created with one Fortran routine in each.

The variable "fp" is a pointer to the file opened by the fopen function. The variable "start" points to the schema structure and the string ("DELALL" or "MODIFY") gives the name of the SEED subroutine call to be used in the Fortran code to be created. Note, that in the two calls to the "idu" routine, fp points to different files for the output and the call contains different strings which will be written out later in the creation of the Fortran program.

The idu routine is the routine which actually produces the Fortran code (for a flow chart refer to figure 4.3). The idu routine takes in three parameters matching the three passed by indelup. They are called here "fp" (as in indelup), "start" (as in indelup), and "srutine" which holds the value of the string passed in.

The start of the code creates the header and overhead information. Note that the subroutine name of the routine being created comes from the string passed in from indelup concatenated with "ITE". In this way, unique subroutine names are created for the Fortran routines being written. At the same time, these routines have names which relate to their purpose.

This code is followed by a call to the commons routine. After the common statement has been created, the loop initialization and control statements are created followed by creation of the computed go to statement. The loop statements are as in GRASP. The computed go to statement is created by traversing the schema structure. As each table structure is visited, a number is written in the computed go to statement corresponding to the line number to go to in that case. This is done by calling the gorite routine. The statement is completed by the code:

```
    "fprintf(fp,"),\n      *        NRS(N)");"
```

The continuation line is created to ensure that the 72 column restriction of Fortran is met.

The remainder of the routine traverses the schema structure and creates the necessary code. As each table structure is visited, statements are generated to create a call to the proper SEED subroutine, check for errors and continue in the loop if no errors are found The variable "numb" holds the appropriate line number. This is incremented by one after each table structure is visited. "srutine" has the value of the string which is the name of the SEED subroutine to be called. This was passed in from the idu routine.

Once this has been completed for each table structure in the schema structure, the final statements are written and the control returns to the indelup routine. The final statements for this routine are identical to those of GRASP and are produced in the same manner. When control returns to the indelup routine, either the second call to idu is made or control reverts to the main program. After each call to idu is made, the appropriate file is closed. Upon return to the calling program, the MODITE and DELITE subroutines have been completed and are available in their respective files.

The "inserter" routine produces the Fortran code for the "INSRTN" routine. Figure 4.4 gives the flow chart for the inserter routine. Inserter takes two arguments as input; "fp" and "start". "fp" is a pointer to the file where the resulting Fortran routine should be output. "start" is the pointer to the schema structure. The routine begins by producing the subroutine heading, the common statement, and the control structure. The control structure for the Fortran program is created by creating a while loop (created with a goto statement) and by creating a computed go to statement. Next, the schema structure is traversed. As each table structure of the schema structure is visited, gorite is called, and the correct line number is written into the go to statement. If the rhook field of the table structure being visited =/= 0, then the list of hook structures pointed to by the rhook field, is traversed and a number is written in the computed goto statement for each hook structure in the list. This will allow inserts into recursive records in the Fortran program.

Now, the schema structure is again traversed to create the actual code. For each tables structure in the schema structure the following code is generated:

```
"fprintf(fp,"\n %d        CALL STORE(%s)",numb,(*begin).aname);
 fprintf(fp,"\n        IF (ERRSTA .NE. 0) GO TO 9999");"
```

The "numb" variable contains the appropriate line number to ensure that it will agree with the line number in the computed go to statement.

As each table structure is visited there are three possibilities to consider. The first case is where the code being created is to insert a record into a set whose only owner is system. This is the simplest case. Here the code to create the proper Fortran is simply:

```
"fprintf(fp,"\n          CALL INSERT(SYSTEM_%s)",(*begin).aname);
 fprintf(fp,"\n          IF (ERRSTA .NE. 0) GO TO 9999");
 fprintf(fp,"\n          GO TO 9990\n");"
```

The next case is more complicated. Here code is created where the record is part of at least one non–system owned set. In this case, a call to insert is generated for every set having this record as a member excluding those sets where it is a recursive occurrence. This is done by traversing the list of hook structures pointed to by the "for_sets" field of the table structure being visited. As each hook structure is visited, if the rhook field of the table pointed at by the hook structure is 0, then the record is inserted into that set. If the rhook field =/= 0, then the table in question owns the recursive occurrence of some record as a member. So the list of hook structures pointed at by the rhook field is traversed. If the table pointed at by the begin structure is pointed at by one of these hooks, no insert is created. This is because the table has a recursive relationship with this other table. If no such relationship exists, then an insert is created.

The last case is where a record is to be inserted into a set where it has a recursive occurrence. In this case, the rhook field of the table structure pointed to by begin is examined. If it is =/= 0, then the list of hook structures pointed at by the rhook field is traversed. For each table structure pointed at by a hook structure in this list, a "STORE" command and an "INSERT" are created. The recursive record here is inserted into the set owned by the record whose name is stored in the table structure pointed by the begin variable.

This completes the third possibility for the type of inserts to be performed. There are of course other options which are available for a normal insert into a SEED database. These options have not been made available here. This is in keeping with the vanilla nature of the schema and database.

The above three cases are considered for each of the table structures in the schema structure. Once the entire structure has been traversed, the final information ending the Fortran control structure (including the "RETURN" and "END" statements) are output and control return to the main program. In the main program, the file which now contains the "INSRTN" routine is closed.

This completes all technical description of C code. The last code discussed depth is the Fortran code created by these C routines. Sample flow charts for the three routines can be found in figure 4.5 (MODITE), figure 4.6 (DELITE), and figure 4.7 (INSRTN). The technical description of these routines follows.

The "DELITE" routine is used to delete records from the database. The main program which calls this routine will have to have certain restrictions. Before a record is deleted, the user must be sure that the correct record to be deleted is the current record. This could be done by calling FIRSTW and supplying the correct FILTER routine. Also in the main program, the NRS array must be filled. This requires, the following numbering scheme. For both DELITE and MODITE, the first record in the DBL schema corresponds to the number 1, the second record to the number 2, etc with the nth record corresponding to the number n. If there is a recursive occurrence of a record type, only the original or the recursive occurrence can be deleted in one call to DELITE. So to delete both a record and the recursive occurrence of the same record type would take two calls to the DELITE routine with the same number used in the NRS array to delete the appropriate records. Before the first call to this routine, either the record or its recursive occurrence must be the current record of the type. Then before the second call to the routine, the currency of this type of record must be changed to reflect the other record's currency. These restrictions are also true of the MODITE routine.

One final note about using the DELITE routine. Whenever this routine is called, assuming everything is done correctly, the appropriate records will be deleted. In addition, all records owned by the deleted record will be deleted if they have no other owner. To ensure these records are not lost unless the user want them lost, the user should be sure that any such records are given another owner.

The DELITE routine takes in two arguments. The first is the "NRS" array containing the numbers corresponding to the records. The last entry in this array must be 0. This will be used to terminate the looping in the routine and cause control to pass back to the calling routine. The second argument "N" is the size of the "NRS" array.

The program begins execution by initializing the variable "I" to 1. This will be used to index through the "NRS" array. Then the test is made to see if NRS(I) = 0. If so, control passes to the return statement. If not, control passes to the computed go to statement. Control then goes to the line number for the appropriate record. The appropriate record is guaranteed by the correspondences between numbers and records as described above.

At each of the different line numbers, a call is made to the subroutine "DELALL" with an argument of the appropriate record name. After each call, the "ERRSTA" variable is checked to ensure there is no error. If there is an error control passes to line numbered 9999 and control returns to the main program. If not, control passes to line 9990. Here the variable I is incremented and control returns to line 1. At line 1, the process repeats until some termination condition is reached. Upon termination, control returns to the calling program and the records whose numbers appeared in the NRS array have been deleted.

The differences in the DELITE routine and MODITE routine are small. The restrictions mentioned above for DELITE hold for MODITE. The control structure down to the number correspondences for records are the same. The obvious difference is that something different is being accomplished. In terms of the routines, this amounts to "DELALL" being replaced by "MODIFY". The other real difference is an added restriction on MODITE. Here, the data to be updated, must be input into the correct variables before the routine is called. This would normally be done with a read into these variables. Upon return to the calling program, the records would be updated instead of deleted. Other than this, these two routines are identical. Of course, in this routine, there is no worry about losing any records by accident as in the DELITE routine, however, the user should be aware that once information has been changed it can no longer be recovered by the system.

The INSRTN routine is very similar to the other two routines. It takes in the same two variables with the same meanings. It contains the same declaration and common statement. It has the same control and looping structure. The first restriction on using this routine is that the user must be sure to load the information of a record into the appropriate variables before calling the routine. The routine will not check this but will assume it has been done. The second restriction is that any record which will be an owner of a set into which an insertion is made must have the correct currency.

Since records and recursive occurrences of records will have different owners, the number correspondences must have facility to determine which is being inserted. Therefore, in this case the number correspondence is different. The first record in the DBL schema is assigned the number one. If it is not the owner of any recursive occurrences of any other records, then 2 corresponds to the second record in the DBL schema. If it is the owner of a recursive occurrence of another record, the first such record listed corresponds to the number 2, the second to the number 3 and so on up to n. Then n+1 would correspond to the second record and the entire process would repeat itself.

A difference between this and the other two programs, is that it takes calls to two different SEED routines to accomplish one insert. For each record to be inserted, there will always be one call to the "STORE" routine to make this information the currency of the record to be inserted. There will then be one call to "INSERT" for each set this record is to be a member of.

This brings up the problem of which sets to make a record a member of. The convention is, if a particular record is not a recursive occurrence of a record type, it is inserted into all sets with which it does not have a recursive relationship. This is used to preserve the vanilla nature of the database. This method requires only one decision to be made, which records to insert. Another possibility here would be to allow selection as to which sets to insert a record into. This could be done by simply changing the numbering correspondence for the computed goto statement and adding a

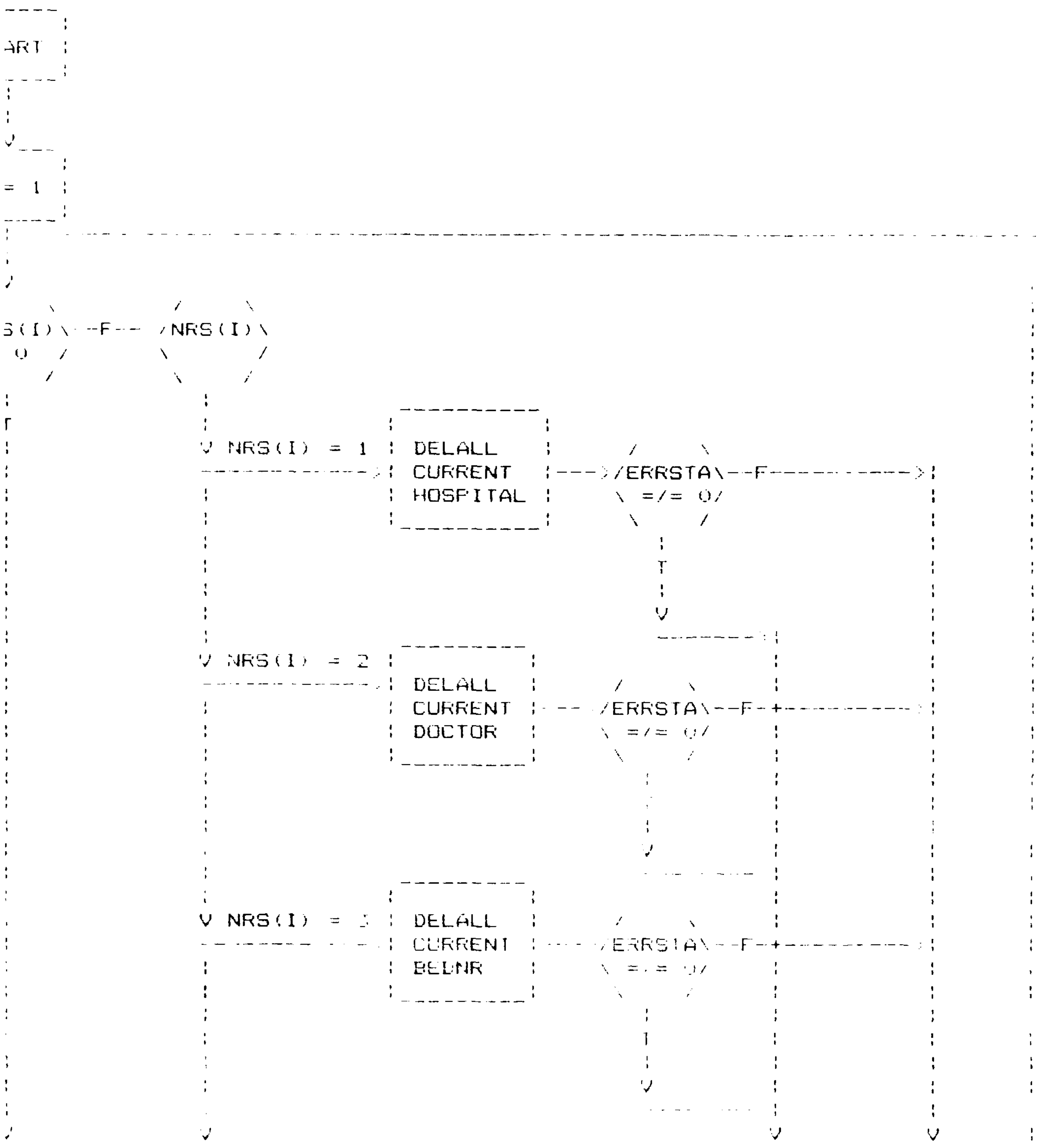


Figure 4.6 Page 1 of 2.

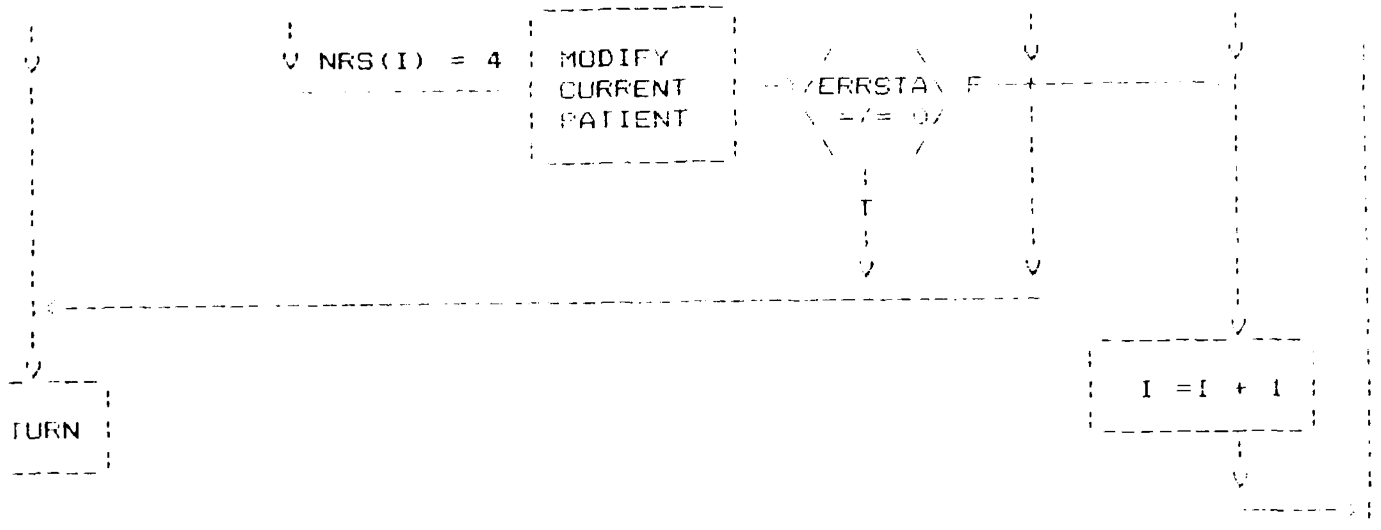


Figure 4.5 Page 2 of 2.

Figure 4.5 A Flow Chart for an Example MODITE Routine.

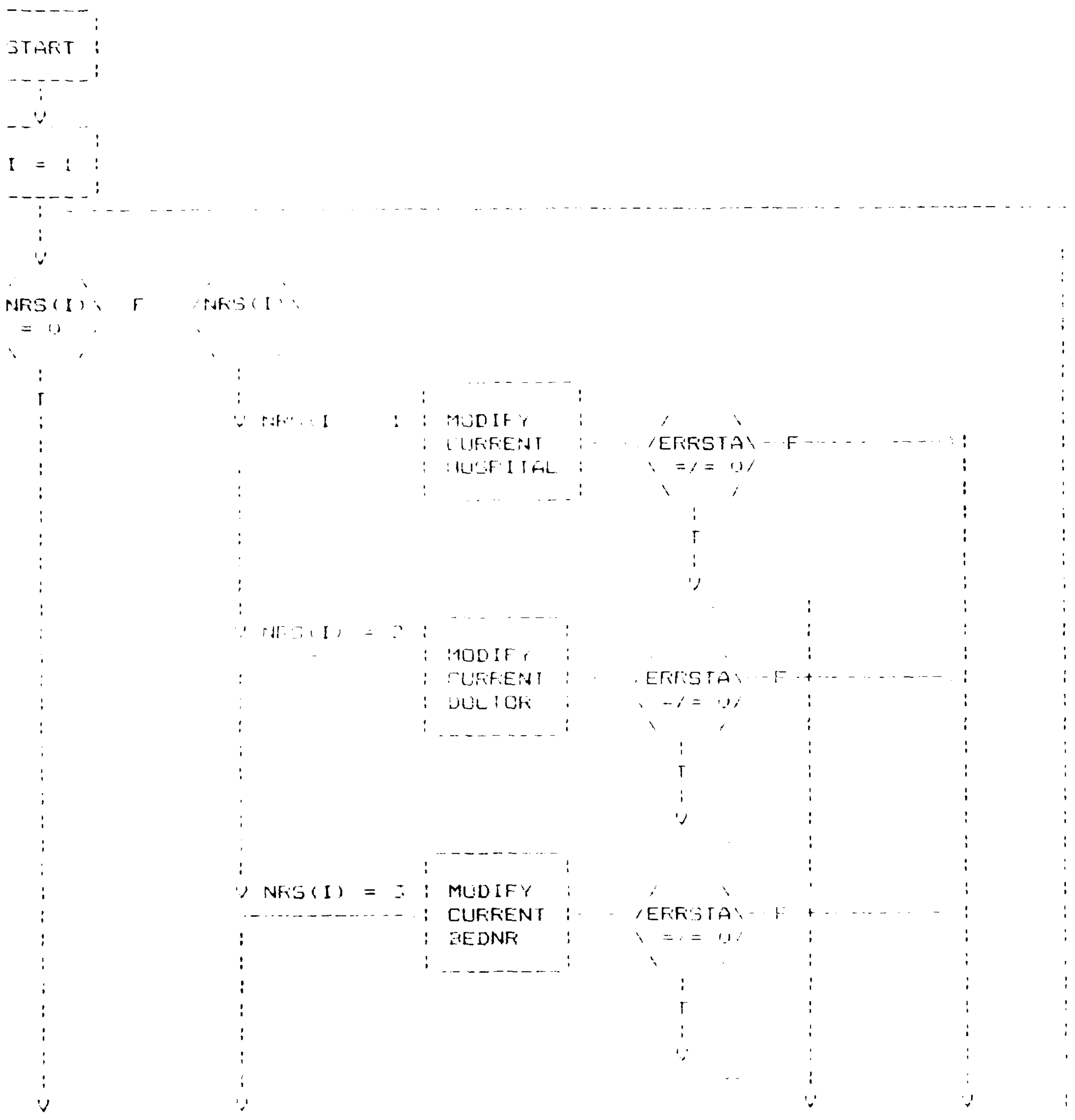


Figure 4.5 Page 1 of 2.

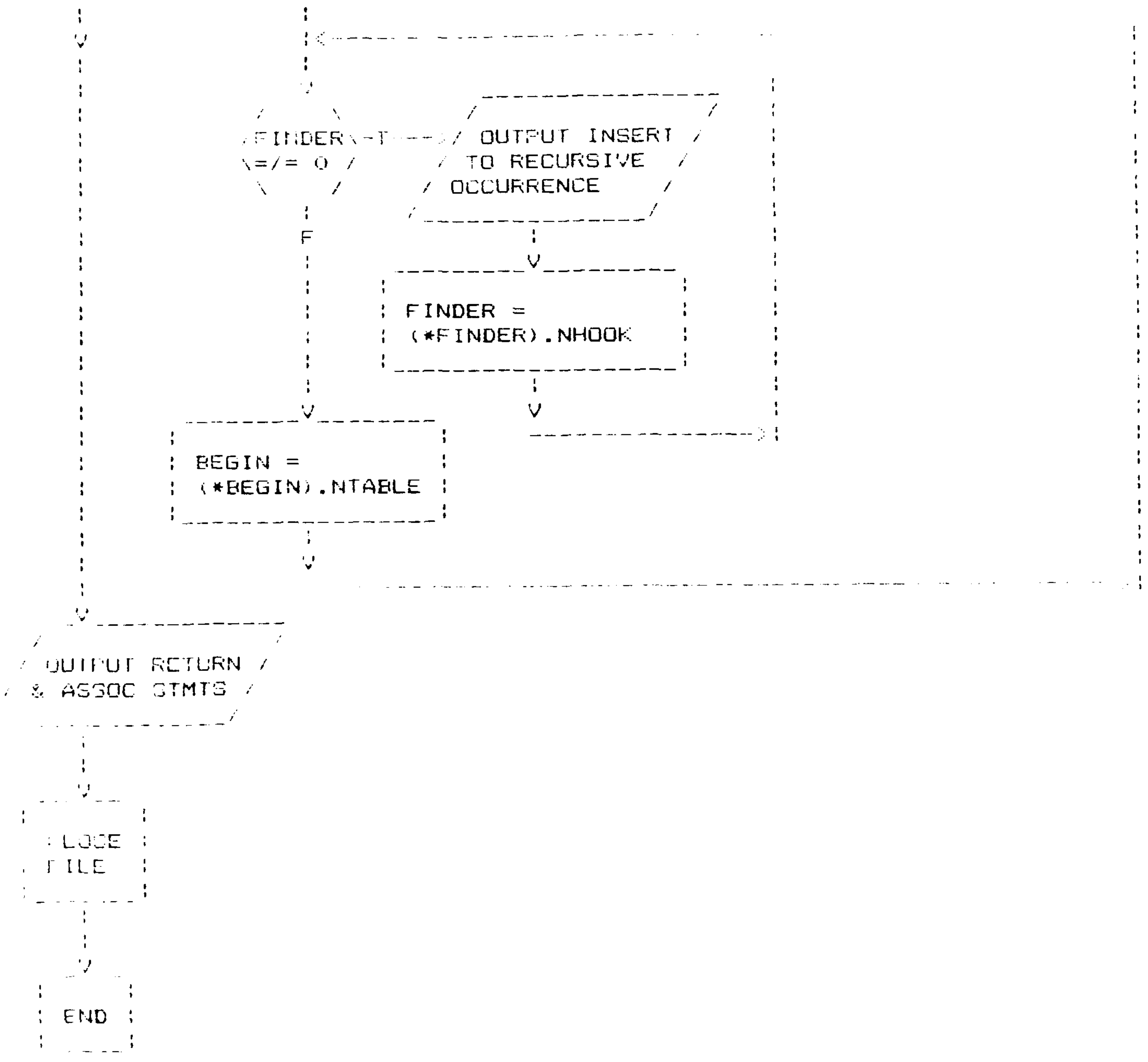


Figure 4.4 Page 4 of 4.

Figure 4.4 The Flow Chart for the inserter Routine.

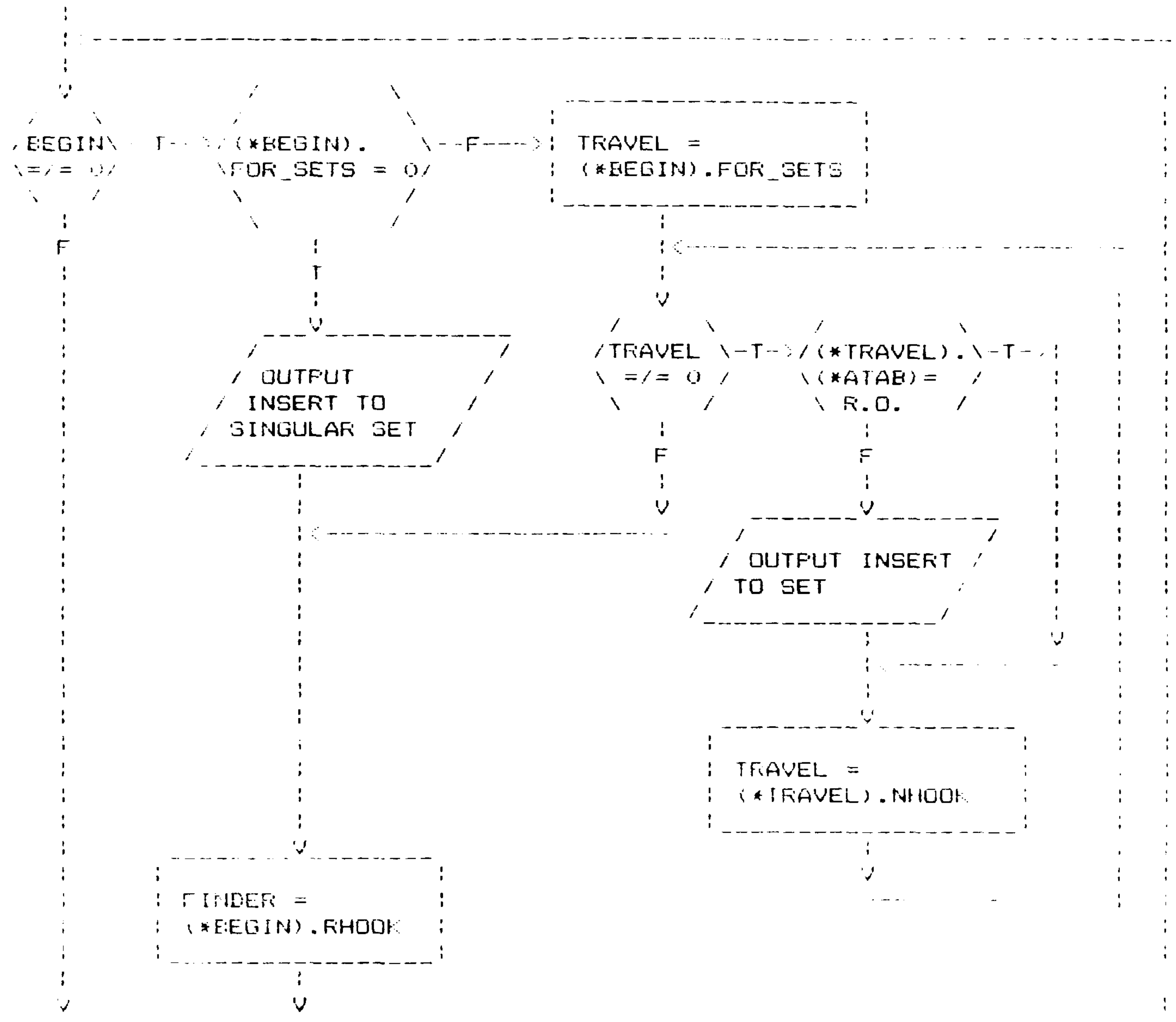


Figure 4.4 Page 3 of 4.

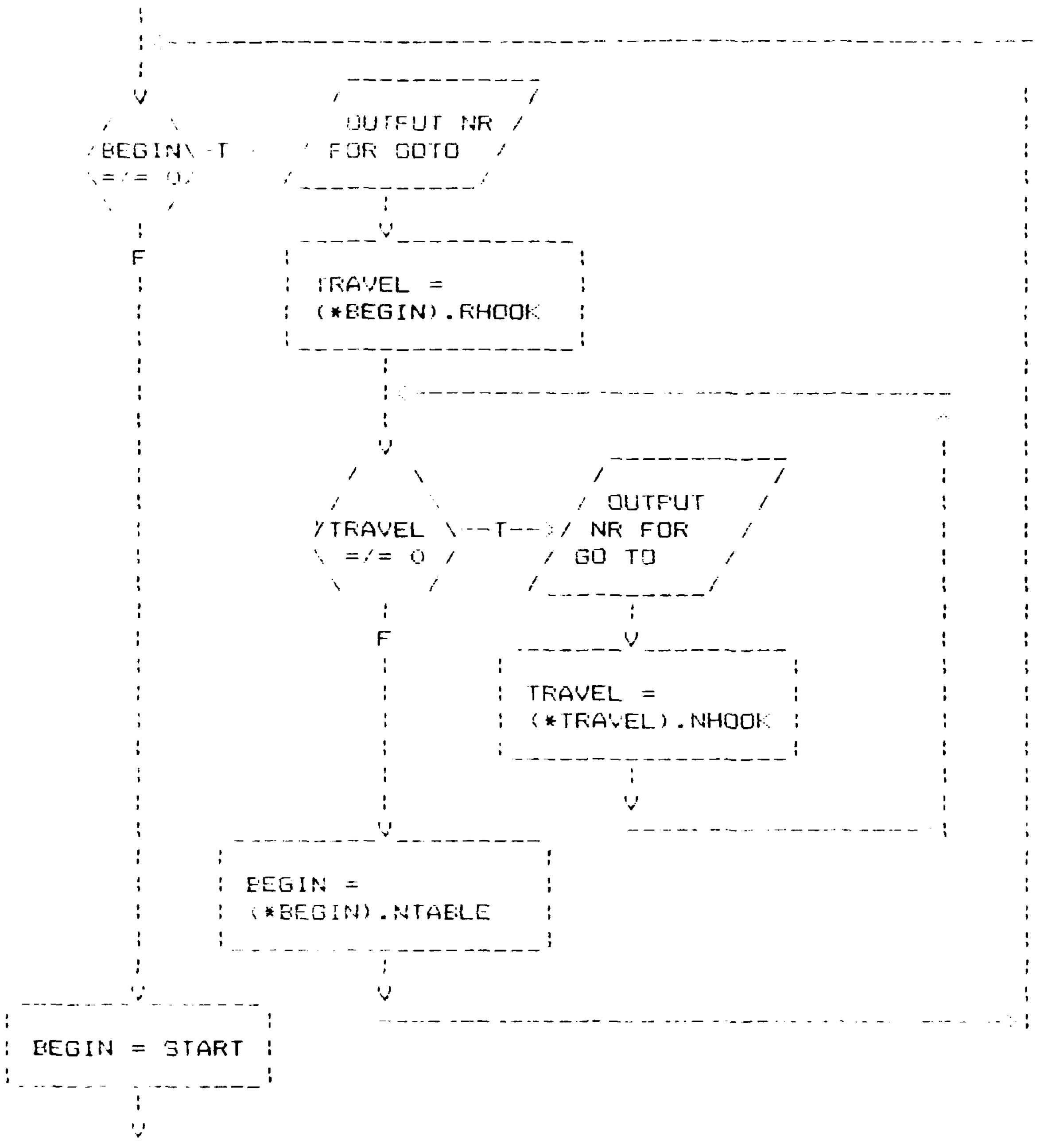


Figure 4.4 Page 2 of 4.

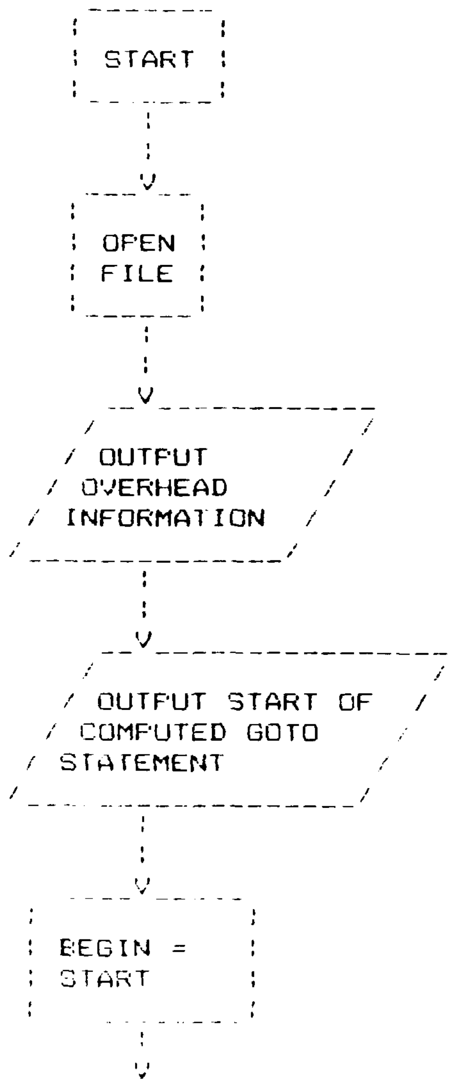

START
OPEN FILE
OUTPUT OVERHEAD INFORMATION
OUTPUT START OF COMPUTED GOTO STATEMENT
BEGIN = START

Figure 4.4 Page 1 of 4.

```
            :
      ______V________
     :               :
     : OUTPUT RETURN :
     : & ASSOC STMTS :
     :_______________:
             :
             :
          ___V___
         :       :
         : CLOSE :
         : FILE  :
         :_______:
             :
             :
           __V__
          :     :
          : END :
          :_____:
```

<u>Figure 4.3 Page 3 of 3.</u>

Figure 4.3b The Flow Chart for the idu Routine When It Produces the DELITE Routine.

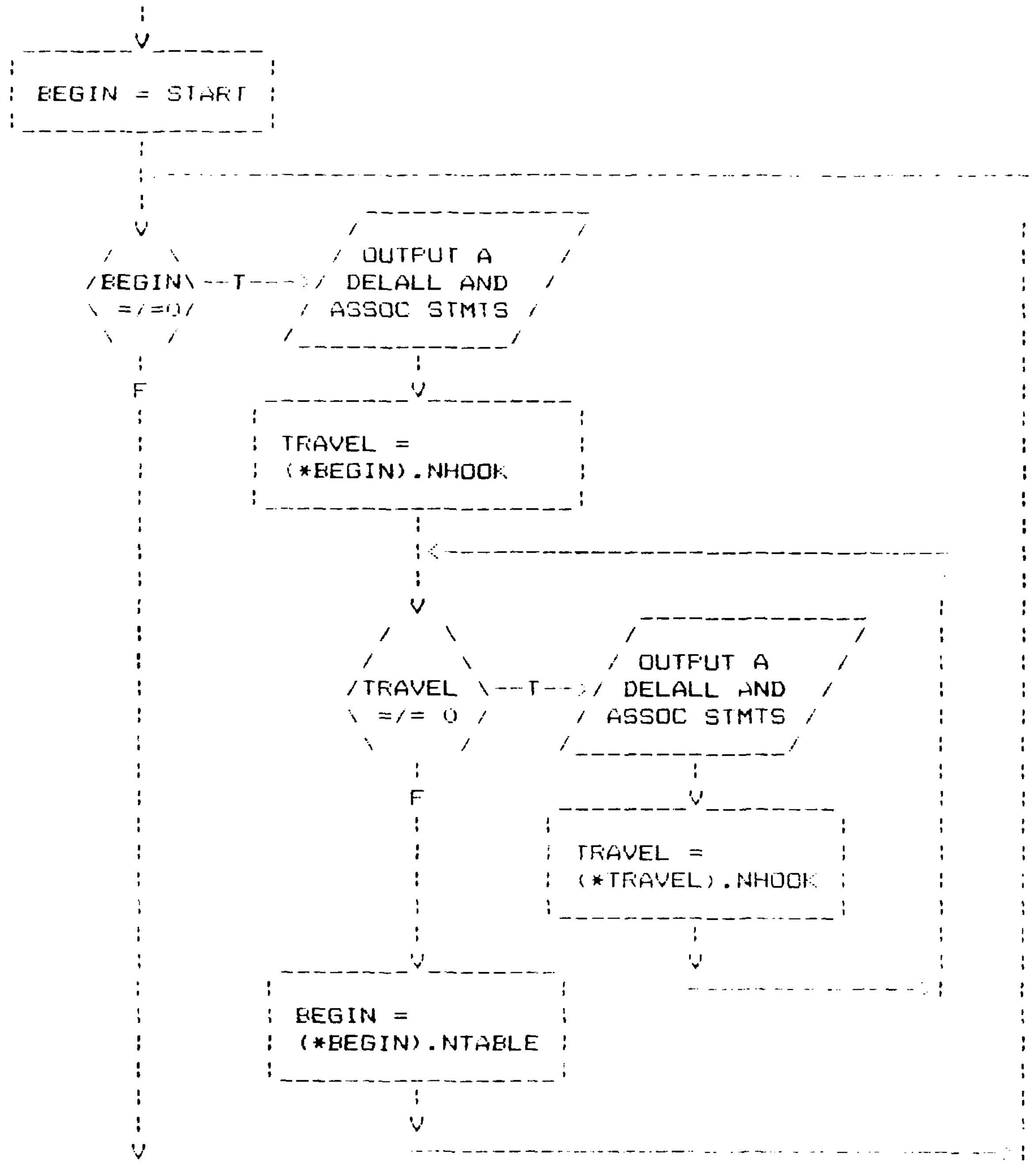


Figure 4.3 Page 2 of 3.

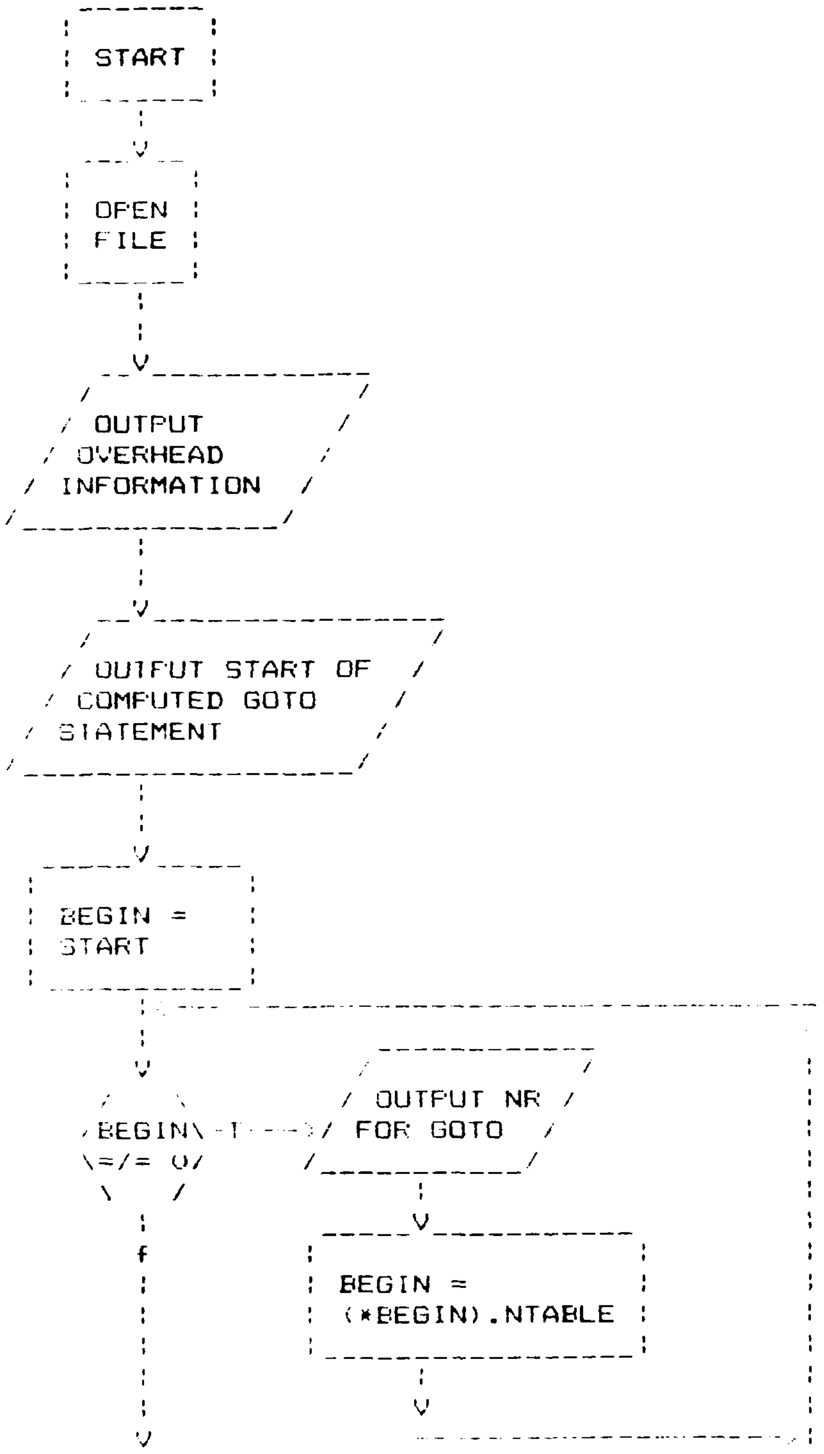


Figure 4.3 Page 1 of 3.

```
      SUBROUTINE  INSRTN(NRS,N)

      INTEGER N,  NRS,  I

      COMMON DANL/NAME,CITY,ZIP,NHOSPITAL,AHOSPITAL,HOSPITAL,
     *             SYSTEM_HOSPITAL,DRNAME,DRNR,NDOCTOR,
     *             ADOCTOR,DOCTOR,HOSPITAL_DOCTOR,BEDNUM,WARD,
     *             NBEDNR,ABEDNR,BEDNR,HOSPITAL_BEDNR,PNAME,
     *             DISEASE,AGE,SEX,VITS,NPATIENT,APATIENT,PATIENT,
     *             DOCTOR_PATIENT,
     *             ERRSTA,CRRECT,CRRUNU,CRCALC,DB

      I = 1

1     IF (NRS(I) .EQ. 0) GO TO 9999

      GO TO (2,3,4,5),
     *    NRS(I)

2     CALL STORE(HOSPITAL)
      IF (ERRSTA .NE. 0) GO TO 9999
      CALL INSERT(SYSTEM_HOSPITAL)
      IF (ERRSTA .NE. 0) GO TO 9999
      GO TO 9990

3     CALL STORE(DOCTOR)
      IF (ERRSTA .NE. 0) GO TO 9999
      CALL INSERT(HOSPITAL_DOCTOR)
      IF (ERRSTA .NE. 0) GO TO 9999
      GO TO 9990

4     CALL STORE(BEDNR)
      IF (ERRSTA .NE. 0) GO TO 9999
      CALL INSERT(HOSPITAL_BEDNR)
      IF (ERRSTA .NE. 0) GO TO 9999
      GO TO 9990

5     CALL STORE(PATIENT)
      IF (ERRSTA .NE. 0) GO TO 9999
      CALL INSERT(DOCTOR_PATIENT)
      IF (ERRSTA .NE. 0) GO TO 9999
      GO TO 9990

9990  I = I + 1
      GO TO 1

9999  RETURN

      END
```

Figure 4.2 An Example INSRTN Routine.

```
      SUBROUTINE DELITE(NRS,N)

      INTEGER N, NRS(N), I

      COMMON/BANK/NAME,CITY,ZIP,NHOSPITAL,AHOSPITAL,HOSPITAL,
     *            SYSTEM_HOSPITAL,DRNAME,DRNR,NDOCTOR,
     *            ADOCTOR,DOCTOR,HOSPITAL_DOCTOR,BEDNUM,WARD,
     *            NBEDNR,ABEDNR,BEDNR,HOSPITAL_BEDNR,PNAME,
     *            DISEASE,AGE,SEX,VITS,NPATIENT,APATIENT,PATIENT,
     *            DOCTOR_PATIENT,
     *            ERRSTA,CRRECT,CRRUNU,CRCALC,DB
      I = 1

1     IF (NRS(I) .EQ. 0) GO TO 9999

      GO TO (2,3,4,5),
     *   NRS(N)
2     CALL DELALL(HOSPITAL)
      IF (ERRSTA .NE. 0) GO TO 9999
      GO TO 9990

3     CALL DELALL(DOCTOR)
      IF (ERRSTA .NE. 0) GO TO 9999
      GO TO 9990

4     CALL DELALL(BEDNR)
      IF (ERRSTA .NE. 0) GO TO 9999
      GO TO 9990

5     CALL DELALL(PATIENT)
      IF (ERRSTA .NE. 0) GO TO 9999
      GO TO 9990

9990  I = I + 1
      GO TO 1

9999  RETURN
      END
```

Figure 4.1 An example DE[illegible] Routine.

```
       SUBROUTINE MODITE(NRS,N)

       INTEGER N, NRS(N), I

       COMMON/BANK/NAME,CITY,ZIP,NHOSPITAL,AHOSPITAL,HOSPITAL,
     *             SYSTEM_HOSPITAL,DRNAME,DRNR,NDOCTOR,
     *             ADOCTOR,DOCTOR,HOSPITAL_DOCTOR,BEDNUM,WARD,
     *             NBEDNR,ABEDNR,BEDNR,HOSPITAL_BEDNR,PNAME,
     *             DISEASE,AGE,SEX,VITS,NPATIENT,APATIENT,PATIENT,
     *             DOCTOR_PATIENT,
     *             ERRSTA,CRRECT,CRRUNU,CRCALC,DB
       I = 1

1      IF (NRS(I) .EQ. 0) GO TO 9999

       GO TO (2,3,4,5),
     *       NRS(N)
2      CALL MODIFY(HOSPITAL)
       IF (ERRSTA .NE. 0) GO TO 9999
       GO TO 9990

3      CALL MODIFY(DOCTOR)
       IF (ERRSTA .NE. 0) GO TO 9999
       GO TO 9990

4      CALL MODIFY(BEDNR)
       IF (ERRSTA .NE. 0) GO TO 9999
       GO TO 9990

5      CALL MODIFY(PATIENT)
       IF (ERRSTA .NE. 0) GO TO 9999
       GO TO 9990

9990   I = I + 1
       GO TO 1

9999   RETURN
       END
```

Figure 4.0 An Example MODITE Routine

few additional lines of code. It may be added in future implementations.

Other than these differences, the routine is very similar to the other two. Should an insert fail, by checking the value of the ERRSTA variable in the calling program, the exact cause of failure can be determined. Again on termination, control returns to the main program. At that time, either ERRSTA =/= 0 or an error has occurred. Again, the exact cause for the error can be discovered by checking the value of the ERRSTA variable.

4.4 THE CODE

This section gives the main "C" routines described in this chapter. These are the "indelup", "idu", and "insrt" routines. Figure 4.3a gives the code for the indelup routine, figure 4.3b gives the code for the idu routine, and figure 4.4 contains the code for the insrt routine.

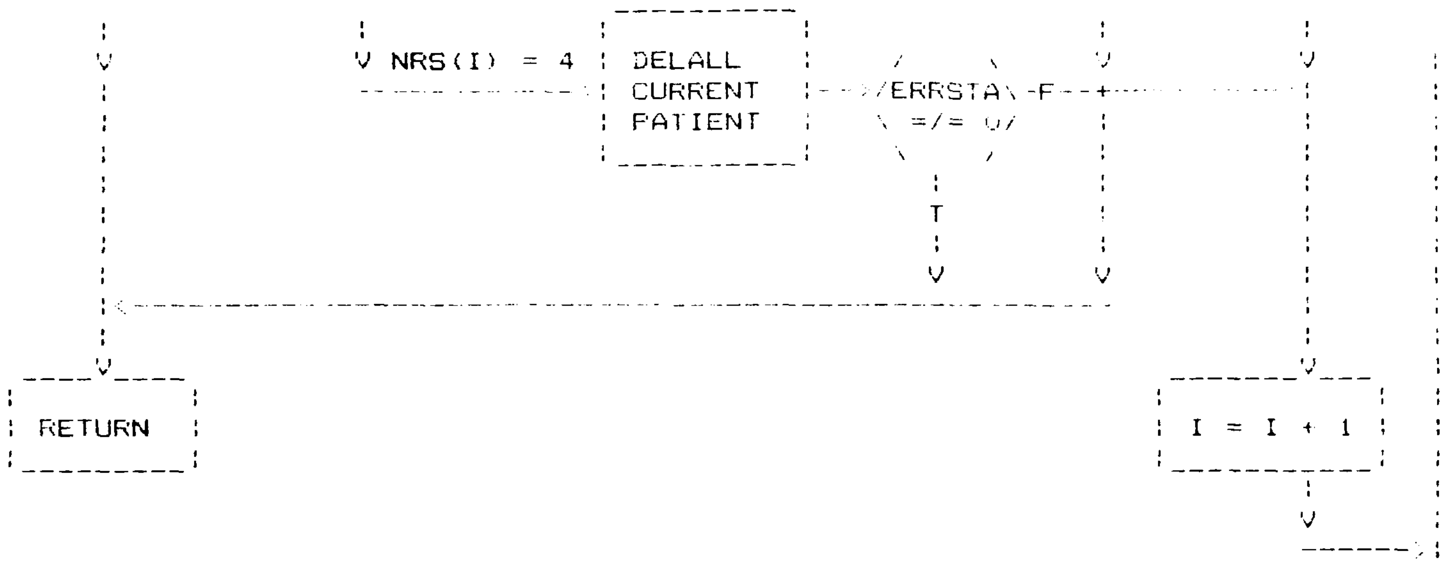


Figure 4.6 Page 2 of 2.

Figure 4.6 A Flow Chart for an Example DELITE Routine.

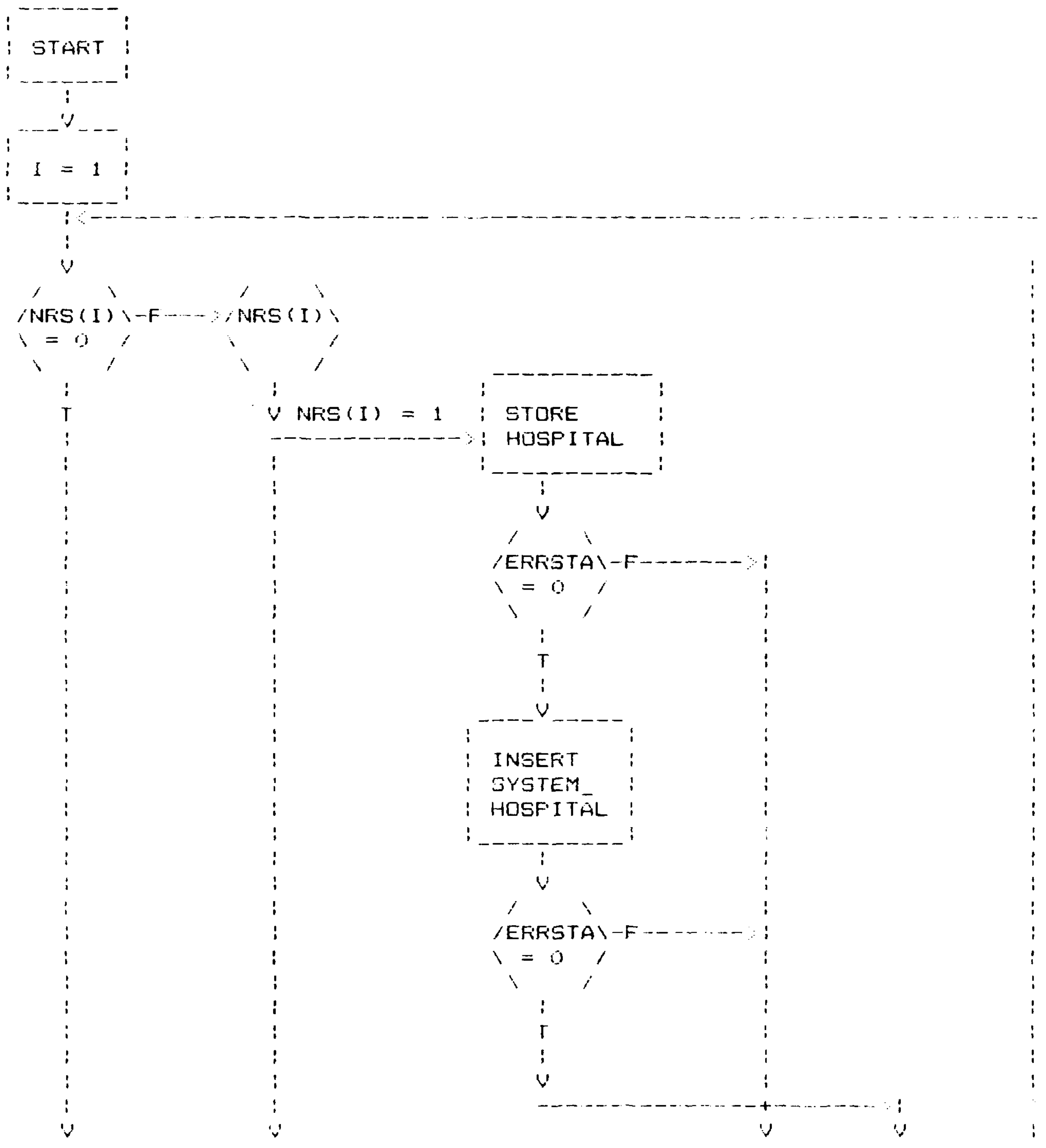


Figure 4.7 Page 1 of 4.

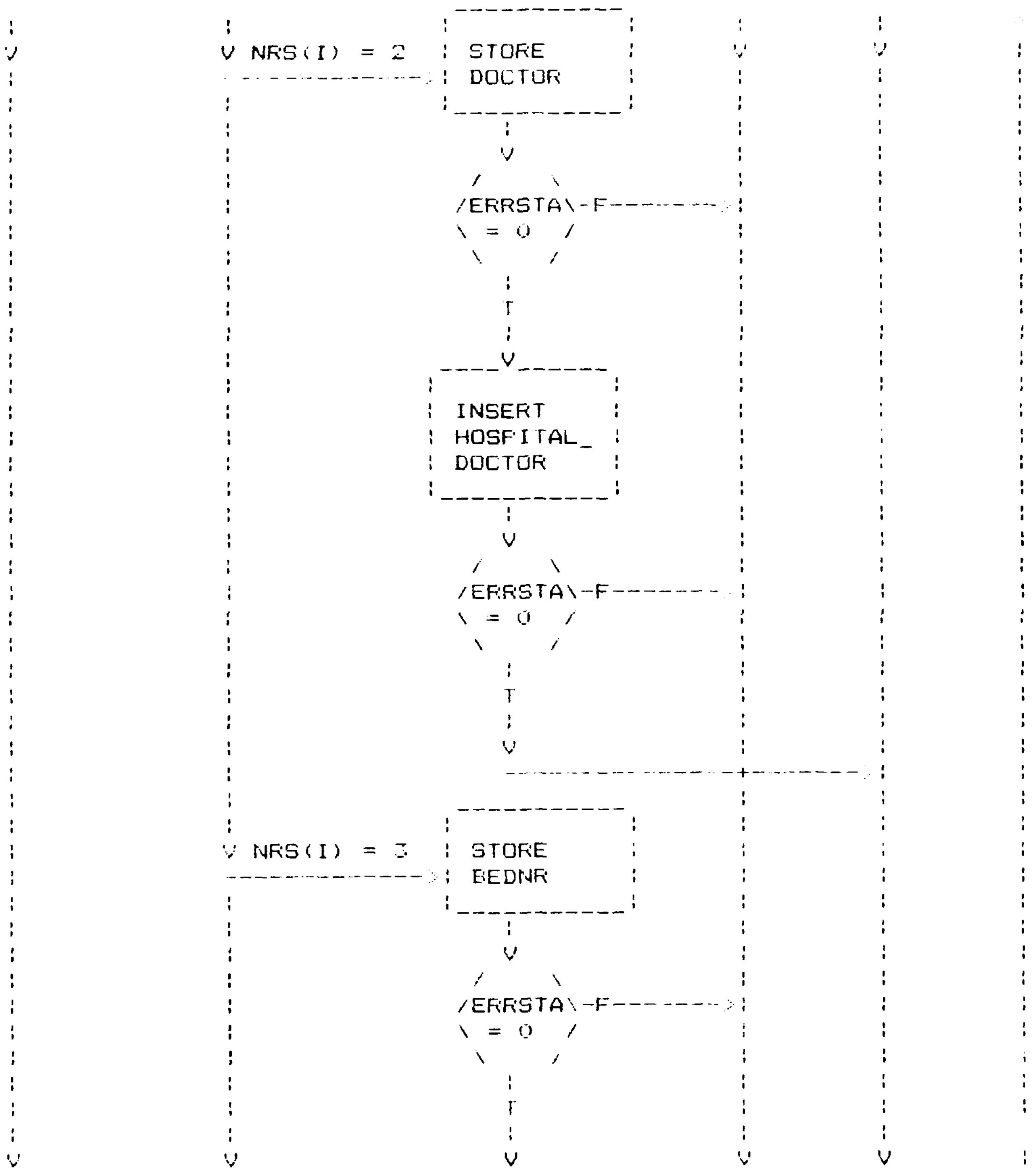


Figure 4.7 Page 2 of 4.

```
:              :                  :              :         :
V              V            ____V______          V         V         :
:              :           :              :      :         :         :
:              :           :  INSERT      :      :         :         :
:              :           :  HOSPITAL_   :      :         :         :
:              :           :  BEDNR       :      :         :         :
:              :           :______________:      :         :         :
:              :                  :              :         :         :
:              :                  V              :         :         :
:              :                /   \            :         :         :
:              :              /ERRSTA\-F-------->:         :         :
:              :              \ = 0  /           :         :         :
:              :                \   /            :         :         :
:              :                  :              :         :         :
:              :                  T              :         :         :
:              :                  :              :         :         :
:              :                  V              :         :         :
:              :                  ---------------+-------->:         :
:              :             ___________         :         :         :
:              :            :           :        :         :         :
:              V NRS(I) = 4 : STORE     :        :         :         :
:              ------------>: PATIENT   :        :         :         :
:                           :___________:        :         :         :
:                                 :              :         :         :
:                                 V              :         :         :
:                               /   \            :         :         :
:                             /ERRSTA\-F-------->:         :         :
:                             \ = 0  /           :         :         :
:                               \   /            :         :         :
:                                 :              :         :         :
:                                 T              :         :         :
:                                 :              :         :         :
:                            _____V____          :         :         :
:                           :          :         :         :         :
:                           : INSERT   :         :         :         :
:                           : DOCTOR_  :         :         :         :
:                           : PATIENT  :         :         :         :
:                           :__________:         :         :         :
:                                 :              :         :         :
:                                 V              :         :         :
:                               /   \            :         :         :
:                             /ERRSTA\-F-------  :         :         :
:                             \ = 0  /           :         :         :
:                               \   /            :         :         :
:                                 :              :         :         :
:                                 T              :         :         :
V                                 V              V         V         :
```

Figure 4.7 Page 3 of 4.

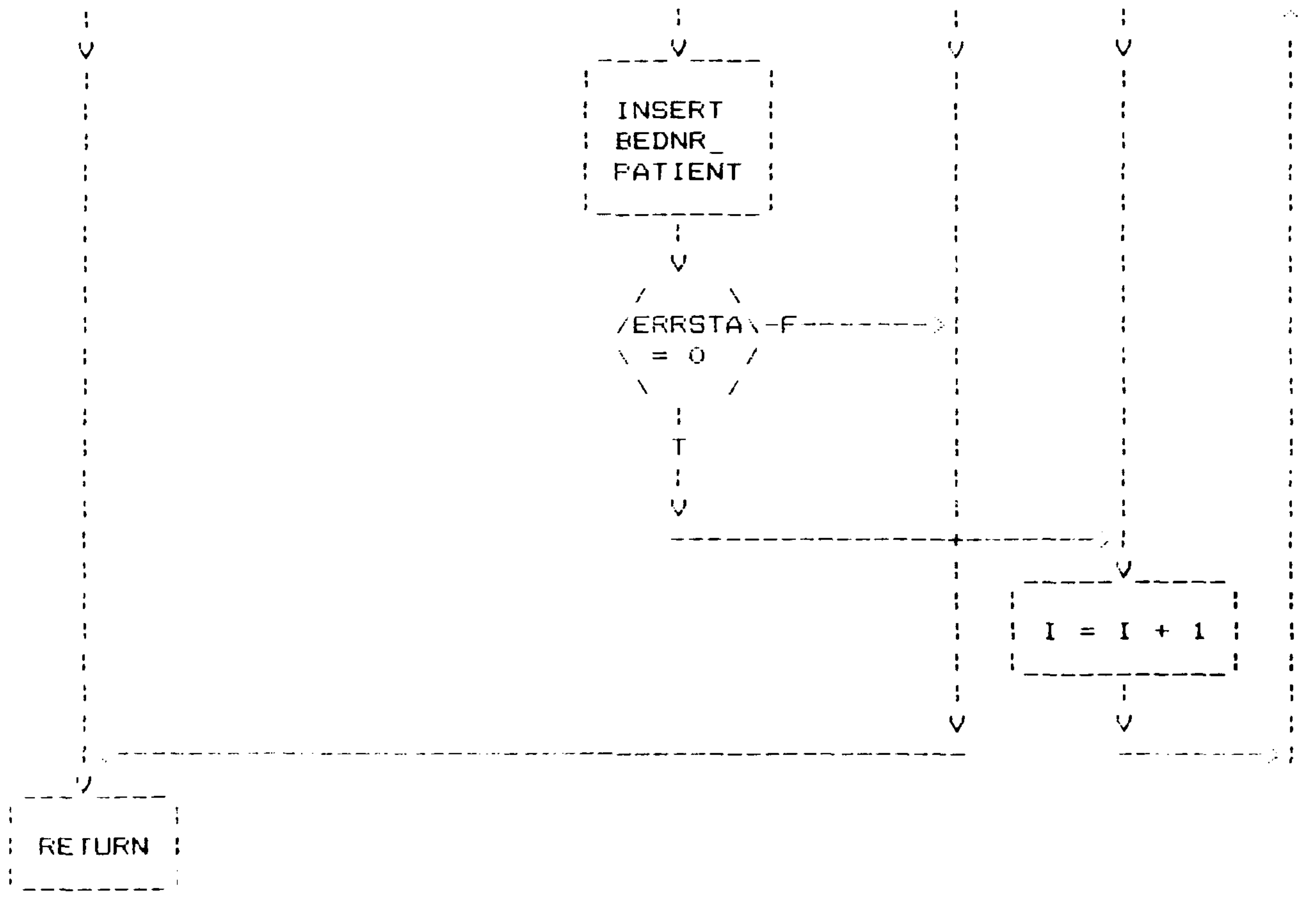


Figure 4.7 Page 4 of 4.

Figure 4.7 A Flow Chart for an Example INSERTN Routine.

```
indelup(start)

/* JOSEPH AULINO */
/* DAVID PROJECT  */
/* JUNE 1984      */
/*                */
/* THIS PROCEDURE CONTROLS THE PRODUCTION OF TWO FORTRAN       */
/* SUBROUTINES USED BY EVERY DATABASE. THE TWO                 */
/* SUBROUTINES ARE SIMILAR. THE DIFFERENCE BETWEEN THEM IS     */
/* WHICH "SEED" SUBROUTINE THE FORTRAN ROUTINE IS TO CALL.     */
/* CALLING A DIFFERENT SUBROUTINE IS ACHIEVED BY SENDING       */
/* THE PROGRAM WHICH ACTUALLY PRODUCES THE FORTRAN THE         */
/* NAME OF THE PROGRAM YOU WANT PRINTED.                       */
/* THE PROGRAM WHICH DOES THE ACTUAL PROGRAM CREATION IS       */
/* CALLED IDU.  IT PRODUCES A PROGRAM TO DELETE FROM           */
/* OR UPDATE A "SEED" DATABASE.                                */
/*                                                             */
/*                                                             */
/* LOCAL VARIABLES:                                             */
/*                                                              */
/*      1.  fopen - OPENS A NEW FILE FOR WRITING.               */
/*      2.  fp - POINTS TO THE ABOVE FILE.                      */
/*                                                              */
/*                                                              */
/* NON-STANDARD SUBROUTINE USED:                                */
/*                                                              */
/*      1.  idu - DOES THE FORTRAN GENERATION FOR THIS          */
/*                ROUTINE.                                      */

struct table *start;
{
FILE *fopen(),*fp;

fp = fopen("DELAL.F","w");
idu(fp,start, "DELALL");
fclose(fp);

fp = fopen("MODIF.F","w");
idu(fp,start, "MODIFY");
fclose(fp);

return;

}      /*****     END OF THE INDELUP ROUTINE      *****/
```

Figure 4.8a The indelup Routine C Code.

```
idu(fp,start,srutine)

/* JOSEPH AULINO */
/* DAVID PROJECT */
/* JUNE 1984     */
/*               */
/* THIS ROUTINE MAKES A SUBROUTINE WHICH CAN DELETE OR UPDATE     */
/* INFORMATION IN A DATABASE.  IT DETERMINES WHICH SUBROUTINE TO */
/* MAKE BY THE VALUE OF THE STRING IN SRUTINE.  THE FORTRAN CODE */
/* GENERATED DECIDES WHICH RECORD IS TO BE UPDATED OR DELETED BY */
/* VALUES PASSED IN THE VECTOR NRS.                              */
/*                                                               */
/*                                                               */
/* LOCAL VARIABLES:                                              */
/*                                                               */
/*      1.  begin - TRAVERSES A LIST OF TABLE STRUCTURES.        */
/*      2.  numb - USED BY GORITE TO HELP PRODUCE LINE NUMBERS.  */
/*      3.  adder  USED BY GORITE TO HELP PRODUCE LINE NUMBERS.  */
/*      4.  sum   USED BY GORITE TO HELP PRODUCE LINE NUMBERS.   */
/*      5.  incre - USED BY GORITE TO HELP PRODUCE LINE NUMBERS. */
/*                                                               */
/*                                                               */
/* NON STANDARD SUBROUTINES USED:                                */
/*                                                               */
/*      1.  commons - TO GENERATE THE NEEDED COMMON STATEMENT.   */
/*      2.  gorite - TO GENERATE THE NUMBERS NEEDED BY THE       */
/*                   COMPUTED GO TO STATEMENT.                   */

FILE *fp;
struct table *start;
char srutine[];
{
struct table *begin;
int numb, adder, sum, incre;

/*****************************************************/
/*                                                   */
/* FIRST, THE BASIC OVERHEAD INFORMATION IS WRITTEN. */
/*                                                   */
/*****************************************************/

fprintf(fp,"\n\n      SUBROUTINE %c%c%cITE",srutine[0],srutine[1],srutine[2]);
fprintf(fp,"(NRS,N)");

fprintf(fp,"\n\n      INTEGER N, NRS(N), I\n");

commons(start,fp);
```

Figure 4.8b Page 1 of 3.

```
fprintf(fp,"\n        I = 1");
fprintf(fp,"\n\n 1      IF (NRS(I) .EQ. 0) GO TO 9999");

numb = 2;
adder = 3;
sum = 12;
incre = 1;

/***********************************************/
/*                                             */
/* NEXT, THE COMPUTED GO TO STATEMENT IS CREATED. */
/*                                             */
/***********************************************/

fprintf(fp,"\n\n       GO TO (");

begin = start;

while (begin != 0)
{
   gorite(fp,&sum,&adder,&numb,incre);
   begin = (*begin).ntable;
}
fprintf(fp,"),\n     *      NRS(N)");


/*************************************************************/
/*                                                           */
/* THE NEXT SECTION OF CODE PRODUCES THE FORTRAN CALLS TO DO */
/* WHATEVER IS BEING REQUESTED WHETHER IT BE A DELETE,       */
/* OR UPDATE.                                                */
/*                                                           */
/*************************************************************/

numb = 2;
begin = start;

while (begin != 0)
{
   fprintf(fp,"\n %d     CALL %s(%s)",numb,srutine,(*begin).aname);
   fprintf(fp,"\n        IF (ERRSTA .NE. 0) GO TO 9999");
   fprintf(fp,"\n        GO TO 9990\n");
   numb++;
   begin = (*begin).ntable;
}
```

Figure 4.8b Page 2 of 3.

```
·*******************************************/
·                                          */
· THE LAST SECTION OF THIS CODE PRODUCES THE */
· ENDING OF THE FORTRAN CODE.               */
·                                          */
·*******************************************/

rintf(fp,"\n 9990   I = I + 1");
rintf(fp,"\n        GO TO 1");
rintf(fp,"\n\n 9999  RETURN");
rintf(fp,"\n        END\n\n");

eturn;

     /*****     END OF THE IDU ROUTINE     *****/
```

Figure 4.8b Page 3 of 3.

Figure 4.8b The idu Routine C Code.

```
erter(fp,start)

JOSEPH AULINO *∕
DAVID PROJECT *∕
JUNE 1984     *∕
              *∕
THIS PROCEDURE CREATES THE FORTRAN CODE NEEDED TO PROPERLY *∕
INSERT ANY NEW RECORD INTO ALL APPROPRIATE SETS IN A       *∕
"SEED" DATABASE.  THE FORTRAN PROGRAM TAKES IN A VECTOR OF *∕
OF NUMBERS WHICH CORRESPOND TO THE RECORD TO BE INSERTED.  *∕
THIS VECTOR IS USED BY A COMPUTED "GO TO" STATEMENT TO     *∕
ACHIEVE THE DESIRED RESULT.                                *∕
                                                           *∕
LOCAL VARIABLES:                                           *∕
                                                           *∕
    1.  sum - USED BY GORITE TO HELP CREATE LINE NUMBERS.  *∕
    2.  adder - USED BY GORITE TO HELP CREATE LINE         *∕
                NUMBERS.                                   *∕
    3.  numb - USED BY GORITE TO HELP CREATE LINE NUMBERS. *∕
    4.  incre - USED BY GORITE TO HELP CREATE LINE         *∕
                NUMBERS.                                   *∕
    5.  flag - DETERMINES WHAT RECORDS ARE INSERTED IN     *∕
               WHICH SETS.                                 *∕
    6.  begin - TRAVERSES A LIST OF TABLE STRUCTURES.      *∕
    7.  travel - TRAVERSES A LIST OF HOOK STRUCTURES.      *∕
    8.  finder   POINTS TO A HOOK STRUCTURE.               *∕
                                                           *∕
NON-STANDARD SUBROUTINES USED:                             *∕
                                                           *∕
    1.  commons - PRODUCES THE NECESSARY COMMON STATEMENT. *∕
    2.  gorite - PRODUCES THE NUMBERS USED IN THE COMPUTED *∕
                 GO TO STATEMENT.                          *∕

E *fp;
uct table *start;

 sum, adder, numb, incre, flag;
uct table *begin;
uct hook *travel, *finder;

FIRST THE STATIC OVERHEAD FORTRAN INFORMATION IS WRITTEN *∕
AND CERTAIN VARIABLES ARE INITIALIZED.                   *∕

intf(fp,"\n\n       SUBROUTINE INSRTN(NRS,N)\n");
intf(fp,"\n       INTEGER N, NRS, I\n");
```

Figure 4.9 Page 1 of 5.

```
= 12;
r = 3;
 = 2;
e = 1;

********************************************************/
                                                      */
EXT, THE COMMON STATEMENT IS WRITTEN BY THE SUBROUTINE */
COMMONS" AND THE SUBROUTINE "GOTORITE" IS USED TO      */
REATE THE COMPUTED GO TO STATEMENT.                    */
                                                      */
********************************************************/

ons(start,fp);

ntf(fp,"\n\n        I = 1\n\n");
ntf(fp," 1      IF (NRS(I) .EQ. 0) GO TO 9999\n");
ntf(fp,"\n        GO TO (");
n = start;
e (begin != 0)

orite(fp,&sum,&adder,&numb,incre);

f ((*begin).rhook != 0)

  travel = (*begin).rhook;
  while (travel != 0)
  {
     gorite(fp,&sum,&adder,&numb,incre);
     travel = (*travel).nhook;
  }

egin = (*begin).ntable;

ntf(fp,"),\n     *      NRS(I)\n");

*****************************************************/
                                                   */
EXT, THE ACTUAL CODE IS PRODUCED TO  STORE (CREATE) */
EW RECORDS AND INSERT THEM IN THE PROPER SETS       */
ASED ON THE INFORMATION GIVEN IN THE NRS VECTOR.    */
                                                   */
*****************************************************/
```

Figure 4.9 Page 2 of 5.

```
= 2;
 = start;
 (begin != 0)

rintf(fp,"\n %d       CALL STORE(%s)",numb,(*begin).aname);
rintf(fp,"\n         IF (ERRSTA .NE. 0) GO TO 9999");

****************************************/
                                        */
SE 1: THIS RECORD IS OWNED ONLY BY SYSTEM. */
                                        */
****************************************/

 ((*begin).for_sets == 0)

 fprintf(fp,"\n        CALL INSERT(SYSTEM_%s)",(*begin).aname);
 fprintf(fp,"\n        IF (ERRSTA .NE. 0) GO TO 9999");
 fprintf(fp,"\n        GO TO 9990\n");


*****************************************************/
                                                   */
SE 2: THE RECORD IS OWNED BY AT LEAST ONE OTHER RECORD. */
      HERE THE RECORD WILL BE INSERTED INTO ALL SETS    */
      IN WHICH IT IS NOT A RECURSIVE MEMBER.            */
                                                   */
*****************************************************/

se

 travel = (*begin).for_sets;
 while (travel != 0)
 {
     flag = 1;
     if ((*(*travel).atab).rhook != 0)
     {
        finder = (*(*travel).atab).rhook;
        flag = 0;
        while (finder != 0)
        {
            if ((*finder).atab == begin)
            {
               finder = 0;
               flag = 1;
            }
```

Figure 4.9 Page 3 of 5.

```
           else
              finder = (*finder).nhook;
        }
     }

     if ((flag == 0) || ((*(*travel).atab).rhook == 0))
     {
        fprintf(fp,"\n        CALL INSERT(");
        fprintf(fp,"%s_%s)",(*(*travel).atab).aname,(*begin).aname);
        fprintf(fp,"\n        IF (ERRSTA .NE. 0) GO TO 9999");
     }
     travel = (*travel).nhook;
  }
  fprintf(fp,"\n       GO TO 9990\n");

********************************************************/
                                                       */
E 3: THIS CASE TAKES CARE OF THE POSSIBILITY THAT THIS */
A RECURSIVE INSERT.                                    */
                                                       */
********************************************************/

f ((*begin).rhook != 0)

   numb++;
   finder = (*begin).rhook;
   while (finder != 0)
   {
    fprintf(fp,"\n %d     CALL STORE(%s)",numb,(*(*finder).atab).aname)
    fprintf(fp,"\n        IF (ERRSTA .NE. 0) GO TO 9999");
    fprintf(fp,"\n        CALL INSERT(");
    fprintf(fp,"%s_%s)",(*begin).aname,(*(*finder).atab).aname);
    fprintf(fp,"\n        IF (ERRSTA .NE. 0) GO TO 9999");
    fprintf(fp,"\n        GO TO 9990\n");
    finder = (*finder).nhook;
  }

mb++;
gin = (*begin).ntable;
```

Figure 4.9 Page 4 of 5.

```
*********************************************************/
                                                       */
Y, THE REMAINING STATIC INFORMATION IS WRITTEN */
HE FORTRAN PROGRAM IS COMPLETE.                       */
                                                      */
*********************************************************/

fp,"\n 9990   I = I + 1");
fp,"\n         GO TO 1");
fp,"\n\n 9999  RETURN");
fp,"\n\n        END\n\n");


****      END OF THE INSERTER ROUTINE      *****/
```

Figure 4.9 Page 5 of 5.

Figure 4.9 The inserter Routine C Code.

BIBLIOGRAPHY

[1] Adiba, M. and D. Portal, "A Cooperation System for Heterogeneous Database Management System", INFORMATION SYSTEMS, Vol. 3, 1978, pp. 209–215.

[2] Cardenas, A. and M. H. Perahesh, "Database Communication in a Heterogeneous Database Management System Network", INFORMATION SYSTEMS, Vol. 5, 1980, pp. 55–79.

[3] Chu, W. and V. T. To, "A Hierarchical Conceptual Data Model for Data Translation in a Heterogeneous Database System", in Chen P. P. (ed.), ENTITY RELATIONSHIP APPROACH TO SYSTEM ANALYSIS AND DESIGN, North–Holland, Amsterdam, 1980, pp. 598–615.

[4] Computer Corporation of America, "A Prototype Chemical Substances Information Network", Technical Report No. CCA–77–19, Contract No. EQ8AC028, August 21, 1979.

[5] Date, C.J., AN INTRODUCTION TO DATABASE SYSTEMS, 3rd. edition, Addison–Wesley Publishing Co., Reading, Mass. 1982, pp. 1–50.

[6] Esculier, C. and A. M. Glorieux, "The Sirius–Delta Distributed DBMS", in Chen P. P. (ed.), ENTITY RELATIONSHIP APPROACH TO SYSTEM ANALYSIS AND DESIGN, North–Holland, Amsterdam, 1980, pp. 616–624.

[7] Gligor, V., T. Jacobs, and G. Luckenbaugh, "The Interconnection of Remote Hetergeneous DBMS; Project Survey Recommended Approach, Selection Criteria", NBS Contract No. NB805BCA0367, April 6, 1981.

[8] International Databas Systems Inc., SEED USER MANUAL, Philadelphia, Pa. 19103, August 1980.

[9] Jacobs, Barry E., APPLIED DATABASE LOGIC I: FUNDAMENTAL DATABASE ISSUES, Prentice–Hall Inc., Englewood Cliffs, N.J. 07632 (Publication Forthcoming).

[10] ----------------, APPLIED DATABASE LOGIC II: HETEROGENEOUS DISTRIBUTED QUERY PROCESSING, Prentice–Hall Inc., Englewood Cliffs, N.J. 07632 (Publication Forthcoming).

[11] ------------------, APPLIED DATABASE LOGIC IV: THE DAVID SYSTEM, Prentice-Hall Inc., Englewood Cliffs, N.J. 07632 (Publication Forthcoming).

[12] Jacobs, Barry E. and Thomas E. Jacobs, "On the Functional Requirements of a Global Data Manager for a Set of Distributed Heterogeneous Databases I", Technical Report TR-1068, University of Maryland, College Park, Md. 20742.

[13] Sylto, Regina, "ERB-6 Data Inventory", National Aeronautics and Space Administration Technical Manual Nr. 82176, NASA, Goddard Space Flight Center, Greenbelt, Md. 20771.

[14] Tsubaki, M. and R. Hataka, "Distributed Multi-Database Environment with a Supervisory Data Dictionary Database", in Chen P. P. (ed.), ENTITY-RELATIONSHIP APPROACH TO SYSTEMS ANALYSIS AND DESIGN, North-Holland, Amsterdam, 1980, pp. 625-646.

[15] Ullman, Jeffrey D., PRINCIPLES OF DATABASE SYSTEMS, Computer Science Press, Rockville, Md. 20850, 1982.