

DATA INDEPENDENT REVISIONS TO
DATA BASE DEFINITIONS

A Thesis

Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Sue Simpson Schwartz

December 1978

ACKNOWLEDGMENTS

The author would like to express her appreciation to Dr. Gerald D. Everett who has been a source of information, assistance, and inspiration throughout this project. The finished form of this thesis is mostly due to his careful reading and red-pencilling of the many drafts.

The Panhandle Eastern Pipe Line Company generously furnished both the computer time and the funds necessary to support the work done on the UPDATE module of the Remote File Management System.

Finally I wish to thank my husband, Andy, for without his support, encouragement, and admonitions of "When are you going to finish your thesis?" I would never have persevered.

DATA INDEPENDENT REVISIONS TO
DATA BASE DEFINITIONS

An Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Sue Simpson Schwartz

December 1978

ABSTRACT

This thesis deals with the question of how to modify a data base definition independent of values which have been entered into the data base. Specifically an attempt was made to derive a scheme which would allow definition modifications of an existing data base to occur with a minimum of reorganization within the data base. The data base management system chosen for examination of definition modifications was the Remote File Management System.

Possible modifications to the definition of a data base consist of changes in data descriptions, the addition or deletion of data descriptions, and changes in the order in which data is related to other data in the data base. Each of these definition modifications is examined in depth and a scheme involving minimal reorganization within the data base is thoroughly detailed for each modification. Each scheme is then compared to the only other known method of handling definition modifications, completely unloading the data and reloading it using the modified definition. In every instance the new schemes for definition modification in the Remote File Management System were found to be superior to the unload-reload method.

TABLE OF CONTENTS

Chapter	Page
1. DATA BASE MANAGEMENT	1
2. THE REMOTE FILE MANAGEMENT SYSTEM	18
3. DEFINITION MODIFICATION IN THE REMOTE FILE MANAGEMENT SYSTEM	42
A. COMPONENT NAMES	43
B. COMPONENT NUMBERS	50
C. ELEMENT TYPE	56
D. LOGICAL ENTRY STRUCTURE	61
E. ADDITION OF NEW COMPONENTS	64
F. DELETION OF COMPONENTS	72
G. ORDER CHANGE WITHIN A REPEATING GROUP	78
H. ORDER CHANGE ACROSS REPEATING GROUPS	82
I. REPEATING GROUP - ELEMENT TYPE CHANGE	89
J. CONCLUSIONS	93
BIBLIOGRAPHY	96
APPENDIX A: CDEFNA TABLE	100
APPENDIX B: CDEFNB TABLE	104
APPENDIX C: CELS TABLE	108
APPENDIX D: CVALDR TABLE	111
APPENDIX E: CVALUS TABLE	114
APPENDIX F: CNAME TABLE	118
APPENDIX G: CENTS TABLE	120
APPENDIX H: CFIND TABLE	123
APPENDIX I: CDATA TABLE	127

LIST OF FIGURES

Figure	Page
1. Program system development effort	3
2. General-purpose system development effort	5
3. Unload-reload method of data base definition modification	16
4. Sample tree	22
5. Data base table building and modification	24
6. Partial reproduction of sample RFMS tables	34
7. CDEFNA-CELS linkage	44
8. CDEFNA-CELS linkage after processing a component name change command	45
9. CDEFNB table entries after the execution of a NUMCHG command	54
10. Element type change truth table	57

Chapter 1

DATA BASE MANAGEMENT

Within the environment of all organizations, many pieces of data are collected daily. These data come from payroll, inventory control, sales, and indeed almost all operations of the organization. The data are facts which are collected from observations or measurements and represent the reality in which this particular organization operates. So much time, effort, and money is spent in the collection and storage of these data that it represents a significant asset of the organization.⁸

This asset is not fully utilized, however, until the data is transformed through meaningful interpretation and correlation into information which can be used in the organization's decision-making process.²⁵ The information thus extracted is of value if it results in better decisions than would otherwise have been made. This implies that the information must be current to reflect the organization's present reality. Current information is based on current data which is captured, processed, and stored so that it will be available while it is still relevant.

A management information system is a means of supplying information based on the collected data to an organization.²⁵

It is an organized method of providing past, present, and projected information related to the internal operations and external intelligence of an organization. Its objective is to provide the most current and accurate information available.

In many cases the management information system is a man-machine system where the data to be interpreted is stored in a computer and retrieved by programs which are written to correlate these data. A combination of people, procedures, data processing equipment, data, and input-output devices make up the management information system.¹⁴ Its purpose is to utilize more fully the range of information-processing capabilities of digital computer systems, while at the same time permitting more extensive use of these capabilities by the non-specialist, nonprogrammer user.¹⁰ A management information system must provide an accurate, timely representation of the organization's reality to the personnel charged with the planning and operation of the organization.¹⁴

Before the advent of management information systems, an organization wishing to implement a new application, for example a new inventory control system, focused its attention on the programs to be written.⁹ The data files to be set up were considered to serve the single purpose of these procedural programs, and little or no consideration was given to any multi-purpose master data files. Each department within the organization maintained its own data files which were set up

and accessed by their own programs. If two or more departments had files on the same data items, effort was duplicated and discrepancies often occurred in the data on these files. Since reports to the organization's management were often based on the data in the various files, any discrepancies led to inconsistent, and sometimes incorrect, reports. The use of management information systems is an attempt to eliminate these data inconsistencies by consolidating data files across departments.

A typical system implementation cycle as described by Bleier and Vorhaus for a management information system is shown in Figure 1.²

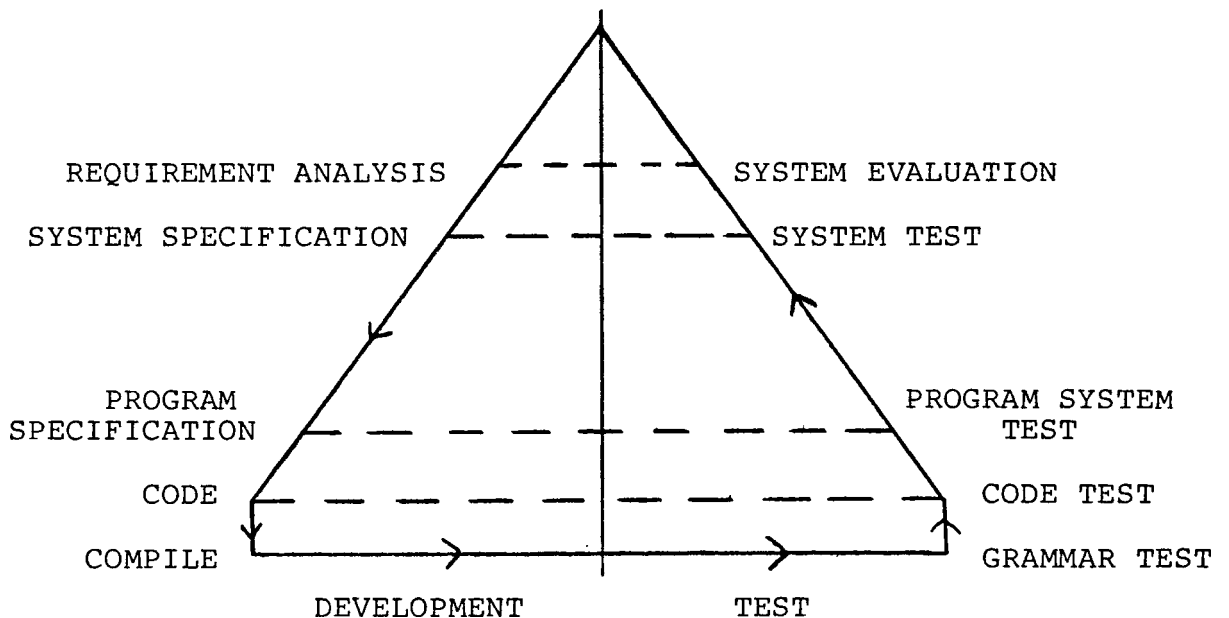


Figure 1. Program system development effort

At the beginning of the cycle, the users traditionally have requirements which may not be completely known, even to themselves. The system analysts translate the recognized needs into system design specifications. From this a program specification is produced which results in code. When the coding is completed, the code is compiled. These steps represent the development phase of the cycle.

During the test phase, each compilation is grammar tested and then the pieces of code are checked. Next, all of the pieces of code are collected into a program and tested as a unit. The program is then tested in the organization's environment and the finished product is evaluated.

Aside from the length of time it takes to implement such systems, the costs in terms of time, money, and manpower are excessive during the part of the cycle from program specification to program system test. System requirements tend to change over a period of time necessitating even more time spent at the bottom of the triangle to accommodate these changes. And often the decisions made by the system analysts during the implementation cycle are not those which would have been made by the system users.

These drawbacks can be overcome by the use of a

general-purpose system, a generalized data base management system, in the implementation cycle. (Figure 2.) The use of the general-purpose system reduces total development time and thus reduces cost. One important characteristic of a data base management system is that it can be constructed with sufficient generality to provide an environment for a wide range of potential uses, and thus avoid the need for numerous specialized subsystems.¹⁰ Furthermore, data bases which are maintained under a general system rather than several, perhaps incompatible, subsystems can probably be more easily adapted to meet future information requirements. And despite the generality features, data base management systems can be designed in such a manner that data bases ranging up to

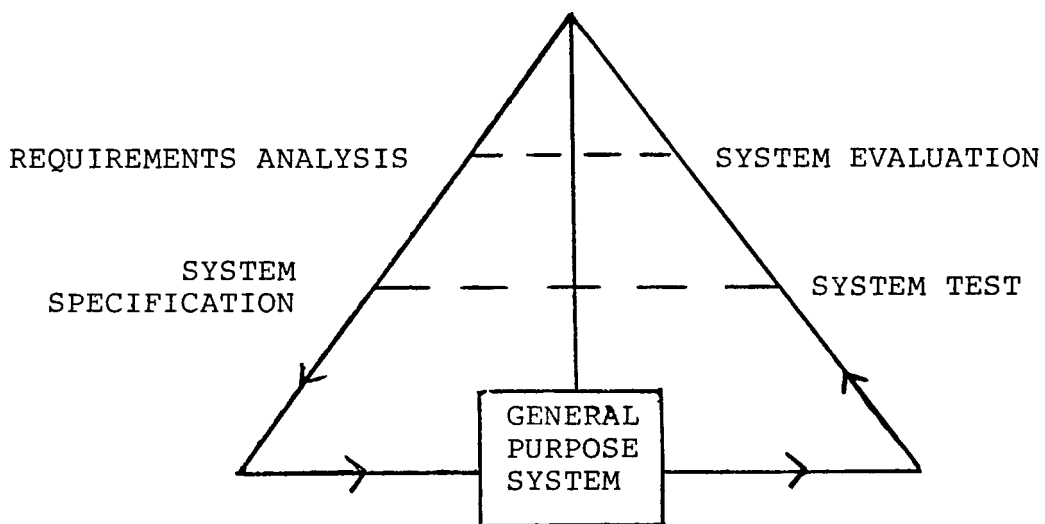


Figure 2. General-purpose system development effort

several hundred million characters in size can be handled efficiently. Indeed, in most systems, the size of the data base is a hardware constraint rather than a software one.

A data base management system is the software tool dedicated to storing, processing, and retrieving the data items for the management information system.⁸ It is a computer-based system which is used to establish, make available for use, and maintain the quality and integrity of a collection of data. A data base management system thus becomes an organizational tool for the user who has a collection of data and some idea of how he wants to use it.⁶ A data base management system is not a necessity of a management information system, but it is required for effective data management in an information system environment.²⁵

Management of a data base must encompass both the control and the use of the data resources of an organization.⁸ The data base management system must improve the accessibility of the data and at the same time maintain its integrity. Preserving the integrity of data implies safeguarding of the data from malicious or erroneous tampering or faulty equipment.²⁵ The system must then become the single door through which all accesses to the data base pass. This single availability of the

functions is desirable for improved performance of the processing algorithms and mandatory for the control of the integrity of the data base.

The concept behind a data base management system is that common data resources can be shared by a variety of users. A shared resource must be available and must respond in a timely and economic manner to diverse users operating in diverse modes to meet diverse needs.⁹ The sharing of the data base implies that different users and different processes are using the same actual data at virtually the same time. Under shared control, concurrent processes may look at a single piece of data in the data base, but only one process at a time may change that piece of data. This contrasts with single application programs and their resultant data bases where each program exercises exclusive control over its data base. This means that with single application programs, no other program or user can look at or change a piece of data.

In order to implement the goal of data sharing, each user of the shared resources must release control of the resources to a common responsible authority and must cooperate in their maintenance. The maintenance of the integrity of the data base must incorporate the concepts of conformance to data definition, validation of update

transactions, physical security and back-up protection, privacy, and the control of concurrent processes.⁹

The data base management system provides a tool for centralizing, coordinating, and integrating the collection and storage of the data. This then leads to increased consistency, reduced redundancy, and less duplicated effort in the capture and maintenance of an organization's data.

As shown in Figure 2, one of the reasons for the development of generalized data base management systems was to decrease the programming effort and the lead time from the formulation of a data base to its availability for use by application programs.¹² A data base management system is a common interface between all users and a set of generalized file processing routines which address the data base. The purpose of this user interface is to allow the user to use descriptive labels, or names, when accessing the data and to allow the use of generalized processing capabilities. The system can then make the data base available to a user who is not familiar with the detailed computer steps the system uses to process his request, and at the same time maintain control over the content of the data base.

The use of a data base management system can lead to reduced effort in responding to changing data base requirements. This makes the system especially attractive in applications which have a large data base with a sophisticated data structure and processing requirements which evolve over a period of time. The one-time development costs of the data base can then be amortized over many applications.¹² However, in some applications, efficiency may be sacrificed in favor of more generalized processing capabilities.

In order for a management information system to meet successfully the needs of its community of users, several elements must be present.¹⁴ The system must have the direction, involvement, and commitment of the system administrators. It must be flexible and evolutionary, and at the same time operate within the limits of the available computer resources. The system must quickly produce meaningful results for its users. All management information systems must contain a reporting system, a data base, and a facility for responding to specific requests. This implies that a data base management system must have a powerful, generalized query language; tools to define, create, revise, and interrogate a data base; and a mechanism to service and control accesses to the data base.

All large scale data base management systems have several features in common. These features are defined by the Codasyl Systems Committee in its reports on generalized data base management systems.^{4,5} These features provide for file definition (data organization), file creation, file updating, and file interrogation. Other features of a data base management system include a report generator and a program language interface.

In the file definition state, the data base management system sets up the basic subsystems of the data base. These subsystems are the nucleus, the schema, and the filter.²² The term nucleus refers to those files which will contain the explicitly represented data items entered by the system user. The schema is the particular definition which underlies the data base. This includes the labels chosen by the user for the data items and their informal definitions. Some of the other descriptions included in the schema are a file of files listing all files that are a part of the system, together with identifying information including file names, the address of the starting entry in each file, the length of each entry, and the file type.¹⁸ The filter is a group of subroutines which protect the data base and its users against false messages and messages which are not meaningful according to the specifications and definitions

embedded in the schema.

During the file creation process, a set of data items is presented to the data base management system to form the initial copy of the file.⁵ File creation also includes a formal statement of the validation conditions which the data entering the file must satisfy. In some systems it was deemed easier and more efficient to validate input transactions as they are received than to monitor continuously the data base against a comprehensive data definition. Therefore, in these systems the data items are checked for conformance to the stored data definition before they are entered into the data base.

Interrogation is the process of selecting and extracting for presentation some part of the whole data base. The interrogation process typically consists of two parts. In the first part the selection criteria, a condition on the data in the file, is expressed, while the second part defines which data items must be copied out of the file into the output list.⁴ In the generalized management information system, the interrogation process includes the processes of data selection, sorting, and report formatting. Data items to be examined for selection may be specified as being logically related by one or more Boolean operations. The user is able to formulate

a query in the language of the system without detailing the sequence of steps used to access the data base and extract the information.⁵ When the user wishes to change what he wants to examine in the data base, he merely changes his selection criteria.¹² This contrasts with the traditional data processing approach of writing a new application program with each change in the object of the interrogation.

The end results of any data base management system are the reports that it produces.²⁶ This is the information that is retrieved from the data base for use in monitoring the performance of the organization and to aid in making decisions which affect the organization. The ability to produce accurate reports quickly and easily is vital for the efficient management of the organization. Elementary data base management systems may search sequential files having simple record structures and provide only rudimentary report formatting facilities.⁵ More complex systems handle several files via "indices" or "links" and provide for elaborate formatting schemes.

The dynamic environment of an organization will require changes in the data items in the data base to keep the files current.²⁶ The timeliness and accuracy of the data base determines the accuracy of the reports produced by it. This implies an on-going activity of the data

base management system, namely the processing of the daily transactions that take place in the organization. Many data base management systems provide a data file updating facility. Updating is the procedure the data base undergoes to change the value content of some part of the data base.⁵ During the update process, input messages are received which define the update "target" via selection criteria.⁴ The update process then executes a two-part built-in algorithm. First the target selection criteria is satisfied, and then the specified action is carried out on the selected data items.

This update process, however, precludes any restructuring of the data caused by modification to the stored data definition. Consider what happens with conventional data base management systems when the data requirements of the organization change.²⁶ The administrators of the management information system must then become concerned with ways in which to define new data items to respond to these changes. The definition of the evolvability of a system is its ability to reach a highly developed state through growth and change.⁹ These changes may cover both expansion and contraction of the data base. A system's ability to change with the requirements of its host organization minimizes its risk of technological obsolescence and also minimizes the cost and disruption

of any redefinition and restructuring of the data base.

Unlimited data base restructuring is almost never provided for in data base management systems. Therefore the user of most present data base management systems must have in mind not only the information he currently wishes to have available, but he also must estimate new data requirements.⁶ In a dynamic environment, data requirements are bound to change. Sometimes these data definition revisions come about as the result of experience gained during a period of use of the data base. Other causes for these revisions might be additional requirements imposed by government regulations or technological advances.

The few data base management systems which do recognize the changing needs of its community of users provide only for very restricted modifications to the stored data definition and a subsequent remapping of the entire data base to conform to the new definition.⁵ This process can indeed be costly and time-consuming for very large data bases and in most cases approaches the creation cost of an entirely new data base in the amount of work required by the computer system.

In the normal course of use of a data base, however, the modifications needed to the data definition may not fall within very restricted bounds. A data base manage-

ment system should provide for the introduction or elimination of a data element, changing of the heirarchical structure within the entry, or changes within a particular element's definition.²² And these modifications should be accomplished with the minimum amount of remapping within the data base.

It is the update process, then, which I wish to address in detail with the objective of introducing a modification to the update procedure. Specifically, this modification will allow the update process to restructure the data and modify the stored data definition with a minimum of change within the data base. This modification will be particularly applicable to the Remote File Management System (RFMS) data base management system as currently implemented on the Univac 1110. Similar modifications could also be made to the CDC and IBM versions.

As an example of a data base restructuring situation, consider an academic environment. Many colleges and universities use large scale data base management systems to maintain their student records. The data items stored for each student might include the student's name, his address, his telephone number, and his unique, university-assigned, student number. These records could cover many years, thousands of students, and might involve millions of separate data items.

What would happen if the student records' administrators were suddenly informed that they must go back and add the social security number of each student to his record? This would involve changing the stored data description of an existing, valued data base and most data base management systems make no provision for such modifications.

As noted above, of the few systems which do allow for modifications of the data base definition, almost all implement this feature by completely unloading the data base, regardless of the size of the data base or of the complexity of the modification involved, and reloading it according to the modified definition, or schema. (Figure 3) In some data base management systems, unloading of the data base involves the dumping of the data values into an output file with no provision for formatting of the data. More sophisticated

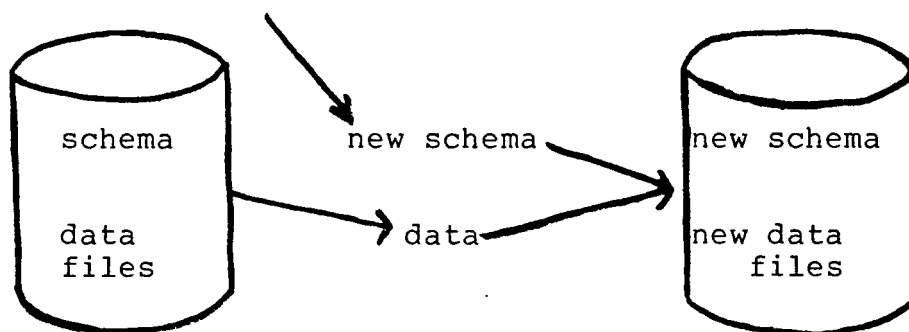


Figure 3. Unload-reload method of data base definition modification

systems, however, usually provide a formatting capability which allows the data values to be output in such a manner that the output file can then be used as the input file for the new data base.

It is possible to modify the Univac version of the RFMS update process so that definition modifications can be made without the unload-reload operation. The ability to modify the update process in this manner is the result of several features which I implemented in this version of RFMS. In order to understand these features and their effect upon the update process, it is necessary to examine in some detail the RFMS program structure. Consideration will also be given to the RFMS data file structures and how they are used by the data base management system.

Chapter 2

THE REMOTE FILE MANAGEMENT SYSTEM

The Remote File Management System (RFMS) is fundamentally a file processing system directed by input messages. All of the data items in the files are specified by the user. The system keeps files of the processing subroutines, the language specifications, the format specifications, and any other special-purpose files. When an input message is received, the translator, directed by a specific language specification, translates the message into a list of operations to be performed. The processor then steps through this list interpretively, accessing or modifying the data base, and invoking any processing routines specified. If any output is generated, it is placed in the output files where output-formatting programs perform the required formatting procedures. The Remote File Management System is a self-contained system. The user of the data does not exercise control over the sequence of the detailed steps the system uses to process his requests, nor the sequence in which data is examined and moved from one level of storage to another.

The computer code for the Remote File Management System includes more than 100 labeled common blocks and approximately 500 subroutines.¹¹ Since no computer could contain at one time

the code generated by all of these subroutines, it was necessary not to require that all of the code be in core at the same time. Furthermore, the RFMS designers did not want to require that the computer be solely dedicated to RFMS whenever the data base management system was loaded.

This led to the partitioning of the RFMS code into approximately 30 overlays. An overlay is a block of code which is written on a file in absolute form and loaded into the computer without relocation. The main overlay is core resident throughout the execution of a RFMS process and it requests the loading of the other overlays as they are needed. Overlays called in by the main overlay are placed in core immediately following the last location of the main overlay.

The 500 RFMS subroutines are divided among the overlays on the basis of "functional requirements" and the overlays are then grouped into modules. These modules conform to the generalized data base management system features as outlined by the Codasyl Committee.^{4,5} In the Remote File Management System the modules are CONTROL, DEFINE, LOADER, RETRIEVAL, REGENT (Report Generator), and UPDATE.¹⁰

CONTROL -- contains the program language interface between the user and the other modules. It maintains

communication between the user and RFMS by supervising user file operations and maintains communication between the different service modules within RFMS so that the correct program module is called to process the user's requests.

- DEFINE -- allows a user to create a data base description, i.e., to name the elements of the entries in his file, to indicate heirarchical relationships among elements, and to specify what type of data will be associated with named data elements.

- LOADER -- provides a flexible capability for creating a new data base, taking as input the data base description supplied by the user to DEFINE and the data values for entries supplied by the user.

- RETRIEVAL -- provides search capability based on one or many search criteria expressed in terms of the elements of the user's data base definition.

- REGENT -- provides flexible report generation capabilities

suitable for formatting RETRIEVAL answers
for batch or remote output.

UPDATE -- provides capability for editing an existing
data base through instructions which add,
delete, insert, and change data values.

A description of the RFMS data files must begin with
a formal definition of some of the terms used to describe
the files.

A tree is composed of a heirarchy of components called
nodes.^{15,17} With the exception of the uppermost node, the
root, every node has one and only one node related to it at
a higher level (parent). Each node may have none, one, or
more nodes related to it at a lower level (children). In
RFMS terminology, the root node is called ENTRY and resides
at level 0. Its children also reside at level 0, but sub-
sequent descendents of the children reside at levels 1, 2,
3, etc. (Figure 4)

The RFMS nodes are specified as being either elements
or repeating groups. An element is the smallest logical
unit of named data in the data base. A node which is specified
as an element may never have any descendent nodes. Actual
data in the data base which are associated with the elements
are called data values. A repeating group is a named node

which is used to associate other descendent nodes. Its children may be elements or other repeating groups. A repeating group never has data values associated directly with it. The root node, ENTRY, is considered the parent (or root) repeating group for the tree.

Throughout the remainder of this paper the term component will be used whenever a description is equally applicable to an element or a repeating group. Components may be described further as having a component name consisting of a descriptive label from 1 to 150 characters long. The component also has a shorthand label, or component number, of from 1 to 6 numbers with no embedded blanks. The component's type indicates to the system the actual format of the data to be associated with the component.

A data set consists of one set of component numbers and their corresponding data values which are the sub-

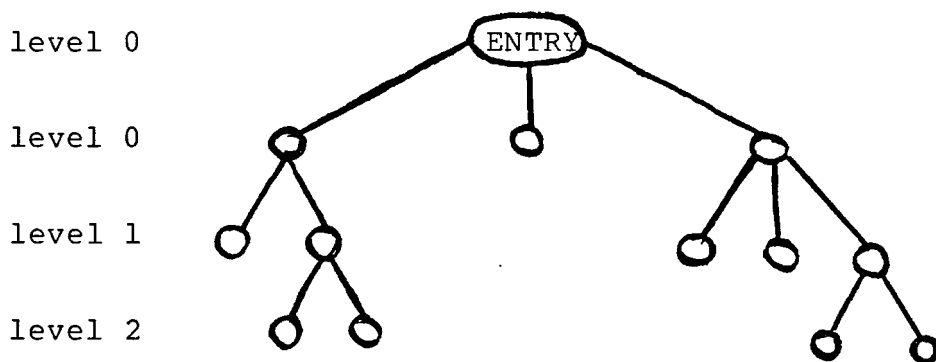


Figure 4. Sample tree

ordinate components of a repeating group or ENTRY.¹⁰ Each component number may appear no more than once in a data set, but every subordinate component need not appear in all of its allowable data sets.

The Remote File Management System utilizes a tree heirarchy when building the data base tables.¹⁰ The logical entry definition is the declaration of component labels which explicitly states each component's type, number, and all component nesting, or heirarchy. This heirarchy is the logical entry structure which is used to describe to RFMS the relationships within the input data which should be established. The RFMS heirarchy is also fully-inverted, which means that every node in the tree can be accessed directly, or used as a key. This allows all components to be located by their contents.

An RFMS data base consists of a set of nine designated random access files called tables. This set can be thought of as being broken into three subsets; the Definition Tables, the Selection Tables, and the Retrieval Tables. The CDEFNA, CDEFNB, and CELS tables are referred to as the Definition Tables. These tables are built and modified by the DEFINE module and contain the logical entry definition for the data base. The CVALDR, CVALUS, CNAME, and CENTS tables are referred to as the Selection Tables. They are built by the LOADER module and changed by the UPDATE

module. These tables contain the actual data values associated with each element in the definition of the data base. The Retrieval Tables are CFIND and CDATA. These tables are also built by the LOADER module and changed by the UPDATE module and contain the data set linkages which associate the unique data values (in the Selection Tables) to their respective logical entry definition elements (in the Definition Tables). (Figure 5)

If a data base definition has been made but data values have not been entered for the defined elements, the RFMS data base consists minimally of the Definition Tables (CDEFNA, CDEFNB, and CELS). After data values have been

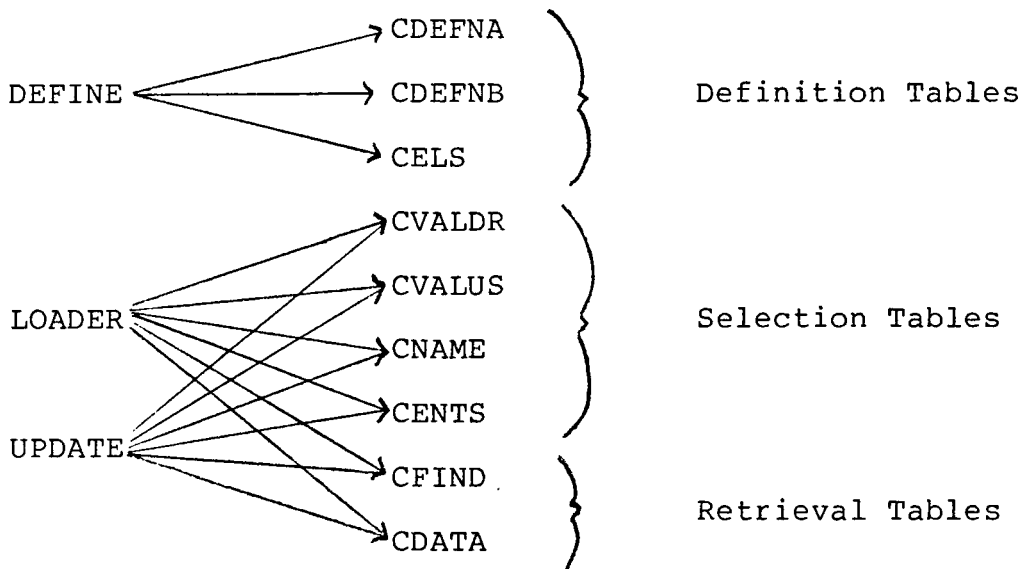


Figure 5. Data base table building and modification

entered into the data base according to the definition held in the Definition Tables, the data base consists of all nine tables.

Upon entry to the DEFINE module to initiate a new RFMS data base, all user input is kept in the form of scratch tables. This allows the user to modify and refine his data base definition as he is building the logical structure. Once he is satisfied with his declaration, the user issues a FINALIZE statement which builds the permanent versions of the CDEFNA, CDEFNB, and CELS tables. Subsequent entries to the data base must then be made through the LOADER (for the initial data values load only), REGENT, UPDATE, or RETRIEVAL modules.

The LOADER module enters the user's data into internal data structures for the newly defined data base. It scans the input strings, builds the Selection Tables (CVALDR, CVALUS, CNAME, and CENTS) and then builds the Retrieval Tables (CFIND and CDATA). In the LOADER string, the data base structure is explicitly detailed by the order in which the data sets occur.

Once values have been entered into the data base, access to it is limited to the RETRIEVAL module, the REGENT module, and the UPDATE module. The UPDATE module allows the user to modify data values or the order in which data sets are associated with their parent repeating

group. The UPDATE module does not allow the introduction of any new nodes to the logical entry structure or any reordering of nodes within the existing structure. The RETRIEVAL and REGENT modules make no modifications to the data base. They merely select and report requested data.

To aid in understanding the relationships between the RFMS tables, the component parts of each table are described in Appendices A through I. The table structures are shown as they are currently implemented by the Univac version of RFMS. The descriptions for the various items within each table are taken from the original RFMS documentation for the University of Texas.¹¹

The CDEFNA table contains the logical entry structure of the data base definition as ordered by the user. Each two-word entry in this table corresponds to one component in the logical entry definition. Items in the CDEFNA and CDEFNB tables contain properties and attributes for each component in the definition and the relationship among components as described by the user. The CDEFNA, CDEFNB, and CELS tables form the map and the pathway to entering and finding all of the current data in the data base.

The order of the CDEFNA and CDEFNB table entries is determined by the user. Words 0 and 1 of these tables are reserved and used by the system as an implied repeating

group component for the level 0 set of components in the logical entry definition. Entry items in words 2 and 3 apply to the first user-defined component description, words 4 and 5 to the second component description, and so on. This two-for-one correspondence between the CDEFN entries and the component descriptions becomes an index and a relative address pointer to the CDEFN tables. This index is called the system-assigned component number and is used throughout the Selection and Retrieval Tables rather than the user-defined component number. The Definition Tables, CDEFNA and CDEFNB, are ordered and indexed in parallel. Items in each entry of the CDEFNA table are described in Appendix A.

The CDEFNB table contains the user-assigned component number and the type of the component as declared by the user in the type description for the component. Each entry in the CDEFNB table supplements and parallels information found in the CDEFNA table for a given component, including words 0 and 1. Items in each entry of the CDEFNB table are described in Appendix B.

Each entry in the CELS table corresponds to a component in the logical entry definition. A CELS entry holds the component name declared by the user with all leading, extraneous (more than one) embedded, and trailing blanks deleted. A CELS entry exists for each entry in the CDEFN

tables excluding the level 0 entry which is reserved for use by the system. The first CELS entry begins at word 2 in the CELS table. Words 0 and 1 are reserved for system use. CELS entries need not be ordered as they are accessed by a relative address pointer found in the CDEFNA table. Entry items of the CELS table are described in Appendix C.

The CVALDR table serves as a page directory to the data values for each element. Each entry in the CVALDR table corresponds to a single partition in the CVALUS table. No two CVALDR entries point to the same CVALUS partition. For each element in the logical entry definition for which data values have been entered, there are as many CVALDR entries as the number of CVALUS partitions created for the element. CVALDR entries need not be ordered, as the set of entries pertaining to a single element are linked by item CVNEXT as shown in Appendix D. CVALDR entries indicate the highest value in a CVALUS partition for an element and thus are used to point to the CVALUS partition where a data value may be found if it exists in the data base for the given element. Items in each entry of the CVALDR table are described in Appendix D.

Each unique data value assigned to a given element is represented by one and only one entry in the CVALUS table. Each CVALUS partition contains entries pertaining

to only one element. Thus if an identical unique data value is assigned more than once to a given element, only one CVALUS entry is created. However, if the same unique data value is assigned to two or more different elements, a CVALUS entry is created and stored in the proper partitions for each of the respective elements. Entries within a partition are ordered by low value first, with respect to the total value (not truncated) in table CNAME. The set of partitions for any given element need not be ordered or consecutive in the CVALUS table since the CVALDR table is a directory that identifies, links in order, and locates the complete set of partitions for each element that has been assigned values. The CVALUS representation of a data value is not necessarily the complete actual value, but item CNAMB in each CVALUS entry points to the full actual value as found in the corresponding CNAME table entry which is available if further matching is needed during a compare operation. The items in each entry of the CVALUS table are described in Appendix E.

The CNAME table contains the full display value string for unique data values. A CNAME entry exists for each unique data value assigned to an element. If the value was assigned to a specific element in more than one data set, one and only one copy of the value is

retained in the CNAME table, and it is pointed to by relative address pointers in the appropriate CDATA table entries. CNAME entries are not necessarily ordered, but there is a one-to-one correspondence between entries in the CNAME and CVALUS tables, with each CVALUS entry having item CNAMB as the relative address pointer to its respective CNAME entry. CNAME entries have no relative address pointers to any other table. Entries in the CNAME table are used in matching when uniqueness is not assured by the corresponding CVALUS entry and are used in generating report output. The items in each entry of the CNAME table are described in Appendix F.

Whenever an identical data value has been entered for an element in more than one data set, a CENTS entry is created to hold all relative address pointers to the CFIND table entries for the respective data sets. If the unique value occurs only once for the element, no CENTS entry is created and the relative address pointer to the single CFIND entry for the specific data set is stored in item CPFNDA of the CVALUS entry for the unique data value.

A CENTS entry may consist of one or more CENTS blocks in one or more partitions. One block is created initially if less than partition size. Additional updates create additional blocks. All blocks pertaining to multiple occurrences of the same unique data value for an element

are linked. No order is necessary among the relative address pointers within a CENTS entry and, likewise, the entire CENTS table need not be ordered as associated blocks are linked by item CLINK. Items in each entry of the CENTS table are described in Appendix G.

The CFIND table is a retrieval tree of linked data sets and logical entries. Its entries preserve the structure of the data sets in a data base. For each data set there exists one CFIND entry. If no values were entered for a data set but other data sets exist at a deeper logical level for the same family of data sets, then a dummy CFIND entry is created for the null parent set. Complete vertical and horizontal heirarchical structure is preserved within a logical entry and between logical entries by three two-way pointers in each CFIND entry. CFIND entries need not be ordered nor need they be consecutive in the CFIND table, as each is linked to its logical associates. Items in each entry of the CFIND table are described in Appendix H.

Each CDATA entry serves as a directory to access actual data values entered for specific elements in a single data set. Every data set in the data base that contains at least one data value has a CDATA entry (accessed by item CPLOC of the corresponding CFIND entry for the set). If the data set is a dummy entry in the

CFIND table, then no CDATE entry exists and item CPLOC =0.

Each CDATE entry is made up of one block of two or more consecutive words. Within a block, it is unnecessary to order the words by their contents. One word exists for each element having a data value in the data set for this CDATE entry. The items in each entry of the CDATE table are described in Appendix I.

In order to show the logic of the tables, consider the following retrieval request:

```
PRINT ENROLLMENT LIMIT WHERE INSTRUCTOR EQ OSBURN  
AND SECTION NUMBER EQ 0002;
```

The RETRIEVAL module examines the conditional portion of the request, the WHERE clause, and finds the first Boolean expression, INSTRUCTOR EQ OSBURN. Using the partial reproduction of the data base tables as shown in Figure 6, a search is made of the CELS table until a match is made on INSTRUCTOR. This CELS entry points to the CDEFNA-CDEFNB entries related to INSTRUCTOR. Having found that entry, the CPVDIR pointer is followed to search the CVALDR table for a pointer to the appropriate CVALUS partition. Having found the appropriate CVALUS partition, the program looks for the value OSBURN. Having found the

value OSBURN in the CVALUS table, the program checks to see if OSBURN occurred only once for the element INSTRUCTOR. Since the CONCE item is set to 1, OSBURN occurs only once as an INSTRUCTOR and his CPFNDA pointer points to a CFIND entry instead of a CENTS entry. The CPFNDA pointer is held temporarily.

The process is then repeated using the second BOOLEAN expression, SECTION NUMBER EQ 0002. The CELS, CDEFNA, CDEFNB, CVALDR, and CVALUS tables are all examined as described above. The CPFNDA pointer to the CFIND table is once again temporarily held.

The program then examines the logical operator and finds it to be an AND. The two CPFNDA pointers are "ANDed" together giving the pointer to the CFIND table for the data set containing both OSBURN and 0002.

Table CFIND contains the pointer to the CDATE entry for the selected data set. The data value for component number 8 (ENROLLMENT LIMIT) is the required output. The program finds the CDATE entry for the data set and looks for component number 8. It then follows the CPVAL pointer to the CNAME table and the data value 75 is printed out.

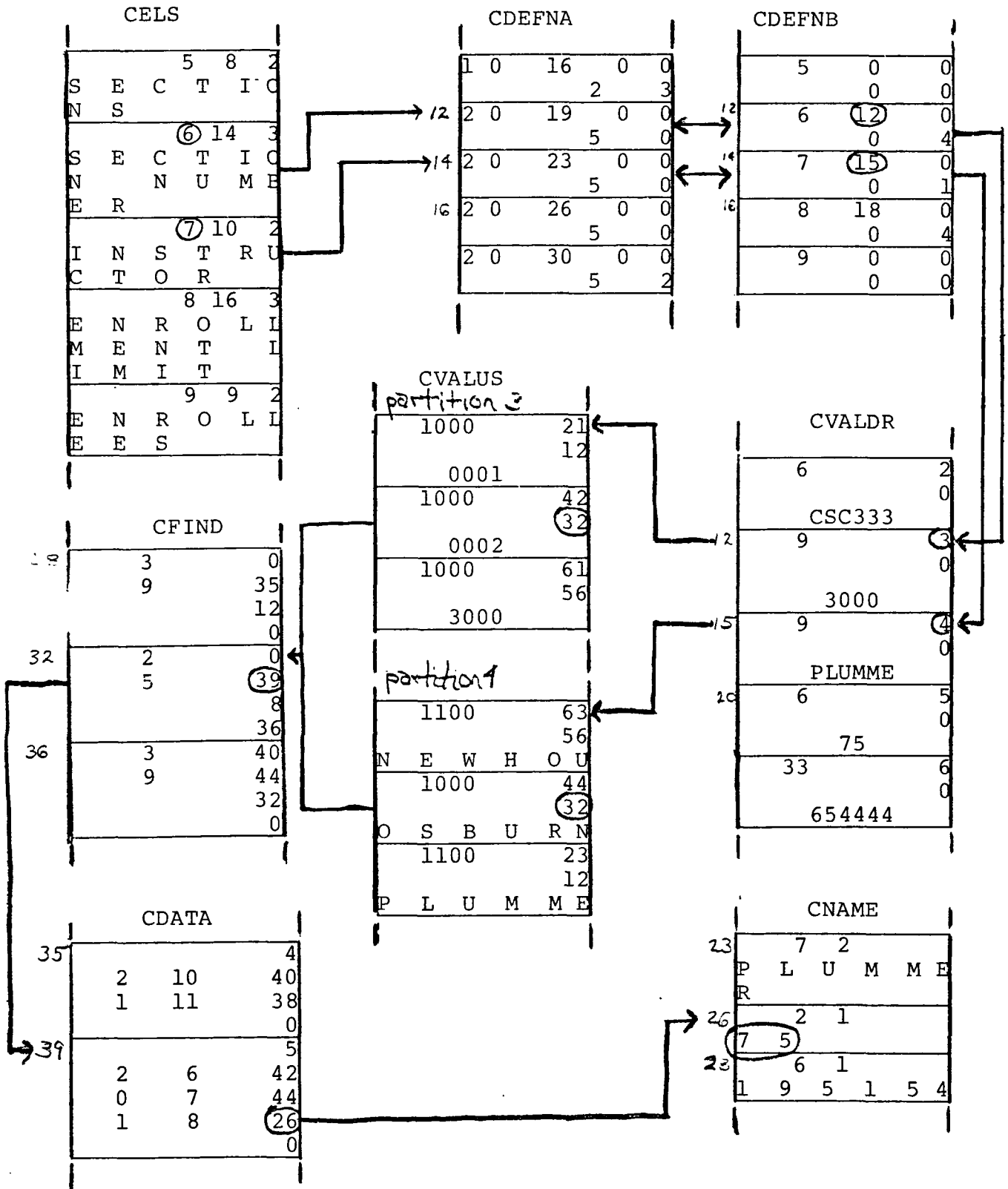


Figure 6. Partial reproduction of sample RFMS tables

The original RFMS was written for the CDC 6600 series of computers. That computer series has a logical memory broken down into 60-bit words. The CDATE table for RFMS under CDC consists of one-word entries, each word containing the relevant information for one element in a given data set. The low-order third of each word points to the relative CDATE address of the next element in that data set. As data sets expand to accommodate an element which previously had no data value in that data set, the added element can be quickly inserted at the end of the CDATE table and linked back to the other members of its data set by this pointer. Data sets can thereby expand and contract with a minimum of change within the CDATE table.

When RFMS was converted to run on the Univac 1100 series computers, a new CDATE table format had to be adopted to accommodate the computer's shorter (36 bit) word length. The new format consists of a header word containing a count of the entries in the data set with the data set entries immediately following. No provision was made originally for expanding or contracting the sizes of the data sets. This condition was justifiable since certain applications originally planned to use only the DEFINE, LOADER, and RETRIEVAL modules. Each load of the data base would thereby be static and expanding or contracting data sets would not be encountered.

With the decision to implement the UPDATE module came the necessity of providing in the CDATE tables for data sets which change size. Since no item had been set up in the CDATE entries to provide for a relative address pointer to the next CDATE entry for that data set, it was decided that the entire CDATE entry for a data set which expands in size would be rewritten. CDATE entries for data sets which contract in size would be left where they were and would be followed by unused words which had previously been a part of that CDATE entry. The question of where to rewrite the expanded data sets and what to do with the unused words within the CDATE table then had to be addressed. More specifically, the problems of available space linkages and garbage collection had to be considered.

The original design specifications for RFMS provided for a common block list, GARBAGE, to contain a pointer to the word after the last word used, the first "available space", in the CENTS, CDATE, CFIND, and CNAME tables. This list was set up by the LOADER module. As the Selection and Retrieval Tables were modified by the UPDATE module, the UPDATE module was to mark the words as deleted that were no longer used in the tables, and link these words into an available space list, or collect the garbage. The pointers from this list would then be held in GARBAGE.

Since the Univac application under consideration originally intended for each load of the data base to be static, the available space linkages for garbage collection, or GARBAGE, had never been used for more than the first available space pointer.

A discussion of what, if any, type of garbage collection should be implemented needs to include an analysis of how the tables are used. Upon entrance to the RETRIEVAL module, three or more CDATE partitions are set aside in the RFMS buffer area. In searching for the data sets that satisfy the retrieval request (or update request), it is assumed that a minimum of swapping of CDATE partitions will occur due to the fact that most requests consist of components which are logically contiguous (or related) and therefore will be somewhat physically contiguous. This physical proximity of data sets is indeed a direct consequence of the stringent ordering requirements of the LOADER module. If data sets which change sizes are rewritten in the last CDATE partition with no regard to ordering, the minimal swapping assumption will break down to the extent that there are changing data set sizes.

Three garbage collection schemes were considered for this Univac implementation.

- (a) No garbage collection. Under this scheme, all CDATA data sets which expand in size would be rewritten in the last CDATA partition. This is the easiest scheme to implement and the fastest to execute in the UPDATE mode as no time is spent marking and linking unused words in the CDATA table. It is a viable alternative if the largest majority of the UPDATE requests do not change the sizes of existing data sets, only their values, and the whole data base is reloaded periodically, theoretically before the slight RETRIEVAL degradation begins to show.
- (b) Partial garbage collection within a partition. This scheme would require the LOADER module to create an overflow area by padding the bottom of each CDATA partition. The percentage of padding could be set by the user to reflect the percentage of data sets he expects to change in size over the life of this load of the data base. The theory behind partial garbage collection is that RETRIEVAL degradation can be minimized by rewriting the changed data sets in the same partition the original data set was in. This could be accomplished by designating the first word of each partition as an available space pointer

(again requiring a change in the LOADER module). More simply, each available word could be marked with a unique code in the high-order bits (another change in LOADER) and a sequential search made down the partition until the available space is hit. Failure to find enough available space in a CDATE partition to rewrite a data set would cause the set to be rewritten in the last partition. This would cause some increase in the execution time of the UPDATE module due to the sequential search and the marking of unused words, but could lessen the RETRIEVAL degradation due to physical separation of data sets.

- (c) Full garbage collection. This scheme requires the most time to maintain. Available space would be linked together throughout the CDATE partitions and data sets which must be rewritten would be placed in the first available space large enough to hold them. This would have the same logical fragmentation effect as the scheme using no garbage collection, only the data sets would tend to be rewritten in the first CDATE partitions instead of the last ones. Due to the degradation of the UPDATE module caused by the building of

the available space linkages, this approach is not a viable alternative unless the physical space requirements for the CDATE table become such that it is critical that each word in the partitions be used.

Examination of the types of update requests that are most frequently received showed them to be almost exclusively changes in existing data values and insertions of entire new data sets in which all the elements already have values. This evaluation led to the choice of no garbage collection for the CDATE table, as changing data set sizes were not anticipated as the result in any significant percentage of the update requests.

When the LOADER module finishes building the Selection and Retrieval Tables, it stores the first-available-space address in the system-reserved words at the top of each table. Since this pointer was already stored in the CDATE table, the decision to do no garbage collection required no revision in the LOADER module. The UPDATE module was simply modified so that expanding data sets were entirely rewritten in the last CDATE partition, the next-available-space address in the CDATE table was updated to reflect this addition, and the CPLOC pointer of the CFIND table was changed to point to the new location for this data set.

The ease with which expanding and contracting data sets could now be handled due to the decision to move expanding data sets to the end of the CDATE table suggested to me the possibility of allowing for redefinition of an existing data base within the confines of the UPDATE module. This would eliminate the need for the time-consuming process of unloading and reloading the data base to accommodate a change in its logical entry definition. With this innovation in handling definition modifications in mind, I began investigating its feasibility for RFMS as currently implemented on the Univac 1110.

Chapter 3

DEFINITION MODIFICATION IN THE REMOTE FILE MANAGEMENT SYSTEM

In order to design a data base definition modification system, several questions had to be addressed. The first question revolved around the actual structure of the tables which comprise the data base and their linkages to one another. Could the present form of these tables accommodate all the various definition modifications, and, if not, what changes needed to be made to the tables and their linkages? The second question dealt with the application (user) programs which were written under various definitions, or editions, of the data base. How could one data base edition differ from another and still be processed by application programs written under a different edition? The final question addressed the relative desirability of this type of data base definition modification. Were there circumstances under which the modifications enumerated below were superior to the only other available definition modification method, the unload-reload method?

The first consideration an attempt to modify the Remote File Management System to accommodate definition changes had to test was whether or not the present structure of the data base tables could support these changes. In order to determine

if the present data base tables could support definition modifications, all of the various types of modifications had to be examined. Possible definition modifications consist of a change in the user-assigned component name, or the user-assigned component number, or the component's type description. Other modifications to the definition include the addition or deletion of an element or repeating group, or a change in the order in which components are defined within the logical entry structure.

A. COMPONENT NAMES

A change in the user-assigned component name could be easily handled within the present confines of the data base tables. The CDEFNA and CELS tables contain the only references to the component name. (Figure 7) The CNAMA item in the CDEFNA table is a relative address pointer to the header word of the CELS table entry where the component name for that entry is stored. Since the component names in the CELS table occupy only as many words as are necessary to hold each component's name and one header word, modifications to component names could be most rapidly accomplished by appending the new component names to the end of the CELS table.

At the end of the initial definition process of a

data base, the DEFINE module presently places in the CNAMA item of word 0 of the CDEFNA table a relative address pointer to the next available address for storing a component name in the CELS table. The routine handling the change in a component's name would simply have to pick up the next available space pointer from word 0 of the CDEFNA table and insert the new name at this point in the CELS table. The CPEL item in the old component name's CELS entry would be copied into the new component name's CPEL item space. The CNAMA item in the component's CDEFNA table entry is

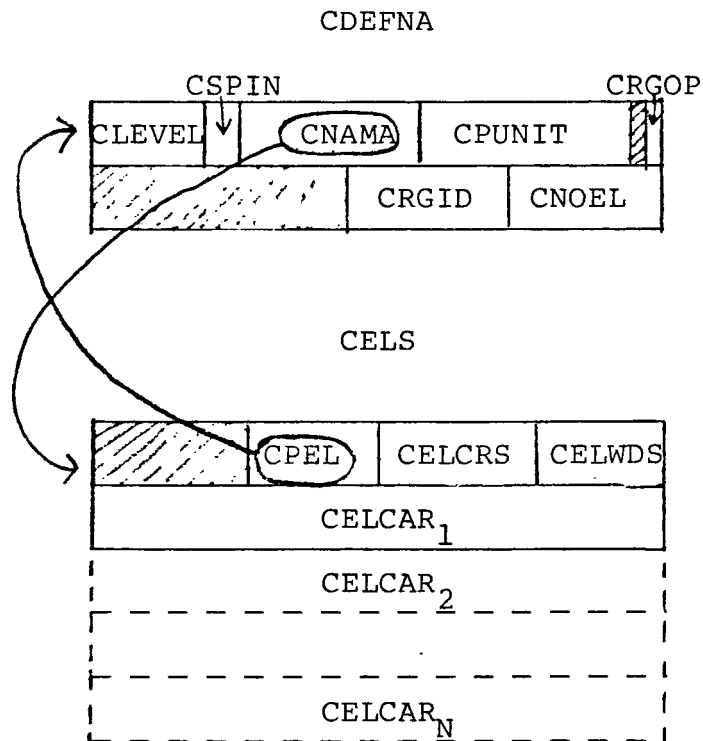


Figure 7. CDEFNA-CELS linkage

then changed to point to the new component name and the CNAMA item in word 0 of CDEFNA is updated to point to the next available space in the CELS table after the new name. (Figure 8)

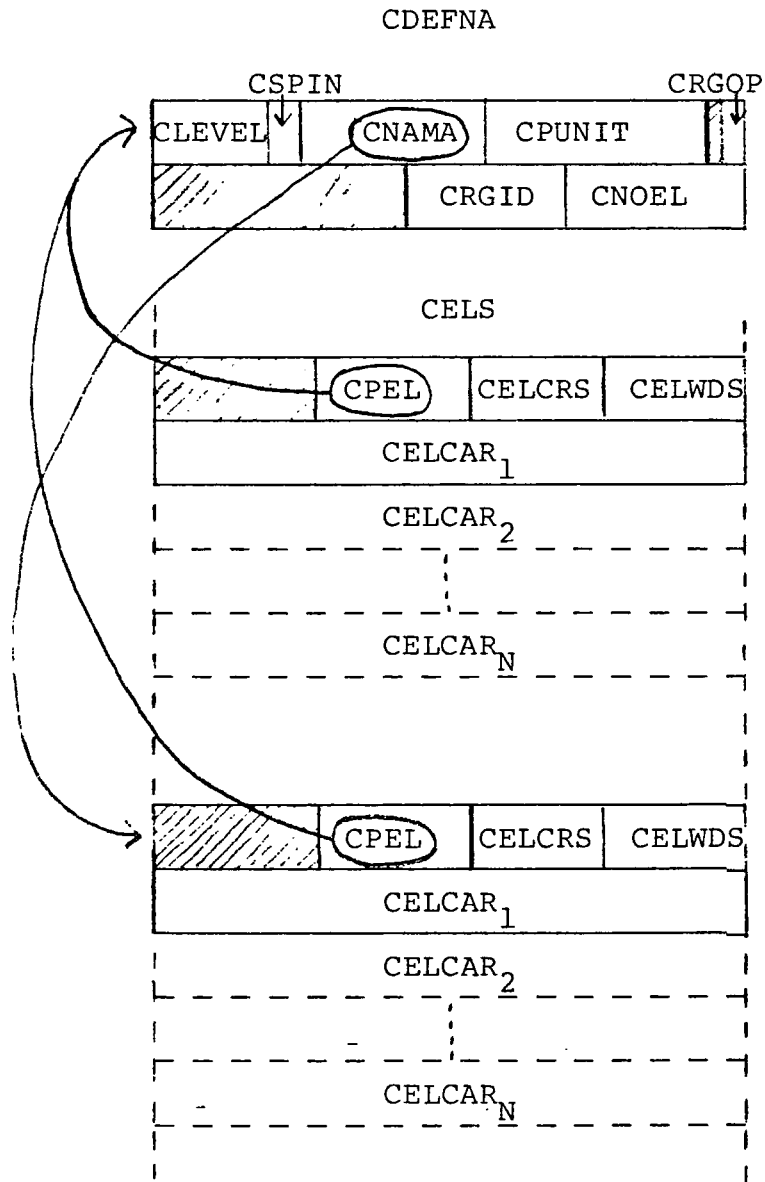


Figure 8. CDEFNA-CELS linkage after processing a component name change command

The command to change a component's name might look like:

```
NAMCHG <C1> EQ <NEWNAME>;
```

where 1 is the user-assigned component number for the component whose name will be changed to NEWNAME.

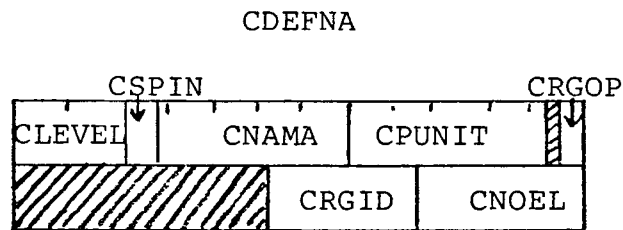
To allow user programs which were written under previous editions of the data base to continue to run under the present edition, CELS table words which are occupied by component names which have been changed must not be overwritten. Consider the following command issued from an application program:

```
PRINT GRADE WHERE INSTRUCTOR EQ JONES AND  
STUDENT NAME EQ BLACK;
```

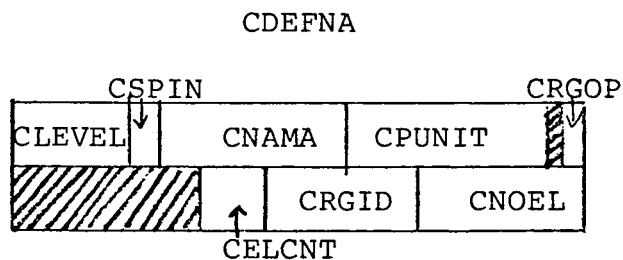
What would happen to this command if under a subsequent edition of the data base the component name INSTRUCTOR were changed to PROFESSOR? In its attempt to satisfy the WHERE clause of the command, the retrieval process searches down through the CELS table until it finds a match on the component name, INSTRUCTOR. If that space in the CELS table had been reclaimed and overwritten, the retrieval search would have failed and the application program would

have received the message that the component INSTRUCTOR was not a member of that data base. By leaving the old name in the CELS table, the retrieval process will find it upon a search through the CELS table.

The decision to leave the old component names in the CELS table led to the question of how to relate the synonymous names for a given component. Since the CDEFNA table contains the pointer to the most recent name associated with a component, it is proposed that an unused portion of each component's CDEFNA entry be modified to hold a count, CELCNT, of the number of names which have been associated with the component. Thus a component's CDEFNA table entry would be changed from



to look like



Item CELCNT, 5 bits long, would be a counter of how many names have ever been associated with a particular component.

To implement this change in the CDEFNA table, the DEFINE module would have to be modified to set all of the CELCNT items equal to one when the data base is defined. The UPDATE module would then increment a component's CELCNT item by one each time the component received a new name.

The addition of the CELCNT item to CDEFNA makes possible the addition of a new UPDATE command, SYNONYMS. The invocation of the SYNONYMS command would cause the entire data base definition to be displayed with each component name followed by any other component names which have ever been associated with the component. The SYNONYMS routine would check the CELCNT of each component as it printed out the component's current name. If the CELCNT item for a component were greater than one, the CPEL item of the CELS entry for the current name would be saved. The CPEL item is a pointer to the CDEFNA entry, or component, associated with this CELS entry. The SYNONYMS routine would then start at the top of the CELS table and search for matches on the CPEL item. Each time a match is made, the component name associated with that CPEL item would be printed. By the time the routine reaches the CELS entry for the component name currently associated with the component, CELCNT synonymous names

for the component should have been printed.

An attempt to change a component's name to one which has been used previously or is currently in use could result in another problem. Application programs trying to access a non-unique name would always get the data values associated with its first occurrence in the CELS table, a clearly unsatisfactory solution. Therefore the CELS table should be used as a legality list for all the component names which have been defined under any edition of the data base. When a request is processed to place an additional name into the data base definition, the CELS table should be checked to see if the new name is a duplicate of a previous component name. If the new name is a duplicate, an error message to that effect should be printed out and the processing of that command terminated. If, however, the new name is indeed unique, the system should allow it to be placed in the CELS table.

Once an unload-reload operation has been performed on a data base, all component names which are not currently defined for the data base are lost. This means that error messages will be generated by application programs which were written under previous editions of the data base and which attempt to access components whose names have been changed. The unload-reload process involves unloading all of the data values in the data base and building new

linkages for all of these data values. Since none of the data values, or their linkages, are altered by a change in the user-assigned component name, the modification to the Remote File Management System which allows for component name changes within the UPDATE module is clearly superior to the unload-reload method of handling changing nomenclature.

B. COMPONENT NUMBERS

The second type of possible definition modification is a change in the user-assigned component number. A close examination of the data base reveals that the user-assigned component number is used in only one place, the CFIELD item of the CDEFNB table. Therefore a modification in a component's user-assigned component number could be easily accomplished by merely changing its CFIELD item.

The command to change a component's number might be:

```
NUMCHG<C7> EQ <15>;
```

where 7 is the old user-assigned component number and 15 will now be that component's user-assigned component number.

Note that a change in a component's user-assigned component number does not require a change in its system-assigned component number. The system-assigned component

number is a number calculated by the system and serves as a relative address pointer into the CDEFN tables for that component. The system-assigned component number is intentionally independent of the user-assigned component number.

An examination of RFMS shows that when processing a query containing a user-assigned component number, the retrieval process searches down through the CDEFNB table until it finds a match in the CFIELD item. The process then follows the CPVDIR item pointer to examine or retrieve the component's data values. If an application program written under a previous edition of the data base tried to access a component number whose CFIELD item had been overwritten by a new component number, the search for the old component number would fail.

To avoid problems with old application programs, it is proposed to further modify the method of handling component number changes so that the old component number is not lost when a new number is assigned to a component. Under this proposal, the existing CDEFNB entry of a component whose user-assigned component number is to be changed would first be rewritten immediately following the last entry in the CDEFNB table. Then the CFIELD item in the original CDEFNB entry space would be changed to the new user-assigned component number. An unused portion of each CDEFNB entry could be modified to hold a pointer,

CFLNUM, back to the original CDEFNB entry for this component.
Thus a component's CDEFNB table entry would be changed from

CDEFNB

CFIELD	CPVDIR	CPAD
	CPLEGL	CTYPE

to look like

CDEFNB

CFIELD	CPVDIR	CPAD
	CFLNUM	CPLEGL CTYPE

Item CFLNUM, 12 bits long, would then be a link from an old user-assigned component number back to the current user-assigned number for the component.

The decision always to leave the current copy of a component's CDEFNB entry, and parallel CDEFNA entry, in its original position in the CDEFN tables meant that a user-assigned component number could be changed without changing the system-assigned component number. This decision is significant since the system-assigned component number is used throughout the CDEFNA, CELS, CFIND, and

CDATA tables, and a change to the system-assigned component number would entail significant changes in these tables.

To implement this change in the CDEFNB table, the DEFINE module would have to be modified to set all of the CFLNUM items equal to zero when the data base is defined. The zeroth word in the CDEFNB table is reserved for use by the system and therefore could not be the relative address of a component whose user-assigned component number could change. The presence of a zero in the CFLNUM item of a component's CDEFNB entry means that the CFIELD item contains the most recent user-assigned component number associated with the component. Thus the command:

```
NUMCHG C7 EQ 15 ;
```

would result in CDEFNB table entries as shown in Figure 9.

As noted in Appendix B, the CFIELD item of word 0 of the CDEFNB table contains the last system-assigned component number, or number of CDEFN entries, and therefore can be used as a pointer to the last entry in the CDEFN tables. Since each entry in the CDEFN tables occupies two computer words, the last CDEFN entry would start at $\text{CFIELD}(\text{word } 0) \times 2$. The CDEFNB entry to be rewritten should then be moved to start at $\text{CFIELD}(\text{word } 0) \times 2 + 2$. After rewriting the CDEFNB entry, the CFIELD item of

word 0 in the CDEFNB table would then be incremented by 1 to show the total number of meaningful entries in the CDEFNB table.

This change in the CDEFNB table structure means that the CFIELD item in word 0 of the CDEFNB table no longer can be assumed to contain a count of the total number of components defined in the schema. This count is not needed, however, as will be shown with the addition of the CDNEXT item discussed in SECTION D. The CFIELD item in word 0 of the CDEFNB table merely becomes a count of the number of entries which have been made in the CDEFN tables and therefore can be used to calculate the relative address

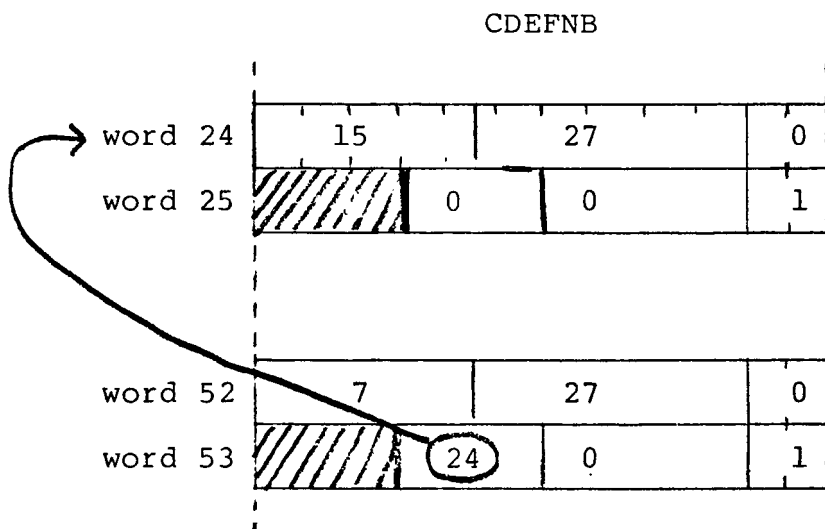


Figure 9. CDEFNB table entries after the execution of a NUMCHG command

of the last meaningful entry in the CDEFN tables.

A change in a component's user-assigned component number presents many of the same problems noted above for a change in a component's name. If a user changed a component number to one that was currently in use, application programs which tried to access that number would always retrieve those data values associated with the first occurrence of that number in the CDEFNB table. To avoid this problem, it was decided to treat user-assigned component numbers in much the same fashion as component names. Requests to place component numbers in the data base definition would be checked against the CDEFNB table for duplicates. If a duplicate component number is encountered, an error message to that effect should be printed out and processing of the command terminated. If the component number is indeed unique, the system should continue processing the command.

Using the unload-reload method of definition modification to change a user-assigned component number results in the loss of the old user-assigned number. Indeed the data base retains no reference to the previous number and this number could be reassigned. A number formerly associated with an element could be reassigned to a repeating group or vice versa. This could lead to the retrieval of meaningless data values by application programs which were

written when a component number was associated with an entirely different component type. The cost and time required to rebuild all of the data base pointers to handle a component number change is clearly not justified when compared to the RFMS modification that rewrites a CDEFNB entry and changes a CFIELD item.

C. ELEMENT TYPE

A third possible definition modification is a change to the type description of an element. As noted in Appendix B, there are six possible type descriptions for an element; NAME, TEXT, DATE, INTEGER NUMBER, DECIMAL NUMBER, and EXPONENTIAL NUMBER. If care is not exercised when changing an element's type description, problems can arise in the data base tables where the element's data values are stored.

As an example, if an element is specified as type DATE, each of its values is converted to a value equalling the number of elapsed days between October 15, 1582 and that date. This number is then stored as a floating point number in item CTEN of the data value's CVALUS entry. In order to convert the data value to the number of elapsed days since October 15, 1582, the data value must be in the form MONTH DAY YEAR and must be a valid date. Consider what would happen if an attempt were made to change an element of type

TEXT with a data value of AMEX RENTAL COMPANY to type DATE. AMEX RENTAL COMPANY is clearly not a valid date and the system would not allow the conversion. Similar problems would arise if an attempt were made to convert a data value with alphabetic characters into a data value of type NUMBER.

To avoid these problems, proposed element type changes must be checked against a legality table similar to the one in Figure 10.

NEW TYPE OLD TYPE	NAME	TEXT	INTEGER NUMBER	DECIMAL NUMBER	EXPONENTIAL NUMBER	DATE
NAME	—	YES	NO	NO	NO	NO
TEXT	YES	—	NO	NO	NO	NO
INTEGER NUMBER	YES	YES	—	YES	YES	NO
DECIMAL NUMBER	YES	YES	NO	—	YES	NO
EXPONENTIAL NUMBER	YES	YES	NO	YES	—	NO
DATE	YES	YES	YES	YES	YES	—

LEGEND

— -- no type change

YES -- type change without data validation

NO -- type change requires data validation

Figure 10. Element type change truth table

Since types NAME and TEXT can be any combination of alphanumeric or special characters, any other element type can be converted to types NAME or TEXT. An element of type NAME could be converted to type INTEGER NUMBER only if all of its data values happened to be integers. Since individually checking each data value of an element to see if it will correspond to a new type description could be very time-consuming, it was decided to disallow all type changes which involved individual data validation.

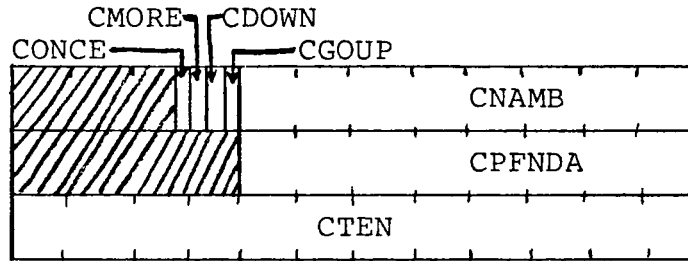
Note that the truth table in Figure 10 applies only to elements which have data values associated with them. If an element has been defined but no data values are currently associated with it in the data base (item CPVDIR in its CDEFNB table entry equals 0), then the element's type description can be changed to any of the types allowed for an element.

Once it has been determined that a proposed element type change is legal (as indicated by a YES in Figure 10), the CTYPE item in the CDEFNB table entry for the element would be changed to the new type. The full data values for the element are stored in display format in the CNAME table, but these display fields do not change if the element type is changed. So the CNAME table is not affected by a change in an element's type description.

The CTEN items in the CVALUS partitions pertaining

to the element whose type is to be changed may have to be modified to correspond to the element's new type.

CVALUS

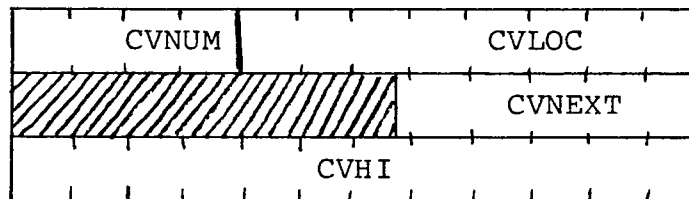


The CTEN items for an element can contain one of two representations of data values assigned to that element. If the element is defined as type NAME or TEXT, the first six display-coded characters of the data values are stored in item CTEN. If the element is defined as a number, then its display value is converted to binary and carried in item CTEN in floating point format. If the element is defined as a date, the date is converted to a value equalling the number of elapsed days between October 15, 1582 and that date and the value is then handled as if it were a type NUMBER. Therefore for changes from type NAME to TEXT or vice versa, no change would be needed in the CVALUS table. The same would also be true for the allowed changes between INTEGER NUMBER, DECIMAL NUMBER, EXPONENTIAL NUMBER, and DATE. For changes from NUMBERS or DATES to NAME or TEXT, however, a process

would have to be invoked which changed all of the NUMBERS or DATES in the CTEN items for that element back to their display values and stored the first six display characters in the CTEN item. Note that this change in the CTEN representation would not necessitate a reordering of the values within the affected CVALUS partitions, as the partitions are ordered by low value first with respect to the total value (not truncated) in table CNAME, and the CNAME table has not been changed.

The final table change needed for an element's type modification will occur in the CVHI item of the CVALDR table.

CVALDR



The CVHI item contains the last CTEN item of the CVALUS partition this CVALDR entry points to. If the representation of the CTEN item has been changed in the CVALUS table, this change should be copied into the CVHI item in the CVALDR table.

The command to change an element's type might be:

TYPCHG <C2> EQ <TEXT>;

where 2 is the user-assigned component number whose new type will now be TEXT.

A change in an element's type description will have no effect on application programs which were written under previous editions of the data base. The data values retrieved for output from the data base are always the full display values contained in the CNAME table, and as shown, these values are not altered by an element's type change.

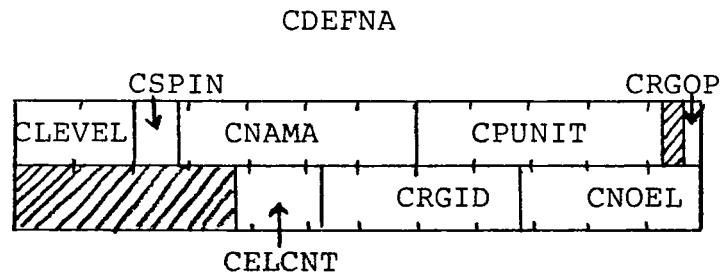
Use of this modification to RFMS does not result in any degradation in retrieval time for components whose type description has been changed. This is due to the fact that this method does not change any of the linkages within the data base, nor does it cause any data sets to be rewritten. The unload-reload operation would have to do all of the conversions for CVALUS and CVALDR as described above, as well as rebuild all of the data base links. The unload-reload method again proves to be more time-consuming and more costly than the RFMS modification.

A modification to the type description of a component which changes an element to a repeating group or a repeating group to an element will be discussed below under changes in ordering.

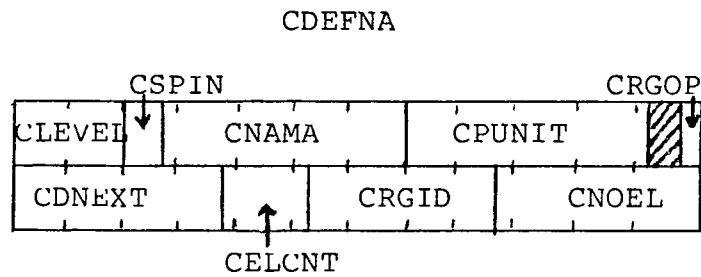
D. LOGICAL ENTRY STRUCTURE

The current version of RFMS maintains in the CDEFNA table the logical entry structure of the data base definition as ordered by the user. This is accomplished by requiring the user actually to input his components in exactly the order he wishes for the schema. Each component is then entered into the CDEFNA table in this order. The addition, deletion, or reordering of components within the data base definition, however, requires a more flexible means of maintaining the logical entry structure.

Since the CDEFNA table currently contains the logical entry structure, it is proposed that an unused portion of each component's CDEFNA entry be modified to hold a relative address pointer, CDNEXT, to the component which succeeds it in the logical entry definition. Thus a component's CDEFNA table entry would be changed from



to look like



Item CDNEXT, 14 bits long, would be a relative address pointer to the component immediately following the present component in the data base's logical entry structure.

To implement this change in the CDEFNA table, the DEFINE module would have to be modified to build the CDNEXT pointers sequentially as it is building the rest of the CDEFNA table. This would be a simple programming change, as the DEFINE module requires that the components be enumerated in the order in which they will appear in the logical entry structure. Each CDNEXT pointer would then point to the two-word CDEFNA entry below it. The DEFINE module should also place a zero in the CDNEXT item of the last CDEFNA entry to mark it as the last component in the logical entry structure.

Additional changes would have to be made in the PRINT and DISPLAY routines of the RETRIEVAL module so that the data base definition could be shown in the order implied by the CDNEXT pointers. As stated above, the present version of RFMS assumes that the components are entered into the CDEFN tables in their correct order. A request to DISPLAY the data

base definition would then only require that the DISPLAY routine step down through the CDEFN and CELS tables, adding each component to the output list as they are encountered. The addition of the CDNEXT pointers implies that the logically contiguous components may no longer be physically contiguous in the CDEFN tables. Therefore the PRINT and DISPLAY routines should be changed to use the CDNEXT pointers as their guide in building an output list.

E. ADDITION OF NEW COMPONENTS

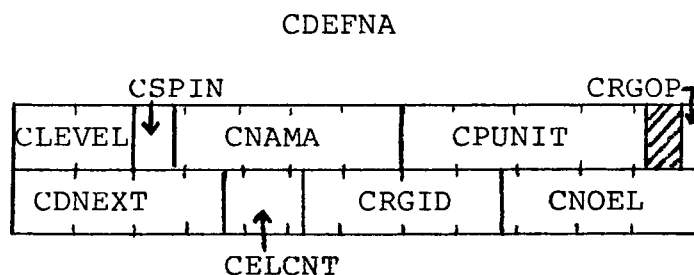
To add a new element to a valued data base, one in which data values have already been loaded, it would be necessary to specify the component which will immediately precede the new component in the logical entry structure, as well as the new element's parent repeating group. The new element's user-assigned component number, name, and type would also have to be specified. The command to add a new element might be:

```
PLACE ELEDEFN AFTER C4;  
3) 10) NEWNAME (INTEGER NUMBER);  
END ELEDEFN;
```

where 4 is the component number of the component which will now precede the new element whose user-assigned component

number is 10, whose name is NEWNAME, whose type is INTEGER NUMBER, and whose parent repeating group is the repeating group associated with the user-assigned component number 3. As noted in Chapter 2, the level 0 components have the repeating group ENTRY, with its implied component number 0, as their parent repeating group. Therefore the new command, PLACE ELEDEFN, would be well-behaved with respect to the addition of a new element to the level 0 set of components, as its parent repeating group could be specified by the number 0.

After checking the CELS and CDEFNB tables for duplications of the component name and number, the CDEFNA and CDEFNB entries could be built for the new element after the last definition in the CDEFN tables. The new CDEFN entries would then be linked to their correct position in the logical entry structure by the CDNEXT pointers.



An examination of the CDEFNA table shows that the new element gets its CLEVEL item from the CLEVEL of its parent repeating group.

$$\text{CLEVEL}(\text{NEWNAME}) \longleftarrow \text{CLEVEL}(\text{Parent Repeating Group}) + 1$$

If the parent repeating group is ENTRY (component number 0),
then

$$\text{CLEVEL}(\text{NEWNAME}) \longleftarrow 0$$


The relative address pointer to the next available space in the CELS table for storing a new component name would be retrieved from the CNAMA item in word 0 of the CDEFNA table and this pointer would become the CNAMA item for the new element. The CRGID item for the new element is the system-assigned component number of its parent repeating group and is calculated from the parent repeating group's relative position in the CDEFN tables. For example, if the parent repeating group is the seventh component in the CDEFN tables, its system-assigned component number would be 7. The CNOEL item for elements is always zero. The CDNEXT item of the new element obtains its value from the CDNEXT item of the component immediately preceding it in the logical entry structure. The preceding component's CDNEXT item then becomes a relative address pointer to the new element, the location of which is calculated from the CFIELD item of word 0 of the CDEFNB table as shown above.

CDNEXT(NEWNAME) ← CDNEXT(Preceding Component)

CDNEXT(Preceding Component) ← CFIELD(word 0) x 2 + 2

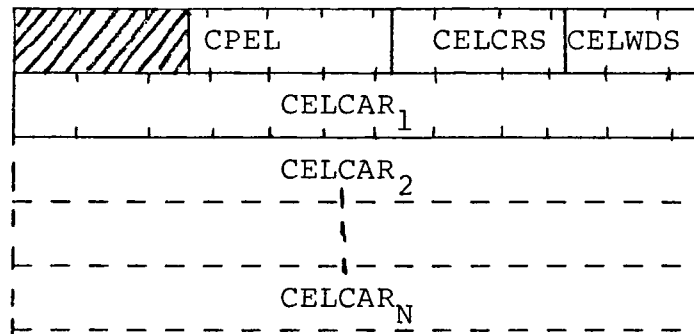
The CSPIN, CPUNIT, and CRGOP items are not implemented in this version of RFMS and are always set equal to zero.

CDEFNB

CFIELD		CPDVIR	CPAD
	CFLNUM	CPLEGL	CTYPE

For the new element's CDEFNB table entry, the user-assigned component number of the element is copied into the CFIELD item space. The component type description of the new element is converted to a number from 1 to 6 to match the RFMS legal element type description and becomes the element's CTYPE item. If the user has specified a percentage of padding for the CVALUS partitions in this data base, this percentage becomes the element's CPAD item. Otherwise, CPAD is set to zero. Since no data values have been entered yet for this element, no CVALDR table entry has been built and CPVDIR is set to zero. CPLEGL is not implemented under this version of RFMS and is always equal to zero.

CELS



In addition to building the CDEFNA and CDEFNB entries for the new element, a CELS table entry must be built. The CPEL item is the system-assigned component number of the new element which serves as a relative address to the element's CDEFN table entries. The user-assigned component name for the new element is then stored in display code in CELCAR(1) through CELCAR(N) with a count being kept in the CELCRS item of the number of characters stored and another count being kept in the CELWDS item of the number of words, N, required to store the name.

After creating the Definition Table entries for the new element, several "housekeeping" functions must be performed to allow the data base to function correctly on all subsequent accesses. The CDEFNA CNOEL item of the parent repeating group of the new element must be increased by 1 to show the correct number of elements currently associated with that repeating group. The CNAMA item of word 0 of the CDEFNA table must be

adjusted to show the next available space in the CELS table for placing a new component name. Namely,

$$\text{CNAMA}(\text{word } 0) \leftarrow \text{CNAMA}(\text{word } 0) + \text{CELWDS}(\text{NEWNAME}) + 1$$

where CELWDS(NEWNAME) contains the number of words used to store the new element's component name and the 1 represents the word taken up by the header word for the new component name in the CELS table. Likewise, the CFIELD item of word 0 in the CDEFNB table must be increased by 1 to indicate the total number of meaningful entries in the CDEFN tables.

The new element would then be defined for the data base, and its data values could be entered into the data base by subsequent calls on the UPDATE verbs ADD or ASSIGN.

The addition of a new repeating group into a data base definition parallels the addition of an element in many respects. To add a new repeating group to a valued data base, it would be necessary to specify the component which would immediately precede the new repeating group in the logical entry structure, as well as the new repeating group's parent repeating group. The new repeating group's user-assigned component number, name, and type would also have to be specified, as well as those of its descendents, if it has any. The command to add a new repeating group might be:

PLACE RGDEFN AFTER C7;

5) 11) NEWRG (REPEATING GROUP);

12) NEWELE (NAME IN 11);

13) SECELE (DATE IN 11);

14) DESCENDRG (REPEATING GROUP IN 11);

16) DESCENDELE (NAME IN 14);

END RGDEFN;

where 7 is the component number of the component which will now precede the new repeating group, NEWRG, whose user-assigned component number is 11, whose parent repeating group is the repeating group associated with the user-assigned component number 5. Note that unlike the new command PLACE ELEDEFN which can add only one element per command into the data base definition, PLACE RGDEFN can add a repeating group and any number of descendents, both elements and repeating groups, of the new repeating group. The only limitations are the system limits of 127 total components defined for the data base and that repeating groups may be nested and declared dependent only up to 64 levels. .

After checking the CDEFNB and the CELS tables for duplications in the component numbers and names, the CDEFNA, CDEFNB, and CELS entries are built for the new repeating group and its descendents exactly as described above for the addition of an element. The CNOEL item in the new repeating group's

CDEFNA table entry would be determined by the actual number of descendent elements declared for the repeating group in this command. The new repeating group and its descendents, if any, would then be defined for the data base, and their associated data values could be entered into the data base by subsequent calls on the UPDATE verbs ADD, ASSIGN, or INSERT.

Application programs written under previous editions of the data base would be affected little by the addition of new components to the definition. Since the old programs do not have the new components' names or numbers in their name list, they would have no knowledge of their existence. Note, however, that any PRINT or DISPLAY command which included the new component's parent repeating group would automatically result in the retrieval and output of the new component, and its descendents if the new component were a repeating group, along with all the other components previously associated with the new component's parent repeating group.

Whereas the unload-reload operation would build new pointers for every data value in the data base, this Remote File Management System modification would rebuild only those pointers which were affected by the addition of new components. For cases involving relatively small additions to large data bases, most of the old pointers in the data base would not be affected by the addition, and a substantial savings could be realized by not rebuilding the entire data base.

F. DELETION OF COMPONENTS

As with the addition of a new element to a data base definition, the ability to delete an element from the data base definition would require a modification to the Definition Tables as they are currently implemented by RFMS. Two approaches are considered to handling the deleted element's Definition Tables entries. The first approach involves actually removing the element's CDEFNA, CDEFNB, and CELS entries, while the second approach merely marks the entries as belonging to a deleted element.

Since the first approach implies a "packing up" of the definition, or moving the components below the deleted element up to fill the space left by the deletion, an examination of the effect of this packing up on the other tables was made. Since a component's system-assigned component number is its relative address in the CDEFN tables, all components below the deleted element would have to be given a new system-assigned component number if they were packed up. The system-assigned component number of an element is used in its CELS entry (item CPEL) and in every CDATA data set (item CPVAR) where the element has been given a data value. If the element were packed up, all of these items would have to be located and changed. Furthermore, the system-assigned component number for each component's parent repeating group

is used in the component's CDEFNA entry (item CRGID) and in the CFIND entries (item CRGNUM) which are built for each of its data sets. If a repeating group appears below the deleted element in the data base definition, its system-assigned component number would be changed during the packing up process. This, of course, means that the parent repeating group references of all of its descendents would also have to be modified. Because of the tremendous volume of system-assigned component number changes which could result from a packing up operation, the first approach has little merit over the unload-reload approach.

The second approach of merely marking an element's CDEFN entries "deleted" proved to be advantageous from several standpoints. First, the system-assigned component number changes would be avoided. Second, the user-assigned component name and number would still be in the Definition Tables and therefore would still be recognizable by application programs which were written under previous editions of the data base. The RETRIEVAL module would have to be modified slightly so that a successful search for a component name or a user-assigned component number would be followed by a check on whether or not the element was defined in the current data base edition. If the element were marked as deleted, an error message to that effect would need to be issued and the processing of the application program should be halted to guard against

the introduction or use of meaningless data.

A new item, CDEF, in each component's CDEFNB entry could be used to indicate if the component is defined for the present edition of the data base.

CDEFNB

CFIELD			CPVDIR			CPAD		
CDEF		CFLNUM		CPLEGL			CTYPE	

Item CDEF, 6 bits long, would be zero for currently defined components and one for deleted components. The DEFINE module would have to be modified to assign zeroes to the CDEF items as the data base definition is being built.

The deletion of an element from the data sets in the data base could then be accomplished easily by using a modification of the UPDATE module's REMOVE verb. The REMOVE verb removes all references to an element from all the CDATA entries, as well as all CENTS, CVALUS, and CVALDR entries relevant to the element. If the removal of an element causes a data set to become empty, REMOVE also deletes its CFIND entry. The deletion of an element's CVALUS and CDATA entries causes all pointers to that element's CNAME data values to be lost, but no attempt is made to actually remove the data values from the CNAME table.

Using the programming already available for the REMOVE verb, a new UPDATE verb, DELETE-ELE, could be formulated. A command using DELETE-ELE might be:

DELETE-ELE <C7>;

where 7 is the user-assigned component number for the element which will be deleted from the data base definition.

In addition to performing the tasks described above for the REMOVE verb, DELETE-ELE must also provide for setting the element's CDEF item to 1 and for decreasing by 1 the CNOEL item of the element's parent repeating group. One final task would be the removal of the element from the CDNEXT chain in the CDEFNA table.

CDNEXT(Preceding Component) ← CDNEXT(Deleted Element)

The "Preceding Component" would be found by searching down through the CDEFN entries, using the CDNEXT items as a pointer to the next entry to examine, until a match was made on the user-assigned component number (CFIELD item in the CDEFNB table). During this search a copy should be kept of the CDEFN relative address of the component which was accessed just before the present component. Upon a successful match on the component number, the address of the preceding

component's CDEFN entries would then still be available and its CDNEXT item could be changed.

The deletion of a repeating group from a data base definition would be accomplished similarly to the deletion of an element with one special addition. The removal of a repeating group must always imply the removal of all of its descendents, as these descendents have no connection to the data base definition except through their parent repeating group.

The UPDATE verb REMOVE TREE, or RT, operates over one or more levels of data sets and removes all references to selected data sets, and their descendent data sets, from CDATA, CENTS, CVALUS, CVALDR, and CFIND. By using the programming already available for the REMOVE TREE verb, a new UPDATE verb, DELETE-RG, could be formulated. The command using DELETE-RG might be:

DELETE-RG <C4>;

where 4 is the user-assigned component number for the repeating group which, with its descendents, will be deleted from the data base definition.

DELETE-RG must then set the CDEF items in all of the deleted components' CDEFNB entries to 1 to show that these components are not defined for this edition of the data base.

The CDNEXT item in the CDEFNA entry which logically preceeds the deleted repeating group would then be replaced by the CDNEXT item of the logically last descendent of the deleted repeating group. If the deleted repeating group had no descendents,

CDNEXT(Preceeding Component) ← CDNEXT(Deleted Repeating Group)

Application programs which were written under previous editions of the data base and which try to access deleted elements or repeating groups would not perform correctly after these components have been deleted. It was deemed so important that the user be aware that his program tried to access a deleted component that the decision was made to stop the processing of the program at that point and issue an appropriate error message.

An unload-reload operation would also result in an error message if an application program from a previous edition of the data base attempted to access a deleted component. The deleted component names are lost in the unload-reload process and are therefore available to be reassigned to a completely different component. As shown above, this could lead to the generation and use of meaningless data values by application programs which were written when the component name was associated with an entirely different component.

The cost and time required to rebuild the entire data base by the unload-reload method just to delete a small number of components is clearly not justified. The modified RFMS method can remove components easily without changing any of the links of other, logically separate components.

G. ORDER CHANGE WITHIN A REPEATING GROUP

A modification to the order in which components are defined within the logical entry structure of a data base might be the result of a desire to change the order in which components are defined within a repeating group. The logical entry structure would also change if a component was made a descendent of a repeating group other than the one it is currently associated with, or if an element were changed to a repeating group or vice versa.

The CDEFNA table, through the CDNEXT item, contains the only map of the order in which components are defined within the logical entry structure of the data base. Changing the order in which components are defined within a repeating group, or ENTRY, then becomes a relatively simple matter of changing their CDNEXT pointers. The command to move a component in the logical entry structure might be:

REORDER <COMPONENTNAME1> TO FOLLOW <COMPONENTNAME2>;

where COMPONENTNAME1 is the user-assigned component name of the component which will be logically repositioned to follow COMPONENTNAME2 in the same parent repeating group. Consider the following example:

```
    RGA
      ELEC
      ELED
      ELEF
    RGB
```

If the user desired to have ELEF as the first element defined under the repeating group RGA, then the command

```
REORDER ELEF TO FOLLOW RGA;
```

would result in

```
COPY ← CDNEXT(ELED)
CDNEXT(ELED) ← CDNEXT(ELEF)
CDNEXT(ELEF) ← CDNEXT(RGA)
CDNEXT(RGA) ← COPY
```

If the component to be moved in the logical entry structure is a repeating group, then the desired repeating group, and

all of its descendents, will be moved. For example:

```
    RGA
      ELEC
      RGD
        ELEF
        ELEG
      ELEH
    RGB
```

If the user desired to have RGD, and its descendents ELEF and ELEG, as the first components defined under the repeating group RGA, then the command:

```
REORDER RGD TO FOLLOW RGA;
```

would result in

```
COPY ← CDNEXT(ELEC)
CDNEXT(ELEC) ← CDNEXT(ELEG)
CDNEXT(ELEG) ← CDNEXT(RGA)
CDNEXT(RGA) ← COPY
```

The last descendent under a given repeating group is found by starting with the desired repeating group and chaining

down through the CDEFNA table using the CDNEXT items, the CRGID items (parent repeating group's system-assigned component number), and the CNOEL items (number of elements defined for a specific repeating group). Indeed, with the exception of using the CDNEXT item as the pointer to the next logical entry to examine, the programming necessary to find the last descendent of a repeating group is already operational in RFMS.

Since the CDATE data sets have no implied logical ordering for components within a data set, a change in the order of definition within a repeating group, or ENTRY, would have no effect on the CDATE table. This means that two different data sets for the same components may or may not have the components listed in the same order in the two CDATE data sets. Indeed, one of the data sets may contain fewer components in its CDATE data set than another data set for the same components if a component has not been assigned a data value for that data set.

Application programs written under previous editions of the data base would continue to run smoothly after a reordering within a repeating group. The only time a user need be aware that a reordering had occurred would be if the program called for a print out of the definition or of a repeating group which included the reordered repeating group as one of the descendents to be output. The definition or descendent data values would be printed out using the new positioning within

the repeating group.

The addition of the REORDER verb to the RFMS UPDATE module leads to a method of handling reordering within a repeating group which is clearly superior to the unload-reload method. The modified Remote File Management System method can reorder the components by changing the CDNEXT pointers of the components involved in the reordering, while the unload-reload method would have to rebuild all of the pointers in the data base.

H. ORDER CHANGE ACROSS REPEATING GROUPS

In order to make an element which has been assigned data values a descendent of a repeating group other than the repeating group with which it is currently associated, it would first be necessary to remove all references to the element from its current data sets. An operation similar to that of deleting an element from the data base definition would occur, with the exception that the element would not be marked as deleted in the CDEFNB table.

Data values are associated with elements in specific data sets. When an element is deleted from the definition of a data set, then all of its data values must also be removed from the data set. Furthermore, if the element will henceforth be associated with a new data set definition, the element's

data values must be added into the new data sets in a specific, user-assigned manner. Since the data base management system has no knowledge of which data values from the element's previous data sets should be associated with which of the element's new data sets, it is the user's responsibility to reassign data values to the element in the new data sets.

The CDNEXT pointers in the CDEFNA table should then be changed to reflect the change in the logical entry structure. The command to move an element to another repeating group might be:

RESTRUCTURE <COMPONENT1> TO FOLLOW <COMPONENT2> IN <COMPONENTRG>

where COMPONENT1 is the user-assigned component name of the element which will now be linked by the CDNEXT pointers to logically follow COMPONENT2 in the repeating group COMPONENTRG. Note that if COMPONENT1 is to be the first logical entry in the repeating group COMPONENTRG, then COMPONENT2 and COMPONENTRG would be the same component name.

As an example:

 RGA

 ELEC

 ELED

 RGB

 ELEF

 ELEG

If the user desired to have ELEC as the second element defined under the repeating group RGB, then the command

RESTRUCTURE ELEC TO FOLLOW ELEF IN RGB;

would result in the following changes to the CDNEXT items:

$COPY \leftarrow CDNEXT(RGA)$
 $CDNEXT(RGA) \leftarrow CDNEXT(ELEC)$
 $CDNEXT(ELEC) \leftarrow CDNEXT(ELEF)$
 $CDNEXT(ELEF) \leftarrow COPY$

Associating an element with a repeating group other than the element's present parent repeating group could result in a change in the CDEFNA CLEVEL item for that element.

$CLEVEL(COMPONENT1) \leftarrow CLEVEL(COMPONENTRG) + 1$

If the new parent repeating group, COMPONENTRG, is the repeating group ENTRY, then

$CLEVEL(COMPONENT1) \leftarrow 0$

The CDEFNA table CRGID item would be changed for the element COMPONENT1 to the system-assigned component number of

the new parent repeating group, COMPONENTRG. The CNOEL item in the parent repeating group's CDEFNA table entry would have to be incremented by 1 to reflect the addition of another element to its descendents. The CNOEL item in the old parent repeating group's CDEFNA entry would already have been decremented by the deletion operation.

The element would then be defined in the new parent repeating group, but would have no data values associated with it. Data values for the element could then be inserted into specific data sets through the use of the UPDATE verbs ADD or ASSIGN.

Moving a repeating group so that it becomes a descendent of a different parent repeating group involves the same operations as outlined above for moving an element. An important extension, however, of the process is that when moving a repeating group, all of its descendents will also move, as they have no connection to the data base definition except through the parent repeating group. An operation similar to deleting a repeating group from the data base definition would occur with all of the descendent data sets removed from the data base. Unlike the deletion operation, however, the repeating group and its descendents would not be marked as deleted in the CDEFNB table.

The same command could be used to associate a repeating group with another repeating group as was used for an element.

RESTRUCTURE $\langle \text{COMPONENTRG1} \rangle$ TO FOLLOW $\langle \text{COMPONENT2} \rangle$ IN $\langle \text{COMPONENTRG2} \rangle$;

If COMPONENTRG1 is the user-assigned component name for a repeating group, then COMPONENTRG1, and all of its descendents, would be logically moved to follow COMPONENT2 in the parent repeating group COMPONENTRG2.

As with an element, a repeating group which has been moved will obtain its CDEFNA CLEVEL item from its parent repeating group. If the CLEVEL item changes for the moved repeating group, then it must also change for all of the descendents of that repeating group.

$$\text{CLEVEL}(\text{Descendent Component}) \longleftarrow \text{CLEVEL}(\text{Parent Repeating Group}) + 1$$

The CDEFNA table CRGID item would be changed for the repeating group COMPONENTRG1 to the system-assigned component number of its new parent repeating group, COMPONENTRG2. However, none of the CRGID items for the descendents of the repeating group COMPONENTRG1 would be changed as their parent repeating group would remain the same. No change would occur in any CNOEL items, as repeating groups are not considered in a count of the number of elements associated with a repeating group.

The repeating group and its descendents would then be

defined as belonging to the new parent repeating group, but no data values would be associated with the descendent data sets. Data values for these data sets would have to be inserted into the data base and linked to their new antecedent data sets by the use of the UPDATE verbs ADD, ASSIGN, or INSERT.

Whereas an element can be logically associated with any parent repeating group in the data base definition, there are restrictions on associating repeating groups with other repeating groups. Namely a repeating group may not be logically moved so that it becomes a descendent of one of its own descendents. This would result in no defined parent repeating group for all of the original descendents of the parent repeating group, and every component in the data base definition must have a specified or implied (ENTRY) parent repeating group. For example, in the data base definition:

```
    RGA
      ELEB
      ELEC
      RGD
        ELEF
        RGG
          ELEH
    RGJ
```

The command

```
RESTRUCTURE RGD TO FOLLOW ELEH IN RGG;
```

would be illegal.

Logically moving elements and repeating groups to associate them with new parent repeating groups can have a major effect on user programs which were written under a previous edition of the data base. Since the moved component is no longer a member of its previous data sets, retrieval requests for those data sets will no longer cause the retrieval of the moved component's data values. And retrieval requests which explicitly call for the moved component in conjunction with another previously logically connected but now disjoint component can lead to indeterminate results.

Because of these inconsistencies in retrieval results over data base editions which have had modifications to the association of components to parent repeating groups, it was decided to flag components involved in a RESTRUCTURE operation. The RESTRUCTURE command would then further result in the placement of a 2 in the CDEF item of the CDEFNB entry for each component moved by RESTRUCTURE. Subsequent retrieval of data values for these components would cause a message to be issued warning of possible indeterminate results due to the RESTRUCTURE.

When using this modification of the Remote File Management System to associate an element or a repeating group and its descendents with a new parent repeating group, the number of links to be broken down for data values which will be deleted can be significant. This number may be small, however, when compared to the total number of pointers in the data base. The unload-reload method has to build all of the data base linkages, with no regard to whether or not a particular portion of the data base was involved in the alteration.

I. REPEATING GROUP - ELEMENT TYPE CHANGE

The final modification to be examined for the logical entry structure is really an extension of a component's type change as discussed in SECTION C. The modification involves changing an element's type description to that of a repeating group or a repeating group's type description to one of the types associated with an element. These types of changes can lead to major alterations in the data base such as the deletion of all the data values associated with an element or the deletion of all of the descendents, and their data values, of a repeating group. To lessen the chance that these data base modifications would occur as the result of a typing error for the command TYPCHG, it was decided to restrict the TYPCHG command to alterations dealing only with changing an

element's description to another legal type description for an element.

For changing an element's type description to that of a repeating group or a repeating group's type description to one of the allowable descriptions for an element, new commands are proposed. The command to change an element to a repeating group might be:

ELERGCHG <C4> EG <REPEATING GROUP>;

where 4 is the user-assigned component number for the element whose new type will now be REPEATING GROUP.

Since a repeating group has no data values associated with it, changing a component type to REPEATING GROUP would involve deleting the data values currently associated with the element. This could be easily accomplished using the programming available for the UPDATE verb REMOVE. The REMOVE verb can be used to remove all references to that element from all of the CDATA entries, as well as all CENTS, CVALUS, and CVALDR entries relevant to the element. If the removal of the element causes a data set to become empty, REMOVE also deletes its CFIND entry.

The CPVDIR and CTYPE items in the CDEFNB entry for the element would then be set to zero to indicate that the element is now a repeating group. The CNOEL item in the CDEFNA entry

of its parent repeating group would be decremented by 1 to show the loss of an element from its descendents. The CNOEL item of the component whose type was changed would remain zero to signify that the new repeating group has no descendents. Descendents could then be associated with the new repeating group by the use of the UPDATE commands PLACE ELEDEFN, PLACE RGDEFN, or RESTRUCTURE.

To change a repeating group to an element, the command might be:

```
RGELECHG <C10> EQ <INTEGER NUMBER>;
```

where 10 is the user-assigned component number of the repeating group which will now become an element of type INTEGER NUMBER.

The first operation to be performed when changing a repeating group to an element would be to delete all of the repeating group's descendents from the data base. This would occur in the same manner as described above for the new UPDATE command DELETE-RG, with the exception that the parent repeating group, in this example component number 10, would not be marked as deleted in the CDEFNB item CDEF. For example

```
1  RGA
   2  ELEB
   3  ELEC
```

```
      4    RGD
          5    ELEF
          6    ELEG
7    RGH
```

The command

```
RGELECHG C4 EQ NAME;
```

would result in

```
CDNEXT(RGD) ← CDNEXT(ELEG)
```

and ELEF and ELEG would be deleted from the data base definition.

The CTYPE item in the repeating group would then be changed to reflect the component's new element type and the CNOEL item of its parent repeating group, RGA, would be incremented by 1 to show the addition of another element to its descendents. The CNOEL item of the new element would be changed to 0, as elements have no descendent elements.

The former repeating group would then be defined as an element in the data base and data values could be associated with it through the use of the UPDATE verbs ADD or ASSIGN.

Changing an element to a repeating group or a repeating group to an element may cause errors in application programs

which were written under previous editions of the data base. If a program tries to retrieve a value for a repeating group (formerly an element), an error message will be printed out since repeating groups have no data values associated with them. And user programs which try to access components which were deleted as the result of being a descendent of a repeating group-turned-element will receive the fatal error message that an attempt was made to access a deleted component.

Although the unload-reload approach for changing an element would also result in the deletion of a former element's data values or a former repeating group's descendents, the savings realized in using the modification described above for RFMS would come as a result of not having to rebuild the entire data base. The investment in building the original data base structure is lost whenever an unload-reload operation is undertaken, as this method can be as costly as the first formulation of the data base.

J. CONCLUSIONS

As shown, the Remote File Management System data base is organized in such a way as to allow for component modifications with a minimum of data reorganization. This is primarily due to the fact that each data value is implicitly associated with its particular data set and data set selection

in the RETRIEVAL module is determined by these associated data values. RFMS provides an independent access path to all of the data values in the data base.

An examination of the preceeding modifications makes it clear that the purpose of providing for data base definition changes is to allow the definition to evolve over its period of use. The data base is modified at the time the definition revision is processed so that all subsequent accesses to the data base will occur under the revised edition.

Experience with evolving data bases has shown that the addition of elements and repeating groups to an existing data base definition is much more frequent than the deletion or reordering of components.²⁴ The proposed modifications to the Remote File Management System would easily handle these additions by placing expanding data sets at the end of the CDATE table. The housekeeping chores for linking the new components to their proper positions in the logical entry structure are minimal and are readily accomplished by the commands PLACE ELEDEFN and PLACE RGDEFN.

During these discussions, no value has been assigned to the storage space which would continue to be occupied by data values belonging to deleted elements. Indeed, the decision to do no garbage collection within the data base was the result of an attempt by the data base administrators and myself to assign relative costs to computer time spent in

garbage collection and to storage space. It was decided that the potentially small storage space to be regained by garbage collection did not justify the computer time which would be spent building and maintaining the garbage collection lists for the various tables.

Periodic unloading and reloading of the data base to bring all of the logically contiguous data sets into physical proximity may be desirable for the sake of efficiency. The decision of when to undertake an unload-reload operation based on a cost-benefit analysis is discussed in detail by Shneiderman.²¹

In the above discussions of the data base definition modifications which might be encountered, comparisons were made between the Remote File Management System using the unload-reload method of definition revision and the Remote File Management System modified to handle definition revisions in the UPDATE module. In all cases the modified RFMS operations were demonstrated to be superior to the only other available method of data base definition modification, the unload-reload method.

SELECTED BIBLIOGRAPHY

SELECTED BIBLIOGRAPHY

1. Abrial, J. R. "Data Semantics," Data Base Management. eds. J. W. Klimbie and K. L. Koffeman. North-Holland Publishing Company, 1974.
2. Bleier, Robert E. and Alfred H. Vorhaus. "File Organization in the SDC Time-Shared Data Management System (TSMS)," Proceedings IFIP Congress 1968, 2:1245-1252, 1969.
3. Cardenas, Alfonso F. Data Base Management Systems. Boston, Massachusetts: Allyn and Bacon, Inc., 1978.
4. CODASYL Systems Committee. "A Survey of Generalized Data Base Management Systems." A CODASYL Systems Committee Technical Report. Association for Computing Machines, May 1969.
5. _____. "Introduction to 'Feature Analysis of Generalized Data Base Management Systems'," Communications of the ACM, 14(5): 308-318, May 1971.
6. Durchholz, R. and G. Richter. "Concepts for Data Base Management Systems," Data Base Management. eds. J. W. Klimbie and K. L. Koffeman. North-Holland Publishing Company, 1974.
7. Everest, Gordon C. "Concurrent Update Control and Data Base Integrity," Data Base Management. eds. J. W. Klimbie and K. L. Koffeman. North-Holland Publishing Company, 1974.
8. _____. "Data Base Management Systems Tutorial," Fifth Annual Midwest AIDS Conference Proceedings, Vol. 1. Minneapolis, Minnesota, May 1974.
9. _____. "The Objectives of Data Base Management," Information Systems. Proceedings of Fourth International Symposium on Computer and Information Science (COINS-72), Miami Beach, Florida, 1972.

10. Everett, Gerald D., C. William Dissly, and W. Terry Hardgrave. Remote File Management System User's Manual. The University of Texas at Austin, Computation Center, Austin, Texas. TRM-16. August 1971.
11. Everett, Gerald D. Systems Programming Documentation, Control-IDS, Internal Design Specifications of the Remote File Management System, Phase 1. The University of Texas at Austin, Computation Center, Austin, Texas. TSD-12. July 1970.
12. Fry, James P. "Data Base Management: An Overview," Generalized Data Base Management Systems. An Intensive Short Course, College of Engineering, University of Michigan. June 1974.
13. Hall, M. and A. M. Feinstein. "Data Base Management Systems," Chemical Engineering Progress, October 1977: 79-82.
14. Kindred, Alton R. Data Systems and Management--An Introduction to Systems Analysis and Design. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.
15. Knuth, Donald E. The Art of Computer Programming 1, Fundamental Algorithms. Reading, Massachusetts: Addison-Wesley, 1968.
16. Lucking, J. R. "A Descriptive Methodology Suitable for Multiple Views of an Information Processing System," Data Base Management. eds. J. W. Klimbie and K. L. Koffeman. North-Holland Publishing Company, 1974.
17. Martin, James. Computer Data Base Organization. Englewood Cliffs, New Jersey: Prentice-Hall, 1977.
18. Salton, Gerald. Automatic Information Organization and Retrieval. New York: McGraw-Hill, 1968.
19. Schenk, Hans. "Implementational Aspects of the CODASYL DBTG Proposal," Data Base Management. eds. J. W. Klimbie and K. L. Koffeman. North-Holland Publishing Company, 1974.

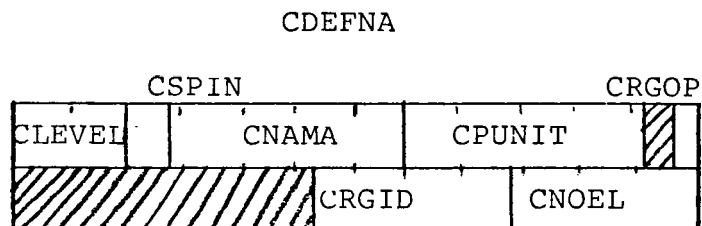
20. Senko, M. E., E. B. Altman, M. M. Astrahan, and P. L. Fehder. "Data Structures and Accessing in Data Base Systems," IBM Systems Journal, 12:30, 1973.
21. Shneiderman, Ben. "Optimum Data Base Reorganization Points," Communications of the ACM, 16(6):362-365, 1973.
22. Sundgren, Bo. "Conceptual Foundation of the Infological Approach to Data Bases," Data Base Management. eds. J. W. Klimbie and K. L. Koffeman. North-Holland Publishing Company, 1974.
23. System 2000. System 2000 Newsletter. System 2000, Version 2.40, CDC CYBER/6000 Series Computers. MRI Systems Corporation, Austin, Texas, October 1975.
24. Taylor, Robert W. and David W. Stemple. "On the Development of Data Base Editions," Data Base Management. eds. J. W. Klimbie and K. L. Koffeman. North-Holland Publishing Company, 1974.
25. Tsichritzis, Dionysios C. and Frederick H. Lochovsky. Data Base Management Systems. New York: Academic Press, 1977.
26. Tymshare. MAGNUM. Cupertino, California: Tymshare, Inc., 1976.

APPENDIX A

CDEFNA TABLE

APPENDIX A

CDEFNA TABLE



(a) CLEVEL item (6 bits long) --

CLEVEL specifies the level of the component (0 is the highest logical level). Level is implied in the logical entry declaration.

(b) CSPIN item (2 bits long) --

This item was never implemented and is always equal to zero.

(c) CNAMA item (14 bits long) --

CNAMA contains the rightmost 14 bits of a relative address pointer to the header word of the CELS table entry which contains the component name for this component. Under this version of RFMS, a maximum of 150 characters is allowed for a user-defined component name, and with this maximum, the rightmost 14 bits of the relative address is sufficient to access any entry in the CELS table.

(d) CPUNIT item (12 bits long) --

This item has never been implemented and is always equal to zero.

(e) CRGOP item (1 bit long) --

This item is not implemented in the Univac RFMS version and is always equal to zero.

(f) CRGID item (10 bits long) --

CRGID specifies the system-assigned component number of the parent repeating group component to which this component (either an element or a lower level repeating group) belongs. All level 0 components (elements and repeating groups) belong to the implied repeating group 0.

(g) CNOEL item (10 bits long) --

This item is a counter stored in a repeating group component's CDEFNA entry which indicates the number of elements belonging to that repeating group. For elements, CNOEL = 0 and is not meaningful. The count of level 0 elements is stored in CNOEL of word 1 and behaves as any other CNOEL repeating group item.

Values of items in words 0 and 1 of CDEFNA implicitly associated with the level 0 set of a logical entry are:

- (a) CLEVEL item -- equals 0 and is meaningful.
- (b) CSPIN item -- equals 0 and is not meaningful in current implementation.
- (c) CNAMA item -- a pointer to the next available address in the CELS table.
- (d) CPUNIT item -- holds the count of the components in the definition.
- (e) CRGOP item -- equals 0 and is not meaningful in current system.
- (f) CRGID item -- equals 0 and is meaningful.
- (g) CNOEL item -- count of level 0 elements.

APPENDIX B

CDEFNB TABLE

APPENDIX B

CDEFNB TABLE

CDEFNB

CFIELD	CPVDIR	CPAD
	CPLEGL	CYYPE

- (a) CFIELD item (14 bits long) --

The CFIELD item represents the user-defined component number for the component at the time that the logical entry declaration was finalized.

- (b) CPVDIR item (16 bits long) --

This is the rightmost 16 bits of a relative address pointer to the first word of the first CVALDR entry for an element. For repeating groups, CPVDIR = 0 and is not applicable. If no data values have been stored for an element, CPVDIR = 0.

- (c) CPAD item (6 bits long) --

CPAD specifies the user-defined percentage padding of the CVALUS table partitions for each element defined. For a repeating group component, CPAD = 0 and is not applicable.

(d) CPLEGL item (12 bits long) --

This item has never been implemented and this field always equals zero.

(e) CTYPE item (6 bits long) --

CTYPE contains a code which specifies the type of the component as declared by the user in the type description for the component.

CTYPE = 0 for REPEATING GROUP

1 for NAME

2 for TEXT

3 for DATE

4 for INTEGER NUMBER

5 for DECIMAL NUMBER

6 for EXPONENTIAL NUMBER

Words 0 and 1 of the CDEFNB table are reserved and used by the system in a manner similar to the use described in the CDEFNA table. Contents of items in words 0 and 1 of the CDEFNB table are:

(a) CFIELD item -- used by the system to contain the number of components defined in the total definition. This number is the last system-assigned component number and is the last meaningful relative address pointer to the CDEFN tables.

- (b) CPVDIR item -- is used by the system to contain the maximum (deepest) level used in the logical entry definition.
- (c) CPAD item -- used by the system to keep CENTS table padding percent as given by the user.
- (d) CPLEGL item -- equals 0 and is not meaningful in the current implementation.
- (e) CTYPE item -- equals 0 to imply that the ENTRY level component is a repeating group.

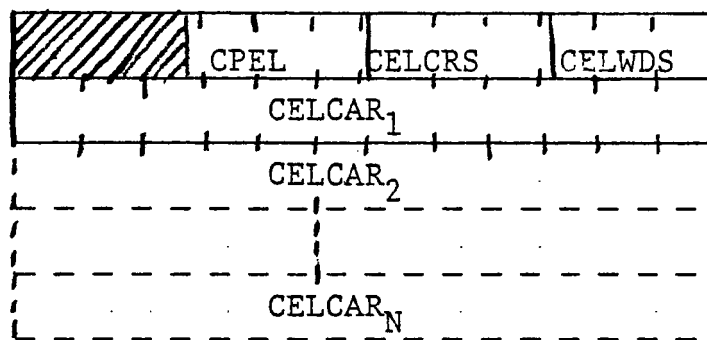
APPENDIX C

CELS TABLE

APPENDIX C

CELS TABLE

CELS



(a) CPEL item (12 bits long) --

CPEL contains the system-assigned component number which is the relative address pointer (or CDEFN entry number) to the CDEFNA and CDEFNB table entries associated with this component name.

(b) CELCRS item (8 bits long) --

This item contains the number of characters in the component name as declared by the user excluding leading, extraneous (more than one) embedded, and trailing blanks. System-supplied trailing blanks in the last word, CELCAR_N, are not included in the count for the CELCRS item.

- (c) CELWDS item (8 bits long) --

CELWDS contains the number of CELCAR words associated with this entry. The word count does not include the header word (first word) of each CELS entry; it is equal to N.

- (d) CELCAR item (36 bits long) --

CELCAR is a string of display coded characters declared by the user as the component name. The character string begins at the left of word $CELCAR_1$ and the last word $CELCAR_N$ is filled with trailing blanks if there are less than 6 characters remaining. CELCAR words do not contain any leading, extraneous (more than one) embedded, or trailing blanks.

Words 0 and 1 of the CELS table are reserved and used by the system.

- (a) CPEL item -- contains the system-assigned component number of ENTRY, namely 0.
- (b) CELCRS item -- contains number of characters in ENTRY, namely 5.
- (c) CELWDS item -- contains number of CELCAR words needed to store ENTRY, namely 1.
- (d) CELCAR item -- contains the display code for ENTRY followed by one blank.

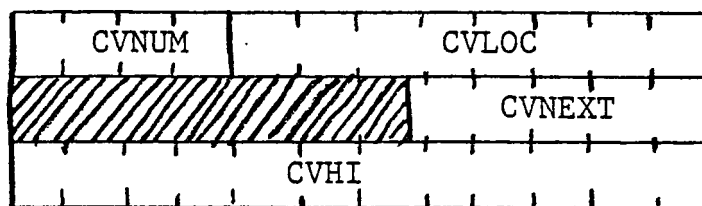
APPENDIX D

CVALDR TABLE

APPENDIX D

CVALDR TABLE

CVALDR



(a) CVNUM item (12 bits long) --

CVNUM contains the number of words containing values in the relevant CVALUS partition for the element.

(b) CVLOC item (24 bits long) --

This item contains the relative address pointer to the first word of the relevant CVALUS partition.

(c) CVNEXT item (16 bits long) --

CVNEXT contains a relative address pointer to the next CVALDR entry for the same element. Whenever a CVALUS partition for the same element exists after the CVALUS partition associated with this CVALDR entry, another CVALDR entry exists for the element in the CVALDR table. New CVALDR entries

are added to the end of table CVALDR and linked by CDNEXT. For the last CVALDR entry in the set of CVALDR entries for an element, item CVNEXT = 0.

(d) CVHI item (36 bits long) --

CVHI contains item CTEN of the last entry of the corresponding CVALUS partition pointed to by CVLOC in this CVALDR entry. CVHI item contains, then, the representation of the highest value in the relevant partition.

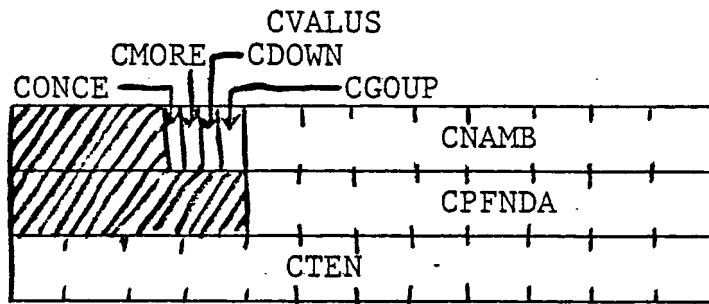
Words 0, 1, and 2 of the CVALDR table are reserved and used by the system.

APPENDIX E

CVALUS TABLE

APPENDIX E

CVALUS TABLE



(a) CONCE item (1 bit long) --

CONCE is set to 1 if the corresponding unique data value occurred only once for a given element; otherwise CONCE = 0.

(b) CMORE item (1 bit long) --

CMORE contains a 0 if the number of characters in the value string is less than or equal to 6; otherwise it is 1 for type NAME or TEXT. For numeric or date types, CMORE item is 0 since the floating point value stored in item CTEN below is complete for comparing regardless of the number of input characters for the data value.

(c) CDOWN item (1 bit long) --

CDOWN contains a 1 if there is a duplicate CTEN item in the CVALUS table entry immediately

below this one within the same partition.

Otherwise CDOWN = 0.

(d) CGOUP item (1 bit long) --

CGOUP contains a 1 if there is a duplicate CTEN item in the CVALUS table entry immediately above this one within the same partition. Otherwise CGOUP = 0.

(e) CNAMB item (24 bits long) --

CNAMB contains a relative address pointer to word 1 of the CNAME table entry which in turn contains the entire value string in display code associated with the representation in this CVALUS entry.

(f) CPFNDA item (24 bits long) --

This item contains a relative address pointer to word 1 of the CENTS table entry for this unique value if CONCE = 0, i.e., if this value occurred more than once for this element. If CONCE = 1, then the CPFNDA item contains the relative address pointer to the single CFIND table entry in which data set this data value occurred.

(g) CTEN item (36 bits long) --

CTEN contains two kinds of representations of data values which have been assigned to an

element. If an element is defined as type NAME or TEXT, then the first six characters of the data value are stored in item CTEN. If the element is defined as a number, then its display value is converted to binary and carried in CTEN in floating point format. If the element is defined as a date, then the date data value is converted to a value equalling the number of elapsed days between October 15, 1582, and that date value, and then handled as if it were a type NUMBER, i.e., number of days elapsed is stored as a floating point number in item CTEN.

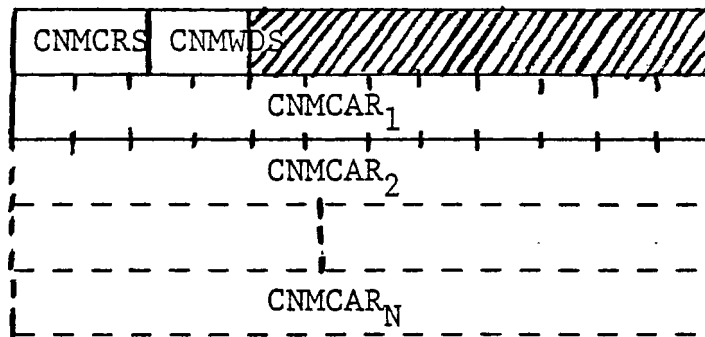
APPENDIX F

CNAME TABLE

APPENDIX F

CNAME TABLE

CNAME



(a) CNMCRS item (8 bits long) --

CNMCRS contains the number of characters in the value string found in this entry's CNMCAR.

(b) CNMWDS item (5 bits long) --

CNMWDS contains the number of words needed by CNMCAR to hold the complete value string; in the above illustration, CNMWDS = N.

(c) CNMCAR item (36 bits long) --

This item contains the string of display coded characters that comprise the unique data value string.

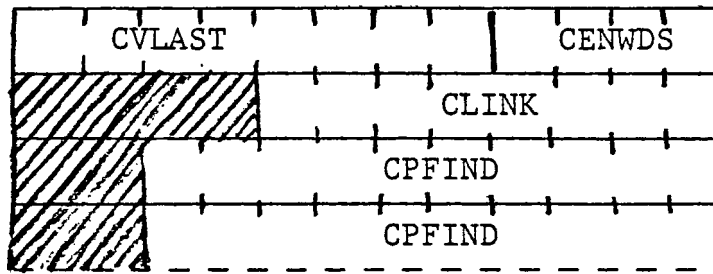
APPENDIX G

CENTS TABLE

APPENDIX G

CENTS TABLE

CENTS



(a) CVLAST item (24 bits long) --

CVLAST contains the relative address pointer to the last CPFIND item in the entire CENTS table for this unique data value. CVLAST item is stored only in the first block in the series of blocks for a given element value.

(b) CENWDS item (12 bits long) --

CENWDS contains the number of CPFIND words in this block.

(c) CLINK item (24 bits long) --

CLINK contains a relative address pointer to the next CENTS block for this unique value if

another block exists. If this block is the only block or the last block, CLINK = 0.

(d) CPFIND item (30 bits long) --

This item contains a relative address pointer to an entry in the CFIND table which, in turn, identifies and associates this occurrence of the unique value for the element with a specific data set in the database structure.

APPENDIX H

CFIND TABLE

to the next level 0 set and thus preserves the horizontal chain of logical entry entrance into the database. CPRGHT of word 0 (the zeroth entry) points to the first level 0 data set in the database.

(c) CRGNUM item (10 bits long) --

CRGNUM contains the system-assigned repeating group component number to which the data set belongs. CRGNUM equals 0 for all level 0 data sets, and is well behaved in that the level 0 elements are treated implicitly as repeating group 0.

(d) CPLOC item (24 bits long) --

CPLOC contains the relative address pointer to the beginning of the CDATA table entry for this CFIND entry if any actual data values have been entered for this data set. If this is a dummy CFIND entry, then no actual values exist for the set, no CDATA entry exists, and CPLOC = 0.

(e) CPUP item (24 bits long) --

This item contains a relative address pointer. For any level N set where N is greater than zero, CPUP points to the CFIND entry for the set at level N-1 with which the level N set is associated as a member of a repeating group. For a level

zero set, item CPUP points to the previous level zero set in the database or equals zero if there is none. CPUP of word 2 (the zeroth entry) of the CFIND table points to the last level 0 CFIND entry (the beginning of the last logical entry in the database).

(f) CPDOWN item (24 bits long) --

CPDOWN contains a relative address pointer also. For any level N data set, item CPDOWN points to the CFIND entry for the first level N+1 data set associated with that level N data set. If none exists, CPDOWN = 0. CPDOWN of word 3 (the zeroth entry) of the CFIND table contains the next available address (relative) for storing a CFIND entry.

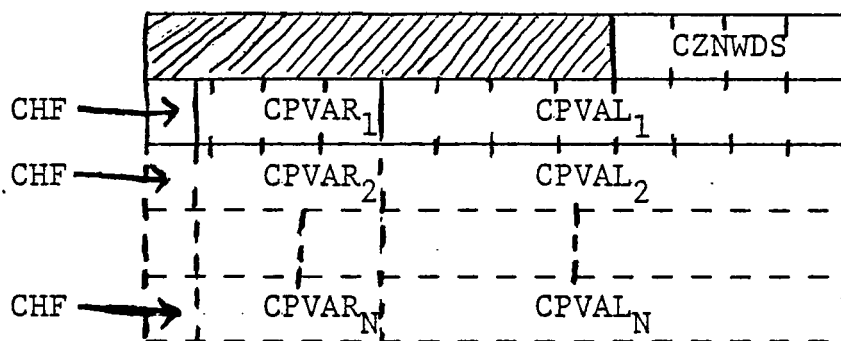
APPENDIX I

CDATA TABLE

APPENDIX I

CDATA TABLE

CDATA



(a) CZNWDS item (12 bits long) --

CZNWDS contains the number of CPVAL words associated with this entry. The word count does not include the header word and is equal to N.

(b) CHF item (2 bits long) --

If the word is the second word for the data set block (the header word being word 1), CHF is set to 2. The CHF items for all subsequent words in this data set block are set to 0.

(c) CPVAR item (10 bits long) --

CPVAR contains the system-assigned component number, i.e., the relative address pointer to the CDEFNA and CDEFNB table entries for the element in this data set which has the value pointed to by the CPVAL item associated with this CPVAR item.

(d) CPVAL item (24 bits long) --

This item contains the relative address pointer to the CNAME table entry which in turn contains the actual data value associated with the element identified in the CPVAR item above.