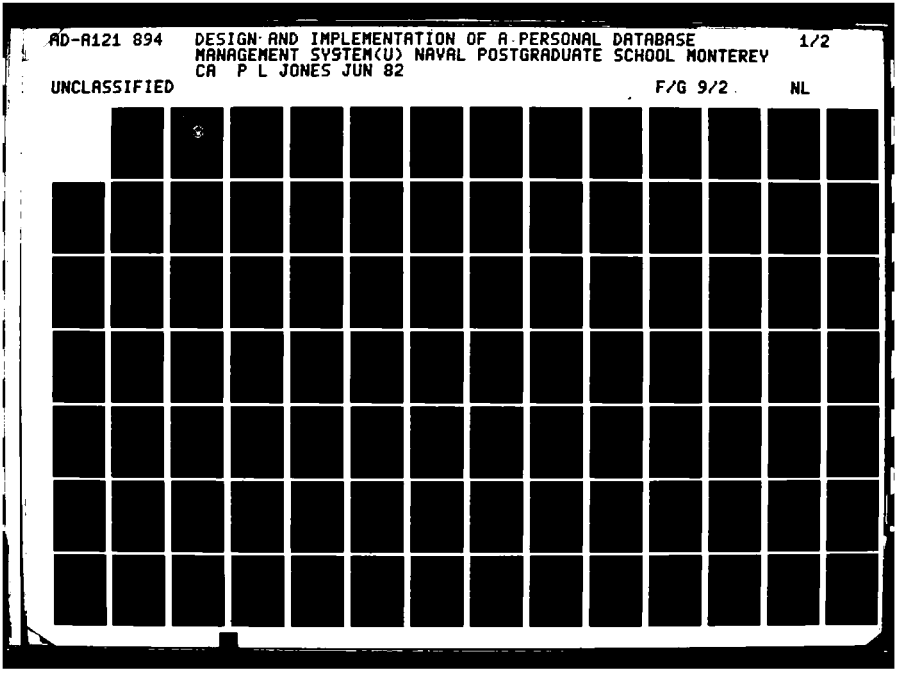
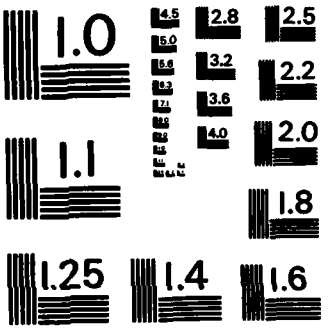
AD-A121 894 DESIGN AND IMPLEMENTATION OF A PERSONAL DATABASE MANAGEMENT SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P L JONES JUN 82 1/2

UNCLASSIFIED F/G 9/2 NL

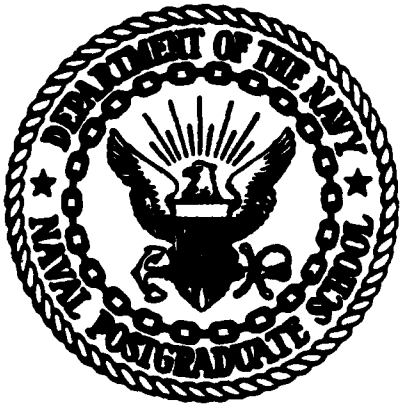




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A121894

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

DESIGN AND IMPLEMENTATION
OF A
PERSONAL DATABASE MANAGEMENT SYSTEM

by

Peter L. Jones

June 1982

Thesis Advisor: Dushan Z. Badal

Approved for public release; distribution unlimited.

DTIC FILE COPY

DEC 30 1982

A

82 11 30 071

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A121844	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Design and Implementation of a Personal Database Management System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June, 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Peter L. Jones		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June, 1982
		13. NUMBER OF PAGES 116
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

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

Microcomputer, Hand-held computer, Database Management System, Non-volatile Memory, FORTH.

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

The Personal Database Management System is a hardware and software system designed to support people's memory and recall processes. It is a small, low power, and inexpensive microcomputer system which employs EEPROM and CMOS technology. The design is based upon how people manage their personal information, which was found to be different from the ways conventional computerized systems manage information.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Approved for public release; distribution unlimited.

Design and Implementation of a Personal Database Management System

by

Peter L. Jones
Captain, United States Marine Corps
B.S., University of Washington, 1975

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1982

DTIC COPY INSPECTED 2

Author: _____

Approved by: _____
Thesis Advisor

_____
Second Reader

_____
Chairman, Department of Computer Science

_____
Dean of Information and Policy Sciences

ABSTRACT

The Personal Database Management System is a hardware and software system designed to support people's memory and recall processes. It is a small, low power, and inexpensive microcomputer system which employs E^2PROM and CMOS technology. The design is based upon how people manage their personal information, which was found to be different from the ways conventional computerized systems manage information.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

DISCLAIMER

Some terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis are listed below following the firm or individual holding the trademark.

Zilog, Incorporated, Cupertino, California:
Z80

FORTH, Incorporated, Hermosa Beach, California:
FORTH

Digital Research, Pacific Grove, California:
CP/M

National Semiconductor, Santa Clara, California:
NSC800

XICOR, Incorporated, Milpitas, California:
NOVRAM Non-volatile Static RAM

Greenwich Instruments Limited, Greenwich, London, UK:
Instant ROM

ACKNOWLEDGMENTS

The author would like to thank the following people without whose help much of this thesis would not have been possible.

Mr. Walter L. Landaker, of the Department of Computer Science Laboratory, who helped in the hardware implementation by doing much of the bread-boarding and managed to get the LCD (which had been received without the promised manual and hardware interface) operational.

Mr. Michael A. Williams, also of the Laboratory, who helped, not only in the hardware implementation, but also in the hardware design. It was he who proposed interleaving the E^2PROM which decreased the average write-time by 400 percent. He also designed the "smart" ports.

Ms. Kathy Yamamaka, of the Department of Computer Science, who went out of her way to help by "greasing the skids" in the Supply Department to ensure that the materials required by this research were received in a timely fashion.

I. INTRODUCTION

One of the factors which limits human performance is the limited capacity of human memory. Memory is commonly considered to be divided into two parts: short-term and long-term. Short-term memory is that part which we can consciously access; it may be compared to the primary store of a computer. It is characterized by rapid access and volatility. Long-term memory is analogous to secondary storage in that it is more permanent in nature than short-term memory and it requires more time and effort to record information to and retrieve information from [1].

Short-term memory is a major limiting factor on human performance because it is the memory which is consciously accessible and thus our working memory, and it is very limited in its capacity. This memory holds units of information for up to thirty seconds. That period may be extended through repetition and rehearsal. The size of short-term memory is approximately seven units of information (plus or minus two). The nature of these units is a function of experience and training. For example, someone familiar with English may find it easy to remember seven English words but difficult to remember seven Chinese ideograms. Thus it is easy to see that the information processing capacity of humans can be easily overloaded. Long term memory limits performance because of the time and effort associated with fetches from and stores to it [1].

The idea behind a Personal Database Management System (PDBMS) is to provide an extension to both short-term memory and long-term memory. A good PDBMS should provide its users with means of storing information and later retrieving it that are faster and more efficient than ordinary human

means. Long-term memory can be extended by allowing users to easily store information which they find difficult to memorize. Numerical information such as phone numbers, safe combinations, and part numbers are examples of information which are usually expensive in the amount of effort required to ensure that they are not soon forgotten. Short-term memory can be extended by providing users with a way to relieve the burden upon its capacity. Instead of having to remember a piece of information or a key (or cue) to retrieving the desired information, a PDBMS can accept the key as input and retrieve the desired information. Once the key has been entered into the system, it may be forgotten, freeing a portion of short-term memory for more information. Also, retrieved information need not be memorized if the PDBMS records it in a manner which allows it to be easily accessed. For example, information recorded on a piece of paper or on a display screen need not be memorized if it is within easy reach.

What should be the characteristics and what are the requirements of a Personal Database Management System? Because it is designed for the storage and retrieval of <u>personal</u> information, it is a single-user system. In order to be useful to a broad range of people, it should permit interaction at different levels, depending on the sophistication of the user. Novice users will be easily discouraged and see very little benefit if a system appears to be illogical and complicated. Also, because of the personal nature of the information in the database, the system should provide security to that information. Finally, in order to be acceptable, it should be small, light-weight, and inexpensive.

This last requirement was taken to indicate that such a system should be built using a battery-driven microprocessor. Current microprocessor technology provides more

computer power than is needed strictly for a PDBMS. So the design presented here incorporates the following additional capabilities: 1) the ability to be used as a calculator, 2) the ability to be programmed by the user, and 3) the ability to be connected into networks or to other devices via an RS232 serial interface.

The PDBMS is programmed in a non-standard version of FORTH. The particular one used here is neither fig-FORTH nor FORTH-79, the two most prevalent versions of FORTH. However, the basis for the language used is 8080 fig-FORTH, version 1.3, which was partially modified to conform with the FORTH-79 standards [2]. Further modifications were made to this based upon hardware characteristics, and the suggestions and ideas of various members of FORTH Interest Group. In spite of this, when referred to in this thesis, the language used in the PDBMS will be called FORTH. One major distinction should be made, however, the PDBMS's base vocabulary is called ROOT, not FORTH.

II. PERSONAL DATABASE CHARACTERISTICS

A. BACKGROUND

The largest part of the information presented in this chapter was derived from detailed study of four personal address books (Appendix B contains detailed statistics from this study). Address books were used as a basis for the preliminary investigation of personal databases because they were found to be more structured, standardized, and easily computerized than other personal databases (e.g., shopping lists, appointment calendars, and things-to-do lists).

The people (some of whom worked with computers daily) interviewed during the study indicated that the maintenance of personal databases is not analogous to management of databases by computer. Indeed, the ways in which a database management system (DBMS) is structured, maintained, and used is very different from the way people manage their personal information. The results of the author's studies and interviews seem to indicate that the essential difference between DBMSs and personal information management is the number of "system" users. It is this difference that is the apparent cause of most all of the other differences.

Because DBMSs are normally organizational tools with many users, records, fields, attribute values, query languages, keys, etc., they must be standardized. Because organizational data is entered and retrieved by many different individuals and thus without standardization, it would be difficult for one person to know of information entered into the system by another, much less retrieve it. On the other hand, personal information is shared by only a few people, if any. An important point here is that in such

a situation where there is only one user, that user knows (or knew at one time) all of the information in the system because he entered it. People record and maintain personal information in an auxiliary store in order to relieve themselves of some of the burdens of recall and recognition. Because long-term memory is generally considered to be permanent [1], the data recorded in auxiliary stores need not be a verbatim copy of the information which is to be retrieved later. Truly personal information needs only to contain enough context-specific cues to enable a person to reconstruct or recall the structure of their semantic memory.

"The Recognition of Previous Encounters," by George Mandler [3] describes semantic structures as an organization of memory (referred to as a "familiarity variable"). These structures represent the familiarity of events (and of the entities which are part of an event), and are unique to each particular event. Further, they are independent of the context in which the event occurs or in which it is embedded. Two sets of independent processes operate upon semantic structures: intra-event processes which are referred to as "integration," and inter-event processes which relate an event to others called "elaboration." Mandler's hypothesis is that recognition is related to integration, which is developed through attentive repetition (rote learning). Recall is related to elaboration, which is strengthened by the establishment of relational links between the target event and other representations in memory[1]. Mandler does not describe how integration and

---

[1]Recognition is the process of going from a familiar event to the context which caused the event to be remembered. Recall is the opposite process, that is, remembering an event from its context. When a person attempts to remember where he knows a familiar face from, he is employing recognition. Recall is what a person attempts to do when he knows his wife told him to get something on the way home, but has forgotten what.

elaboration manifest themselves except in an abstract way. They must involve the establishment of cues which act as keys to semantic structures whether they might be direct (as one would expect in the case of integration) or indirect (as might be the case for elaboration) access. It is these cues which must be available to a person in order to retrieve the desired events and entities. It is this that makes personal databases different from DBMSs.

Even though only the minimum number of cues need be saved in order to retrieve information, the author's studies revealed that usually more than the minimum required cues are recorded. For example, there is usually no need to record one's parents' city and state of residence, yet every address book contained this, as well as other unnecessary information. This is probably due in part to the fact that address books are not always personal databases, sometimes they are family documents. Appointment calendars appeared to be the tersest of all the personal databases studied. An example entry for March 10 might be, "Rebecca 11:30" which is a reminder that Rebecca has an appointment with Dr. Feeney at the Pediatric Group, 698 Cass Street, 11:30 A.M., on March 10th.

In order to establish a common ground for comparison, the following terms will be used throughout this thesis.

- Personal Database Management System (PDBMS): a computer based system for managing personal information. The information managed by this system is organized into files containing records.

- Manual Database (MDB): a manually maintained file of personal information. Because these databases are normally not systematically managed as a group, there is no MDBMS analogous to a PDBMS. Each MDB is separate and distinct from all other MDBs; an address book, appointment book, etc., are each MDBs.

- File: a relationship between records. An MDB is a file. All records in a file are of the same format and related by the their grouping into the same file.

- Record: an entry in a file. In an address book each time a person or an organization is added to the "address book file," a new record is added.

- Field: an entry in a record. In general, all records in the same file have the same fields (and thus structure). In an address book, the fields are usually called "name," "street," "city, state, and zip code," and "telephone number."

B. GENERAL CHARACTERISTICS

As stated before, people do not generally view personal data as a database in the same sense as information in a computerized database. Each MDB tends to be viewed as a distinct entity, unrelated to any other MDB. Thus there is no notion of a database management system (DBMS) since the MDBs are not managed together as a group. As a result there is often redundant information in MDBs when they are viewed as a group. For example an address book and an appointment calendar probably both contain redundant information about an individual's insurance agent, realtor, dentist, etc. Even though the possibility for joins and Cartesian products exists, they are not only not performed, but the concepts behind these operations are apparently incomprehensible to the layman.

The existence of separate MDB's or files can be intuitively explained by three reasons. First, and most obviously, is that the amount of effort required to maintain even a partially integrated database manually costs more than the value gained by having such a database. Maintaining such a database requires the establishment of all possible desired relationships before the implementation of the database followed by the maintenance of complicated and troublesome cross-indexes. Less effort is required to check one's appointment book for appointments and then go to one's address book to obtain the phone number to call in order to confirm an appointment; or if the requirement for a

confirmation was foreseen, to simply duplicate the phone number in the appointment book.

The second reason is more subtle and might be related to the ideas expressed in reference [3]. Even though the same entity (person, organization, etc.) may be included in more than one file, the different occurrences may represent different views of that same entity; that is, file entries are context-sensitive. When comparing address book records to appointment calendar records, it is very common to find that the address book entry for an individual is more formal than an appointment book entry for the same individual. For example "Richard Elton" might appear as "Richard and May Elton" in an address book, "Rich" in an appointment book, and "Lt. Elton" in a personal note. This context-sensitive nature of entries seems to indicate that integrating a personal database is much more difficult than in the case of traditional DBMSs.

The last reason is that inconsistencies between personal MDBs (i.e., files) due to replication (redundancy) of data is easily managed. This is not only because of the individual and aggregate file sizes, but also because of the nature of the data. The issue of size is obvious; the important characteristic of the data which aids in solving the problems of inconsistency is that the keys used for access are closely related, if not identical, to cues used to reconstruct semantic structures. For example, when a person receives a change to his friend Pat's phone number, it will probably prompt him to make a change in his address/phone book. What changed was not the entity "Pat" but just a value of one of the entity's attributes. So for the most part, the cues (which are context-free) associated with "Pat" remain unchanged. There is a good possibility that all occurrences of the old phone number will not be updated. Later when he comes across an occurrence of the

old number, it will elicit many of the same cues related to "Pat" as would the address book entry. Chances are that he will remember that the number was changed and was recorded in his address/phone book. It will be then that the inconsistency will be corrected, if it is at all. Perhaps people rely upon this and intentionally do not make any great effort to seek out inconsistencies.

1. Files

Manually maintained files are apparently organized in two ways: sequential access and direct-keyed access. MDBs which are direct-key accessed are normally recorded in a commercially procured file or document. Examples of these files are address books which are designed to be keyed on the first letter of a surname in the "name" field or appointment books which are designed to be keyed on a date. Sequentially maintained files are commonly kept on less rigidly structured media such as notepads, chalk boards, or scraps of paper. Information is usually entered chronologically. Shopping lists, things-to-do lists, etc., are examples of sequentially organized files. Another distinction between the two file types is the time-value of the information stored in them. Indexed files usually contain information which is to be retained for a longer period of time than that contained in sequential files. It was not uncommon to find address book entries which were more than ten years old.

2. Records

With the exception of personal notes, records within any particular file tended to be fairly uniformly formatted. There is generally a core of fields which contain a value in almost all records. However many records contained additional fields beyond the "core-fields." In the case of

address books these fields were inserted into the pre-printed record formats by writing them vertically, placing them in an unused, unrelated field, or placing them into another record. The "core-fields" in address books are: "name," "street," "city," "state," "zip code," "area code," and "telephone exchange and number." Typical additional fields contain information such as:

- Account, Model, Serial, Policy, and Social Security Numbers.
- Additional Phone Numbers (e.g., "home," "work," "marketing department," "service," "account inquiries," etc.).
- Birthdays and Anniversaries.
- Additional Names (e.g., children's names, points of contact).
- Cards and Favors Sent and Received.
- Additional Miscellaneous Information (e.g., "When in Seattle," "Neighbors in Monterey," or "Uncle Bob's brother-in-law").

In the case of address books, record deletion appears to be an unpredictable event and probably a function of the medium upon which it is recorded. Bound address books contain many more entries whose validity are questionable. Many of these appear to be retained not only because they were entered in ink, thereby making deletion a messy affair, but for sentimental reasons. Many of the very old entries are for high school and childhood friends. Address books which permit easy deletion of records appear to contain fewer old entries, but because deletions are not recorded it is not easy to attribute this effect to the ease of deletions.

3. Fields

Even though the fields' types and numbers appear to be fairly standardized, the contents of the fields is not. Fields appear to be variable length with no restriction on content. Graphic, non-alphanumeric symbols such as hearts, check-marks, and "happy faces" are not uncommon. Some files contain indicators of the validity of the information in the field (e.g., "?" or "as of Dec 81"). Abbreviations are not consistently used in the same file; for example, one address book examined contained all of the following entries:

Street	St.	Str.	
Avenue	Ave.		
Virginia	Virg.	Va	VA
Mr. & Mrs.	Mr/Mrs	Mr. and Mrs.	

C. DESIGN IMPLICATIONS

It appears obvious that a PDBMS and a DBMS are not the same. As such, it is reasonable to construct a PDBMS differently from a DBMS. Because a PDBMS is used as an aid to recall contexts from memory, and the cues to these are unique to each context [3], not only should the system have no restrictions such as fixed field lengths and attribute values, but additionally it should:

- Allow the user to use any word as a key.
- Be able to recognize and compensate for misspelled keys.
- Be able to take into account keys which are synonyms and refer to the same entity (for examples see the description of fields, above). Also it should have the ability to discriminate between homonyms which appear to be the same but refer to different attributes or entities (for example, "CT," as an abbreviation for "Court" in a street address versus "CT," as an abbreviation for "Connecticut").

When interviewing laymen, it was found that they easily understand the concepts of "file" and "record," but not "field." This suggests that perhaps people conceptualize an entity as a synergistic sum of its attributes rather than as a relationship between attributes. Thus a record is the smallest logical unit with which people normally deal because it, as a whole, contains the cues necessary to reconstruct semantic structures. The number of fields in a record may be related to an individual's ability to "integrate" the corresponding semantic structure [3].

Because a PDBMS is an aid to an individual's recall, it should faithfully preserve information entered and retrieve it by logical means. If text compression or compaction[2] is employed it must be transparent to the user. Logical retrieval means that if the user feels that he has given sufficient information to specify the desired data, the system should be able to either retrieve the data or give a comprehensible reason why it could not be retrieved.

A PDBMS should be "user friendly" and require very little effort on the part of the user. This means that persons who have no need or desire to understand computers, DBMSs, etc., should be able to use the system. Further, file, record, and field formats should be easily specified without the need for a plethora of technical details. Entry and retrieval of data should also be fast and easy. Most people who are not specifically trained on computers tend to have much less tolerance for poorly engineered computer systems or ones requiring a technical expertise than do the

---

[2]Text compression and compaction involve removing redundant information from text so that it can be stored using fewer resources than if the original text had been stored. The difference between the two is that an exact copy of the original text is recoverable after compression, whereas it is not from compaction.

system's designers or computer scientists [4]. Above all, a computerized system must be better in every way than the corresponding manual system [1].

III. HIGH LEVEL PDBMS SYSTEM DESCRIPTION

A. SOFTWARE

When the user first receives the PDBMS, he sees only two functions: a calculator and a database management system. As the user learns how the system works, it is possible for him to expand the system incrementally until eventually he can reprogram a large portion of the system itself in FORTH and/or assembly language.

Many of the keys on the PDBMS's keyboard are programmable. They are initially used to allow the user to enter commands by simply pushing a key. Instead of typing "RECORD" when using the database management function, the user needs only to push the "SHIFT" and "R" keys and the system will enter the word "RECORD" for him.

1. The Calculator Function

The calculator which the user initially receives is much like any other calculator. Two major ways in which this function differs from most standard calculators is that a series of arithmetic operations may be entered at once, and that the user may create and use variables. Unlike most calculators, the action of most of the keys on the PDBMS is simply to enter textual data into the system. The PDBMS does not interpret most of the input until the ENTER key is pressed. So the following two key sequences have the same effect, i.e., to add two to three and obtain five.

2	2
<enter>	<space>
+	+
<enter>	<space>
3	3
<enter>	<space>
=	=
<enter>	<enter>

Like in FORTRAN, variables are created when they are first used. If a word or a character is found in the input which the calculator cannot recognize and it is to the right of an equal sign, it assumes that it is a variable declaration and creates one. If an unrecognizable word or character is encountered to the left of an equal sign, an error condition is signalled.

2. The Database Management Function

The database management function allows the user to create files and records, delete files and records, retrieve records, and use keys (i.e., passwords) to seal records and other keys as a means of providing data security. The user is not required to deal directly with the technicalities of database data structures, he only needs to know that files are a collection of records, all having the same format. Files appear to the user to be separate and disjointed, similar to MDBs. The procedure for creating a file requires only that the user specify the file's name and the names of the fields within the records of the file. The user is led through the process of file creation and record retrieval by system prompts.

Records may be retrieved by using any word (or group of words) contained within them. The only restriction on this is that the user must specify which field is to be

searched for the target word(s). This restriction should not seem unnatural to the user but, rather, necessary. Because any word is a possible key attribute, the user must be able to specify the context of the target word. By specifying the field name with queries, the user is able to retrieve a record using Mr. York's last name without also retrieving all of the records containing "New York."

B. DATA STRUCTURES

The PDBMS uses some data structures which might be considered unusual when compared to other database applications. Some of these are characteristic of FORTH and others are used because of the nature of the system.

1. Dictionaries

Two different dictionary structures are used in the PDBMS. One dictionary is that which is associated with FORTH. The second is conceptually more like a dictionary, as a layman might think. A FORTH dictionary is simply a linked list of FORTH definitions. The definitions are maintained in chronological order by their time of creation. These definitions typically describe the following basic FORTH word-types: colon definitions, constants, variables, user variables, and vocabularies. Colon definitions are FORTH definitions which are defined in terms of previously created definitions, similar to procedures and functions in other languages. Vocabularies are "sub-dictionaries" and are used to delimit the scope of definitions.

The other dictionary is called the DB dictionary and it is used to store the words entered and contained in the database. Words are entered into the dictionary and looked-up by hashing to a linked list using the first letter or digit of the target word, and then traversing the list,

which is alphabetically sequenced. Punctuation is not stored in the DB dictionary.

2. Files

Files are completely inverted. They contain only administrative data, and indices and pointers into the DB dictionary. Information which is retrieved from the database is reconstructed a word at a time by looking words up in the dictionary (punctuation is stored directly in the database in its ASCII format). Memory for files, the DB dictionary, and sealed keys (discussed later) are allocated from a heap so that none of these data structures occupy contiguous memory. A file is defined as a FORTH vocabulary and its definition contains pointers to the first and last records in the file. Records are maintained as a circular, doubly linked list. The fields are defined as FORTH constants in their respective file's vocabulary. Their value is an ID number which is used to relate the fields in the database to the names assigned to them by the user.

3. Logical Records

To the user a record appears to be a collection of information related to a particular entity. The fields help to organize the data by grouping it. The logical record itself is variable in length. The first set of bytes in a record contain the record's access descriptor, which is variable in length. This is followed by the links (or pointers) to the previous and next records in the file. Following these pointers are the fields which are fixed in number (as determined in the file's definition), but are each variable in length. Fields are separated by an end-of-field (EOF) marker. Because records contain a fixed number of fields, the last EOF serves as a end-of-record marker.

4. Fields

Fields are a continuous string of bytes which represent the data contained in the field. Punctuation appears in its ASCII format (one character per byte). Words are represented by two bytes, the first contains the word's initial letter (or digit) which is used to hash into the DB dictionary. the second byte is a number used to identify the particular member of the linked list hashed to representing the target word.

5. Keys

Keys may be thought of as passwords which are used to secure records, FORTH screens, and other keys (called sealed keys). These objects (i.e., records, screens, and keys) all have access descriptor fields which contain information about what keys are necessary to access the particular object. Keys allow the user to construct fairly complex access mechanisms.

C. HARDWARE

Figure 3.1 is a simple picture of the layout of the PDBMS's hardware. The system makes extensive use of CMOS technology so that it can be battery driven. There are six major components in the system.

1. Erasable Programmable Read-Only Memory

Erasable programmable read-only memory (EPROM) occupies the system's low memory and contains the PDBMS's operating system. There are 16K bytes of EPROM in the system. As its name implies, its contents cannot be altered by the user.

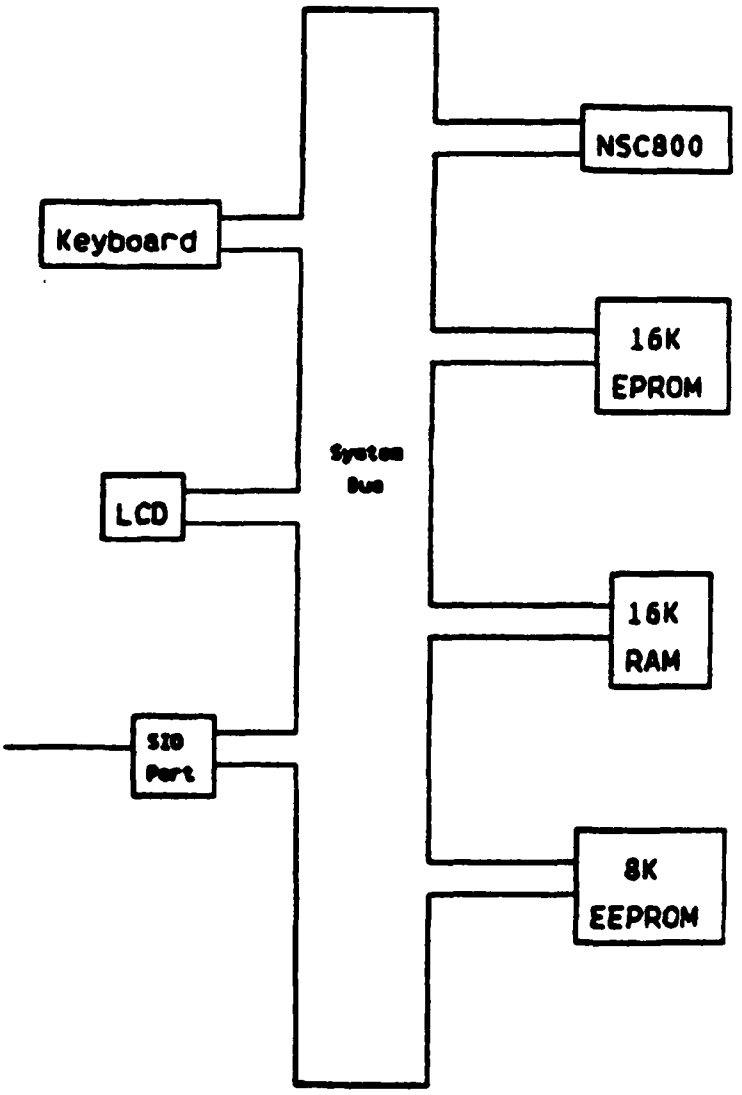


Figure 3.1 PDBMS Hardware Configuration.

2. Random Access Memory

Random access memory (RAM) is used by the user as his workspace. System parameters and data structures which change according to the runtime environment are also maintained in RAM. There are 16K bytes of RAM.

3. Electrically Erasable Programmable Read-Only Memory

Electrically erasable programmable read-only memory (EEPROM or E²PROM) serves as the system's secondary storage. The unique characteristic of E²PROM is that it can be erased (i.e., written into) under software control, as RAM can, but it is non-volatile (i.e., its contents are not lost when the power is turned off). Part of the E²PROM is not accessable to the user because it is used by the system for E²PROM memory management, and database management and storage. What is not used by the system is available to the user as FORTH screens.

4. Liquid Crystal Display and Keyboard

The liquid crystal display (LCD) serves as the system's console. It contains two rows of 20 characters. It is attached directly to the system's bus and any data written into memory beginning at address C000H appears on the LCD. The keyboard provides the means by which the user can directly input data into the system. It is connected to the system's bus via a parallel I/O port.

5. Central Processing Unit

The PDBMS uses an NSC800 microprocessor operating at a clock rate of 1 Mz. This is a CMOS microprocessor which is downwardly compatible with the Z80. It was chosen as the system's CPU because of its low power consumption and the availability of software. The slow speed is not an issue

with this system because of the naturally slow nature of human-computer communications.

6. RS232 Serial I/O Port

This port allows the user to interface his system with other systems.

IV. DETAILED PDBMS SYSTEM DESCRIPTION

A. CONVENTIONS AND NOTATION

The nature of words in FORTH does not lend them to be referred to by enclosing them in quotes, so instead they will appear in upper-case boldface. However, because boldface punctuation is often hard to distinguish from standard text punctuation, the following eight FORTH words will be enclosed in braces:

: . , ; ! ? ' "

Additionally FORTH words composed entirely of strings of these characters will be enclosed in braces (for example, {."}).

Finally, to avoid ambiguity, the following conventions will be used when using the three words "key," "word," and "dictionary." When there is a possibility of confusing the FORTH meaning of "word" (described below) and the accepted computer term "word" (i.e., two bytes or 16 bits on the 8080 and Z80 microcomputers), the former "word" will be called a "word" or a "FORTH word," whereas the latter "word" will not be used, instead "two bytes" will be used. Adding further possibilities for confusion is the third meaning of "word." This third meaning is the usual English connotation of "word" and these "words" are data in the PDBMS. The ubiquitous FORTH response, "OK," and words entered by the user as responses to the system prompts and as data to be included into the database are "words" in this third class. Data words of this type will be called "uwords." Because uwords entered into the database may be altered before they are entered into the database dictionary, the words which reside

TABLE I

BNF Definition of Uword and Wordd

```
uword ::=        <wordd><punctuation>|<punctuation>
punctuation ::=  ,|.|/|+|=|-|<space>|*|(|)|:| ... etc.
space ::=        20H
wordd ::=        <wordd><char>|<char>
char ::=         1|2|3|4|5|6|7|8|9|0|A|B| ... |X|Y|Z
```

in the database dictionary will be referred to as "wordds." Table I shows the BNF definitions of both uword and wordd.

In order to distinguish between a "key" on the keyboard and a "Key" which is used as a password to SEAL and UNSEAL data objects, the latter "Key" will always begin with a capital "K." Finally, because many of the system data structures are not only maintained as FORTH dictionaries (also referred to as vocabularies); but wordds are stored in a data structure which is not a FORTH dictionary but which may also be rightfully called a dictionary, the following convention will be followed. When the possibility of ambiguity may exist, the dictionary being referred to will be prefaced by its name (e.g., root dictionary, DB dictionary, etc.).

B. PHYSICAL MEMORY AND I/O PORTS

1. Hardware and I/O Ports

Physical memory is that memory in which FORTH programs execute. This memory lies entirely within the user's address space. The PDBMS's physical memory consists

of a little more than 32K bytes (see Figure 4.1). The lower memory (0000H to 3FFFH) is EPROM, and the high memory (4000H to 7FFFH) is RAM. Additionally there are 256 bytes of memory located at addresses C000H through C0FFH; the first 40 bytes of these 256 bytes represent the 2 lines of 20 characters on the liquid crystal display (LCD). The contents of these memory locations are interpreted as ASCII encoded data and are mirrored on the LCD. Thus the LCD is directly addressable via the system's bus. Finally, memory locations FF00H to FFFFH comprise the virtual E²PROM window. When a segment is accessed from E²PROM by writing its segment number to the segment register and "powering up" the E²PROM, it appears at these addresses and may be read from and written to. When E²PROM power is off these addresses are invalid.

There are two ports which are directly associated with the user's address space and accessible to him. One port is a read-only port used to receive data from the keyboard (it is envisioned that the keyboard will eventually be tied directly to the system's bus). This port is located at FBH. The other port is a UART port configured for an RS232 serial interface and is located at FAH.

Finally three locations are set aside as jump vectors. These are predetermined by the NSC800 hardware in interrupt mode 1 which mimics the Z80. The cold boot vector is located at 00H. The non-maskable interrupt (NMI) jump vector is found at 66H. This interrupt is generated by two conditions: whenever the system is "turned off" by the user and whenever the system is reset (via the reset button). Because of the slow nature of the E²PROM, it may be possible for the user to turn the power off or reset the system before a write-cycle involving a large block of data has been completed. The virtual memory manager is the ultimate recipient of NMIs. Upon receiving one, it waits for the

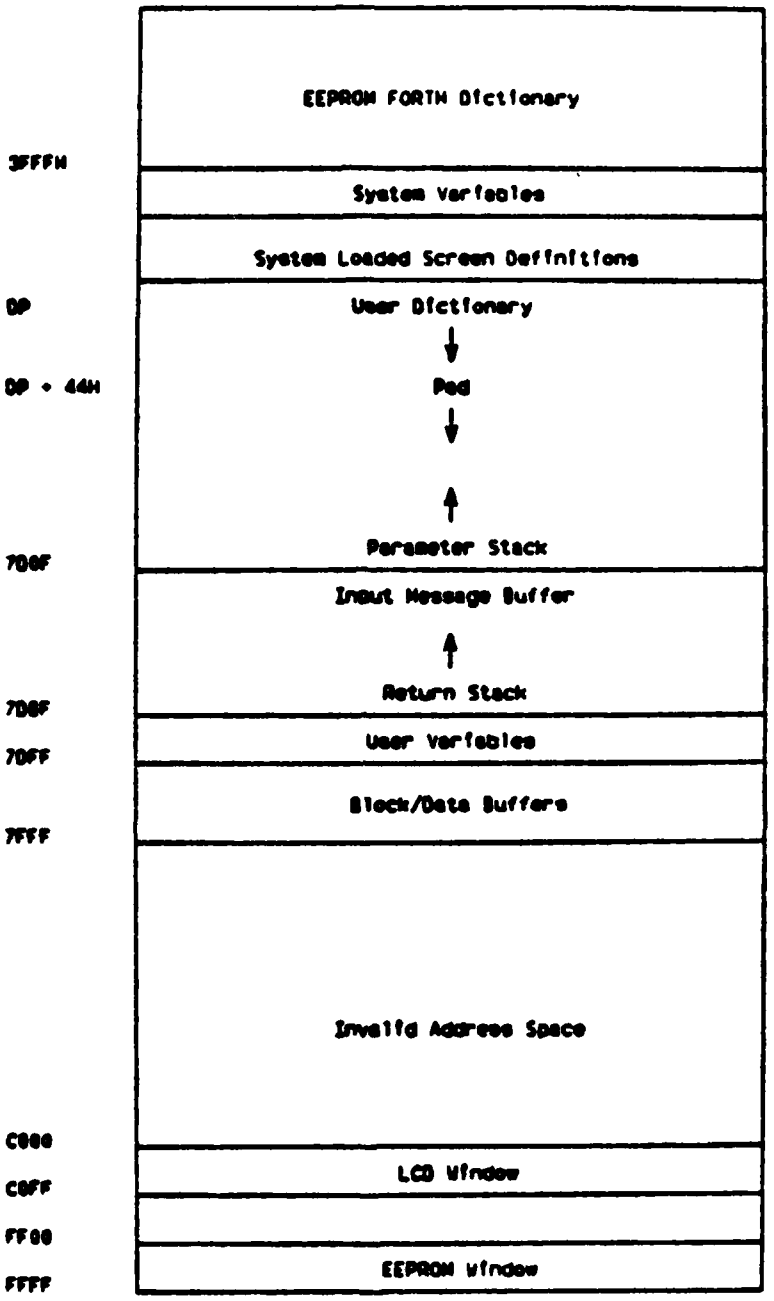


Figure 4.1 PDBMS Physical Memory Map.

write-cycle to be completed and then sets bits 1, 0, and 4 of the control port accordingly. After doing that, a jump to warm boot is executed. Setting bit 4 to one when the power switch is in the on position has no effect, so the same interrupt handling routine correctly handles both interrupt sources. Ten seconds after an NMI generated by the power-off condition, the hardware automatically shuts itself off, if it is still on at that time. The third location is 38H which contains the maskable interrupt (MI) vector. Both the keyboard and E²PROM generate interrupts which vector here; the device requiring service is determined by reading the status register (described below).

2. Data Structures

Figure 4.1 shows the allocation of physical memory to data structures in the PDBMS. It varies from the configuration in Figure A.1 only in that it has data buffers and pointer buffers. These buffers share memory with the buffer blocks. Block and data buffers are not used concurrently so they do not occupy the buffer area at the same time[3]. The data buffers are used for encoding and decoding individual database records. Records are read into the buffers as they appear in E²PROM (less key ID numbers and administrative pointers) and then are decoded into their ASCII representation which is placed into the current record buffer and the LCD window. Probably only a portion of the record fits into the 40 character LCD. The first two bytes of each data buffer contain the resident record's virtual pointer (FFFFH indicates an empty buffer).

---

[3]Even if the PDBMS is designed so that it LOADs definitions from screens during execution of database operations, there is no problem. This is because the block buffers are not used during a LOAD; the E²PROM is simply read directly without using a buffer.

The pointer buffers serve several purposes. During retrieval operations buffer number one holds the pointers to records to which the user is authorized access and which have satisfied all query conditions processed so far. The second buffer holds pointers to records to which the user is authorized access and which satisfy the current query condition being processed. After the completion of the processing of each query condition the intersection or union of the two buffers (depending upon the query) of the two buffers is placed into buffer one.

C. VIRTUAL MEMORY AND CONTROL PORTS

1. Hardware

In the PDBMS, E²PROM is used as secondary storage. A total of 8K bytes of E²PROM is included and it is segmented into 32 segments, each 256 bytes in size. Segments (analogous to FORTH blocks) are further divided into physical records 16 bytes in size. Figure 4.2 shows the bus interface of the Intel 2816 E²PROM chips. As in standard FORTH, the user and user programs deal with physical addresses only. The user can only refer to virtual memory by using screen numbers. However, some PDBMS words use two byte virtual addresses to access physical records in virtual memory. Only assembly language coded words ("low-level" words) can directly fetch and store bytes in E²PROM via the window.

PDBMS virtual addresses consist of two bytes. One byte contains a segment number and the other a physical record number within the segment. Because only four bits are needed to designate a physical record, if it were technically feasible the system could accommodate 512K bytes of E²PROM.

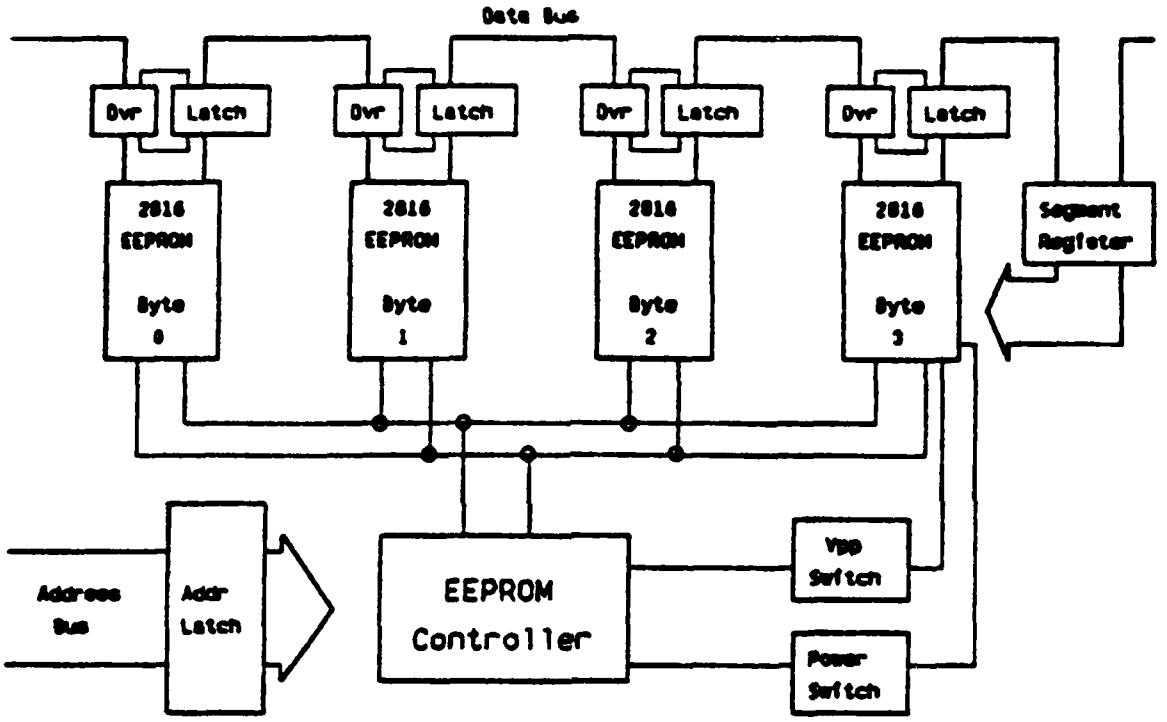


Figure 4.2 2816 E²PROM Configuration.

Only 15 of the 16 bits are used for virtual addresses. The high bit (bit 7 of the Most Significant Byte—MSB) is used to differentiate virtual from physical addresses in E²PROM and RAM. Virtual addresses which move from E²PROM to RAM and vice versa must pass through low level FORTH words which ensure RAM and E²PROM virtual addresses never get mixed in with each other. E²PROM virtual addresses have their high bit set to zero while RAM virtual addresses have their high bit set to one. Thus virtual addresses appear to be out-of-range references within the domain in which they occur. For example, if an address referenced inside an E²PROM segment is less than 8000H, then it is a virtual address to another segment. Intra-segment addresses are always greater than or equal to FF00H (all of which have a high bit of one). This means that, as in standard FORTH, "programs" cannot be executed directly from secondary storage but must be LOADed first. This allows <u>all</u> code field addresses (CFA) to be interpreted as physical addresses, whether they occur in RAM, EPROM, or E²PROM, so there is no problem associated with storing constants and variables in E²PROM. Care must be exercised to ensure that LCD window addresses are never used in the same RAM context as RAM virtual addresses since they would be indistinguishable from each other.

The E²PROM can be read in 450 usec, however it requires 20 msec* to write one byte (all of the bytes on each chip may be erased in one 10 msec operation). Additionally the 2816 must be strobed with a 21 volt pulse during the write process. This means that E²PROM cannot be

---

*Intel literature states that their E²PROM requires 10 msec per write, which is true. However, in order to ensure that the data is properly recorded, the addressed byte should contain FFH before it is written into if a write requires a zeroed bit to be changed to one. Thus writing involves two write operations: one to set the target byte to FFH, and a second to write the desired value.

treated the same as RAM. Other non-volatile memories were considered for this design, such as NOVRAM and Instant ROM. Both of these alternatives can be treated almost as if they were RAM, however they were judged unsuitable. NOVRAM was not found to be a feasible choice because of its small size. The largest NOVRAM chip contains only 256 bytes, thus 8K of NOVRAM cannot be battery powered because of the large number of chips that would be required. Instant ROM was also found to be undesirable because it contains its own battery power. The on-chip battery is guaranteed for three years, and this is hardly suitable for a permanent database. Currently available hand-held computers use concepts similar to Instant ROM, they use CMOS memories which are constantly refreshed, even when they are turned "off."

The E²PROM and the PDBMS is controlled through three control ports. One port, the segment register, is used to select the desired segment. This port is located at F8H and is write-only. The second port is the status register. It is located at F9H and it is read-only; it reflects the system's current status. Figure 4.3 shows the status port's configuration. Complementing the status register is the control register which is a write-only port located at F9H. The control register is used to effect system changes. This port is described in Figure 4.4. These ports, as well as all other ports, are "smart" ports in that they only accept instructions from code being executed from EPROM. It does this by checking the program counter which the NCS800 places on the address bus prior to fetching an opcode fetch. If the A15 and/or A14 lines of the address bus are high the next instruction is ignored. E²PROM power and write-power are turned on and off by setting bits 0 and 1 accordingly. Whenever either of these bits is set to one, bit 7 of the status register is set to zero. After the chips have been powered-up, bit 7 of the status register is set to one, so

Bits	Flag Meanings	Boot-up Values
7	1: EEPROM ready 0: EEPROM not ready	1
6	1: EEPROM write-power is on 0: EEPROM write-power is off	0
5	1: EEPROM interrupt pending 0: No EEPROM interrupt pending	0
4	Not used	n/a
3	Not used	n/a
2	1: Keyboard interrupt pending 0: No keyboard interrupt pending	0
1	1: UART receiver ready 0: UART receiver not ready	n/a
0	1: UART transmitter ready 0: UART transmitter not ready	n/a

Figure 4.3 Status Port Flags (IN 9FH).

is bit 6 or 5 (depending upon whether bit 0 or 1 of the control register had been set). Additionally, whenever bit 7 is set to one (except during a cold boot of the system), an NMI is generated. When bit 7 of the control register is set to one, bit 7 of the status register goes to zero. When the E²PROM write-cycle has been completed, bit 7 goes high and an NMI is generated.

Changes in bits 0 and 1 of the status register do not generate interrupts, but when bit 2 goes high (indicating keyboard input) an NMI is generated. Reading the status register resets bit 2 to zero.

Notice from Figure 4.2 that the four 2816 chips are interleaved so that all addresses equal to zero, mod four, are on the first chip (i.e., those addresses whose last hexadecimal digits are 0, 4, 8, or C). Those equal to one, mod four, are on the second chip, etc. This arrangement facilitates fast writing of blocks of data to E²PROM because four contiguous bytes may be written simultaneously. Thus in the best case (when four contiguous bytes are written) the average write-time per byte is approximately 5 msec and an entire segment can be written in 1.25 seconds. Actually more time is required, but the additional time is minor when compared to the gross nature of the E²PROM write-time. The additional time involves reading and comparing the contents of the E²PROM to the appropriate buffer's contents (data or block buffer). The entire write-cycle algorithm is shown in Table II.

2. Organization and Data Structures

The 8K bytes of E²PROM are divided into two types of segments: system segments and block (or screen) segments. System segments are owned by the system and cannot be directly accessed by the user or his programs. Block segments are those which contain screens, in the usual FORTH

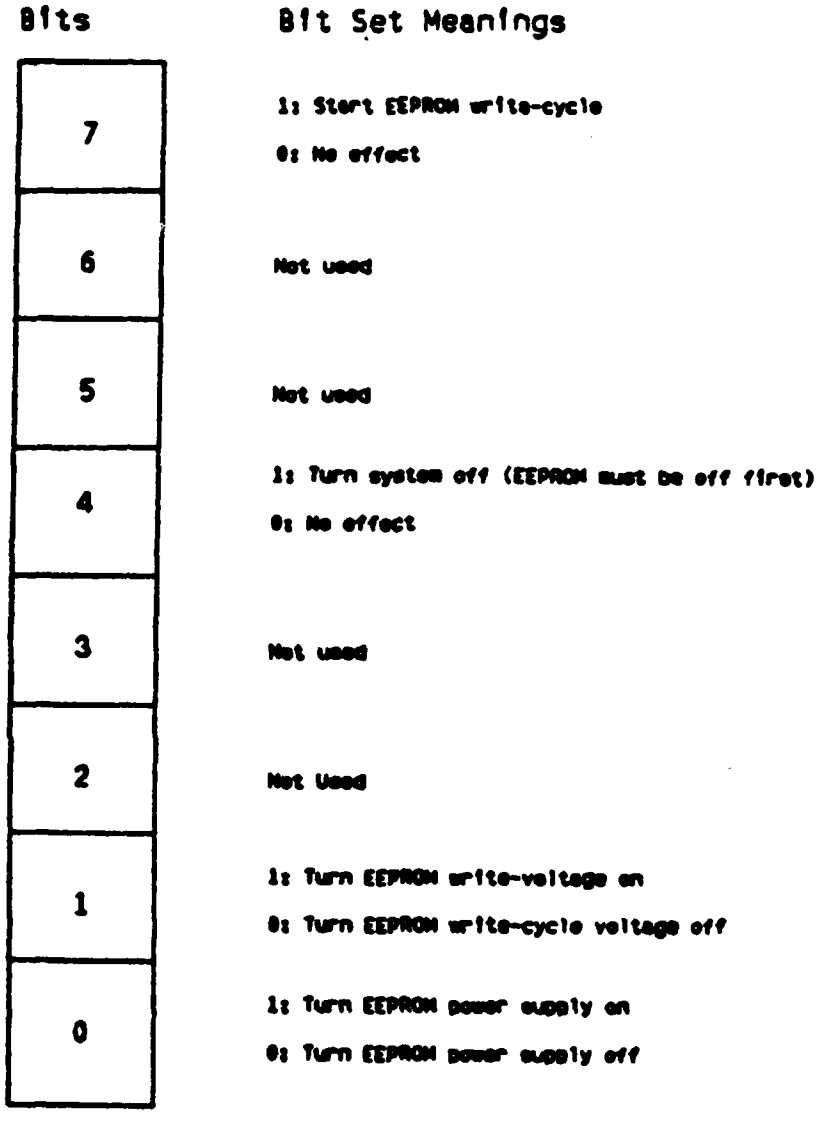


Figure 4.4 Control Port Flags (OUT 9FH).

TABLE II

Virtual Memory Write-cycle Algorithm

```
J = START_OF_SEGMENT;
REPEAT UNTIL NO_MORE_BYTES;
 DO I = J TO J+3;
    READ E²PROM_BYTE(I);
    IF BUFFER_BYTE(I) ≠ E²PROM_BYTE(I) THEN DO;
       IF BUFFER_BYTE(I) & E²PROM_BYTE(I) ≠ 0 THEN
          E²PROM_BYTE(I) = FFH;
       E²PROM_BYTE(I) = BUFFER_BYTE(I);
    END DO;
 END DO;
 CONTROL_PORT_BITS(7) = 1;
 LOW POWER HALT; /* WAIT FOR INTERRUPT */
 DO I = J TO J+3;
    READ E²PROM_BYTE(I);
    IF BUFFER_BYTE(I) ≠ E²PROM_BYTE(I) THEN
       SIGNAL(E²PROM_WRITE_ERROR);
 END DO;
 J = J + 4;
END REPEAT;
```

sense, and are available to the user. Blocks are allocated sequentially in a round-robin fashion by the memory manager. This means that the next segment to be allocated is the next higher unallocated segment after the last allocated segment. When the 32nd segment is reached, allocation begins again from the first segment not initially assigned to the system (i.e., when the software was placed into the system). This scheme is used in an attempt to more uniformly distribute

the E²PROM use. If a "lowest available segment algorithm" were used, there would be a higher probability that portion of E²PROM assigned to the low numbered segments might "burn out" (E²PROM is limited to 10,000 write operations to each individual byte).

a. System Segments

System segments are those which are used by the PDBMS for virtual memory management data structures and the database. The user cannot directly access these segments because any segment allocated to the system is not placed in the block number dictionary. System routines address these segments directly (i.e., they "know" the physical segment numbers whereas the user knows only virtual block or screen numbers). At least four segments are dedicated to the system; the system and the user compete for the remaining segments (less system message screens) which are allocated on a first-come, first-serve basis. Additional system segments (beyond the dedicated four) are used to accommodate the expanding database. Because the database resides in system segments, the user cannot see their physical structure; he is limited to viewing it through the PDBMS. The first four segments are structured as described below.

(1). Parameter Table. This segment contains a collection of system parameters and tables. For example, most of the cold boot parameters are loaded from here. Also located here is the vocabulary table.

(2). Key Sub-Dictionary. Security in the PDBMS is provided in part by Keys. These Keys are used to seal records, blocks, and other Keys. These Keys are maintained in a linked list dictionary as a separate VOCABULARY. The Key vocabulary definition is located in EPROM. The code pointer of each Key points to the run-time code for CONSTANT which is located at docon. Thus when the Key is executed,

it returns the contents of its two byte parameter field address (PFA). The value held in the PFA may have two meanings. If the value returned is less than 128, then it is the Key's identification number (ID). If it is greater than 128, then the value returned is a virtual pointer to a sealed record containing the Key's ID number. The Key ID value, FFH is reserved for the null Key, while the value 00H is reserved for the system's Key. Also the value FEH is used as a substitute ID for the ID value of deleted Keys' IDs in access descriptors. The use of Keys is discussed in greater detail in Chapter VI. The Key vocabulary, besides containing Keys, contains words; these words are stored in EPROM.

(3). Block Number Dictionary. The segment containing this is divided into three parts. Four bytes are set aside as the segment allocation table, four bytes are used as the segment allocation sequencer table, and the rest of the segment is used as a vocabulary for virtual block numbers. Each bit in the segment allocation table represents a segment. If a bit is set to one, the corresponding segment has been allocated. The sequencer table has only one bit set, the one corresponding to the last segment allocated.

The virtual block numbers are maintained as a FORTH vocabulary, as are the Keys. Also like the Key vocabulary, the definition of the block number vocabulary is located in EPROM. However, unlike the Keys, virtual block numbers are fixed length name, one byte constants. This allows virtual numbers to be assigned to all of the originally unallocated segments. This limits block numbers to four characters in length. This dictionary is static and always contains 28 entries. Entries are removed from the dictionary by blanking out their virtual number (i.e., the entry's name field) and setting the smudge bit so they will

not be found. When a virtual block number is entered by the user, the entire dictionary is searched. For example the following keyboard entries would trigger searches of the dictionary for "1" and "25" respectively.

```
1 LIST
25 LOAD
```

If "1" had not been found in the dictionary a block buffer (located in physical memory) would have been allocated to virtual block "1." The virtual number "1" would not be entered into the block number dictionary until it was written to E²PROM. If "25" had not been found the usual FORTH error condition would have been raised.

(4). The Database Segment. This block is broken into two parts. The first contains a jump table into the DB dictionary. There is one jump vector for each printable ASCII character allowed by the system (a maximum of 64). A character's jump vector is hashed to using the following equation on the character's hexidecimal value (called "char").

Location of jump vector =

((char - 32H) * 2) + FF00H

If the vector is equal to zero, then the character is punctuation (as described in Table I). Punctuation is not stored in the DB dictionary. If the vector is equal to FFFFH (uninitialized E²PROM), then there are currently no wordds in the dictionary starting with that letter. Otherwise the vector is the virtual address of the first physical record in an alphabetical linked list of wordds beginning with that letter. The next four bytes of this segment contain a bit map of the segments. Like the segment

allocation table, a bit is set to one if the corresponding segment belongs to the database.

The second half of this database segment is used for the beginning of the file and field name vocabulary. Field entries are simply FORTH constants which return their field ID number (0 to 255). File entries are modified FORTH vocabulary definitions (they contain five extra bytes used to store pointers to the first and last records in the file, and a field count). The field names are entries into the "file vocabulary" to which they belong. This allows FORGET to be used to delete files. Of course FORGET is not sufficient by itself; the virtual memory allocated to the forgotten entries must be turned back to the system. Because of the nature of record entries in the PDBMS, fields cannot be individually forgotten. As with the Key vocabulary, the file vocabulary definition, as well as some other words, reside in EPROM.

When information is added to the database, it expands in three ways. First the file and field vocabulary grows to accommodate new file and field definitions. This dictionary may spill into additional segments. Allowing this dictionary to exist in more than one segment creates some problems which must be specifically addressed by the interpreter/compiler. Off-segment references can only address 16-bit physical records, so entries of this type cannot be positioned in a "format-free" manner. Thus entries in this vocabulary are all placed in memory taking the physical record into consideration (i.e., beginning on a physical record boundary). A benefit of this is that the entries may be mixed into the same segments with the DB entries, file logical records, and sealed Keys.

The database itself may be considered a totally inverted file system. Records contain only PDBMS information and pointers to dictionary entries of wordds

which appear in the record. Figure 4.5 shows a typical entry in the PDBMS. The system knows how many fields are in the currently open file, so it uses the last field's end-of-field (EOF) as the end of record marker (EOR). The EOF is the same character as the null Key, making FFH (blank E²PROM) a general system end-of-data marker. When a logical record is broken over a physical record boundary, the last two bytes of the physical record contain a pointer to the next physical record.

Fields are strings of ASCII characters followed by an entry ID number. The ASCII letters are the initial letter of the wordds (i.e., transformed uwords) originally entered into the record by the user. The letters are used to hash to the jump vector table on the first segment of the database. DB dictionary entries are maintained in an alphabetical linked list. The correct wordd corresponding to the uword entered into the record is found by matching the ID number following the letter used as input to the hash function to the ID number of a wordd on the linked list hashed to. Punctuation is not followed by an ID number and the record decoding routines "know" not to look for an ID number in the record because punctuation jump vectors are equal to zero.

Figure 4.6 shows a typical dictionary entry. This structure is an expanded and modified version of the one used in Craig language translators [5]. The entries are designed to take advantage of the alphabetical nature of English language dictionaries. The first byte contains a zero and is ignored when traversing the DB dictionary during a wordd look-up. It is placed there to prevent an accidental retrieval by non-dictionary routines which always treat the first byte as a Key. The second byte, the copy byte, contains the number of leading characters in the current wordd which match the leading characters

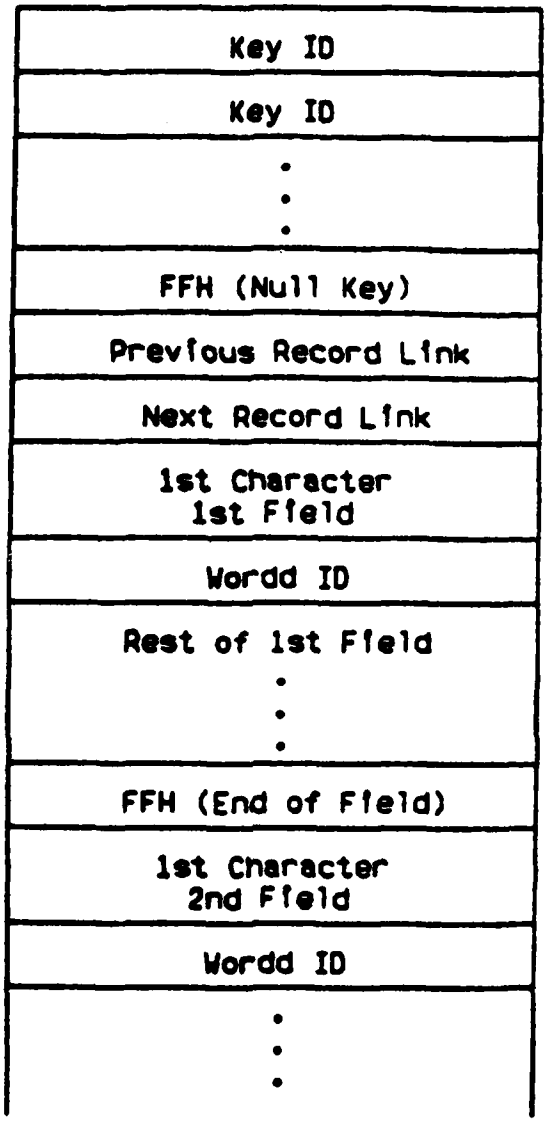
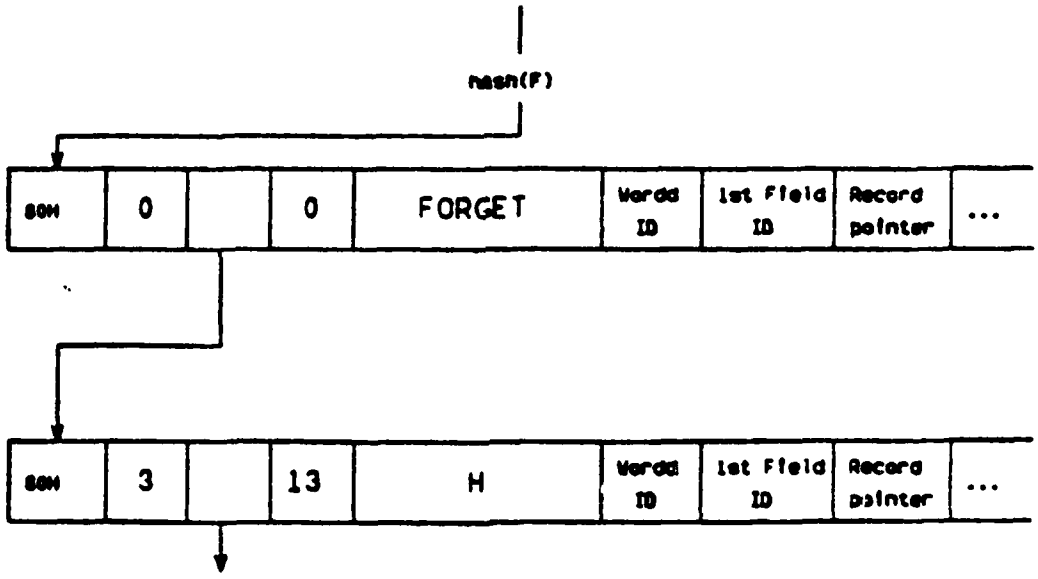


Figure 4.5 Database Physical Record Structure.

in the previous wordd on the linked list. The link bytes contain a pointer to the next wordd in the linked list. The add byte contains a number, which when added to the "copy byte + 1" character of the previous wordd yields the correct "copy byte + 1" character of the current wordd. The bytes following the add byte contain the ASCII characters of the current wordd after the "copy byte + 1" character. The last character's high bit is set to one as an end of string delimiter. If there are no characters following the "copy byte + 1" character then the byte following the add byte contains FFH (which translates to an ASCII delete). The wordd ID byte contains the wordd's ID number. This is used when decoding records. Figure 4.6 shows how the DB entries for "FORGET" and "FORTH" would appear if they were consecutive entries and "FORGET" was the first "F wordd." Following the last unique character is a linked list of field ID numbers with pointers to records containing the field associated with its corresponding field ID. These field numbers and pointers are used in retrieval operations. Records are retrieved by specifying field names and uwords. Obviously punctuation cannot be used for retrieval since only wordds are stored in the DB dictionary.

Figure 4.7 shows how the dictionary is traversed to find the desired wordd. Uwords are reassembled in the PAD by making the changes indicated by the copy byte, add byte, and unique characters as the list is traversed. That is, when the DB dictionary linked list is entered, the first wordd in the list is copied out into the PAD. If this is the not target wordd, then the second entry in the linked list is moved to. Using the information in the copy byte, the add byte, and the unique characters, the second wordd in the list is constructed. In moving from "FORGET" to "FORTH" as shown in Figure 4.6, "FORGET" would be written into the PAD as the first wordd in the linked list of "F wordds."

Typical DB Dictionary Entry



"FORGET" & "FORTH" as DB Dictionary Entries

Figure 4.6 Structure of a DB Dictionary Entry.

When the search continued past "FORGET" because it was not the target wordd, the first three letters in the PAD would be left because the copy byte of the second entry is 3. Then 13 would be added to the fourth letter (G) because that is the contents of the add byte. This would change the fourth letter from a "G" to a "T." Then the fifth letter, and any subsequent ones, would be replaced by the the unique characters (in this case "T" would be overwritten with an "H"). At this point the PAD contains the wordd "FORTH."

Once a wordd has been placed into the dictionary, its first physical record is never returned to the system to be reallocated. If all instances of a wordd are removed from the database, the high bit of the copy byte is set to one. Subsequent searches of the dictionary will not "see" a wordd if its copy byte contains a negative number (two's complement). Because the dictionary is a linked list, this memory may be reused in the same list by reattaching it at a different point in the list. When the first record is reused, the new wordd placed in it uses the ID number assigned to the first wordd to use the record. This is done to make ID assignment easier and to stave off the possibility of running out of ID numbers[5]. Physical records other than the first may be returned to the system when a wordd is deleted.

In segments acquired by the system to accommodate database expansion, only 15 physical records are used for the database. The first record (record 0) contains administrative information such as a record allocation map for the segment.

---

[5]The maximum ID number is 255. The statistics in Appendix B indicate that, even in an aggregate of four address books, the maximum number of unique wordds is not that large.

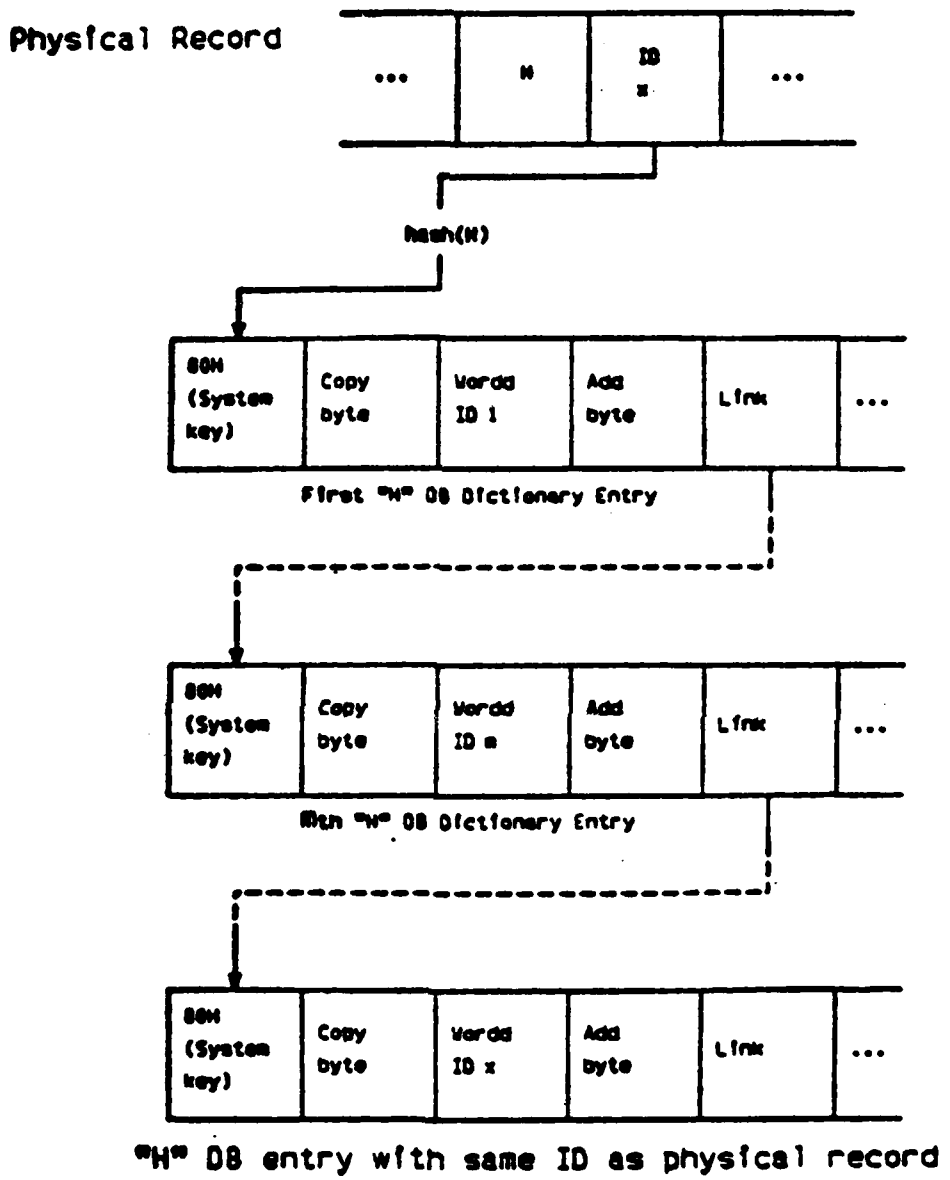


Figure 4.7 DB Dictionary Wordd Look-up.

b. Screen Segments

These segments belong to the user for use as FORTH screens. A screen segment is divided into two parts. The first physical record contains the screen's access descriptor. The rest of the records contain the part of the segment the user sees as a screen. A screen consists of 16 rows of 15 characters. This is much smaller than the standard FORTH screen which is 16 rows of 64 characters. The smaller screen is better suited to the 2 row by 20 character LCD.

When the system is first initialized (i.e., when the software is first placed on the hardware), some of the screen segments are used to store system messages, as in standard FORTH. Additionally, some screens are used to store some of the definitions used in the PDBMS, particularly those used with the naive user interfaces. This allows the user to eliminate or change these definitions and system messages as he sees fit.

V. THE DEVICE DESCRIPTION

At the time of this writing, the PDBMS is in the process of being prototyped. This first prototype is not intended to meet all of the desired characteristics of a PDBMS. For example, it cannot be hand-held because it is bread-boarded and a standard keyboard is used; additionally it requires more than one power supply because not all of the CMOS components have been received. What is described in this chapter is the outline of the final prototype as it is envisioned at the present time. For the most part, this is a description of the PDBMS as it would appear to the user.

A. THE HARDWARE

From the user's point of view, the hardware consists of four major components: 1) the enclosure, 2) the display, 3) the keyboard, and 4) the electronics inside. These aspects involve how the system physically appears to the user, not how he perceives it to work.

1. The Enclosure

The enclosure should be as small as possible and yet still be useful. The major constraints upon how small the PDBMS can be made are the size of the display and the keyboard. The minimum practical size available with currently available products is approximately 9 inches (23 cm) by 4 inches (10 cm) by 1 inch (2.5 cm). This is the average size of most of the hand-held computers today, such as those made by Panasonic, Radio Shack, and IXO [6 and 7]. These systems tend to weigh around 14 ounces (400 gm). Their size seems to be the smallest practical one in order

to keep the keys far enough apart to minimize the chances of hitting the wrong key or hitting two keys at once[6]. It is doubtful that the display will be shrunk; if anything, future displays will be larger and allow smaller fonts, thus allowing more information to be shown. Ultimately, it could be possible for the display to dominate the front of the PDBMS if voice input were incorporated. This would most certainly require a large display because function keys would probably not be used (or even desired) and the system would be expected to echo all vocal input so that the user could verify that he had been correctly understood.

The back of the enclosure opens to allow batteries to be changed and E²PROM to be added in or taken out. This last feature would not only allow the user to expand his memory (or treat it like a floppy disk, i.e., interchangeable secondary storage), but also allow the transportation of software and data from one PDBMS to another by a means other than through the RS232 port. The hardware and software of the first prototype do not include an ability to add more E²PROM, but the required modifications are minor.

It should be mentioned that the current implementation of Keys does not gracefully support the transportation of sealed objects from one system to another by physical transportation. There is no way to guarantee that security would be uniformly enforced, independent of the system in which the objects are found, because key assignments are local in context.

---

[6]The size of the keys is really unimportant so long as the user feels comfortable using them. This normally is taken to mean that the keys should not be physically uncomfortable to use and they should provide some sort of tactile and audible response upon being struck.

2. The Display

The current display is an LCD which contains two rows of 20 characters each. This is larger than the displays in most of the currently available hand-held computers. These normally have one row of 16 to 20 characters. It was felt that two lines were the minimum acceptable number of lines for the PDBMS. Two lines allow user commands and responses to appear on one line and the system responses and prompts to appear on the other. This allows the user to compare his commands and responses with the system's. Ideally the PDBMS should have a larger display. The largest LCD displays available at this time have four lines with 40 characters per line, however these are too expensive to be compatible with cost criteria of the PDBMS[7].

3. The Keyboard

Most of the keys should be 3/16 inch (0.5 cm) square and protrude from the keyboard background by 1/8 inch (0.3 cm). The keys are separated by 1/4 inch (0.6 cm). These dimensions are used on most of the Hewlett-Packard calculators for the arithmetic keys (i.e., + - + x). Using them as an example, the author found that keys were easily differentiated from one another, and two or more keys were almost never pushed simultaneously. The keys should be arranged by function with the background colored differently for the letters, numbers, and special function keys, similar to what was done on the Quasar and Panasonic computers [6]. The on/off switch should be away from the other keys and be a sliding switch, not a push switch. This should be done to

---

[7]LCD is the only flat display technology presently available which is power efficient enough to be used in a good battery powered system. LED and plasma displays are much less power efficient.

help prevent the accidental switching on or off of the power.

The letter keys should be arranged in the standard "QWERTY" format, not only because of the entrenched place in the English speaking world [1], but also because it has been found to be more effective than previously thought relative to some keyboards designed using human engineering principles, especially with novice users [8]. At the present only upper-case letters are planned to be provided to the user for text entry. Below is a list of the keys and their functions.

a. Letter and Digit Keys

These keys act in the usual and expected fashion; they are used to enter the ASCII representation of the desired character. Input from these keys is handled as it normally would be in any FORTH system. The letter keys may also be used as "function keys." When shifted, using the shift key, the ASCII code for the key's lower-case equivalent is generated. These "illegal" characters are treated similarly to LaFORTH words; that is, they are interpreted immediately upon input [9]. Initially the function accomplished by these words is to place into the input message buffer and the LCD window the ASCII string representation of other words; they do not appear in the input message buffer or on the LCD*. For example, in the database management application a shift-G causes the word GET to be placed in the message buffer and the LCD window so when the return key is eventually pushed, WORD will find GET in the buffer, not shift-G. Notice that the keys may perform different functions depending upon the current vocabulary.

---

*When they must be displayed, as in their colon definitions, they are displayed in "reverse video."

b. Mathematical Keys

These keys are similar to the shifted lettered keys, however they act as input immediate words without shifting them. That is, they always cause a search of the current vocabulary. This was done so that the user can choose to use either infix or postfix notation (infix notation is the default definition of these keys in the "naive" calculator vocabulary). These keys include the following five keys:

\+ - x ÷ %

c. Special Function Keys

These keys are the usual terminal editing keys, and with the exception of the "NEXT" keys, they are not programmable. The keys are described below.

(1). Enter. This key causes a carriage return and line-feed to be placed into the input which is reflected upon the LCD. This causes the interpreter to begin parsing the input.

(2). Del. This causes a control-H to be input and acts as a character deletion key. It backs up the cursor one position and displays a space on the LCD.

(3). →. This moves the cursor to the right one character position without effecting the contents of the LCD window or the message buffer.

(4). ←. This moves the cursor to the left one character position without effecting the contents of the LCD window or the message buffer.

(5). Shift. This is a non-locking shift key used with other keys to elicit their alternate definitions.

(6). X≥. This deletes all input from, and including, the current cursor position to the end of the line.

(7). NEXT↑ and NEXT↓. These keys are used to scroll the display to the next line above or below, respectively. In the database application, the shifted NEXT keys are used to scroll to the next field above and below the current field. This allows fields to include carriage returns and line-feeds so that a field need not be constrained to one logical line on the display.

B. THE SOFTWARE

When the user initially receives the system, he is presented only with two functions: a calculator and a database manager. He does not have direct access to ROOT. This was done to help prevent the user from inadvertently destroying the system before he understands it. For example, it prevents him from redefining or forgetting a word accidentally. The user can expand the scope of the system gradually as he learns more about it until he can, if he chooses, run it strictly in FORTH (or even redesign the system to a great extent). This flexibility is gained by using FORTH execution vectors. In the case of interfacing with different levels of users, there is a different version of FIND for each level of user sophistication. So as the user becomes more adept with the system, the vector associated with FIND is simply made to point to a new, more powerful version of FIND's run-time code. The version initially available to the user only searches the limited calculator and database management vocabularies; the ROOT vocabulary is not searched. The version available to the most sophisticated user includes a modified version of the standard FORTH FIND. All FINDs have been modified to be a little more user friendly. Instead of reporting the usual, "IS UNDEFINED," when a word is not found, the PDBMS reports the current vocabulary's name as well. So for example if

the user entered a {:} when he was using the database vocabulary where it is undefined, the system would report, "NOT DATABASE WORD." Notice that this message may fall off the right-hand side of the display for some words; but the first word of the message should cue the user to the error and if he then realizes that he has forgotten what the current vocabulary is he can move the display to the right using the cursor control keys.

There is no editor in the "initial" system because all of the needed functions are available through the keyboard keys, making the PDBMS a full-screen editor, albeit a small screen editor. There is an editor vocabulary which is defined in the PDBMS after ROOT and ASSEMBLER. This editor is only needed once the user has begun working directly with screens. Table 5.1 shows the vocabulary structure of the PDBMS. The concept of sealed vocabularies[9] is employed; however notice that some words link one vocabulary temporarily to others. For example, SEAL causes a search of the Key vocabulary. SEAL and UNSEAL are defined in the DB vocabulary to be themselves (i.e., they simply point to their definitions in ROOT). This allows them to be used by the naive user without directly accessing the root vocabulary. E²PROM permanent vocabularies (i.e., Key, file, and virtual block) are not linked through each other or those vocabularies defined in RAM. Thus FORGETting a definition in RAM which precedes a file, block, or key definition will not erase any E²PROM definitions[10].

---

[9]These are vocabularies which confine word searches to themselves, and usually FORTH. The FIND used in fig-FORTH searches all parent vocabularies of the current vocabularies. The calculator and database vocabularies are totally sealed in that not even the root vocabulary is searched.

[10]A sometimes problematic feature of standard FORTH is that all definitions are actually maintained in one straight linked list; vocabularies only describe search paths through the one list. The traditional FORGET simply deletes all definitions created after the definition to be

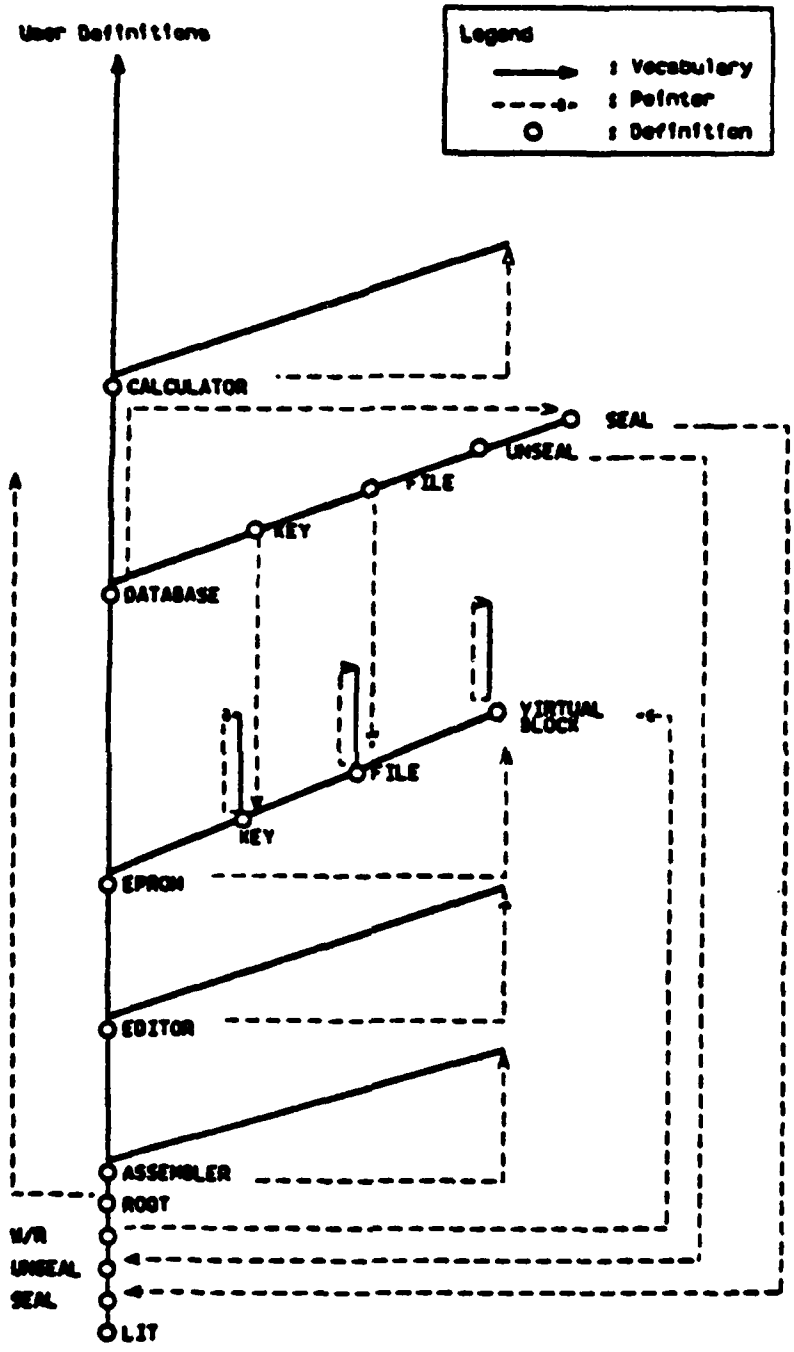


Figure 5.1 PDBMS Vocabulary Structure.

1. The Calculator

Initially the calculator is entered by pushing shift-C. This places the user into the calculator context whose vocabulary contains redefinitions of +, -, x, and ÷ so that they are infix operators. FIND has been modified so that if a word is not found and an equal sign has been previously interpreted, a constant is created. This allows the user to store temporary results by creating "variables" simply by using an undefined word. For example,

```
1 + B = A
```

would cause "A" to be created. If "B" had not been previously defined an error condition would be raised when it was not found in the dictionary. The equal sign is an input immediate which causes "A" to be created, if need be, and sets up an execution vector to cause the ENTER key to store the top of the stack into "A."

Because a derivative of FORTH is used, floating point arithmetic is not used. The system defaults provide the user with a fixed two digits behind the radix point. Like FORTH, the user may choose any base (radix) for arithmetic operations, within the limits of the number of input symbols available.

2. The Database

Initially the database management system is entered by pushing shift-D. This vocabulary allows users to create files, create records, retrieve records, update records, delete records, and delete files. Additionally the user may

---

forgotten—even if they are not in the current vocabulary. When there are multiple vocabularies, this can create dangling pointers in vocabulary definitions.

create and delete Keys, and use Keys to lock records and other Keys.

a. Keyboard Key Definitions

When the user is placed into the database context the NEXT keys are redefined as described before. Besides those two keys, the following shifted characters are defined. These keys are described below. The word which appears on the display and in the input message buffer when the key is pushed is shown in parentheses.

(1). D (DELETE). This is used to delete a file, record, or Key. There are three different DELETEs, one in each the DB, file, and Key vocabularies. Each delete effects only those elements in its respective vocabulary. The delete in the file vocabulary deletes files, the one in the Key vocabulary deletes Keys, and the one in the DB vocabulary deletes the current record.

(2). F (FILE). This word changes the context for the interpretation of the words following it in the input stream so that the file vocabulary is searched. The context is reverted to the DB ("calling") vocabulary when the first word not found in the file vocabulary is encountered. The last filename mentioned before the context is switched out of the file vocabulary becomes the "current file."

(3). G (GET). This is used to initiate a record retrieval. Table III shows a typical record procedure. First the user is asked if the current file is the one to be searched, or asked for a file if there is no current file. Then the user is presented with the names of the fields of the records in the file so the user can enter values which are to be used as key attributes for retrieval. If the user does not desire to enter a value for a particular field, he simply presses the ENTER key. The query in

Table III is a request for any record in the ADDR-BK file which contains "TABETHA" in its NAME field and "MONTEREY" or "VA." in its CITY/ST field. Before actually performing a retrieval operation, the user is asked if he still desires to do the retrieval allowing him to abort a query if he has realized that he has made a mistake.

TABLE III
Record Retrieval

```
GET
FILE ADDR-BK?
YES
NAME?
TABETHA
STREET?
<enter>
CITY/ST?
MONTEREY  VA.
PHONE?
<enter>
MISC?
<enter>
GET?
YES
1 RECORD FOUND
PUSH NEXT
```

(4). H (HIDE). This is used to make a Key which has been made known through a UNSEAL operation, unknown.

(5). K (KEY). This word changes the context for the interpretation of the words following it in the input stream so that the Key vocabulary is searched. As with the shift-F, the context reverts to the calling vocabulary when the first word not in the Key vocabulary is encountered. This word does not effect any Keys or the Key vocabulary, it is only used as a prefix word for MAKE and DELETE.

(6). M (MAKE). This word, like DELETE exists in the DB, file, and Key vocabularies. Each different version creates a record, file, and Key respectively.

(7). N (NO). This is used as an answer to appropriate system prompts.

(8). P (PUT). This is analogous to SAVE-BUFFERS and FLUSH in that it writes the current record to secondary storage.

(9). R (RECORD). This word is included for consistency reasons. It is used to preface DELETE and MAKE when the user wishes to use the *DB* definitions of these words. The DB DELETE and MAKE must be prefaced by RECORD so that there is less chance of an accidental record deletion.

(10). S (SEAL). This is used to seal a Key or the current record. It is simply defined as:

```
: SEAL ROOT SEAL ;
```

This allows the user access to the root word SEAL without directly accessing the root vocabulary.

(11). U (UNSEAL). This word is used to unseal all objects sealed with one or more Keys. It, like SEAL, is simply defined in terms of the root word UNSEAL.

(12). Y (YES). This is used as an answer to appropriate system prompts.

b. File Creation

Files are created simply by using the words FILE and MAKE. Upon entering shift-F (or FILE) and shift-M (or MAKE), the user needs only to follow the system's prompts. Table IV shows the file creation sequence. The user's input is underscored. The user always gets an additional field called "miscellaneous" added to the bottom of all records. This is included because it was found that people's personal data does not normally fit a uniformly structured record.

c. File Deletion

File deletion is simply effected by the sequence shown in Table V. File deletion is not a trivial matter since the E²PROM is organized as a heap with physical records containing a mixture of sealed Keys, DB dictionary entries, and records from various files. First of all, a user cannot delete a file unless he has unsealed all of the records in it, so DELETE must make one pass of all the records in the file to ensure that they are all unsealed. If all of the records are unsealed, then a second pass is made of the records reallocating all of the physical records back to the system (i.e., setting their corresponding bit to zero in the record bit map). Additionally, on this pass the first byte of each physical record is set to 80H (the system's Key) while the second byte is set to FFH (the null Key). Then the DB dictionary must be searched for all references to the deleted field numbers, and these must be removed. When a field reference is removed from a wordd's list of field IDs, the hole created by this deletion is filled by moving the last entry on the list up to the vacated spot. Physical records vacated by this operation are returned to the system. Finally the file's vocabulary and its field entries can be forgotten. Obviously file deletion is a lengthy and complicated process.

TABLE IV

File and Key Creation

File Creation

FILE MAKE
NAME?
ADDR-BK
FLD 1 NAME?
NAME
FLD 2 NAME?
STREET
FLD 3 NAME?
CITY/ST
FLD 4 NAME?
PHONE
FLD 5 NAME?
<enter>
FLD 5 MISC OK

Key Creation

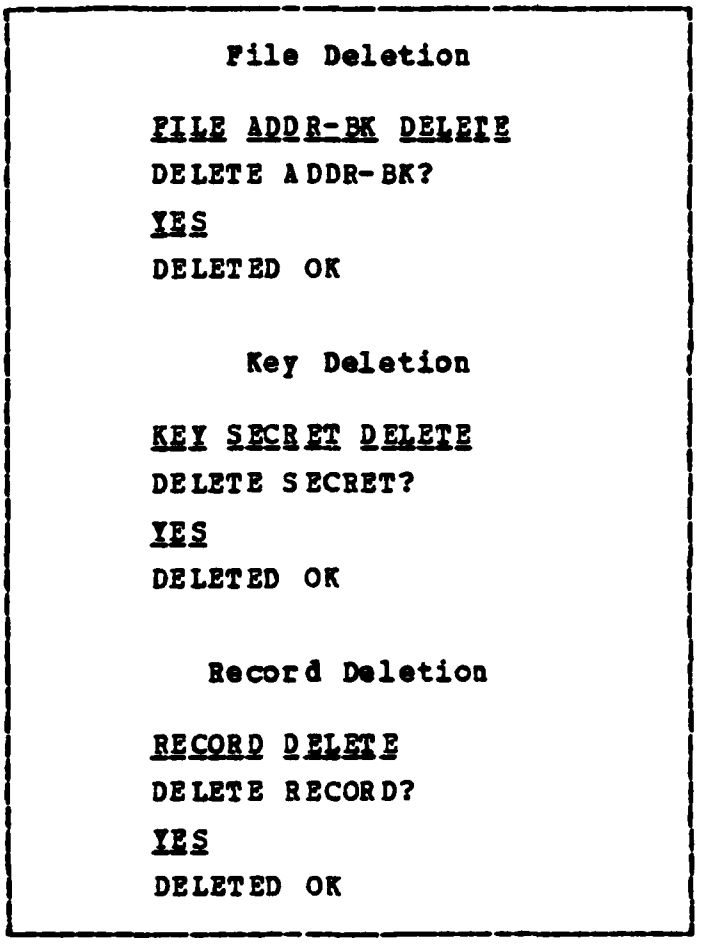
KEY MAKE SECRET
OK

d. Key Creation

Creation of a Key is very simple, as shown in Table IV. The example shows the creation of a key named "SECRET." All that is required to create a Key is the addition of "SECRET" into the Key dictionary as a constant and initializing it to the next available Key ID number.

TABLE V

File, Key, and Record Deletion



e. Key Deletion

Key deletion is accomplished in the same manner by which files are deleted, as shown in Table V. Also like file deletion, the mechanics of Key deletion are not the equivalent to a straightforward FORGET. Before a Key can be deleted from the dictionary, all occurrences of the key in the various access descriptors must be located and changed to reflect the Key's deletion. This entails searching the access descriptor of each screen, record and sealed Key and

converting the deleted Key's ID to FEH (the deleted Key ID). After this is done the Key is deleted from the dictionary. A sealed Key's physical record is returned to the system, after setting the first byte to 80H (the system Key) and the second byte to FFH (the null Key).

f. Record Creation

To the user record creation dialogue is similar to the one associated with file creation. What is involved is collecting the desired data, encoding it[11], finding physical records to hold the logical record, and finally linking the record into the parent file's linked list of records. Currently the linked lists of records are maintained in chronological order (i.e., as a circular queue). This may be frustrating in some applications where the user would like to peruse the database in some specified order. For example, it is not possible to view the records of an address book alphabetically by surname, unless they were originally entered in that order. Because of the unformatted nature of the fields, it is very difficult to sort a file by key attributes.

It would not be too difficult to allow the user to specify a record ordering other than chronological. This could be done by allowing the user to flag a wordd in the record as the sort-key-value (for example the last word in a record starting with the character "@"). Then when the record was PUT into the database, it would be inserted into the file's linked list alphabetically relative to the other "@-wordds" in the file's other records. So the user could

---

[11]This includes converting the uwords to punctuation and wordds, and then the addition of the wordds into the DB dictionary.

maintain the file sorted by surname by prefacing all surnames with a "@"[12].

TABLE VI
Record Creation

```
RECORD MAKE
NAME?
JOHN DOE
STREET?
3249 15TH ST NW
CITY/ST?
PORTLAND, ORE.
PHONE?
<enter>
MISC?
<enter>
OK
```

Table VI shows a typical record creation sequence. Notice that no phone number was given; a null entry is signalled by hitting the ENTER key. Also notice that there is an implicit "current file." This file is the last one referred to after the last use of FILE; had no file been explicitly referenced before a record creation was attempted, the PDBMS would have requested a file name. If the file was not found, the user would have been asked if he desired to create a file or abort the record creation.

---

[12]This may not appeal to many users, but it would not necessarily have to appear in the name field. The "@-surname" could be placed in the "miscellaneous" field.

g. Record Deletion

Record deletion is requested by the user in the same fashion as file and key deletion. Record deletion involves first removing the record from the file's linked list by making the two records adjacent to the current record point to each other. These links are found in the current record's previous and next link bytes (see Figure 4.5). Then all of the wordd references to the record in the DB dictionary must be deleted. Finally the physical records are returned to the system after setting the first byte to 80H and the second to FFH.

h. Update

Only records may be updated; files and keys cannot. Records are simply updated by GETting them, modifying them using the cursor control keys, and then PUTting them back. Like FORTH, once a change has been made to a record, it is marked as being updated, whether or not the change is later undone in the same editing session. Once a record has been marked as updated and it is PUT, the updated record is added as a new record, and the old record is deleted. This is not quite as drastic as it may sound. The old record is used as a template for encoding the new record. Wordds which are unchanged can be copied from the old record directly into the new record. The old record also contains all of the pointers into the DB dictionary where new virtual addresses must be substituted, so the dictionary must be searched only when a new wordd is added. Record update is actually a record creation and deletion operation.

It could be possible to allow file editing (i.e., the addition and deletion of fields) by performing the same type of operations as are employed in record update

(i.e., creating a new file, transferring the appropriate data from the old file into the new file, and then deleting the old file). However, this was considered too complicated and slow to justify its inclusion for what would probably be a rare event. Besides, by always including a "miscellaneous field" in all records, it was felt that this would probably not be a very necessary operation.

VI. SYSTEM SECURITY DESIGN

As stated earlier, security is important in a PDBMS because of the personal nature of the information it may contain. However, the type of security afforded in this design is probably better suited for a larger system. Probably all that is required for such a system as the PDBMS is a simple mechanism which employs one Key or password. This allows the user to hide anything he desires at one level of security (i.e., one either has access to all of the data or has access to only a subset of the data). The PDBMS uses a much more elaborate system. This was done to test two things: the feasibility of securing FORTH, and the feasibility of implementing a security mechanism similar to the one described in reference [10]. FORTH was chosen as the language to implement the PDBMS with no firsthand knowledge of the language. Because it is an interpreted language, it was felt that there would be no problems with securing the system. However, after receiving the FORTH documentation and software many doubts were raised about whether the language could be secured.

At first one thing which seemed essential to securing the PDBMS was the restriction of the user's ability to use assembly language. If the user can write words in assembly language using physical addresses and ports (the only way to write such words on the NSC800 since it does not support segmentation and privileged modes) there is almost no limit to what he can do. All standard FORTHs are very close to the hardware and allow words to be written in assembly language, besides FORTH. As a matter of fact, it is so close to the machine, that in 8080 fig-FORTH and FORTH-79, it is impossible to prevent the programmer from writing

assembly language defined words without changing FORTH to such an extent that it is no longer the same language. In these two systems, the words which are used to specify code definitions (;CODE, CODE, END-CODE, and {;S}) are all high-level words (i.e., words written in FORTH as contrasted to low-level words which are written in assembly language), as is the assembler. As far as the author can determine, there is no low-level word which can be "hidden" from the user without having a detrimental effect and which is required for entering assembly language defined words.

The word "hidden" is enclosed in quotes in the previous paragraph because no word can be hidden from a user in his address space. "Hidden" means that the user neither knows of the hidden word's existence or doesn't know where to find its definition, nor can he execute it directly. A word in FORTH which can be located can be executed even if it is not in the FORTH linked list word dictionary (one simply puts the address of the first executable byte onto the parameter stack and evokes EXECUTE). If a user is to be allowed to program in FORTH, he must be allowed to access words in the ROOT dictionary, and in order to access words, their names must appear in the dictionary since FORTH searches the dictionary by name. This makes it very easy for a user to traverse the dictionary and look at its contents and at the location of words. It would not be hard, though probably tedious, to find a word not included in the dictionary by checking for unaccounted gaps between words in the linked list or finding a reference to a code field address of a word which does not appear in the dictionary. If one were to seriously consider hiding words, the best way to do this would be to remove all of the headers (the name and link fields) from all of the dictionary entries. Such a system could not be extended because no words in the dictionary could be found (since the name and link fields are necessary

to search for a word). If the PDBMS was to be secured there had to be another method which either prevented the use of assembly language or worked regardless of the fact that the user could use assembly language.

In the PDBMS, FORTH could possibly have been secured entirely by using software and still allowed the user to program in FORTH, however it would have undoubtedly been a very limited subset of the language. Such a system would have not needed EPROM; instead a cold boot could have loaded the system in from E²PROM. Verifying such a system would have surely been a problem. Instead the PDBMS relies on both hardware and software to enforce system security.

A. HARDWARE SECURITY MEASURES

In multi-user systems hardware support of security is essential; in truly secure systems it must be verified that there are parts of the system that no one but system administrators can access. In the PDBMS the hardware and software enforce security to such an extent that even the owner of the system cannot access parts of the system at all[13]. This is desirable because it not only prevents other persons who are not the owner of the PDBMS from compromising or destroying the system, but it also prevents the user from "terminally crashing" the system. Many of the system's boot parameters are stored in EPROM and E²PROM. If these were lost, the system could not be booted up.

It is the interaction of the EPROM and the "smart ports" which is the hardware portion of the system's security. Simply, the ports which control access to virtual memory, the keyboard, and the RS232 port only accept instructions

---

[13]The PDBMS has not been proven correct and secure in the sense of the ways described in references [11 and 12]. However, the author believes that it can be made secure and rigorously proven to be so.

executing from EPROM, as discussed in Chapter IV. Because EPROM is read-only, the user is forced to use procedures in it to access these external devices. Thus if the procedures in EPROM can be verified that they are not only correct, but they are also unsubvertable, then the PDBMS can probably be made secure[14].

B. SOFTWARE SECURITY MEASURES

The hardware in itself does not guarantee a secure system; there must be some verified software which operates it. There are three different aspects of the software in the PDBMS which are used to provide security. A fourth aspect is mentioned here which is related to security but is not involved in system security per se. The first three items are: straight-through code, maintenance of system parameters and tables in E^2PROM, and Keys. The fourth item is the FORTH concept of execution vectors.

1. Straight-through Code

Contrary to FORTH programming style, words which are involved with port access must be low-level and indivisible. This means that these words must not be defined in terms of other words, i.e., they cannot be colon definitions, they must be code definitions. For example, it seems obvious that one would like to write the following low-level words for use in other system management words because they would be very commonly used:

---

[14]A correct procedure is one that does only what it is designed to do; nothing more and nothing less. Unsubvertability is a stronger condition than correctness in that it means that even combinations of modules of correct code and portions of modules cannot be caused to be made to interact incorrectly. This is a concern in the PDBMS since the user can read and execute the system's source machine code.

```
E²PROM_ON          ( Turns E²PROM power on )
E²PROM_WRT_ON      ( Turns E²PROM write power on )
WRT_E²PROM         ( Initiates an E²PROM write )
E²PROM_WRT_OFF     ( Turns E²PROM write power off )
E²PROM_OFF         ( Turns E²PROM power off )
```

However, as mentioned before, if a word exists in the user's address space, he can find it and execute it. This means the user could find E²PROM_ON and E²PROM_WRT_ON, and execute them from EPROM. Then using his own assembly language routines, he could manipulate the contents of the E²PROM. The only way to prevent this is to create a minimum set of virtual memory management words which, once execution of any one of them begins, never branches out of the word or returns to the inner interpreter without first turning off access to the ports. Also these words should be written so that if the user jumps into the center of their code, they are still correct.

The first requirement is fairly easy to achieve because these words are resident in EPROM, thus because they cannot be altered, if a user jumps to, or into, them it can be assured that he cannot effect the execution of the words. The second requirement is much more problematic. Satisfying this means that the actions of these code sequences can maintain system security regardless of the actions performed before and after their execution, and regardless of whether the entire sequence is executed (i.e., the user jumps into the middle of a code sequence). For example, the user must not be able to use the code of one word (whether it is the entire code sequence or a part of it) to set up the segment register to point to the Key dictionary, and then by using another word, retrieve the Key dictionary.

2. Maintenance of System Parameters and Tables in E²PROM

By controlling access to E²PROM it is possible to use parts of it to store information which the user should not have access to. Chapter IV discusses the information which is stored out in E²PROM which is not accessible to the user. The locations of the parameters and beginnings of these tables are static so that they may be referred to directly by using their segment number and E²PROM addresses (FF00H through FFFFH). These references are found in EPROM where they are visible to the user. The insurance that the user cannot directly access these segments must be incorporated into the design of the straight-through code. The code must be written so that when control is passed from the word to the inner interpreter, the user is left with no more information about the tables and parameters than he is authorized access to. Any routines which do system table and parameter maintenance are designed so that they work directly on the E²PROM and never bring the contents of these segments into RAM. This makes it easier to ensure the security of system segments.

The above is not entirely true of the PDBMS. During retrieval operations, virtual addresses are brought into the data buffers. Thus the user can gain some information about the maintenance of the system's segments by dumping the contents of these buffers. This information is kept in RAM because it is a "write-intensive" operation. Additionally it must be left in the buffers after the system is finished with processing the query because the virtual addresses must be used to find the records which satisfy the query conditions. The current record's virtual address is needed so that if it is updated the location of the old record can be found and deleted. Thus the user can gain access to the

virtual addresses of records to which he is authorized. Allowing the user access to the virtual addresses of all of the records which satisfy a query gives him some information from which he can make inferences about the allocation of physical records, including those to which he is not authorized access. How much information can be gained through inference seems to be limited by the fact that the segments in which these records occur contain not only records (which can use varying numbers of physical records), but sealed Keys and DB dictionary entries (which also use varying numbers of physical records). Additionally if any deletions or updates ever occurred, the physical records may no longer be allocated in a sequential and chronological manner. Thus in a mature (i.e., one which has processed a number of Key and record additions and deletions) system, it is questionable that much meaningful inference can be done. Of course, the problem can be avoided entirely by keeping all of these virtual addresses in E²PROM at the expense of system speed and possible E²PROM "burn-out."

3. Keys

The proper implementation of Keys relies heavily upon the preceding hardware and software base. Keys are very simple--nothing is fetched from E²PROM unless the proper Key(s) has been UNSEALed (or made known). The operations associated with SEAL and UNSEAL effect the Key dictionary but have no effect upon sealed objects. As mentioned earlier, Keys are maintained in a dictionary as constants. When a Key is UNSEALed, the high bit of its character count byte is set to one. When a data object fetch is requested, the object's access descriptor field is "computed" to see if the requisite Keys have been previously made known.

The access descriptor fields are limited to the first physical record for screens (15 Keys), 15 Keys for a sealed Key (one physical record less one byte for the sealed Key's ID), and no limit for database record (since they are permitted to cross physical record boundaries). However for consistency, from the user's point of view, 15 Keys is the limit for all system objects. The Keys may be "anded" and "ored" with each other to form complicated access mechanisms. This may be further extended by adding layers of sealed Keys. For example if access to the current record required the Keys "CONFIDENTIAL" and "ACCESS," or the Keys "SECRET" and "ACCESS," the current record could be sealed as follows:

KEY CONFIDENTIAL ACCESS & SECRET & | RECORD SEAL

or

KEY CONFIDENTIAL SECRET | ACCESS & RECORD SEAL

where "&" is a logical "and" and "|" is a logical "or." If CONFIDENTIAL's ID was one, SECRET's two, and ACCESS's three, and the second example above had been used to seal the record, then the record would have four key bytes which would contain:

01H 02H 83H FFH

Notice that the high bit of ACCESS's ID was set to one. This signifies that it is to be "anded." A zero high bit signifies the Key is to be "ored." Unique "access paths" are described in both the SEAL process and the access descriptor because they are specified using reverse Polish notation.

When an attempted fetch of a record is made, the fetch algorithm starts by setting a fetch flag to true (the value one). Then it simply reads each Key ID from the access descriptor and searches the Key dictionary to see if the Key is known (i.e., the high bit of its character count is set to one). If the Key is known, the search returns a one, otherwise a zero. The result of the search is "anded" or "ored" with the fetch flag according to the high bit of the byte in the access descriptor. When the null Key is found in the access descriptor, the value of the fetch flag determines whether the object is sealed or unsealed.

Since the Key dictionary entries are maintained as a FORTH dictionary and FORTH dictionaries are searched by name, it may seem that searching the dictionary using the Key's ID may be difficult. It is, in fact, faster than searching by name. This is because of the structure of the dictionary entries which allow the Key's value to be retrieved easily because it is located in the byte immediately following the CFA. Searching by name is slower because it involves string comparisons.

At the root of the Key dictionary (i.e., that entry whose link is equal to 0000H) is the definition of MAKE. Below MAKE are all of the other colon definitions in the Key vocabulary. After the last colon definition is the definition of the system Key. This is a constant like the other Keys but its value is 80H and its count byte contains a 00H. This means that its name's length is zero, and thus it has no name and cannot be found by a name search of the dictionary. Because it cannot be found, it can never be UNSEALed or made known, so the high bit of its character count will always remain zero. Below the system Key are the definitions of the null Key and the deleted Key. These Keys' values are FFH and FEH respectively and their character count bytes are equal to 80H. This means that they

also have no name and they always remain UNSEALed or known. Because these three Keys' values are greater than 127, they are always "anded" into any Key ID list in which they appear.

Changing a deleted Key's ID number wherever it occurs in an access descriptor list results in a "sensible" condition. That is, all other Keys are still required in their same logical relationship except that Key (or relation) which preceded the deleted Key which now takes the place of the relation between itself and the deleted Key. A major problem with deleting a Key is that the user may not realize the data objects which he is effecting or how he is effecting them. This is an unresolved problem in the PDBMS and it is more complicated than it appears on the surface.

Finally, there is one last important operation which concerns maintenance of the Key dictionary: making Keys unknown. The user can make Keys unknown on an individual basis by using HIDE. For example,

KEY SECRET HIDE

makes "SECRET" unknown and seals any objects which are sealed with SECRET. Whenever an non-maskable interrupt is generated, the virtual memory manager makes all Keys whose character count is greater than 80H unknown.

4. Execution Vectors

Execution vectors are used in the PDBMS to allow the user to interact with only that part of the system which he understands. However, they can be used to provide system security to an extent. Simply, if a user does not know how to change a vector's value (or a collection of vectors) or what value to change it to, the situation is similar to needing a password to access a more powerful system. At the

lowest level it is easy to prevent a user from using more of the system than is desired. If the user is constrained to a vocabulary which does not contain words which would allow him the make colon definitions (e.g., {:}) or access memory directly (e.g., {!}, {@}, etc.) the inner working of the system can be hidden from him. Making a user more privileged simply means giving him the name of a word which changes the values of the execution vectors (of course this word cannot appear in a listing of the vocabulary). As the system to which the user gains access becomes more powerful, it becomes progressively harder to provide system security by using execution vectors without relying upon hardware.

APPENDIX A
THE LANGUAGE FORTH

A good description of the concepts upon which FORTH is based may be found in reference [13]. FORTH is a stack-oriented, threaded, interpretive language. It is noted for its compact size and fast execution (compared to other interpreted languages such as BASIC). The 8080 fig-FORTH model (version 1.3) occupies less than 9K bytes of memory (which includes the first page of memory occupied by CP/M). Residing in that 9K is the FORTH interpreter, compiler, dictionary, and a line editor. There are two "generic" FORTHs. The older version is usually referred to as "fig-FORTH," the newer version is usually referred to as "FORTH-79." FORTH-79 was designed to be a standard which establishes the minimum requirements of the language. Specifically reference [2] states that the purpose of FORTH-79 is

> ... to allow transportability of standard FORTH programs in source form among standard FORTH systems. A standard program shall execute equivalently on all standard FORTH systems.

The bibliography contains a list of sources used by the author while learning FORTH. Anyone who seriously desires to understand the language should have at least some of these books and pamphlets.

A. WORDS

The basic unit of the language is a "word." Words can be "colon definitions" (analogous to functions and procedures in other languages), variables, and constants. New

words are defined in terms of previously defined words, making the language extensible. Defined words are kept in a linked list called the "dictionary." The dictionary is maintained as a stack (First-In-First-Out or FIFO) so that the newest words are searched first. Thus previously defined words can be redefined. Dictionary entries are pruned by using the word FORGET. When a word is "forgotten," all words defined after it are also forgotten. Rather than a straight linked list, the dictionary can be extended in a tree structure where branches denote different contexts. Table VII is a list of the FORTH-79 required words. The words in lower-case are dictionary entries for the run-time code for the corresponding compiling word.

B. SYSTEM DATA STRUCTURES

Figure A.1 depicts the standard FORTH memory organization. The user dictionary grows up towards high memory while the parameter stack grows down towards the dictionary. The unused portion of memory separating the two is called the pad. The beginning of the pad moves up in memory with the dictionary pointer (DP). It is usually located 44H bytes in front of the DP. Likewise, the input message buffer grows up in memory according to the size of the input message while the return stack grows down towards the message buffer.

The parameter stack is used for mathematical data manipulations and parameter passing. The data on the stack is operated upon using reverse Polish (or postfix) notation, similar to Hewlett-Packard calculators. The return stack is used by FORTH for storing the interpreter pointer (the address of the next higher context, i.e., the calling word). The pad is used primarily for string manipulations. System variables are those variables maintained and used by FORTH

TABLE VII

FORTH-79 Required Word Set

Nucleus Words

!	*	*/	*/MOD
+	+1	+loop	-
/	/MOD	0<	0=
0>	1+	1-	2
2-	<	=	>
>R	?DUP	@	ABS
AND	begin	C!	C@
colon	CMOVE	constant	create
D+	D<	DEPTH	DNEGATE
do	does>	DROP	DUP
else	EXECUTE	EXIT	FILL
I	if	J	LEAVE
literal	loop	MAX	MIN
MOD	MOVE	NEGATE	NOT
OR	OVER	PICK	R>
R@	repeat	ROLL	ROT
semicolon	SWAP	then	U*
U/	U<	until	variable
while	XOR		

Interpreter Words

#	#>	#S	'
(	-TRAILING	.	79-STANDARD
<#	>IN	?	ABORT
BASE	BLK	CONTEXT	CONVERT
COUNT	CR	CURRENT	DECIMAL
EMIT	EXPECT	FIND	FORTH
HERE	HOLD	KEY	PAD
QUERY	QUIT	SIGN	SPACE
SPACES	TYPE	U.	WORD

Compiler Words

+LOOP		."	:
;	ALLOT	BEGIN	COMPILE
CONSTANT	CREATE	DEFINITIONS	DO
DOES>	ELSE	FORGET	IF
IMMEDIATE	LITERAL	LOOP	REPEAT
STATE	THEN	UNTIL	VARIABLE
VOCABULARY	WHILE	[	[COMPILE]
]			

Device Words

BLOCK	BUFFER	LIST	EMPTY-BUFFERS
LOAD	SCR	UPDATE	SAVE-BUFFERS

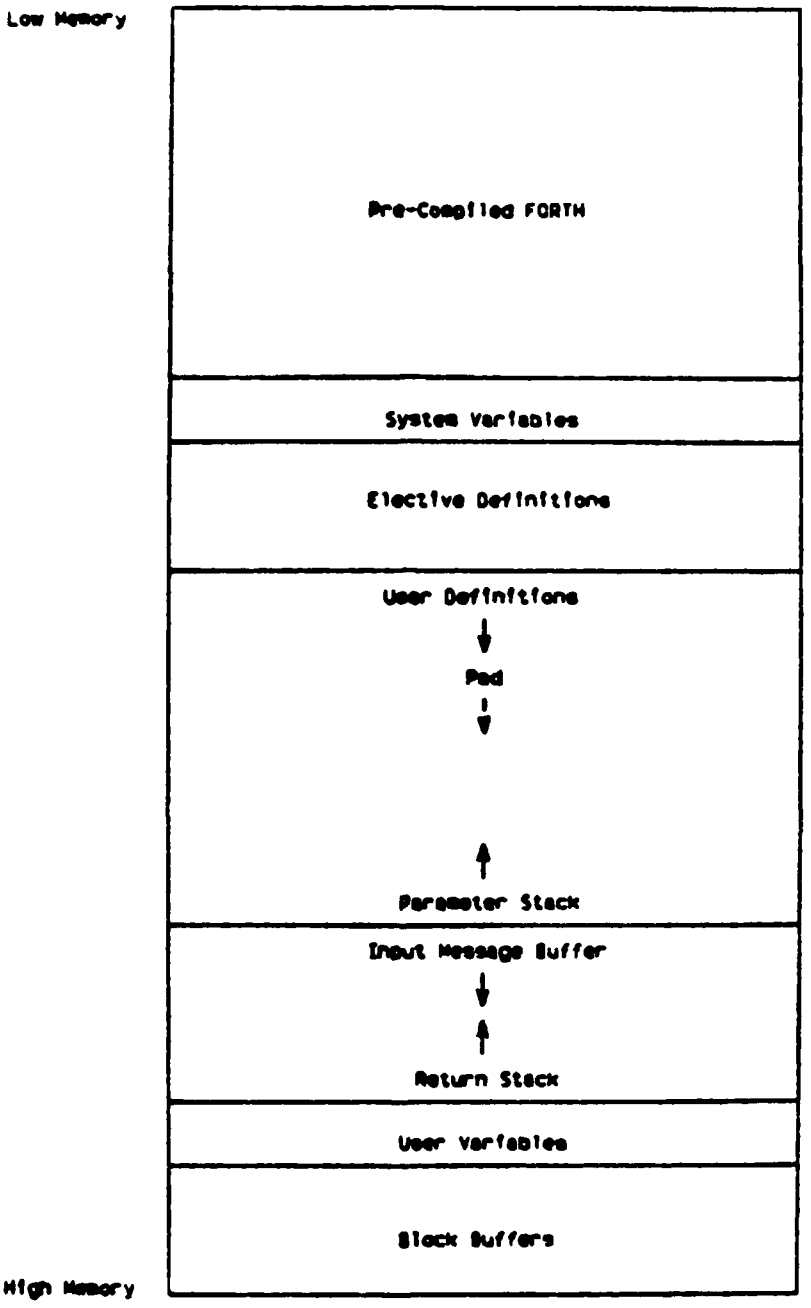


Figure A.1 Standard FORTH Memory Map.

and not directly accessible to the programmer. User variables are declared, maintained and used by the system, but are directly accessible to the programmer. Examples of system variables are cold boot parameters and CP/M disk interface parameters while examples of user variables are the dictionary pointer the current radix (called BASE), and the current execution state (called STATE).

The number of block buffers is dependent upon the amount of physical memory available. Standard FORTH blocks are 1K bytes in size and are stored in secondary storage, thus giving FORTH what its users call virtual memory. FORTH automatically allocates buffers as they are needed according to which buffers have not been allocated yet, the age of the contents of occupied buffers, and whether any buffers contain updated data. Blocks containing FORTH "programs" are commonly referred to as "screens" because they are formatted for CRT display; i.e., 16 lines of 64 characters.

C. THE MECHANICS OF FORTH

There are less than 70 assembly routines in FORTH-79, most of which are less than 20 instructions long. When FORTH words are interpreted, it is these routines which ultimately are executed, except in the case of user code defined words. All words in FORTH contain a code field address (CFA) which is a pointer to an assembly language routine which defines the word's run-time behavior. A constant's CFA points to constant which is an assembly language routine which places the contents of the two bytes following the CFA on to the parameter stack. A code defined word's CFA simply points to the byte following the CFA—the beginning of the word's code definition.

The CFA of a colon definition points to colon. See Figure A.2 for the structure of a colon definition in the PDBMS. This routine has different actions, depending upon the specific version of FORTH (i.e., whether the system increments the interpreter pointer before executing a word, or after). In general though, colon pushes the current value of the interpreter pointer (which points to the current word being executed in the post-incrementing systems) onto the return stack and then sets the interpreter pointer equal to the contents of the first two bytes following the current word's CFA. These two bytes contain a pointer to the CFA of the first word in the currently executing word's parameter field address (PFA). Thus the execution of a word describes an inorder traversal of a tree of FORTH words used to define a word and all words used in those definitions, etc. Leaves on this tree are code defined words, constants, variables, user variables, and other data types; nodes are colon definitions.

Complementing colon is semicolon. This is the runtime code of {;} which is the last word in every colon definition. What semicolon does is simply pop the return stack and sets the interpreter pointer equal to the popped value. This causes execution to move one layer higher in the tree described above. The topmost word in the tree is QUIT, which is an infinite loop. So when the interpreter completes the execution or compilation of a word, execution returns to QUIT which loops waiting for more input.

The heart of FORTH is the inner interpreter. In the 8080, Z80, and NSC800 all this short code routine does is take the interpreter pointer and push it into the program counter. This technique of passing control from word to word makes FORTH almost incomprehensible until the entire system is entirely understood. Because FORTH uses almost no subroutine calls and jumps, flow of control is not

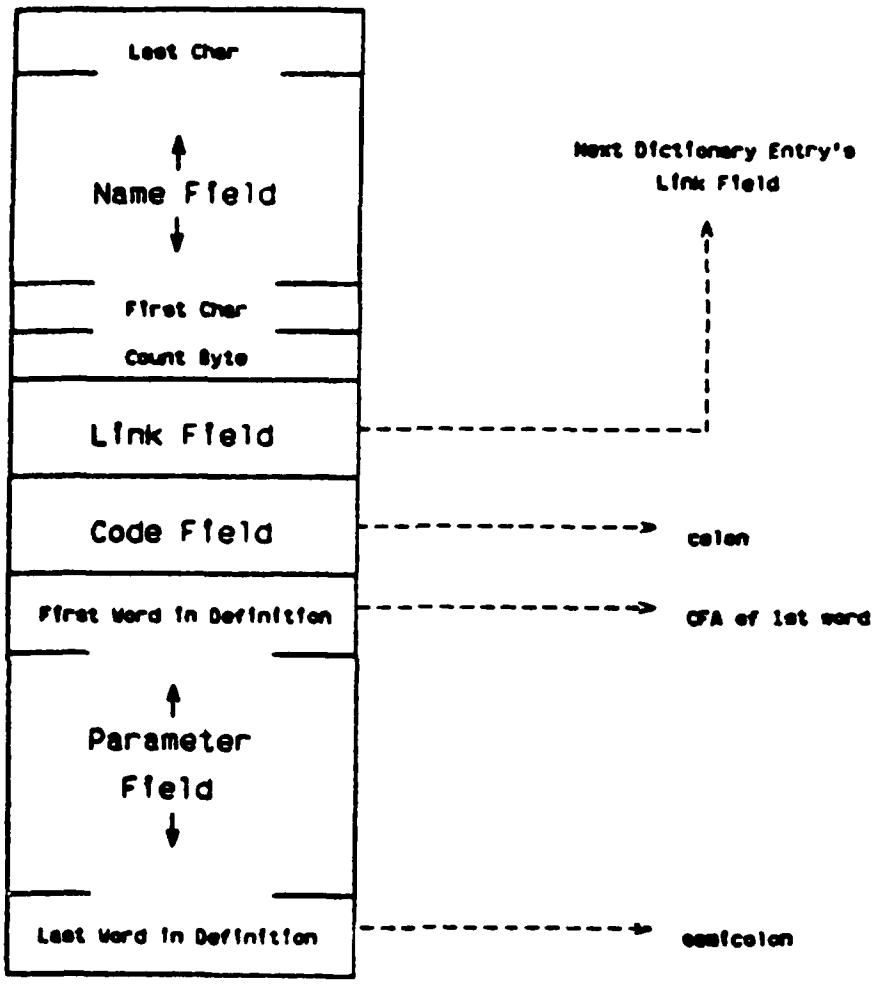


Figure A.2 Structure of a PDBMS Colon Definition.

immediately apparent. In 8080 fig-FORTH (version 1.3) almost the entire FORTH system past the first 1K bytes consists of "DB" and "DW" instructions[1s]. Like LISP, most of FORTH consists of data structures which can be used as data or executable code.

[1s]The "DB" (Define Byte) and "DW" (Define Word) instructions are 8080 assembly language psuedo-instructions which are used to insert data into code areas. For example, the FORTH message "OK" (followed by a carrage return and line feed) is inserted into the source code of FORTH by using the "DB" instruction as follows.

```
DB     'OK',0DH,0AH
```

APPENDIX B
STUDY STATISTICS

A. BACKGROUND

In order to understand what might be involved in a Personal Database Management System, four address books were studied in detail. The results of this study served as a basis for much of the design of the PDBMS. It should be pointed out that the results of this study are probably not indicative of the American population as a whole. The books were not selected on any scientific basis and had the following important characteristics which probably skewed the findings:

- All of the books belonged to friends and neighbors of the author in California. Thus many addresses, zip codes, area codes, etc., had common values.
- All of the books were kept for families and not individuals. The effect of this in uncertain, but because of this entries in these books fell into four distinct categories:
  - The husband's relatives (characterized by similar names, cities, states, zip codes, etc.).
  - The wife's relatives (having the same characteristics as mentioned above).
  - Local friends (characterized by similar cities, state, zip codes, telephone area codes and exchanges, etc.).
  - Non-local friends (which had little in common, except perhaps the military in many cases).
- All of the families had at least one member in the armed forces. This seemed to introduce many acronyms and abbreviations which are probably not very common in civilian spheres. This probably also accounted for a larger than usual number of "non-local friend entries."

B. METHOD OF ANALYSIS

Each of the books was recorded into a file of its own in a fashion which changed it as little as possible from the original. Non-alphabetic and graphic symbols were represented by their closest ASCII equivalent, if there was one. Otherwise an alternate such as "@" was chosen. Statistical analysis was performed on these files but is not included because it included lower-case letters and a large number of spaces (used for formatting). It was felt that these conditions made these first sets of files inappropriate for use with the PDBMS.

After the above files had been created, the files were then copied to another set of files. In transferring the data, all lower-case letters were converted to upper-case and multiple spaces were removed. Tables VIII, X, XI, XV, XVI, and XVII present the results of the analysis of these files.

Finally this second set of files was copied to a third set using a transformation which was designed to reduce the skewedness of the letter and digit distributions. This was done at a time when it had not yet been decided not to use text compression. Many text compression techniques require knowledge of the distribution of the symbols. It was hoped that something close to the letter distribution of standard English would be obtained. The tables which use the label "After" reflect the data gained from analyzing this last set of files. The distribution of the letter frequencies for English were gotten from reference [14]. What follows are the rules applied to the second set of files to produce the third set. They are listed in the order in which they were applied.

- Remove all redundant surnames.

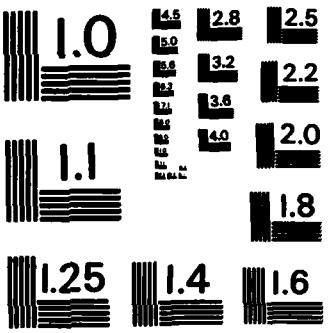
AD-A121 894 DESIGN AND IMPLEMENTATION OF A PERSONAL DATABASE MANAGEMENT SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P L JONES JUN 82 2/2

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

- Remove all redundant city names for cities in the same state. Any form of the name is removed (including abbreviations) leaving the longest form.
- Remove all redundant zip codes.
- Remove all redundant telephone exchange numbers within the same area code.
- Remove all area codes and state names.
- Remove the first three digits of each zip code remaining. These digits indicate the post office's geographical region (the first digit) and major city or distribution point (second and third digits).

The data in the first and second sets of files, though obviously address book data, could not be used as a representative sample of the "average" American address book. For example, 310 (6 percent) of the wordds in the address books refer to the states of California, Maryland, North Carolina, New York, Virginia, and Washington. This would probably serve as a poor basis for predicting the contents of the address book of someone living in Chicago. For this reason the above transformation was used in an attempt to remove the influence of family names and geographical locations from the data yielding a sample more representative of an "average" address book. Because the PDBMS is not designed to handle only one specific person's information, an average address book was needed in order to determine the utility of algorithms and data structures. If the address books had been found to contain almost no redundancies, then the idea of using a DB dictionary probably would have been discarded.

C. RESULTS OF THE ANALYSIS

In the tables appearing in this appendix, the words "wordd," "char," and "punctuation" are used to connote the definitions ascribed to them in Table I. The word "character" is used to mean all printing ASCII characters and the space. All percentages, except those in Table X, reflect the percentage of all characters.

1. General Statistics

The difference between the number of unique wordds in Tables VIII and IX is a result of the reduction of zip codes to their last two digits. The differences are equal to the number of unique zip codes. Also notice that the sum of the unique wordds in the four books is not equal to the number in the total column. This is because the total shown is the number of unique wordds in all four books as a whole. Lastly, the reduction of the number of characters includes not only those chars in the deleted wordds, but also the punctuation following the ends of and between the wordds deleted during the creation of the third set of files.

2. Wordd Length

Table X indicates that the PDBMS, as it is designed, is not as efficient with memory, when compared to a system which simply inserted plain text (i.e., did not use a DB dictionary, etc.). Between the DB dictionary and the logical records, every unique wordd in the PDBMS requires at least nine bytes (seven for the DB dictionary entry and two in the logical record). Wordds which are duplicates of wordds previously entered into the PDBMS require five bytes (three in the DB dictionary used for the field ID and the pointer to the physical record, and two in the logical record used for the first letter of the wordd and the

TABLE VIII

General Statistics - Before

	Book 1	Book 2	Book 3	Book 4	Total
Records	80	129	88	111	408
Fields	340	472	346	350	1508
Characters	6173	8409	5908	6248	26738
Chars	5049	6639	4809	5163	21660
Wordds	1119	1579	1134	1129	4961
Unique Wordds	749	958	740	723	3170

TABLE IX

General Statistics - After

	Book 1	Book 2	Book 3	Book 4	Total
Records	80	129	88	111	408
Fields	340	472	346	350	1508
Characters	5502	7053	4928	5134	22617
Chars	4385	5325	3834	4069	17613
Wordds	1008	1329	941	925	4203
Unique Wordds	722	912	704	678	3016

wordd's ID). Using the numbers in Table X, the average wordd length in the four books is 4.37 chars. In order to be better than or equivalent to a system using plain text in

TABLE X

Wordd Length Distribution

Wordd Length	Frequency	%
1	310	6.25
2	728	14.67
3	939	18.93
4	800	16.13
5	936	18.87
6	427	8.61
7	348	7.01
8	243	4.90
9	116	2.34
10	61	1.23
11	36	0.73
12	16	0.32
13	1	0.02

records requires highly redundant information. The four books together require approximately 34K bytes of storage as plain text (this includes administrative overhead). However this does not include the storage required for indices needed to provide random access; only sequential access is possible with only 34K bytes of storage. Based upon the data derived from the four books, the PDBMS would require approximately 45K bytes to store the same information (27K bytes for the dictionary and 18K for the files; again including administrative overhead). However, unlike the 34K bytes above, this 45K bytes includes storage dedicated to providing random access.

TABLE XI

Char Statistics - Before

	Book 1	%	Book 2	%	Book 3	%	Book 4	%
A	367	5.945	478	5.684	399	6.754	428	6.850
B	101	1.636	145	1.724	79	1.337	106	1.697
C	166	2.689	248	2.949	114	1.930	193	3.089
D	127	2.057	195	2.319	139	2.353	142	2.273
E	473	7.662	459	5.458	338	5.721	414	6.626
F	58	0.940	50	0.594	33	0.559	53	0.848
G	64	1.037	82	0.975	55	0.931	73	1.168
H	106	1.717	170	2.022	103	1.743	113	1.809
I	218	3.532	244	2.902	196	3.318	224	3.585
J	30	0.486	31	0.369	37	0.626	35	0.560
K	57	0.923	55	0.654	61	1.033	76	1.216
L	227	3.677	250	2.973	187	3.165	272	4.353
M	118	1.912	183	2.176	134	2.268	132	2.113
N	294	4.763	375	4.460	223	3.775	297	4.754
O	254	4.115	372	4.424	236	3.995	283	4.529
P	78	1.264	111	1.320	61	1.033	76	1.216
Q	4	0.065	1	0.012	5	0.085	2	0.032
R	289	4.682	424	5.042	250	4.232	360	5.762
S	187	3.029	301	3.580	225	3.808	254	4.065
T	232	3.758	285	3.389	185	3.131	235	3.761
U	109	1.766	78	0.928	64	1.083	88	1.409
V	69	1.118	77	0.916	60	1.016	78	1.248
W	55	0.891	66	0.785	100	1.693	60	0.960
X	17	0.275	49	0.583	17	0.288	11	0.176
Y	99	1.604	98	1.165	79	1.337	97	1.553
Z	16	0.259	8	0.095	14	0.237	8	0.128

TABLE XII

Char Statistics - After

	Book 1	%	Book 2	%	Book 3	%	Book 4	%
A	333	6.052	379	5.374	280	5.682	337	6.564
B	81	1.472	129	1.829	74	1.502	97	1.889
C	151	2.745	150	2.127	98	1.989	127	2.474
D	122	2.217	164	2.325	122	2.476	119	2.318
E	421	7.652	368	5.218	304	6.169	357	6.954
F	56	1.018	40	0.567	28	0.568	39	0.760
G	58	1.054	70	0.993	51	1.035	56	1.091
H	100	1.818	139	1.971	100	2.029	102	1.987
I	212	3.853	208	2.949	165	3.348	194	3.779
J	30	0.545	30	0.425	37	0.751	35	0.682
K	54	0.982	49	0.695	45	0.913	69	1.344
L	186	3.381	221	3.133	161	3.267	233	4.538
M	107	1.945	145	2.056	104	2.110	113	2.201
N	242	4.398	285	4.041	214	4.343	243	4.733
O	229	4.162	334	4.736	200	4.058	233	4.538
P	76	1.381	97	1.375	54	1.096	66	1.286
Q	3	0.055	1	0.014	5	0.102	2	0.039
R	265	4.816	377	5.345	227	4.606	319	6.214
S	175	3.181	241	3.417	192	3.896	208	4.051
T	213	3.871	250	3.545	149	3.024	188	3.662
U	102	1.854	69	0.978	61	1.238	76	1.480
V	67	1.218	63	0.893	59	1.197	65	1.266
W	43	0.782	53	0.752	51	1.035	50	0.974
X	15	0.273	47	0.666	17	0.345	11	0.214
Y	74	1.345	74	1.049	59	1.197	68	1.325
Z	10	0.182	8	0.113	11	0.223	8	0.156

TABLE XIII

Digit Statistics - Before

	Book 1	%	Book 2	%	Book 3	%	Book 4	%	Total	%
0	179	2.9	253	3.0	191	3.2	124	2.0	747	2.8
1	222	3.6	219	2.6	170	2.9	150	2.4	761	2.8
2	144	2.3	271	3.2	194	3.3	157	2.5	766	2.9
3	91	1.5	180	2.1	123	2.1	100	1.6	494	1.8
4	111	1.8	137	1.6	112	1.9	118	1.9	478	1.8
5	95	1.5	158	1.9	138	2.3	87	1.4	478	1.8
6	89	1.4	144	1.7	108	1.8	92	1.5	433	1.6
7	107	1.7	169	2.0	84	1.4	58	.93	416	1.6
8	101	1.6	154	1.8	131	2.2	69	1.1	455	1.7
9	98	1.6	119	1.4	164	2.8	98	1.6	479	1.8

TABLE XIV

Digit Statistics - After

	Book 1	%	Book 2	%	Book 3	%	Book 4	%	Total	%
0	138	2.5	192	2.7	114	2.3	74	1.4	519	2.3
1	144	2.6	179	2.5	129	2.6	105	2.0	557	2.5
2	117	2.1	177	2.5	134	2.7	98	1.9	526	2.3
3	77	1.4	150	2.1	103	2.1	65	1.3	395	1.7
4	94	1.7	102	1.4	92	1.9	67	1.3	355	1.6
5	75	1.3	113	1.6	99	2.0	66	1.3	353	1.6
6	76	1.4	105	1.5	79	1.6	54	1.1	314	1.4
7	78	1.4	112	1.6	72	1.5	40	.78	302	1.3
8	85	1.5	113	1.6	79	1.6	51	1.0	328	1.5
9	75	1.4	81	1.1	65	1.3	34	.66	255	1.1

TABLE XV
Punctuation Statistics

	Book 1	%	Book 2	%	Book 3	%	Book 4	%	Total	%
space	724	12	1057	13	720	12	866	14	3367	13
[illegible]	14	.23	6	.07	6	.10	8	.13	34	.13
[illegible]	1	.02	0	0.0	0	0.0	0	0.0	1	-
[illegible]	45	.73	38	.45	63	1.1	0	0.0	146	.55
[illegible]	3	.05	2	.02	9	.15	3	.05	17	.06
(	12	.19	44	.52	11	.19	4	.06	71	.27
)	12	.19	44	.52	11	.19	4	.06	71	.27
[illegible]	0	0.0	2	.02	0	0.0	0	0.0	2	.01
+	0	0.0	0	0.0	0	0.0	70	1.1	70	.26
,	102	1.7	166	2.0	76	1.3	74	1.2	418	1.6
-	98	1.6	124	1.5	121	2.0	16	.26	359	1.3
.	73	1.2	210	2.5	61	1.0	36	.58	380	1.4
/	4	.06	9	.11	1	.02	2	.03	16	.06
:	27	.44	0	0.0	0	0.0	0	0.0	27	.10
;	2	.03	0	0.0	0	0.0	0	0.0	2	.01
?	0	0.0	0	0.0	3	.05	1	.02	4	.01
other	6	.10	68	.81	17	.29	1	.02	92	.34

3. Char, Digit, and Punctuation

Tables XI, XII, XIII, XIV, XV, and XVI present data on the symbols found in the four address books. Notice from Table XVI that it is obvious that these books are not samples from normal English text. For the most part, the books are "fairly uniform" in their use of letters and digits; this is not the case with punctuation. Book 1 is distinctive in that it is the only one where a dollar sign, colons, and semicolons appear. Book 2 uses an unusually large number of "other" punctuation characters. These punctuation characters are those which were used to represent graphic, non-alphabetic symbols. Book 4 is unlike the others in that it uses the plus sign as the abbreviation for the word "and" whereas the other books use the ampersand. Book 4 also contains a relatively small number of parentheses, dashes, periods, and "others" compared to the other books.

4. Initial Letters

Tables XVII and XVIII show the distribution of all alphabetic wordds in the four books as a whole by their first letter. What is shown in the "Most Frequent Wordds" column are those wordds which account for approximately 30 percent of the total number of wordds starting with the letter in the corresponding first column. Notice that surnames, cities, and states do not appear in Table XVIII because all but one occurrence of them remains in the third set of files. One noticeable exception is the towns of Westminster. The wordd appears in Table XVIII because three different towns occur in the four different books (Westminster, California; Westminster, Colorado; and Westminster, Maryland). As proof of the skewed nature of information notice the large number of occurrences of the

TABLE XVI

Comparison with Standard English

	Before		After	
	Observed	Expected	Observed	Expected
A	418.00	322.76	332.25	273.98
B	107.75	60.52	95.25	51.37
C	180.25	121.04	131.50	102.74
D	150.75	161.38	131.75	136.99
E	421.00	524.49	362.50	445.22
F	48.50	80.69	40.75	68.50
G	68.50	60.52	58.75	51.37
H	123.00	242.07	110.25	205.49
I	220.50	262.24	194.75	222.61
J	33.25	20.17	33.00	17.12
K	66.25	20.17	54.25	17.12
L	234.00	141.21	200.25	119.87
M	141.75	121.04	117.25	102.74
N	297.25	282.42	246.00	239.73
O	286.25	322.76	249.00	273.98
P	81.50	80.69	73.25	68.50
Q	3.00	10.09	2.75	8.56
R	330.75	262.24	297.00	222.61
S	241.75	242.07	204.00	205.49
T	234.25	363.11	200.00	308.23
U	84.75	121.04	77.00	102.74
V	71.00	40.35	63.50	34.25
W	70.25	60.52	49.25	51.37
X	23.50	20.17	22.50	17.12
Y	93.25	80.69	68.75	68.50
Z	11.50	10.09	9.25	8.56

X^2 Statistic Before: 466.89

X^2 Statistic After: 387.44

abbreviations for the states of California (CA), North Carolina (NC), New York (NY), and Washington (WA). The large number of P's and O's can be accounted for by the large number of occurrences of the uword "P.O." as an abbreviation for post office.

These two tables also support the premise that these address books are not from normal English text. The English words "THE," "OF," and "AND" make up 13.75 percent of all words in English text. These same words make up less than one percent of the wordds in the address books. In fact, less than one percent of the wordds in the four address books are the 46 most frequently occurring words in the English language. These 46 words account for more than 41 percent of all words in English text [15].

TABLE XVII

Initial Letters of Wordds - Before

	No. of Unique Wordds	Total No. of Wordds	Most Frequent Wordds	Count
A	71	221	AVE AVENUE APT	47 18 18
B	124	281	BOX BILL BELLMORE	68 14 11
C	129	349	CA C CO CT	89 18 14 10
D	71	179	DR DRIVE D DAVE DAVID	29 11 7 7 7
E	42	89	E EVANS	19 10
F	48	90	FPO F FL FRANKLIN	7 5 5 5
G	59	78	GROVE GARDEN GEORGE GARY	6 5 4 3
H	73	103	HENRY HOME HARRY HELEN	4 4 3 3
I	21	36	IN INC I	5 5 3
J	54	128	JOHN J JIM	14 12 12
K	36	63	KAREN KENNETH KY	5 5 5
L	72	135	LANE LINDA LOS LOUISVILLE LT	10 10 5 5 5

TABLE XVII

continued

	No. of Unique Wordds	Total No. of Wordds	Most Frequent Wordds	Count
M	109	289	MRS MR MD MASS MOREHEAD	36 24 19 14 12
N	56	232	NC NY N NEW NORTH	51 51 18 17 13
O	33	109	O OAK	41 10
P	78	175	P PITTSFORD PAUL	37 10 9
Q	3	3		
R	84	206	RD RT ROAD	40 16 12
S	133	340	ST S SAN SEATTLE STREET	47 27 17 17 17
T	36	72	TOMISSER TEXAS TOM TX	8 4 4 4
U	13	21	UNCLE	3
V	24	57	VA VALLEY VIRGINIA	9 8 5
W	52	165	WA W	54 13
X	0	0		
Y	5	29	YORK	9
Z	7	8	ZUMA	2

TABLE XVIII

Initial Letters of Wordds - After

	No. of Unique Wordds	Total No. of Wordds	Most Frequent Wordds	Count
A	71	203	AVE AVENUE APT	47 18 18
B	124	246	BOX BILL BOB	68 14 5
C	129	234	C CO COURT CIR CT	18 11 7 6 6
D	71	167	DR DRIVE D DAVE DAVID	29 11 7 7 7
E	42	78	E EAST ELIZABETH	18 4 4
F	48	74	F FRANKLIN FEBRUARY	5 4 3
G	59	69	GEORGE GARY	4 3
H	73	97	HENRY HOME HARRY HELEN	4 4 3 3
I	21	36	IN INC I	5 5 3
J	54	127	JOHN J JIM	14 12 12
K	36	59	KAREN KENNETH KATHY KATIE	5 5 3 3
L	72	113	LANE LINDA LT L LA	10 10 5 4 4

TABLE XVIII

continued

	No. of Unique Wordds	Total No. of Wordds	Most Frequent Wordds	Count
M	109	245	MRS MR M MARY MIKE	36 24 7 7 7
N	56	120	N NORTH NEW NO	18 13 7 5
O	33	101	O OAK	41 10
P	78	157	P PAUL PARK	37 9 7
Q	3	3		
R	84	197	RD RT ROAD	40 16 12
S	133	302	ST S STREET SMITH SUE	47 27 17 6 6
T	36	57	TOM THE	4 3
U	13	18	UNCLE	3
V	13	48	VALLEY VISTA	5 4
W	52	98	W WEST WESTMINSTER	13 6 3
X	0	0		
Y	5	10		
Z	7	8	ZUMA	2

LIST OF REFERENCES

1. Shneiderman, Ben. SOFTWARE PSYCHOLOGY Human Factors in Computer and Information Systems, Winthrop Publishers, Inc., 1980.

2. The FORTH Standards Team, FORTH-79, FORTH Interest Group, October 1980.

3. Mandler, George. "The Recognition of Previous Encounters," American Scientist, Vol. 69, March-April 1981.

4. Embley, David W. and Nagy, George. "Behavioral Aspects of Text Editors," Computing Surveys of the ACM, Vol. 13, No. 1, March 1981.

5. Heckel, Paul and Schroeppel, Richard. "Software techniques cram functions and data into pocket-sized uC applications," Electronic Design, Vol. 8, April 12, 1980.

6. Williams, Gregg and Meyers, Rick. "The Panasonic and Quasar Hand-Held Computers," BYTE, Vol. 6, No. 1, January 1981.

7. Morgan, Chris. "A Revolution in Your Pocket," BYTE, Vol. 7, No. 4, April 1982.

8. Hirsch, R. S. "Procedures of the Human Factors Center at San Jose," IBM Systems Journal, Vol. 20, No. 2, 1981.

9. Stuart, LaFarr. "LaFORTH," Proceedings of the 1980 FORML Conference, FORTH Interest Group, 1980.

10. Gifford, David K. "Cryptographic Sealing for Information Secrecy and Authentication," Communications of the ACM, Vol. 25, No. 4, April, 1982.

11. Bell, D. E. and LaPadula, L. J. Secure Computer Systems: Mathematical Foundations, ESD-TR-73-278, Vol. I-III, The MITRE Corporation, 1973.

12. Bell, D. E. and Burke, E. L. A Software Validation Technique for Certification: the Methodology, ESD-TR-75-54, Vol. I, The MITRE Corporation, 1974.

13. Kogge, Peter M. "An Architectural Trail to Threaded-Code Systems," Computer, Vol. 15, No. 3, March, 1982.

14. Kahn, David. The Codebreakers, The MacMillan Company, 1967.

15. Montgomery, Edward B. "Bringing Manual Input into the 20th Century: New Keyboard Concepts," Computer, Vol. 15, No. 3, March 1982.

BIBLIOGRAPHY

Brodie, Leo. Starting FORTH, Prentice-Hall, Inc., 1981.

Cassady, John. fig-FORTH Assembly Source Listing, FORTH Interest Group, September 1979.

Derick, Mitch and Baker, Linda. FORTH Encyclopedia, Mountain View Press, Inc., 1982.

"FORTH From A to Z," Digital Design, Vol. 12, No. 1, January 1982.

Gray, Jim. An Approach to End-User Application Design, Tandem TR 81.1, Tandem Computers Incorporated, 1981.

Haydon, Glen B. "Elements of a FORTH Data Base Design," FORTH DIMENSIONS, Vol. III, No. 2, July/August 1981.

Heckel, Paul. "Designing Translator Software," Datamation, Vol. 26, No. 2, February, 1980.

Huffman, David A. "A Method for Construction of Minimum-Redundancy Codes," Proceeding of the I.R.E., September, 1952.

James, John S. "FORTH for Microcomputers," SIGPLAN Notices, Vol. 13, No. 10, October 1978.

Kilbridge, Dave. "Forgiving Forget," FORTH DIMENSIONS, Vol. II, No. 6, March/April 1981.

Laxon, Henry. "Techniques Journal: Execution Vectors," FORTH DIMENSIONS, Vol. III, No. 6, March/April 1981.

Mayer, Richard E. and Bayman, Piraye. "Psychology of Calculator Languages: A Framework for Describing Differences in Users' Knowledge," Communications of the ACM, Vol. 24, No. 8, August 1981.

Radue, J. E. "Text Compression Techniques," Quaestiones Informaticae (South Africa), Vol. 1, No. 1, June 1979.

Ragsdale, William. fig-FORTH Installation Manual, FORTH Interest Group, November, 1980.

Rather, Elizabeth D. and Moore, Charles H. "The FORTH Approach to Operating Systems," ACM '76, Proceeding of the Annual Conference, 1976.

Raghbati, Hassan K. "An Overview of Data Compression Techniques," Computer, Vol. 14, No. 4, April, 1981.

Reisner, Phyllis. "Human Factors Studies of Database Query Languages: A Survey and Assessment," Computing Surveys of the ACM, Vol. 13, No. 1, March 1981.

Smith, Robert L. FORTH-79 Standard Conversion (Version 1.2), Mountain View Press, 1981.

Zipf, George K. Human Behavior and the Principle of Least Effort, Hafner Publishing Company, 1965.

1981 FORML Proceedings, Vol. I-II, FORTH Interest Group, 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Defense Logistic Studies Information Exchange U.S. Army Logistics Management Center Fort Lee, Virginia 23801	1
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
5.	Captain Peter L. Jones Marine Corps Central Design and Programming Activity Marine Corps Development and Education Command Quantico, Virginia 22134	4
6.	Associate Professor Dushan Z. Badal, Code 52ZD Department of Computer Science Naval Postgraduate School Monterey, California 93940	3
7.	Professor Gordon H. Bradley, Code 52BZ Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
8.	Lieutenant Commander D. R. Shoop Department of Computer Science Carnegie Mellon University Pittsburgh, Pennsylvania 15213	1
9.	Captain Moedjiono Dispullahtal JL. Gatot Subroto 101 Jakarta-Pusat, Indonesia	1
10.	Lieutenant Ricardo Arana C. Central de Procesamiento de Datos Ministerio de Marina Lima - Peru	1

11. Lieutenant Richard T. Holdcroft 1
Department Head Course
Surface Warfare Officer School
Newport, Rhode Island 02840

12. Lieutenant Colonel Paul A. Fritsche (Retired) 1
16 Cottonwood Lane
Pittsford, New York 14354

13. Lieutenant Eduardo Bresani 1
Central de Procesamiento de Datos
Ministerio de Marina
Lima - Peru