1992

The Coatings Database : a design and development summary

Mitchel Frank Ludwig
Lehigh University

AUTHOR:

Ludwig, Mitchel F.

TITLE:

The Coatings Database-A Design ad Development Summary

DATE: October 11, 1992

The Coatings Database

A Design and Development Summary

by

Mitchel Frank Ludwig

A Dissertation
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computer Science

Lehigh University
1992

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

August 18, 1992

Date

Advisor in Charge

EECS Dept. Chairperson

Contents

Table of Figures

Chapter 1

Introduction

The *Coatings Database* was designed as a tool to aid in the determination of specific packaging polymers and passivation coatings for IC chips. The package was to provide users with a simple to operate graphical interface from which they could select from a list of various limiting conditions, and be provided with a selection of IC packages that met those conditions. In its final stage, the application was to provide users with data relating to the properties of the coatings to be used, as well as providing test methods for determining these properties.

The requirements of the system posed a long series of problems which needed to be solved before the application could be developed. The system was initially designed to run in the MSDOS environment on an IBM PC compatible computer. The user interface would be designed under the Microsoft Windows environment, which would allow for a simple *point and click* user interface. Due to the size and nature of the data for the application, a relational database package would also be required. The system as it was envisioned would need to create data that was portable across system platforms, and thus the database decided on would either have to be one supported on multiple platforms, or must provide utilities for converting database files to a format readable by such a package.

To provide users with the flexibility needed to meet their demands, the *Coatings Database* was divided into three distinct groups; Data Acquisition, Data Storage, and Search and Analysis. Together, the three would form a single application which would provide an easy way to determine the proper IC coating type for a specific application, as well as allowing for the addition of new data and test conditions as they became available.

1.1 Data Acquisition

The data and test requirements to be acquired by the *Coatings Database* package was to come from a variety of different sources. Each of the different manufacturers who would eventually use the software had their own test formats, as well as their own lists of existing data. As new data became available, there needed to be a simple method for addition of this new data into the system. It also had to be possible to import data from existing database files into the system, and to be able to enter data manually from the computer keyboard (see figure 1.1).

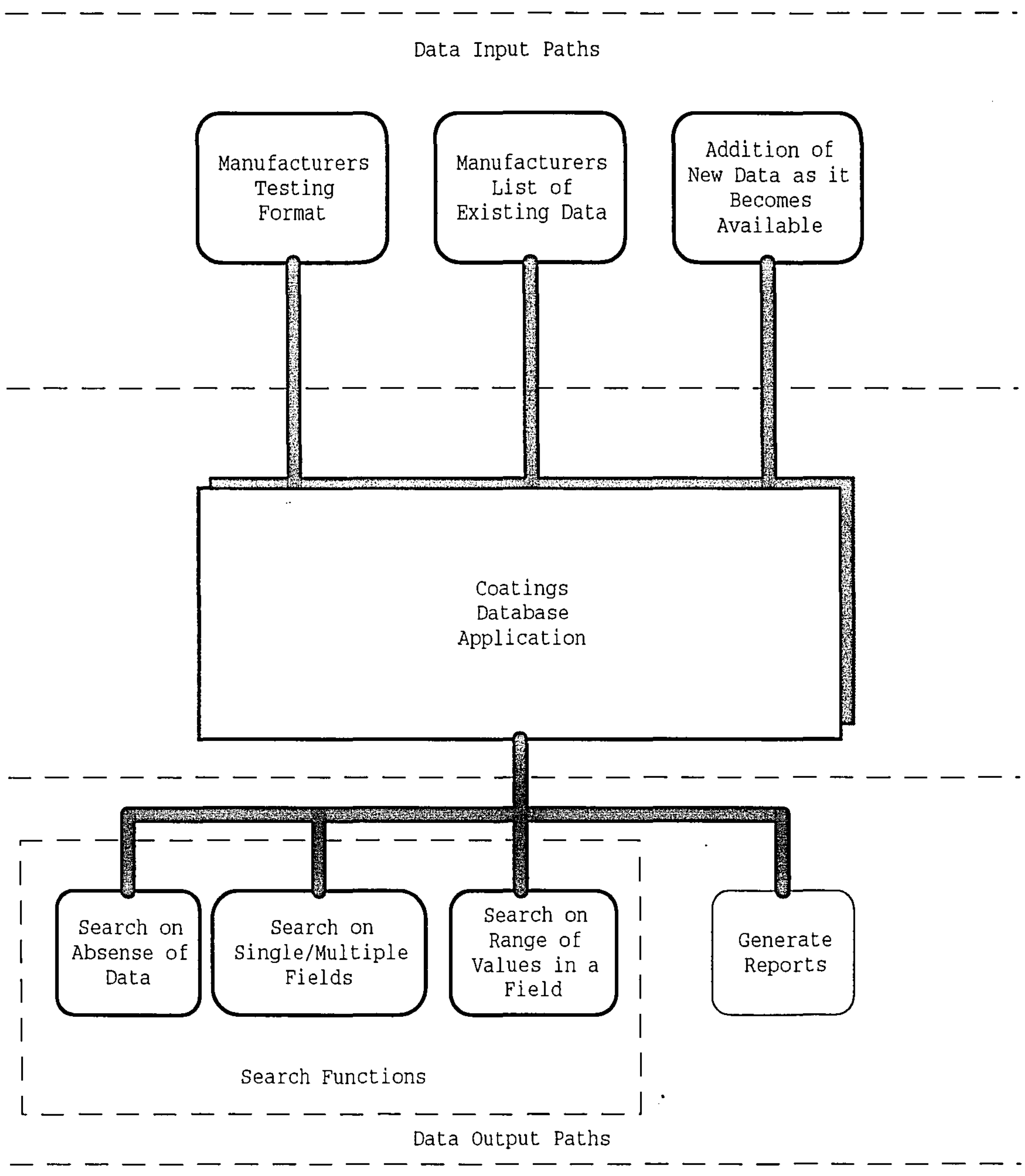

Data Input Paths
Manufacturers Testing Format
Manufacturers List of Existing Data
Addition of New Data as it Becomes Available
Coatings Database Application
Search on Absense of Data
Search on Single/Multiple Fields
Search on Range of Values in a Field
Generate Reports
Search Functions
Data Output Paths

Figure 1.1 : Coatings Database Data Input/Output Paths

As package coating data was made available, it became obvious that sections of data that were included in one package type might or might not appear in others. Whether this was due to the unavailability of the data, or just to its omission was unimportant. In either case, a facility needed to be made available so that this "void" data could be easily located and reported to the user. This was necessary because, in many cases, the "voids" in particular data fields would make the database incomplete. Additionally, the way in which the "voids" in coating packages were handled needed to be determined. If a particular coating package met all the requirements provided by a user, except for the existence of the "voids" in certain fields, should these coatings be displayed for the users as possible matches; and if so, should the user be informed of the existence of the voids?

The manufacturers of the various coatings needed to be contacted to determine why the "voids" existed. They responded with three explanations for the "void" data.

1. The non-existence of certain data fields was due to their being unimportant as far as the package in question was concerned.

2. The omissions were due to the unavailability of the data at the time it was obtained.

3. The data available was far to ambiguous to be of any use, so "void" data was used in its place.

Independent of the explanation of the "void" fields, methods needed to be developed to properly handle their existence within a database.

Often, in the place of a static entry to the database, manufacturers would provide us with ranges for values for a specific field. In many cases, these range values provided an alternative to specifying "void" entries for a field where single values for that field were unknown. We were often faced with situations where the values provided produced ambiguous results due to the uncertainty introduced by the ranges. In order to determine an accurate response to user queries, the data provided to us by the manufacturer needed to be better defined.

The eventual users of the system needed to be contacted in order to provide us with an accurate range of test methods for the various coating packages. Different test plans needed to be developed in order to accommodate the varying requirements of each individual manufacturer. The handling of the "void" and range data fields in individual coating types needed to be handled for each of the test plans. Questions posed concerning the handling of theses types of data included :

1. Should items that passed all of the test requirements, except for the presence of "void" fields, be included or excluded from the list of acceptable coating types?

2. Should items that, due to the presence of a range value, passed all the test requirements, be marked differently than items that passed the tests without the need for range values?

3. When testing a set of range values, how precise should the comparison be?

The ability to update; both by the addition of new coating data as it became available, and also by the correction of incorrect or outdated data on existing coating types, needed to be supported. An interface that was both informative and simple to operate needed to be designed to facilitate easy entry and re-entry of data into the database.

1.2 Data Storage

The question of data storage needed to be addressed. It was found that there were approximately twenty different data fields that could appear in a particular coating type. This was further compounded by the fact that many of the data fields were not static entries, but were ranges of values. An efficient and easy to manage method for the storage of data needed to be developed for the *Coatings Database*. The storage method had to provide simple methods for both the storage of new data in the system, as well as the extraction of old data under a varying set of circumstances. The method for the storage of range values also needed to be considered; would they be stored in two separate fields, or would some method for their storage in a single field be developed?

1.3 Search and Analysis

The methods to be used for the searching on, and analysis of, the data contained within the fields of the database needed to be decided on. Four distinct types of searches needed to be supported (see figure \ref{iopaths}).

- **Search on single field.** The application needed to provide a method for searching the individual fields of the database for specific values. *(e.g. Search for the entries manufactured by company XYZZY).*

- **Search on multiple fields.** The application needed to provide a method for searching multiple fields of the database for specific values. *(e.g. Search for the entries manufactured by company XYZZY with coating type ABCDEF).*

* **Search on absence of data.** The application needed to provide a method for searching for the absence of requested data in specific fields. *(e.g. Search for the entries that have no coating type).*

* **Search on a range of values in a field.** The application needed to provide a method for searching for a range of values that might exist in a single field. *(e.g. Search for the entries that have a dielectric constant between XYZ and ABC).*

In addition to the four search methods described above, the ability to combine search types together was a fifth consideration. *(e.g. Search for entries manufactured by company XYZZY that have a void in the coating type field.)*

The interface to the above searching methods was to be designed under the Microsoft Windows operating system. The application needed to provide a seamless interface to whatever database tool was used to store and manipulate the data. Additionally, the user application had to provide an easy to use interface for search and analysis of the data contained in the various databases.

Chapter 2

Deciding on a Database

A decision needed to be made as to the database we would use in the development of the *Coatings Database* application. The database selected needed to provide us with a large amount of flexibility, while still providing enough power to comply with the demands the application would place upon it. Many different database tools were available to us which would function in the MSDOS environment. They varied in their power, flexibility, and cost. A database needed to be located that would meet the following specifications.

* Interfacing with the C programming language : The *Coatings Database* application was to be written in the C programming language, and would be run under the Microsoft Windows environment. The database tool decided on would need to provide us with some means of interfacing it with our application.

* Ability to use multiple databases in a single application : Due to the large number of data fields that were required for the *Coatings Database*, and the ways in which they were to interact, it was thought that a multiple database file approach might be taken. If this were the case, the database tool would need to be able to link between databases to retrieve necessary information. The selected database tool needed to provide us with the ability to perform searches using information that might lie in separate files.

* Database functionality : The database tool selected would need to provide us with the functions necessary to satisfy the requirements of the application. The search and analysis functions that would need to be supported included Search on single/multiple fields; Search on absence of data; and Search on ranges of values in a single field.

* Portability to other machine platforms : Although initially designed and written for the MSDOS environment, plans were made to eventually port the *Coatings Database* to other platforms. The database tool selected would need to provide a simple, if not transparent, method for porting existing database information, and if possible the actual *Coatings Database* source code itself, to another platform.

2.1 Selecting the Database

Taking the requirements for our database into consideration, the Paradox®[1] relational database package was selected for our application. Included in the Paradox package :

1 Paradox® is a registered trademark of Borland International

* **The Paradox Personal Programmer™** --- "The Paradox Personal Programmer™ is a powerful application generator that enabled you to develop custom single-user database applications without programming."[2]

  The Paradox Personal Programmer™ would allow our application to call user generated "scripts" to be used for designing the test methods for the *Coatings Database*.

* **The Paradox Application Language™** --- The "structured programming language" for Paradox.

  The Paradox Application Language™ would allow the application to make SQL calls to the Paradox database to complete user defined queries.

* **The Paradox Engine** --- "A comprehensive library of C functions and Pascal procedures and functions that can be called from programs within C or Pascal. These functions let you manipulate Paradox tables in both single-user and multiuser environments."[3]

  The Paradox Engine would provide the link between the applications Microsoft Windows based user interface and the database engine that would store the data for the *Coatings Database*.

---

2 Paradox 3.5 book page 2.

3 Paradox 3.5 book page 2

The Paradox Engine would form the backbone of the interface between the *Coatings Database* and the Paradox Database environment. By including the Paradox Engine in the application, a powerful and easy to operate user interface could be designed under the Microsoft Windows environment. The combination of the C language, Microsoft Windows, and the Paradox Engine offered us :

* "A high level of control.

* Access to hardware and the operating system.

* Open-ended functionality. (limited only by your own code or the availability of third-party libraries)

* Open-ended user interface.

* Size and efficiency."[4]

Using the Paradox Application Language\trademark along with the Paradox Engine, we would be able to design and execute the test scripts from within the *Coatings Database* application. As new test methods became available, instead of rewriting the application, new scripts could be written in the Application Language to be included within the program.

The Paradox relational database package provided our group with the tools necessary to comply with the requirements set out for the project. The use of the Paradox Engine, along with the Paradox

4 P.E User page 11

Application Language\trademark, would provide the *Coatings Database* with the database power necessary to complete the tasks required of it.

2.2 Designing the Database

Once the Paradox\registered database was decided upon, it was necessary to choose the method in which the data would be stored within the database files. The two available methods of database design were the *hierarchical* and *relational* database models. Before deciding on the better approach, it was necessary to understand their differences.

2.2.1 Hierarchical Database Design

"A *hierarchical* database organizes its contents in a hierarchical model resembling a tree. The hierarchical "tree" not only identifies the data elements in the database but also defines the relationship among these data elements."[5]\footnote{DBASE p 5} The hierarchical model of database design offers us two main approaches to the definition of the database. They are the "one-to-one" and the "many-to-many" approaches.

In the "one-to-one" approach, the bottom up view of the database tree is a "one-to-one" relationship (see figure 2.1). Each leaf of the tree is connected to one branch above it, and that branch is connected to only one higher branch, until the main branches connect to the one and only root. This method of database design is designed for small applications where little or no data is duplicated in different branches of the tree. As a database designed in this way gets larger, the chance of repetition of

5 DBASE p. 5

data grows, and the redundancy in the database grows as well. This results in reduced efficiency in both the storage and access time of the database.

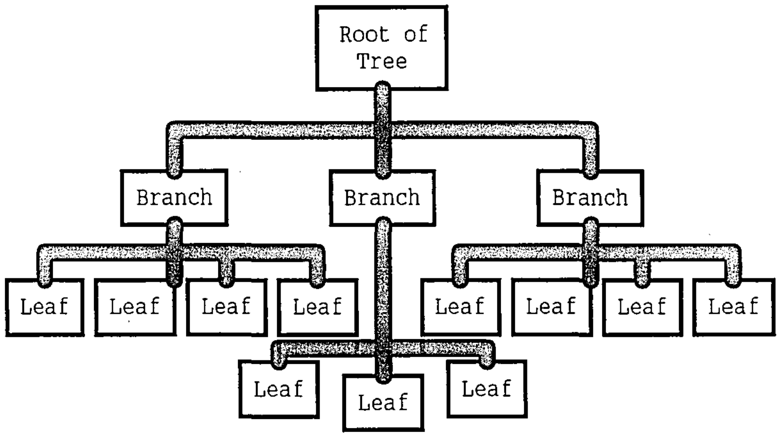


Figure 2.1 : Hierarchical "One-to-One" Tree Model

In the "many-to-many" approach, both the top down and bottom up views of the database show the same "many-to-many" relationship between leaves and branches. In this approach, a single data record is created for each unique addition to the database, and additional links to that entry are made when duplicity is called for (see figure 2.2). This results in databases smaller in size, but with greater complexity.

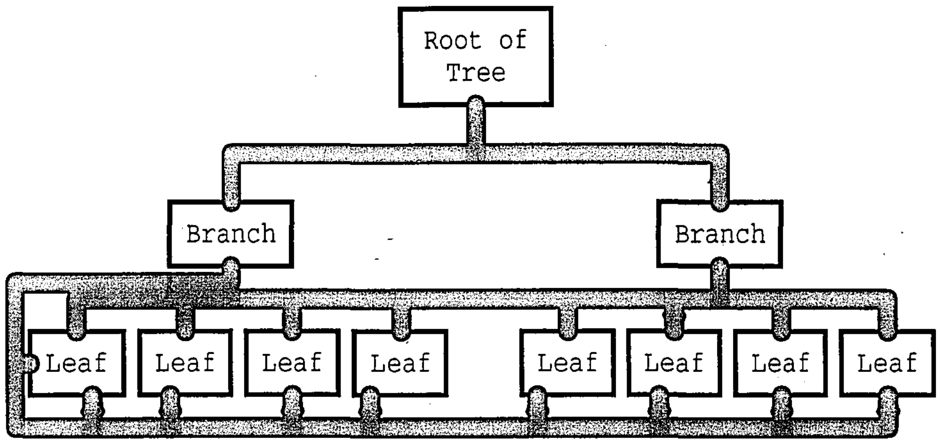


Figure 2.2 : Hierarchical "Many-to-Many" Tree Model

2.2.2 Relational Database Design

	Column 1	Column 2	Column 3	Column 4
Row 1	Data Field	Data Field	Data Field	...
Row 2	Data Field	Data Field	...	...
Row 3	Data Field	...	...	...
Row 4	...	...	...	...
Row 5	...	...	...	...

.
.
.

Row M	Data Record

.
.
.

Row W				
Row X				
Row Y				
Row Z				

Figure 2.3 : Relational Model

"A *relational* database organizes its data elements in a two-dimensional table consisting of rows and columns. Each row contains information belonging to one entry in the database. Data within a row is divided into several items, each occupying one column in the table."[6] In a relational database, the data is set up in a series of two-by-two matrixes, allowing for easy access to individual data items on a row-column basis. In order for a relational database to be effective, the data to be stored must be compatible with this row-column approach to storage and analysis.

"A *relational database file* consists of two main parts. One part defines the structure of the data records, and the other part contains the data itself."[7] "Data records" hold the data items for single entries, forming the "row" groupings for the relational database. "Data fields" are storage units for individual data items within a data record, and form the "column" items within each row grouping (see figure 2.3).

2.2.3 Selection of a Database type for the *Coatings Database* Application

It was theorized that a relational database could be designed to handle multiple groupings of the row-column format by making the data fields of the top level database into data records for lower level database tables. This resulted in a database design where the *hierarchical* approach was used to order data by importance, and the *relational* approach was used to store the individual data items. The use of both database design methods allowed us far more flexibility than either of the two would have had on their own. As a result, it was decided that the merging of the two approaches would best serve the needs of the *Coatings Database.*

6 Dbase book p 7

7 Dbase book p 7

Chapter 3

Development of the Application

Before development of the *Coatings Database* was begun, the project was divided into three different sections; development of the Paradox databases, design and implementation of the C language to Paradox interface, and development of the user interface in the Microsoft Windows environment. In order to insure the three sections would operate correctly when merged, the specifications for the three individual parts needed to include information describing the methods in which the interaction between the parts would be managed.

3.1 Database Design

During the development of the *Coatings Database* application, the increasing amount of data the system was to handle forced us to modify the design many times. The methods used for the storage of coatings data was changed three times over the course of the system's development.

1. <u>Initial database design :</u> A single large database was designed to handle all the data for the application.

2. <u>Multiple database design :</u> The multiple database design was implemented to allow for grouping of data by commonality.

3. <u>Addition of range data :</u> Support for range values is added using links to additional storage databases.

The fields that needed to be managed by the database(s) were:

* <u>Product Name</u>
* <u>Package Type</u>
* <u>Company Name</u>
* <u>Dielectric Strength</u>
* <u>Dielectric Constant</u>
* <u>Dissipation Factor</u>
* <u>Volume Resistivity</u>
* <u>Tensile Strength</u>
* <u>Elongation</u>
* <u>Shore Hardness</u>
* <u>Moisture</u>
* <u>Glass Transition Temperature</u>
* <u>Linear CTE</u>

* Thermal Conductivity

3.1.1 Initial Database Design

In the initial design of the database for the *Coatings Database* application, a single database was used to store all the data collected. In order to support the availability of search functions on each of the different fields, it was necessary to establish numerous keys for the database. A "key is a unique identifier for the table--that is, a column (or combination of columns) with the property that, at any given time, no two rows of the table contain the same value in that column (or combination of columns)."[8] A single *primary key* consisting of the Product Name, Package Type, and Company Name was coded into the database, and was used in combination with each of the other fields to form a set of eleven distinct "keys" that could be used for searches on the databases.

Searches on single values were accomplished by executing the search using the "key" associated with the field to be searched on. In situations where the specified field search did not result in a single result, the user was given two choices. The first choice available was to further limit the search parameters. By entering values for one or more of the parameters which made up the *primary key*, a more restricted search could be run until a single result was determined. The second choice available was to display the entire set of database records which satisfied the specified search parameters.

Many of the search methods used involved the use of multiple fields to define the parameters for the search. In these cases, the "key" choice which offered the greatest degree of uniqueness was chosen as the search "key". After the search parameters had been entered into the system, and the initial search

8 Dbase SYS 234

executed, the fields not included in the chosen "key" were applied as restricting parameters to the set of database records which had passed the initial search. In the multiple field search, as with the single field search, results involving multiple records could be resolved by either further restricting the search parameters or by displaying the entire set of records which satisfied the search parameters specified. Addition and modification to the database were accomplished using the *primary key*. Once the *primary key* had been entered, the database was searched to determine if the record being entered was a new record, or already existed in the database.

If the *primary key* was found to already exist, it was assumed that a modification of the record was to be done. In this case, the record was displayed, and through keyboard input, could be altered and saved. During the saving of the record, the *primary key* was again checked. If it had been altered during the editing session, a new record was created using the new *primary key* and support fields. If the *primary key* was unchanged, the record was updated to reflect the new values. When the *primary key* was not found, the record was assumed to be a new entry into the database. Upon completion of the new record entry, the *primary key* was used to place the new record in the correct location in the database.

The use of a single database provided a useful short term solution to the design of the database for the *Coatings Database* application. As the data to be stored and processed by the system grew, the use of the single database became a hindrance, rather than a help. Queries of any sort, whether to locate specific entries in the database, or to add or modify the database, took longer and longer as the database grew. Thus, it was necessary to convert the *Coatings Database* application to a multiple database design.

3.1.2 Multiple Database Design

Once a single database design was proven to be inadequate for the *Coatings Database* application, we began considering the different paths available by using multiple databases. In order to develop an efficient multiple database application, we needed to know the ways in which the data for the *Coatings Database* was distributed. Once this was known, the database fields could be separated over a group of databases in such a way that each of the individual databases would provide a link back to a main database, while providing enough information to be useful on its own.

Package Type	Number of References
Polymide	15
Urethane	6
Acrylic	4
Epoxy	28
Poly-Silox	4
Elastom	24
Rubber	4
Gel	8

Figure 3.1 : Package Type Counts

Using the data provided to us by the initial group of manufacturers involved (see figures 3.1 and 3.2), we were able to determine a method for breaking up the data within the database into more manageable sections. Analysis of the data showed that there were two different ways in which we could break up the available data that would allow us to better handle the data as it was processed, while still providing us with a coherent set of data files.

Company Name	Package Type	Number of References
Dupont	Polymide	7
Chase	Urethane	6
	Acrylic	4
	Epoxy	1
Ciba Geigy	Epoxy	1
A.I. Technology	Epoxy	1
Oxy SIM	Poly-Silox	1
Emerson & Cummings	Epoxy	3
Amoco	Polymide	5
General Electric	Elastom	23
	Rubber	4
	Gel	8
IPN Indestries	Epoxy	1
Ablestik	Polymide	2
	Poly-Silox	3
HB Fuller	Epoxy	1
Master Bond Inc.	Epoxy	1
Epoxy Technology	Polymide	1
	Epoxy	17
Dexter	Epoxy	2
Castall	Elastom	1

Figure 3.2 : Package Type Counts by Company Name

1. The data could be broken up according to the company which produced the product. By breaking the data up by company name, the addition of new data would be simplified. Each manufacturer would need only to provide us with a new and updated version of the data file for their company, which could then be integrated into the main database.

2. The data could be broken up according to the package type of the product. The division of data accoring to the package type would simplify the data in the area of "void" field determination. It had already been proven that the importance of a "void" data fields was dependant upon the package type the void appeared in. By separating the data by package types, we were able to make the testing of the "void" fields better tailored to the individual package types.

It soon became obvious that separating the data by either of these methods provided only minor relief from the every growing database. By implementing both of the methods we would be able to provide a database system which would both remain small enough to be easily managed, and powerful enough to complete the required queries in a short amount of time. The division of data along these lines would also give the databases a degree of simplicity that neither of the design strategies could provide on their own. It was therefore decided that the original single database would be divided into three groups of databases (see figures 3.3 and 3.4).

Company Name	Database File

Main Database

Coating Type	Database File

Coating Type File Location Database

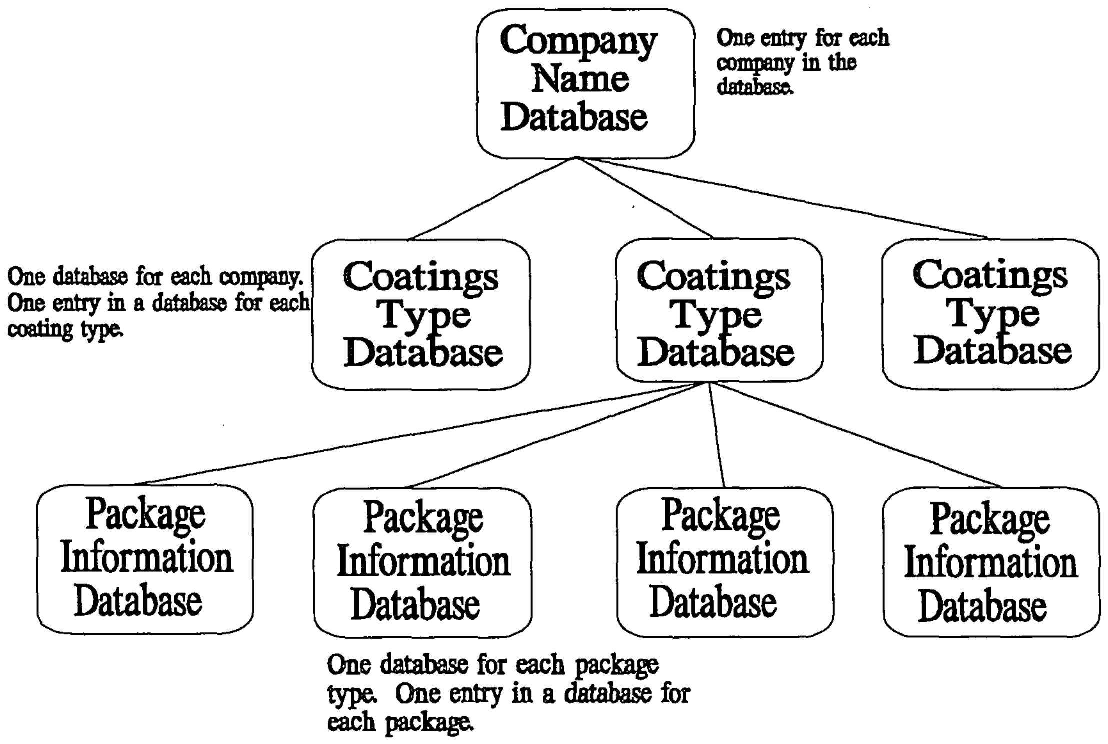
1	2	3	4	5	6	7	8	9	10	11	12	13

Company Data Database

Field Descriptions :

1. Product Name
2. Dialectric Strength
3. Dialectric Constant
4. Dissipation Factor
5. Volume Resistivity
6. Tensile Strength
7. Elongation
8. Shore Hardness
9. Moisture
10. Glass Transition Temperature
11. Linear CTE
12. Thermal Conductivity
13. Comment Field

Figure 3.3 : Multiple Database Field Descriptions



Company Name Database
One entry for each company in the database.
One database for each company. One entry in a database for each coating type.
Coatings Type Database
Coatings Type Database
Coatings Type Database
Package Information Database
Package Information Database
Package Information Database
Package Information Database
One database for each package type. One entry in a database for each package.

Figure 3.4 : Multiple Database Flow Path

* A main "Company" database would be generated with fields containing the Company Name and a link to a file containing information pertaining to packages produced by that company.

* A "Coatings Type" database containing fields for the different coating types manufactured by a company, and a link to a file containing information pertaining to the packages specific to each coating type. Each company that has an entry in the "Company" database will have a separate "Coatings Type" database.

* A "Package Information" database containing fields for the different parameters which make up an individual coatings package. For each coating type that has an entry in the "Coatings Type" database, a separate file will be generated. Additionally, companies

which produce packages with the same coating type will have separate files. Fields in the "Package Information" database will include *: Product Name, Dialectric Strength, Dialectric Constant, Dissipation Factor, Volume Resistivity, Tensile Strength, Elongation, Moisture, Glass Transition Temperature, Linear CTE, and Thermal Conductivity.*

The conversion of the database structure for the *Coatings Database* from a single to a multiple design allowed us to better manage the amount of data generated for the system. The multiple database design was a better defined format for the storage of the information, and retrieval and queries based on the new format were completed at a much higher rate than had been the case while using the single database design.

3.1.3 Addition of Range Data

The conversion of the *Coatings Database* to a multiple database allowed us to process far more data than had been possible when the system was used with a single database. With the introduction of the new data, it became obvious that range data, data fields that consisted of a range of values instead of a single static value, needed to be accounted for by the system.

To accommodate the addition of ranges to the application, modifications needed to be made to the database structures. Analysis showed that the range values took one of three forms.

1. The range values involved were open ended ranges such as (x>10.4) or (y<3.0).

2. The range values involved were closed ranges such as (0.0<x<4.1).

3. The range values involved were multiple static values such as (x = 4.0, x = 10.4, or x = 9.6).

In addition to a single representative of one of the above, the combination of the individual range types needed to be supported.

On its own, the Paradox® Database had no support for the types of range value fields we needed to add to the *Coatings Database*. It was therefore necessary to design and implement our own functions to handle these types of data fields. Three different designs were initially discussed that would allow us, with a minimum amount of change to our existing database structures, to support the range values that might or might not be introduced as data for the various fields of the database.

1. The individual data fields could be modified, when warranted, to support a link to another set of databases which would contain the range values. Using this design, if a data field was to contain a static value, that value would be entered into the data field. If, on the other hand, a range of values was entered, an identifier would be placed into the data field in question pointing to another database which would hold the set of range data.

   The "range data" database would contain two fields. The first field would be an identifier representing the type of range data to be represented by the data record. The supported identifiers were *GTHAN*, *LTHAN*, and EQTO, representing whether the range of acceptable values was greater than, less than, or equal to the value in the second field. The combination of the data records in this database would form the range of acceptable values for the field.

2.. If a limited support of range values was acceptable, each of the single data fields in the "Product Information" database could be expanded into two fields. The first data field would represent the minimum value, the second, the maximum. An identifier for infinity would be added, allowing for ranges such as (x > 10) to be supported. Identity operations such as (x = 10) were supported by setting the minimum and maximum fields to the same value. Range data generation was limited using this method to one of the thee range data formats per record. Combinations of range data formats in a single record were not supported.

3. The "data type" for the individual fields could be changed from a numeric to a character field. This would allow the generation of a string which could represent the range of acceptable values. In this design, the range of values (x = 10, 20 < x < 25, x = 60) would be represented by the string "x = 10, 20 < x < 25, x = 60." Support of this type would necessitate the design of a parser for the resulting data fields, as well as a processor for converting the numeric data back to the string format.

The conversion of the numeric data fields to character fields was chosen as the method we would use to implement range data for the *Coatings Database.* This method involved only minor changes to the database code for the application, and offered us the greatest deal of flexibility out of the three choices. The addition of the range values would not cause the database size to grow as quickly as it would have using the database design involving the separation of the range values into separate fields. Additionally, unlike the linked database design, the addition of the range values in this design did not require the design and creation of a new set of databases to maintain the data. Using the field conversion, the only new code that had to be generated were functions to parse character strings in order to determine the scope of ranges for a particular field, and functions to convert ranges of values to their appropriate string equivalent for storage.

3.2 Implementation of the Database using the *C* Language

With the decision as to the type and format that the database(s) for the *Coatings Database* would take made, the next step was to decide how the software would be designed. The Paradox Engine provided us with a set of *C* callable functions that allowed us to execute Paradox database commands from within a *C* program. Using the Paradox Engine, we were able to design our application in a powerful programming language, while still retaining absolute control over the functionality of our database.

Due to the nature of many of the database requirements for the *Coatings Database*, in many cases the functions provided in the Paradox Engine were not flexible enough to allow us to perform the tasks required of us. In these cases, it was necessary to design functions which called series of Paradox Engine functions to perform a specific task. Functions to be designed included :

* A function to read and write the individual databases needed to be designed. Although the Paradox Engine functions supported the reading and writing of individual fields of a

database, functions which would allow access to entire records of a database at once needed to be implemented.

* A function to open a database needed to be provided. The Paradox Engine functions for deleting, creating, opening, and setting up the key fields existed as individual functions. For the *Coatings Database*, it was necessary that these individual functions be combined into a single routine.

* A simple method for handling the existence of the varied database table types needed to be designed. To support future work, the functions designed needed to be flexible enough that the introduction of a change in a database type would not require large amounts of re-coding. In order to provide this feature, *C* records needed to be designed to hold the size, format, and primary key information of the individual database fields.

* Functions for searching on a database needed to be designed. The **field search** and **key search** functions provided in the Paradox Engine needed to be combined in a single routine which would allow, based on passed parameters, the required types of searches.

In order to provide the *Coatings Database* with the required functionality, it was necessary to first understand the abilities and limitations of the functions that were available to us. The Paradox Engine functions that we needed in order to provide the above functionality were :

* **PXTblOpen(char *tblName, TABLEHANDLE tblHandle, RECORDNUMBER *nRecsPtr)**

  ``**PXTblOpen** will open the table (or index) specified by the base name *tblName*. If **PXTblOpen** is successful, a table handle is returned via *tblHandle* and the function returns PX_SUCCESS."[9]

* **PXTblClose(TABLEHANDLE tblHandle)**

  ``**PXTblClose** closes a table previously opened with **PXTblOpen**. **PXTblClose** takes one argument, the table handle obtained when the table was opened."[10]

* **PXTblCreate(char *tblName, int nFields, char *fields, char *types)**

  ``**PXTblCreate** tries to create a new (empty) table with the path and name given by *tblName* and with a record structure determined by the four standard field arguments."[11]

* **PXTblDelete(char *tblName)**

  ``**PXTblDelete** deletes the table named in the string *tblName*, together with any associated family objects that might exist."[12]

* **PXGetAlpha(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, int bufSize, char *dest)**

  ``**PXGetAlpha** tries to retrieve the alphanumeric string stored in the field specified by *fldHandle* (where the first field starts at 1) in the record transfer buffer referenced by *recHandle*. If successful, the string is returned in the **char** array *dest*."[13]

* **PXGetDate(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, DATE *datePtr)**

  ``**PXGetDate** tries to get the **long int** date value of the date field (internal Paradox format) specified by *fldHandle* (where the first field starts at 1) from the record transfer buffer referenced by *recHandle*. The value is returned via *datePtr*."[14]

* **PXGetDoub(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, double *doubptr)**

  ``**PXGetDoub** tries to get the **double** value of the numeric field specified by *fldHandle* (where the first field starts at 1) from the record transfer buffer referenced by *recHandle*. The value is returned via *doubPtr*.[15]

---

[9]Paradox Engine Book, p 215.
[10]Ibid, p. 199.
[11]Ibid, p. 203.
[12]Ibid, p. 206.
[13]Ibid, p. 122.
[14]Ibid, p. 123.
[15]Ibid, p. 126

* **PXGetLong(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, long *longPtr)**

  ``**PXGetLong** tries to get the **long** value of the numeric field specified by *fldHandle* (where the first field starts at 1) from the record transfer buffer referenced by *recHandle*. The value is returned via *longPtr*."[16]

* **PXGetShort(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, short *shortPtr)**

  ``**PXGetShort** tries to get the **short int** value of the numeric field specified by *fldHandle* (where the first field starts at 1) from the record transfer buffer referenced by *recHandle*. The value is returned via *shortPtr*."[17]

* **PXKeyAdd(char *tblName, int nFields, FIELDHANDLE *fldHandles, int mode)**

  ``**PXKeyAdd** tries to create an index on a table with the base name given in the ASCII string *tblName*."[18]

* **PXKeyDrop(char *tblName, int indexID)**

  ``**PXKeyDrop** tries to delete the index specified by *indexID* for the table named *tblName*."[19]

* **PXPutAlpha(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, char *str)**

  ``**PXPutAlpha** tries to assign the string *str* to the field specified by *fldHandle* (where the first field starts at 1) in the open record transfer buffer referenced by *recHandle*."[20]

* **PXPutDate(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, DATE val)**

  ``**PXPutDate** tries to assign the **long** date value *val* to the field specified by *fldHandle* (where the first field starts at 1) in the open record transfer buffer referenced by *recHandle*."[21]

* **PXPutDoub(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, double Dval)**

  ``**PXPutDoub** tries to assign the **double** value *Dval* to the field specified by *fldHandle* (where the first field starts at 1) in the open record transfer buffer referenced by *recHandle*."[22]

---

[16]Ibid, p. 127.
[17]Ibid, p. 129.
[18]Ibid, p. 132.
[19]Ibid, p. 134.
[20]Ibid, p. 159.
[21]Ibid, p. 162.
[22]Ibid, p. 163.

* **PXPutLong(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, long val)**

  ``**PXPutLong** tries to assign the **long** integer value *val* to the field specified by *fldHandle* (where the first field starts at 1) in the record buffer referenced by *recHandle*."[23]

* **PXPutShort(RECORDHANDLE recHandle, FIELDHANDLE fldHandle, short i)**

  ``**PXPutShort** tries to assign the **short** integer value *i* to the field specified by *fldHandle* (where the first field starts at 1) in the record buffer referenced by *recHandle*."[24]

* **PXRecAppend(TABLEHANDLE tblHandle, RECORDHANDLE recHandle)**

  ``**PXRecAppend** tries to append the contents of the record transfer buffer specified by *recHandle* into the open table specified by *tblHandle*."[25]

* **PXRecDelete(TABLEHANDLE tblHandle)**

  ``**PXRecDelete** tries to delete the current record from the table referenced by *tblHandle*."[26]

* **PXRecFirst(TABLEHANDLE tblHandle)**

  ``**PXRecFirst** tries to change the current record handle of the target table specified by *tblHandle*. If successful, the new current record for the table will be the first record of the table."[27]

* **PXRecGet(TABLEHANDLE tblHandle, RECORDHANDLE recHandle)**

  ``**PXRecGet** tries to transfer the data in the current record associated with the open table referenced by *tblHandle* to the open record transfer buffer referenced by *recHandle*."[28]

* **PXRecGoto(TABLEHANDLE tblHandle, RECORDNUMBER recNum)**

  ``**PXRecGoto** tries to change the current record handle of the target table specified by *tblHandle*. If successful, the new current record will be that given by *recNum*."[29]

* **PXRecInsert(TABLEHANDLE tblHandle, RECORDHANDLE recHandle)**

  ``**PXRecInsert** tries to insert the contents of the record transfer buffer specified by *recHandle* into the open table specified by *tblHandle*."[30]

---

[23]Ibid, p. 164.
[24]Ibid, p. 166.
[25]Ibid, p. 167.
[26]Ibid, p. 174.
[27]Ibid, p. 175.
[28]Ibid, p. 176.
[29]Ibid, p. 178.

* **PXRecLast(TABLEHANDLE tblHandle)**

  ``**PXRecLast** tries to change the current record handle of the target table specified by *tblHandle*. If successful, the new current record will be the last record of the table."[31]

* **PXRecNext(TABLEHANDLE tblHandle)**

  ``**PXRecNext** tries to change the current record handle of the target table specified by *tblHandle*. If successful, the new current record will be the next record following the current record of the table."[32]

* **PXRecPrev(TABLEHANDLE tblHandle)**

  ``**PXRecPrev** tries to change the current record handle of the open table specified by *tblHandle*. If successful, the new current record becomes the record before the previous current record."[33]

* **PXRecUpdate(TABLEHANDLE tblHandle, RECORDHANDLE recHandle)**

  ``**PXRecUpdate** tries to update the current record of the open table *tblHandle*. If successful, the contents of the record transfer buffer specified by *recHandle* are posted (transferred) to the current record associated with the specified table handle."[34]

3.2.1 Development of the Table Definitions File

The *table definitions file* was developed in order to allow for the easy modification of the tables withing the *Coatings Database* application. The file consisted of arrays of structures that contained the various fields necessary to build a database file. The three arrays contained in the file were the *Table Information Array*, the *Field Name Array*, and the *Field Type Array*. The use of a single file to hold all the information necessary to design the database files made the modification of a database an easy task. Additionally, changes to the database that resulted in a need for the recompilation of key program modules were easily centralized because of the use of a single file for all the database information.

[30]Ibid, p. 179.
[31]Ibid, p. 181.
[32]Ibid, p. 182.
[33]Ibid, p. 186.
[34]Ibid, p. 187.

The *Table Information Array* is an array of structures designed to maintain the information necessary to oversee the creation and management of the individual database files. The individual fields of the array structure are :

* **Table Name** - The name of the database. This name will be used during creation of the database as the name for the database. Upon successful creation of the database, the filename *filename.db* will appear in the root directory for the application.

* **Table Size** - The number of individual fields in the database. This field is used to determine the size in fields of the database during creation.

* **Table Key Length** - The number of key fields in the database. Starting at field one (1), this variable is used to determine the number of fields to be included in the *primary key* for the database.

The *Field Name Array* is an array of character strings that are used during the creation of a database. Each entry in a single row of the array is a character string representing the name of a database field for a particular database. An entire row of field names form the name list for a specific database. The number of field names included is equal to the value of the *Table Size* field in the *Table Information Array*.

The *Field Type Array* is an array of character strings the are used during the creation and management of a database. Each entry in a single row of the array is a character string representing the data type for a particular field of a database. The specific database in question is determined by the row number, while the column number determines the specific field in question. As with the *Field Name Array*, the number of entries in each row of the *Field Type Array* is equal to the *Table Size* field in the *Table Information Array*.

A sample of the *table definition file* for a specific database can be seen in figure 3.5.

	Table Name	Table Size	# of Keys
Table Information Array[0]	XYZZY	5	2
Table Information Array[1]			
Table Information Array[2]			
Table Information Array[3]			

Field Name Array[0]	Name	Address	City	State	Zip Code
Field Name Array[1]					
Field Name Array[2]					
Field Name Array[3]					

Field Types Array[0]	A80	A80	A40	A2	A10
Field Types Array[1]					
Field Types Array[2]					
Field Types Array[3]					

Figure 3.5 : Sample Table Definitions File

3.2.2 Development of Database Read/Write Functions

In order to simplify the reading and writing of database files in the *Coatings Database*, it was decided that some sort of generic ***GET*** and ***PUT*** data functions needed to be developed. Using the data provided by the various arrays in the *table definitions file*, it was possible to create a set of generic functions which would facilitate the reading and writing of the database files. Three functions were necessary.

* **inMoveData(int inTableName, char *chRecord, int inMode)**

The **inMoveData** function is designed to move data back and forth between a specified database and the record transfer buffer assigned to it. Created for the *Coatings Database* application, the **inMoveData** function takes as variable input the integer index of the database file to work with as *inTableName*, the work record to be processed as *chRecord*, and the function to perform as *inMode*. The *inMode* parameter can be set to either ***GET*** or ***PUT***, allowing data to be both extracted from, and saved to, the specified database. The **inMoveData** function performs all functions on the current record of the database, and assumes that the database index passed refers to a database file currently open.

* **inGetData(int inTableName, int inFieldNumber, char *chData)**

The **inGetData** function is designed to extract data from the specified field in the record transfer buffer. The **inGetData** function takes as input the table to operate on in *inTableName*, the field number to get data from in *inFieldNumber*, and the storage location for the data in *chData*. The **inGetData** function is an intelligent general ``Get" function. When invoked, the *field types array* is read based on the current table and field numbers. The resulting field type is parsed to determine the variable type and length. Once determined, the appropriate data get function *(PXGetAlpha, PXGetDate, PXGetDoub, PXGetLong, PXGetShort)* is called to retrieve the data in question. The data is formatted using a call to the *sprintf* command, and the resulting field information is returned in *chData*. The **inGetData** function works on the current record transfer buffer. It assumes that a call to **inMoveData** has already been made to locate the record to work with. In most cases, the **inGetData** function is only called from within the **inMoveData** function.

* **inPutData(int inTableName, int inFieldNumber, char *chData)**

  The **inPutData** is designed to dump specific fields onto the record transfer buffer identified in *inTableName*. The *inFieldNumber* field is used to determine which field in the databse is to have new data ``put" into it, and the *chData* variable holds the new data for the record transfer buffer. By referencing the *field types array*, the correct type for the database field is determined. Using the appropriate ASCII to (integer/long/float) function, the data is converted into a format suitable for saving. Once this has been performed, the appropriate data put function *(PXPutAlpha, PXPutDate, PXPutDoub, PXPutLong, PXPutShort)* is called to record the data on the current record transfer buffer. The **inPutData** function performs no work with the database file itself. Like **inGetData**, a call to **inMoveData** is needed to perform the recording of the current record transfer buffer to the correct record of the database file.

3.2.3 Development of Database Open/Close Function

There was a need in the *Coatings Database* for a single compact function which would handle the opening and closing of individual database files. The function developed **inDBFile(int inTableName, int inMode)** was passed the index to the database file in *inTableName* and the operation to perform in *inMode*.

Valid operations included :

* PX_OPEN - Open the specified database file.
* PX_CLOSE - Close the specified database file.

* PX_KEYADD - Add the *primary key* information to the database file.
* PX_KEYDRP - Drop the *primary key* information from the database file.
* PX_CREATE - Create the specified database file if it does not already exist.
* PX_DELETE - Delete the specified database file from the disk.

The individual operations could be *ored* together to form combinations of functions to perform specific operations. For example, to open a specific database file, and create it if it doesn't already exist, the *inMode* parameter would be **PX_CREATE || PX_OPEN**. The individual **PX_????** mode commands, once inside the **inDBFile** subroutine, call their respective Paradox Engine functions *(PXTblOpen, PXTblClose, PXTblCreate, PXKeyAdd, PXKeyDrop, PXTblDelete).*

3.2.4 Development of Database Search Functions

In order to simplify the steps necessary to perform a search in the *Coatings Database*, the many search functions provided in the Paradox Engine were combined into a single function **inSearchDB**, that would perform both *key* and *field* searches on the databases in our application. As passed parameters, **inSearchDB** took the index to the database file to work with in *inTableName*, the stream of data to work with in *chData*, a numeric variable in *inVar* and a mode of operation in *inMode*.

The value represented by the *inVar* variable was dependant upon the type of data in the indexed database. If the database was a keyed database, *inVar* represented the number of key fields to include in the search being done. Thus, if *inVar* was equal to 2, fields 1 and 2 in the *primary key* would be used for the search. If the database was not a keyed database, *inVar* represented the number of the database field to be used in the search. Thus, if *inVar* was equal to **3**, the search would be performed using field 3 as the search path.

The *inMode* parameter allowed the user to specify the type of search to be performed. Valid values for *inMode* were :

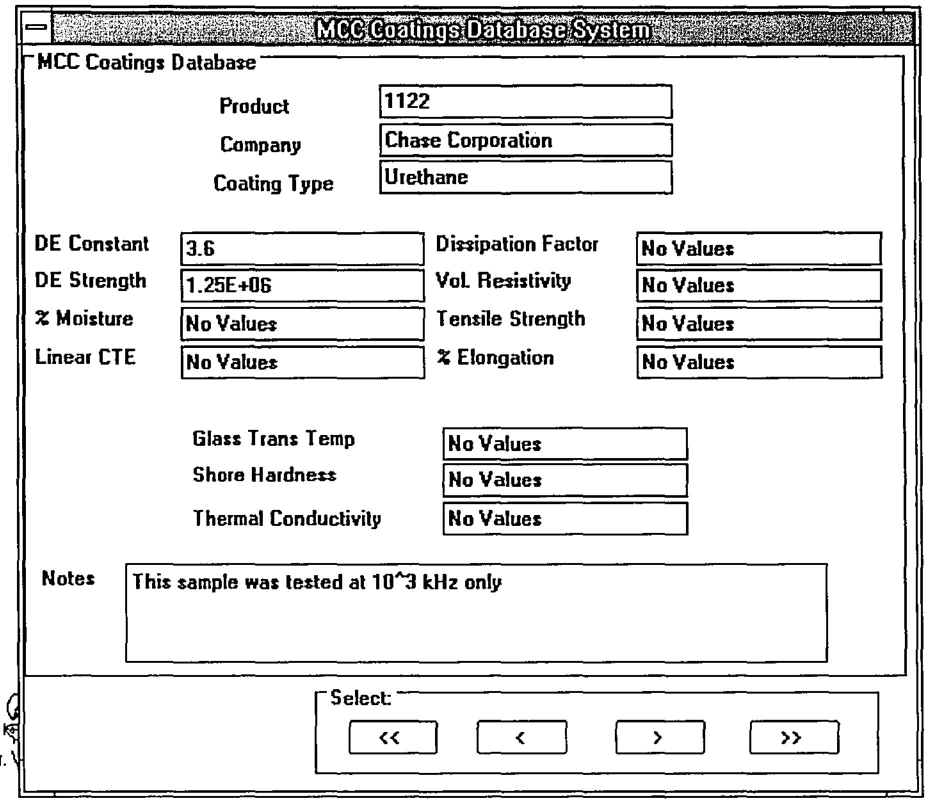
* **DB_SEARCHFIRST** - Search the database for the first occurrance of the specified search match.
* **DB_SEARCHNEXT** - Search the database (starting at the current record) for the next occurrance of the specified search match.
* **DB_SEARCHALL** - Search the database (starting at the first record) for all occurrances of the specified search match.
* **DB_USEKEY** - Use the key fields of the database for the search.
* **DB_USEFIELD** - Use the numbered fields of the database for the search.

The parameters **PX_USEKEY** and **PX_USEFIELD** are modifiers to the search methods. They are designed to override the default settings for the search mode for a specific database. Using these modifiers, it is possible to perform a field specific search on a keyed database. It is not possible, however, to perform a keyed search on a database that is not keyed.

3.3 Development of the Graphical Users Interface

The development of the *graphical users interface* (GUI) for the *Coatings Database* formed the single most important development process of the entire application. If the GUI desgined was difficult to use, or was unable to easily provide the users with the functionality they desired, the entire application would suffer. Because of this, the Microsoft Windows™ was chosen as the platform to be used. The familiarity most users have with the Windows environment, along with the standardization present among the applications designed for Windows, allowed us to create a functional and simple to use interface for the *Coatings Database.*

To simplify the operation of the *Coatings Database*, the system was designed to provide the user with the quickest access to the functions they would require. The main menu of the system (see figure 3.6) constantly displays information for the current coating. The entire list of available fields is presented for the user at this level. Simple viewing of the individual records in the database can be accomplished using the <, <<, >>, and > push buttons located in the bottom center of the display region. The < and > buttons allow for the scrolling of the database one record in either direction (previous or next record). The << and >> pushbuttons will move the current database record location to the beginning or ending record of the database.


MCC Coatings Database System

MCC Coatings Database

Product	1122
Company	Chase Corporation
Coating Type	Urethane

DE Constant	3.6	Dissipation Factor	No Values
DE Strength	1.25E+06	Vol. Resistivity	No Values
% Moisture	No Values	Tensile Strength	No Values
Linear CTE	No Values	% Elongation	No Values

Glass Trans Temp	No Values
Shore Hardness	No Values
Thermal Conductivity	No Values

Notes: This sample was tested at 10^3 kHz only

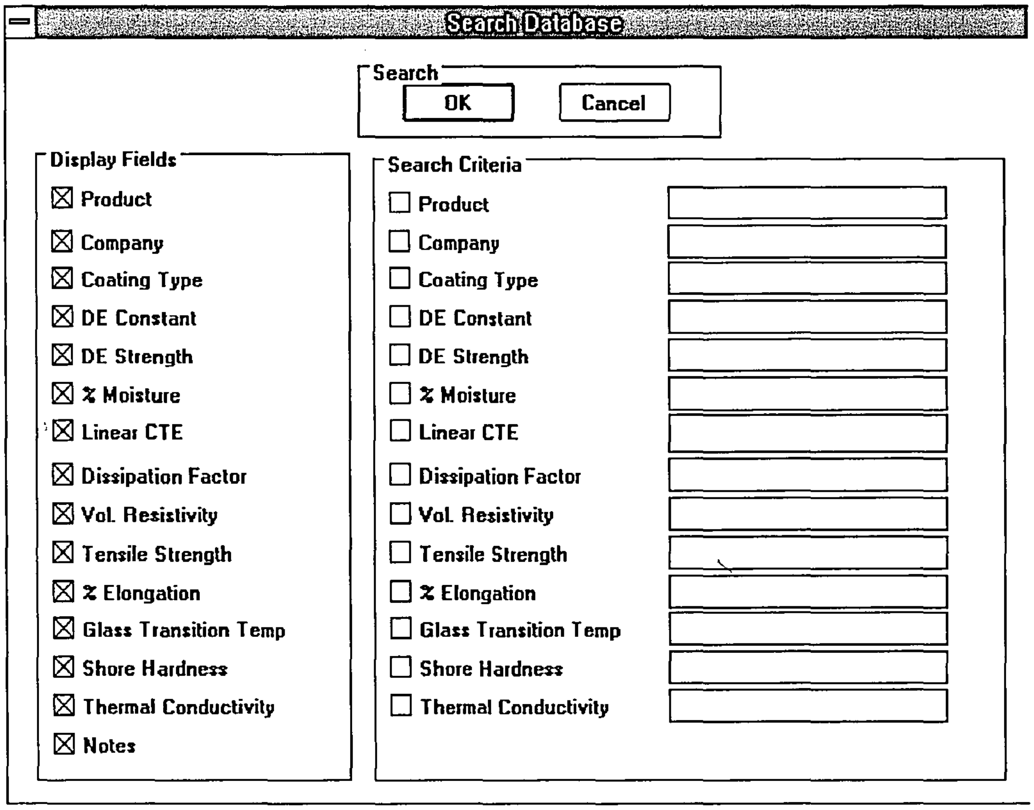
Figure 3.6 : *Coatings Database* Main Menu

Access to the *Coatings Database's* search functions is done using the main menu located across the top of the viewing screen. Selection of the **Search** menu will overlap the *Coatings Database* main menu with the search menu for the system (see figure 3.7). When activated, the search menu provides the user with two sets of selectable fields.

* Display Fields - The user can select the fields that are to be displayed once the search through the database has been completed. The data records that match the current search specifications will be displayed according to the fields designated in this menu.

* Search Criteria - The user can select the search criteria for the database search. The selection of a search criteria is a two step process. By clicking (marking) the mouse over a search criteria to be included in the current search of the *Coatings Database*, the current search parameter is included in the search. The user will be prompted for the

contents of the field to search on. In cases where a numeric value is required, a dialog box will pop up requesting the minimum and maximum ranges for the search field. Once selection of the *Search Criteria* and *Display Fields* has been made, the search can be performed by selection the **OK** button for the search. To cancel the search select the **CANCEL** button.



Display Fields

- ☒ Product
- ☒ Company
- ☒ Coating Type
- ☒ DE Constant
- ☒ DE Strength
- ☒ % Moisture
- ☒ Linear CTE
- ☒ Dissipation Factor
- ☒ Vol. Resistivity
- ☒ Tensile Strength
- ☒ % Elongation
- ☒ Glass Transition Temp
- ☒ Shore Hardness
- ☒ Thermal Conductivity
- ☒ Notes

Search Criteria

- ☐ Product
- ☐ Company
- ☐ Coating Type
- ☐ DE Constant
- ☐ DE Strength
- ☐ % Moisture
- ☐ Linear CTE
- ☐ Dissipation Factor
- ☐ Vol. Resistivity
- ☐ Tensile Strength
- ☐ % Elongation
- ☐ Glass Transition Temp
- ☐ Shore Hardness
- ☐ Thermal Conductivity

Figure 3.7 : *Coatings Database* Search Menu

Once the search has been performed, the system will display a text window which will contain all of the records in the database which match the search criteria specified. In many cases, the window is both too wide and too long to be displayed all at once. In these cases, a horizontal and vertical scroll bar will be displayed across the bottom and right edges of the screen. To move around the display window, click on the arrows located on the two scroll bars. A highlighted block within the range of the scroll bars is used to inform the user as to their current location in the display window.

END OF

TITLE