May 1971

An Extended Data Management Facility for a General Purpose Time-Sharing System

Frank A. Manola
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/828
For more information, please contact libraryrepository@pobox.upenn.edu.
An Extended Data Management Facility for a General Purpose Time-Sharing System

Abstract
Contemporary software forces users to choose between general purpose operating systems with primitive data management facilities, and special-purpose data management systems with little support of programming languages and interactive computation. This is because integrated systems combining these two features have not been available. This report presents the design of an integrated system known as the Extended Data Management Facility (EDMF) which combines the facilities of the RCA Time Sharing Operating System (TSOS) with novel data-handling and access-control capabilities.

The novelties of the EDMF include a generalized file structure, a generalized record organization, an efficient record retrieval algorithm, and access control to files, records within files, and fields within records. A comprehensive set of on-line commands and system macroinstructions are provided for defining and processing files, records, and fields, and for access control. Special programming aids are also provided for defining blocks of commands, called procedures, for later execution.

Keywords
record organizations

Comments
University of Pennsylvania
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
Philadelphia, Pennsylvania

TECHNICAL REPORT

AN EXTENDED DATA MANAGEMENT FACILITY
FOR A GENERAL-PURPOSE TIME-SHARING SYSTEM

by

Frank A. Manola

May 1971

Submitted to the
Office of Naval Research
Information Systems Branch
Arlington, Virginia

under
Contract N00014-67-A-0216-0014
Research Project NR 049-153

Reproduction in whole or in part is permitted for any purpose of the
United States Government

Moore School Report No. 71-24
AN EXTENDED DATA MANAGEMENT FACILITY
FOR A GENERAL-PURPOSE TIME-SHARING SYSTEM

Abstract

Contemporary software forces users to choose between general-purpose operating systems with primitive data management facilities, and special-purpose data management systems with little support of programming languages and interactive computation. This is because integrated systems combining these two features have not been available. This report presents the design of an integrated system known as the Extended Data Management Facility (EDMF) which combines the facilities of the RCA Time Sharing Operating System (TSOS) with novel data-handling and access-control capabilities.

The novelties of the EDMF include a generalized file structure, a generalized record organization, an efficient record retrieval algorithm, and access control to files, records within files, and fields within records. A comprehensive set of on-line commands and system macro-instructions are provided for defining and processing files, records, and fields, and for access control. Special programming aids are also provided for defining blocks of commands, called procedures, for later execution.
Contemporary software forces users to choose between general-purpose operating systems with primitive data management facilities, and special-purpose data management systems with little support of programming languages and interactive computation. This is because integrated systems combining these two features have not been available. This report presents the design of an integrated system known as the Extended Data Management Facility (EDMF) which combines the facilities of the RCA Time Sharing Operating System (TSOS) with novel data-handling and access-control capabilities.

The novelties of the EDMF include a generalized file structure, a generalized record organization, an efficient record retrieval algorithm, and access control to files, records within files, and fields within records. A comprehensive set of on-line commands and system macro-instructions are provided for defining and processing files, records, and fields, and for access control. Special programming aids are also provided for defining blocks of commands, called procedures, for later execution.
<table>
<thead>
<tr>
<th>KEYWORDS</th>
<th>LINK A</th>
<th>LINK B</th>
<th>LINK C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record Organizations</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Fixed-Format Records</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Record with Delimiters</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Extended Data Management Facility (EDMF)</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Indexed Sequential Access Method (ISAM)</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Record Control Block (RCB)</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Tree-Structured Record</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Access Control Capabilities</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Command Language Procedures</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>Service Status Block (SSB)</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
<tr>
<td>File Status Block (FSB)</td>
<td>ROLE</td>
<td>WT</td>
<td>ROLE</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>INTRODUCTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 2</th>
<th>RECORDS AND RECORD ORGANIZATIONS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Introductory Definitions</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2. Record Organization</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>3. Fixed-Format Records</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>4. Records with Variable-Length Fields</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>5. &quot;Free&quot; Format Records</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>6. The Formatted Record and the Concept of a Template</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>7. The EDMF Record</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>7.1 The Header</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>7.2 The Record Control Block</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>7.3 The Text</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>8. The EDMF Template (Record Format Block)</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>9. Record-Template Connections</td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 3</th>
<th>FILES AND FILE STRUCTURES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Overview</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>2. Factors Determining File Structure</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>3. Keywords</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>4. Sequential and Index Sequential File Structures</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>5. Associative File Structures</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>5.1 Hash Coding</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>5.2 Pointers, Indexes, Lists and Directories</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>5.3 The Multilist File Structure</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>5.4 The Inverted File Structure</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>5.5 The Generalized File Structure</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>5.6 The EDMF Implementation of the Generalized File Structure</td>
<td>49</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (continued)

6. Parallel Record Retrieval
   6.1 Selection of Prime Keywords
   6.2 Elimination of Duplicate Record Retrievals
   6.3 Reduction of Movement of the Head-Mechanism

CHAPTER 4  PRIVACY PROTECTION AND ACCESS CONTROL

1. Introduction
2. Typical File Protection and Its Access Control
3. Desirable Privacy Protection and Access Control Capabilities
4. Deficiencies in Commercial Systems
5. EDMF Privacy Protection and Access Control Facilities

CHAPTER 5  COMMAND LANGUAGE PROCEDURES

1. Introduction
2. Standard Procedure Creation and Use
3. Extended Procedure Creation and Use
   3.1 Passing Parameters to Programs in a Procedure
   3.2 Defining and Substituting Formal Command Names and Their Parameters
   3.3 Local Abbreviations and Command-Program Mix in Procedures

CHAPTER 6  EDMF FUNCTIONS AND PROGRAMS

1. The OPEN Function
2. Control Blocks and Access Management
   2.1 Service Status Block (SSB)
   2.2 Interrogation of the List of Service Status Blocks
   2.3 The File Status Block (FSB)
   2.4 Creating and Finding a File Status Block
3. The CLOSE Function
TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.</td>
<td>The RETRIEVE Function</td>
<td>102</td>
</tr>
<tr>
<td>5.</td>
<td>The CONTINUE Function</td>
<td>105</td>
</tr>
<tr>
<td>6.</td>
<td>The STORE Function</td>
<td>109</td>
</tr>
<tr>
<td>7.</td>
<td>The UPDATE Function</td>
<td>114</td>
</tr>
<tr>
<td>8.</td>
<td>The DELETE Function</td>
<td>115</td>
</tr>
<tr>
<td>9.</td>
<td>Access Control Functions and Routines</td>
<td>116</td>
</tr>
<tr>
<td>9.1</td>
<td>The Authority Item</td>
<td>116</td>
</tr>
<tr>
<td>9.2</td>
<td>Storage, Retrieval, and Update of Authority Items</td>
<td>119</td>
</tr>
<tr>
<td>9.3</td>
<td>Access Control Considerations for File Creation and Deletion</td>
<td>122</td>
</tr>
<tr>
<td>9.4</td>
<td>Assignment of Access Rights</td>
<td>123</td>
</tr>
<tr>
<td>9.5</td>
<td>Revocation of Access Rights</td>
<td>124</td>
</tr>
<tr>
<td>9.6</td>
<td>Display of Access Rights</td>
<td>125</td>
</tr>
<tr>
<td>9.7</td>
<td>Access Verification and Control of User Requests</td>
<td>125</td>
</tr>
<tr>
<td>7.</td>
<td>SYSTEMS PROGRAMMING OF THE EDMF</td>
<td>127</td>
</tr>
<tr>
<td>1.</td>
<td>Interface Considerations</td>
<td>127</td>
</tr>
<tr>
<td>1.1</td>
<td>The Basic EDMF Input/Output Operations</td>
<td>132</td>
</tr>
<tr>
<td>1.2</td>
<td>Interfacing the TSOS File Protection Facilities with the EDMF</td>
<td>133</td>
</tr>
<tr>
<td>2.</td>
<td>Systems Programming Techniques</td>
<td>134</td>
</tr>
<tr>
<td>2.1</td>
<td>Program Coding</td>
<td>134</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Choice of Systems Programming Language</td>
<td>134</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Use of Interactive Program Creation Techniques</td>
<td>135</td>
</tr>
<tr>
<td>2.2</td>
<td>Program Testing</td>
<td>136</td>
</tr>
<tr>
<td>2.2.1</td>
<td>System Generation and TESTSYS</td>
<td>136</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Interactive Debugging</td>
<td>138</td>
</tr>
<tr>
<td>2.3</td>
<td>System Backup</td>
<td>139</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (continued)

CHAPTER 8 FUTURE WORK AND EXPERIMENTATION 141

1. Use of the EDMF to Support Advanced Applications 141

2. Use of the EDMF as an Experimental Tool 143

2.1 Multiple Record Format Types 144
2.2 Memory Acquisition and Release 145
2.3 Program Substitution Facility 145
2.4 List Processing Capability 146
2.5 Lower-Level I/O 146

REFERENCES 148

APPENDIX: SYSTEM FLOWCHART OF EDMF ROUTINES 152
CHAPTER 1
INTRODUCTION

Contemporary general-purpose time-sharing systems provide programming language, file-handling and interactive facilities to many users simultaneously; and special-purpose information storage and retrieval systems provide advanced data-handling capabilities allowing the user to store and retrieve data by logical expressions of keywords which characterize the data. The power inherent in these two developments, however, cannot be fully realized unless integrated systems which combine these features become available. This is because general-purpose time-sharing systems do not provide for the advanced data-handling capabilities available on information storage and retrieval systems, while information storage and retrieval systems do not provide the user with comprehensive sets of programming languages and interactive computational facilities. It is felt that an integrated system with both types of capabilities would be able to handle a far wider range of applications than either type of system alone could.

In this report, the design of an integrated system known as the Extended Data Management Facility (EDMF) is presented. This Facility combines the RCA Time Sharing Operating System (TSOS) with novel data-handling and access control capabilities to be discussed in sequel.

The EDMF provides a generalized record organization which separates structural information common to all records from information which differs from record to record within the file. This organization enhances
flexible use and creation of records, reduces the time required for real-time modification of records, and minimizes record storage requirements. The generalized record organization is discussed in Chapter 2.

The EDMF also provides a generalized file structure which includes among other new file structures the contemporary indexed-sequential, inverted, and Multilist as special cases. The generalized file structure allows the user to determine the optimum structure or to select from among a large number of standard structures for his file. Along with the generalized file structure, the EDMF provides an efficient record retrieval algorithm which minimizes the movement of the access mechanism of direct access storage devices with moveable heads. The generalized file structure and retrieval algorithm are discussed in Chapter 3.

Control of user's access to data is provided by the EDMF and is discussed in Chapter 4. The access control facilities provide protection and control for whole files, records within files, and fields within records. Information access rights are associated with individual users, rather than with files as in most contemporary systems. These features enable an owner of information to define the access rights of each potential user of his information on an individual basis, to partition a file into sections with different protection measures, and to determine the information in a record that each user is entitled to receive.

A set of processing functions (available through on-line commands and system macro-instructions) is provided by the EDMF for specifying file structures and record organizations, for creating, modifying, and
deleting files, and for storage and retrieval of information using logical and arithmetic expressions of keywords. Other functions enable the file owner to exercise privacy measures and access control over his data. These EDMF functions are described in Chapter 6.

Special programming aids are provided for creating and executing blocks of system commands and user programs (called "procedures") which can be stored for repetitive use. These aids are discussed in Chapter 5.

In Chapter 7, the techniques for the implementation and testing of the EDMF are presented. The appendix contains a flowchart showing the interrelated EDMF program modules.

Finally, in Chapter 8, the EDMF is discussed in two roles: as a system programming tool for experimenting with future data management and information systems on the one hand, and as an operating system for the support of advanced and diverse applications on the other hand.
CHAPTER 2
RECORDS AND RECORD ORGANIZATIONS

1. Introductory Definitions

Under the common usage of the term, a field or data item, is the smallest piece of information distinguishable by the data management system and application programs referring to it. That is, the field is the smallest unit of information of interest to the user. A field may be a string of characters or numbers.

When discussing fields, it is often useful to separate the term describing or naming the field from the actual contents of the field. This is done by referring to the term describing the field, or field identifier, as an attribute, and the contents of the field as the value of the attribute. For example, the first five columns of a punched card containing a FORTRAN statement are used for the statement number. Thus, the attribute of this field may be "statement number", while the possible values of this attribute consist of all 5 digit integers (including such numbers as 00001). This means that a field may be considered to be the pair of the attribute describing the field and the particular value contained in the field, i.e., an attribute-value pair.

A logical or user record is a collection of attribute-value pairs in which there is a well-defined relationship among the attribute-value pairs of the collection. This relationship is called the record organization. An example of a particular type of record organization, a tree structure, is shown in Figure 2.10. It is
important to distinguish between the logical record and the physical record which is used by a particular data management system to store and retrieve the logical record, and which represents the logical record by various combinations of bits and bytes in computer storage. The user must supply not only the characters which make up the attributes and values of logical records being stored, but also must provide sufficient information to describe the logical record organization. However, the fact that logical records are represented in the form of physical records may be transparent to the user. In other words, the user always sees his records as logical records and only the data management system handles them as physical records.

Naturally, the physical record has its own organization, which determines where and how the information from the logical record is physically placed. In the following sections of this chapter, the term 'record' will be used to mean 'physical record' and various types of physical record organizations will be discussed, from the simplest to the most complex ones. It will be shown that the simple types of physical records are inadequate to store logical records with complex record organizations which are essential in modern data processing in such a way as to provide efficient storage and retrieval and space utilization. It will then be shown that a particular physical record organization, the EDMF record and template, has great advantages in storage and retrieval, economical use of storage space, and speed of updating.
2. Record Organization

Once a processing program has been given a record, it should require only two pieces of information in order to process a field of the record: where the field is in the record, and how long the field is in basic machine units, i.e., words, characters, bytes, etc. This information is syntactic or structural information, as opposed to semantic information. The semantic information concerns what the processing program is to do with the data in a field once the field is located. The structural information may be placed in a) in the record itself, b) implicit in the program, c) explicit in the program, d) in a separate control block, or e) in a combination of these locations. It is the purpose of this section to discuss record organization, and explore the various types of records whose structural information could fall into the above categories.

3. Fixed-Format Records

The simplest type of record is typified by the FORTRAN input card. In the FORTRAN card, there are four basic fields: columns 1-5 used for the statement number, column 6 used for continuation, columns 7-72 used for the FORTRAN statement, and columns 73-80 used for card sequencing or other user data (see Figure 2.1). The important fact is not that there are only four fields, but that the four fields are each fixed length, have fixed position with respect to each other, occur a fixed number of times on the physical medium (once each), and occur in the same place in each physical record (80-column punched card). With this record organization the processing program always knows where to find any given field, because it is in the same place in every record. Thus,
1. columns 1-5: attribute-STATEMENT NUMBER, value - 5 digit statement number
2. column 6: attribute-CONTINUATION INDICATOR, value - character denoting continuation or blank
3. columns 7-72: attribute-FORTRAN STATEMENT, value - any FORTRAN statement
4. columns 73-80: attribute-USER DATA, value - anything

Figure 2.1 A Fixed-Format Record
the organization of the record can be specified into the program implicitly, that is, the program instructions can be written under the assumption that the individual data fields are in the same place in each record and are of fixed length. For example, the FORTRAN compiler does not have to search the 80-column input records for the statement number, but rather uses the first 5 columns of the record as a matter of course (provided they are non-zero and not a comment).

Unfortunately, most data is not of the same simple organization as FORTRAN statements. Naturally occurring data may have repeating fields which occur a variable number of times, may often not occur at all, and may have variable lengths. Such data could not easily fit into the record organization described above, if at all. Thus there is a lack of flexibility in the fixed-format record organization which often necessitates the use of more complicated record types.

4. Records with Variable-Length Fields

An extension to the type of record organization described above is to allow variable-length fields, while still requiring rigid order of fields and fixed repetitions. In order to do this, the processing program must be informed how long each field is, so it can find where one field ends and another starts. This can be done by providing explicitly a length indicator for each field, as shown in Figure 2.2, or by having a delimiting character (mark the end of each field) so that the program can determine for itself how long the field is, as in Figure 2.3. In the past many data processing applications have made use of delimiting characters especially on such equipment as the IBM 1400 series, where special hardware facilitated the identification of
<table>
<thead>
<tr>
<th>length-field 1</th>
<th>field 1, ( n_1 ) bytes</th>
<th>length-field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>field 2, ( n_2 ) bytes</td>
<td>length-field 3</td>
<td>field 3, ( n_3 ) bytes</td>
</tr>
</tbody>
</table>

* Note that length indicator fields must be fixed length - their structure is implicit in the program.

Figure 2.2 Record With Explicit Length Indicators

<table>
<thead>
<tr>
<th>field 1, ( n_1 ) bytes</th>
<th>field 2, ( n_2 ) bytes</th>
<th>( n_3 ) bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>delimiter, 1 byte</td>
</tr>
</tbody>
</table>

Figure 2.3 Record With Delimiters
delimiters and their use by the programs. However, recently the increased use of universal character sets which allow any possible bit combination to be legitimately used as data prevents any character to be used in data fields as a delimiter.

Even when variable-length fields are allowed, most of the record structural information is implicitly contained in the processing program. The record organization is thereby made very rigid in the sense that variable repetitions of fields are not possible, fields must occur in the same order in all records, and all fields must occur in all records. This is because the programming language which specifies the record organization cannot handle records with sufficient generality and different programs are required to handle different types of records. This can be a serious handicap when dealing with certain types of records, such as data on books (multiple authors, no authors, editors but no authors) or inventory records (variable suppliers, prices). Even more general structures may be required for such applications as graphics, or certain system files. Consequently, any time any major portion of the record organization is implicitly specified in a program, there is inflexibility, which leads to restrictions on the applicability of the data management programs on the one hand, and a high (and in many cases redundant) programming effort and use of storage space on the other. It would thus be an advantage to a data management system to be able to handle extremely flexible record organizations, such as those described in the next section by placing little structural information implicitly in processing programs.
5. "Free" Format Records

The previous paragraph described a situation in which the program was assumed to know most of the record organization, and it was shown that this could lead to very inflexible record organizations. One possible alternative is to assume that the program knows only that certain fields may be present in the record, and that it must find out which fields are actually present in any given record through structural information contained in the record itself. In other words, as much structural information as possible is placed in the record. This means that both length information for each field and some sort of field identification information must be present so that in scanning a record the processing program knows not only how long the field is but also which field it is. Such information could be provided by placing the attribute name in the record for each field, as shown in Figure 2.4, along with the length information. This would allow completely free format records, as the processing program would be able to find any given field by scanning for the appropriate attribute.

The disadvantage of this approach is that each record becomes extremely large, since not only all the values of the field must be present but all their attributes (field names) as well. This is especially disadvantageous when fields repeat (occur several times). In this case repeated attribute names are redundant. Furthermore, it is also well to remember that, when records are collected into a file, the usual case is that all the records have similar attributes, because they contain the same type of information. For example, all records in a file of library books are likely to contain the attribute "Author".
<table>
<thead>
<tr>
<th>FIELD1</th>
<th>length of field 1</th>
<th>field 1, $n_1$ bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD2</td>
<td>length of field 3</td>
<td>field 3, $n_3$ bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>field 2, $n_2$ bytes</td>
</tr>
</tbody>
</table>

Figure 2.4  Record With Explicit Length Indicators and Field Names
Thus it is reasonable to expect that there are only a limited number of different attributes in a file, even though this number may be large. Now, suppose there are 100 attributes in a file with 10,000 records. Each of the 100 attributes would be repeated quite often, and very likely some would be repeated the full 10,000 times (any field that appeared in every record would do so). Thus the attribute names are repeated in the file a large number of times in records of this type, resulting in much wasted space.

6. The Formatted Record and the Concept of a Template

The last section presented a free-format record in which structural information was interspersed with the actual data values contained in the record. There is another type of record organization, called the formatted record, in which all the structural information is placed in one part of the record, and the data values placed in another part. In such a record, the structural information is contained in a series of format entries which would then be linked in some way to the actual data values. It is important to note, however, that the formatted record of this type does not overcome the extreme space penalties imposed by the retention of all structural information within the record itself. It merely concentrates all the structural information in one place.

To circumvent this problem a novel approach is employed for the formatted record organization by separating generalized structural information about all the records in a file from the individual records, and placing this information in a central control block. This information including attributes (field names), lengths of fixed length fields, and other non-varying information are placed in this central control
block, which serves as a template for all the records of a file. Other information, which varies from record to record, such as lengths of variable-length fields, as well as sufficient linkage information to allow structural information in the template to be connected with actual values (data items) in the records, must be retained in the records themselves. When planned properly, the amount of retained information is less than the average amount of information that would have been present in a free-format record. Hence this approach not only retains the flexibility of the free-format record but also minimizes the amount of storage space for redundant information. Of course, it would be possible to construct a formatted record without the flexibility of the free-format records. For example, a template could be constructed for a fixed format record such as the FORTRAN statement, which might contain the field names. However, this in itself would not make the record any more flexible. It is when the generality of the free-format record is desired without the space penalty and the concept of record template is implemented, that the formatted record organization can truly result in greater flexibility and usefulness. A more complete discussion of the theory of the formatted record can be found in [19,12,13].

7. The EDMF Record

The record used in the EDMF is a formatted record of the most general type. When used with its associated template, the EDMF record provides flexibility without space penalties, as well as features which allow it to be easily updated, and which allow for future inclusion of record reorganization capabilities. Figure 2.5 shows the form of the EDMF record. As can be seen, the record consists of three parts, a header,
<table>
<thead>
<tr>
<th>3 bytes</th>
<th>Size of Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>Count</td>
</tr>
<tr>
<td>5 bytes</td>
<td>Disk Address (ISAM Key)</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Pointer to Text (values) relative to 1st byte of record</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Control Information</td>
</tr>
</tbody>
</table>

- **HEADER**

<table>
<thead>
<tr>
<th>2 bytes</th>
<th>Format number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 bytes</td>
<td>Size of value</td>
</tr>
<tr>
<td>1 byte</td>
<td>blank</td>
</tr>
<tr>
<td>1 byte</td>
<td>Control Information</td>
</tr>
<tr>
<td>3 bytes</td>
<td>Relative address of first value</td>
</tr>
</tbody>
</table>

- **RECORD BLOCK**

<table>
<thead>
<tr>
<th>2 bytes</th>
<th>Format number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 bytes</td>
<td>Size of value</td>
</tr>
<tr>
<td>1 byte</td>
<td>blank</td>
</tr>
<tr>
<td>1 byte</td>
<td>Control Information</td>
</tr>
<tr>
<td>3 bytes</td>
<td>Relative address of second value</td>
</tr>
</tbody>
</table>

... 

<table>
<thead>
<tr>
<th>5 bytes</th>
<th>Pointer (ISAM Key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n bytes</td>
<td>Value of attribute</td>
</tr>
</tbody>
</table>

... 

**Notes:**

1. The Count field is used when a record is broken into several pieces for I/O.

2. The Control information in the header is used among other things to indicate that a record is to be deleted. This is also true for the Control information in the Record Control Block.

3. The Size of Value in the RCB appears even though the format may be fixed-length.

**Figure 2.5 The EDMF Record**
a Record Control Block, and a text section. Each of these parts will now be discussed in detail.

7.1 The Header

The header of the EDMF record contains information used by the lowest level EDMF I/O programs and important in dealing with the record as a whole. Some of this information is easily explained. For example, the size of record entry is a 3 byte field which contains a binary representation of the size in bytes of the entire EDMF record. The Pointer to Text entry gives the relative address of the first byte of the text section of the record, counting from the first byte of the header. EDMF programs which need to find the record text may do so knowing only the address of the first byte of the whole record by using this Pointer to Text entry. The Control Information entry is used to contain various internal codes associated with the record. For example, if the deletion bit in the Control Information entry is set, then the record has been logically deleted from the file, even though it may still be physically present.

Before discussing the Count and Disk Address entries, it is important to discuss how the EDMF record is stored on physical storage devices, and how I/O is performed in the EDMF. The EDMF uses the RCA Indexed Sequential Access Method (ISAM) as its basic I/O method. The reasons for this choice are discussed in Chapter 6. In simple terms, ISAM allows records to be stored and retrieved on the basis of one 8-digit (decimal) key assigned to each ISAM record, and provides the programming to accomplish the storage and retrieval to and from the physical devices. Thus, an ISAM record may be read into main storage from
secondary storage by calling ISAM with the 8-digit key of the record desired. ISAM, however, imposes many restrictions on the records which it will handle, such as length, format, etc. The EDMF, therefore, uses ISAM records to implement the much more general EDMF record. Since EDMF records have any length up to 16,777,216 bytes while ISAM records are restricted to a length of 255 bytes, EDMF records may have to be broken up into a number of ISAM records before being written to disk. An exact description of this process may be found in [10], which describes the basic I/O mechanism of the EDMF. The Count field is used to hold a number telling the EDMF how many ISAM records were required to hold the entire EDMF record, and the Disk Address field holds the ISAM key of the first ISAM record in the sequence of ISAM records holding this EDMF record. Chapter 6 of this paper contains a more precise description of this process, along with the reasons why this method of implementation was chosen.

7.2 The Record Control Block

The Record Control Block (RCB) is the portion of the EDMF record which contains the structural information which varies from record to record, and at the same time serves to connect the actual values (data items) in the text section of the record with the structural information in the template. There is an entry in the Record Control Block for each value contained in the record. The RCB entries appear in the logical order in which the values are supposed to be processed. Thus, the RCB is an exact guide as to how the record text is to be processed.

The connection between the EDMF record and the template is established by means of the Format Number entry. This format number refers to an entry in the Record Format Block, the EDMF template,
with the same format number, and which contains structural information
about the field associated with that particular entry. Thus the Record
Control Block entry contains the number of the template entry which is
to control processing of the field (value) referred to by the RCB entry.
Further discussion of this connection will be found in Sections 2.8 and
2.9.

The Size of Value and Control Information entries in the RCB entry
are used in the same way as the corresponding entries in the record
header. Since, for variable length fields, the length may change from
record to record, the Size of Value entry is necessary in each record.
Notice that a blank byte is kept for future expansion.

The Relative Address entry contains the address of the value to
which the RCB entry refers, relative to the first byte of the record
text (recall that the address of the first byte of the record text
relative to the first byte of the entire record is contained in the
header). The use of relative addresses all through the records allows
for the easy updating of records. For example, consider the simple
record shown in Figure 2.6 and suppose you wish to add a field to the
record (to simplify the figure, only fields mentioned in the description
are shown, although all relative addresses are calculated as if all
fields, e.g., control information were present). Let the field to be
added be 30 bytes in length and be associated with the format number 3.

Since this is an EDMF record, the addition of a field means that
not only must a value (the contents of the field) be added, but also
a Record Control Block entry to link the value to its associated entry
<table>
<thead>
<tr>
<th>Format Number</th>
<th>Size of Value</th>
<th>Relative Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>25</td>
</tr>
</tbody>
</table>

| 0            | 5 byte value for format 1 |
| 10           | 10 byte value for format 2 |
| 25           | 3 byte value for format 4 |

**Figure 2.6 EDMF Record**
in the Record Format Block. For reasons discussed in Section 2.9, order is important in the Record Control Block, the appropriate RCB entry must therefore be added to the RCB in the proper order. By assuming that the format numbers reflect the required order, the new RCB entry is entered between the last two RCB entries in Figure 2.6. However, since the RCB entries use relative addresses to point to their values, the actual 30 byte value to be added to the record may be added at the end of the text section rather than between the last two text entries. Now, only the Pointer to Text entry in the record header must be changed to reflect the fact that the RCB is longer by one entry, and the changes are complete.

The updated record is shown in Figure 2.7. Notice that none of the old RCB entries are changed, due to the use of the relative address technique. In fact, only one existing piece of information, the Pointer to Text, had to be changed in adding a new field to the record (and this change consists of adding a constant to the existing value).

7.3 The Text

The text section of the EDMF record contains the actual values of the attributes of the record, i.e., the data items. Associated with each value is a pointer which contains the ISAM key of the next record in the file containing that attribute-value pair, in a manner to be discussed in Chapter 3 on file structures. For each value, there is an entry in the RCB which points to it, although because of the required ordering of RCB entries, the text entries need not be in any specific order. Naturally, in a typical record, the text section is by far the largest section of the record.
<table>
<thead>
<tr>
<th>Field Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Record</td>
<td>122</td>
</tr>
<tr>
<td>Pointer to Text</td>
<td>54</td>
</tr>
<tr>
<td>Format Number</td>
<td>1</td>
</tr>
<tr>
<td>Size of Value</td>
<td>5</td>
</tr>
<tr>
<td>Relative Address</td>
<td>0</td>
</tr>
<tr>
<td>Format Number</td>
<td>2</td>
</tr>
<tr>
<td>Size of Value</td>
<td>10</td>
</tr>
<tr>
<td>Relative Address</td>
<td>10</td>
</tr>
<tr>
<td>Format Number</td>
<td>3</td>
</tr>
<tr>
<td>Size of Value</td>
<td>30</td>
</tr>
<tr>
<td>Relative Address</td>
<td>33</td>
</tr>
<tr>
<td>Format Number</td>
<td>4</td>
</tr>
<tr>
<td>Size of Value</td>
<td>3</td>
</tr>
<tr>
<td>Relative Address</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>5 byte value for format 1</td>
</tr>
<tr>
<td>10</td>
<td>10 byte value for format 2</td>
</tr>
<tr>
<td>25</td>
<td>3 byte value for format 4</td>
</tr>
<tr>
<td>33</td>
<td>30 byte value for format 3</td>
</tr>
</tbody>
</table>

Figure 2.7 EIMF Record After Update
8. The EDMF Template (Record Format Block)

In preceding sections, mention was made of "structural information" which was to be stored in the template, without describing exactly what information this was. In the EDMF, the template, called the Record Format Block (RFB), contains a set of entries containing this structural information, one entry, called a Format, for every field which may appear in any record of the file (as noted before, there is only one RFB, or template, for a file). Figure 2.8 shows the form of the Record Format Block. Notice that the RFB is similar in form to the EDMF record, having a header, a table of contents which is similar to the Record Control Block, and a series of format entries similar to the text in an EDMF record. The RFB uses the same relative addressing scheme as does the EDMF record, and for the same reason; it simplifies updating the RFB. Each piece of information appearing in the format entry will now be explained in detail:

The format number in each format is used as a link with the format numbers appearing in the Record Control Block of each EDMF record. The format number required for any processing function may be found and matched in either the EDMF record or in the Record Format Block. The Type of Format entry is a code which can be used to determine how the information associated with that format is to be processed, and is not currently used in the system. Sufficient space (4 bytes) has been reserved to allow the address of a format processing program, or other address, to be placed in this location. The Repetition Number and Size of Value entries are only useful if the corresponding information is of fixed size. For example, if a field repeats a fixed number of times,
Record Format Block

<table>
<thead>
<tr>
<th>4 bytes</th>
<th>Control Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>Pointer to first format</td>
</tr>
<tr>
<td></td>
<td>relative to first byte of RFB</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Last format number assigned</td>
</tr>
<tr>
<td></td>
<td>HEADER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 bytes</th>
<th>Format number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>Control information</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Relative address of first format</td>
</tr>
<tr>
<td></td>
<td>TABLE OF CONTENTS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 bytes</th>
<th>Format number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>Control information</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Relative address of second format</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 bytes</th>
<th>Format number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>Type of format</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Level number</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Repetition number</td>
</tr>
<tr>
<td>3 bytes</td>
<td>Size of value</td>
</tr>
<tr>
<td>1 byte</td>
<td>Control information (see Figure 2.9)</td>
</tr>
<tr>
<td>2 bytes</td>
<td>blank</td>
</tr>
<tr>
<td>4 bytes</td>
<td>Field protection data</td>
</tr>
<tr>
<td>n bytes</td>
<td>Full attribute name</td>
</tr>
</tbody>
</table>

|         | FORMAT ENTRY |

Notes: 1. All relative addresses in the Table of Contents are relative to the first byte in the first format, hence a pointer to the first format is placed in the header. This arrangement obviates the need for changing relative addresses in the Table of Contents if new formats are added to the block.

2. Format numbers appear in the Table of Contents in order of their appearance in file records.

3. The Type of Format field may be used to indicate a program which processes the format.

4. Like the size of value entry, the repetition number will not appear in the format if the format may repeat a variable number of times. Variable repetition is indicated by a bit in the control information.

Figure 2.8 The EDMF Template
Record Format Block
Control Information

The control information byte denoted by an asterisk in the format entry of the Record Format Block is specified below.

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 (reset) repetition number is variable</td>
</tr>
<tr>
<td></td>
<td>1 (set) repetition number is fixed</td>
</tr>
<tr>
<td>1</td>
<td>0 value size is variable</td>
</tr>
<tr>
<td></td>
<td>1 value size is fixed</td>
</tr>
<tr>
<td>2</td>
<td>0 attribute is not in the directory</td>
</tr>
<tr>
<td></td>
<td>1 attribute is in the directory</td>
</tr>
<tr>
<td>3</td>
<td>0 attribute optionally appears in a record</td>
</tr>
<tr>
<td></td>
<td>1 attribute appears in every record</td>
</tr>
<tr>
<td>4,5</td>
<td>00 value is decimal (unpacked)</td>
</tr>
<tr>
<td></td>
<td>10 value is alphabetic</td>
</tr>
<tr>
<td></td>
<td>01 unassigned</td>
</tr>
<tr>
<td></td>
<td>11 unassigned</td>
</tr>
<tr>
<td>6,7</td>
<td>unassigned</td>
</tr>
</tbody>
</table>

Figure 2.9 EDMF RCB Control Information
then this information is contained in the format for that field. If the size of value for a field is fixed, then this information is also contained in the format for that field. Otherwise, this information must be found in the Record Control Block of each record associated with that RFB. The Field Protection Data entry contains information used by the EDMF access control routines in connection with the security checking procedures covered in Chapter 4. The Full Attribute Name entry simply contains the attribute name for the field to which the format refers. This name is connected to the values with which it is associated during EDMF retrievals and output in various forms to the user, to enable him to identify the fields.

The Control Information entry is explained in Figure 2.9. The first bit tells EDMF processing routines whether the repetition number is fixed, and can therefore be found in the RFB, or variable, and must be found by scanning the RCB of each record. The second bit performs a similar function for the value size, informing the EDMF routines where to look for this information. The third bit informs the EDMF whether the attribute is in the directory (and may therefore be used as part of a keyword in retrieval requests) or not. The fourth bit informs the EDMF whether or not the field is required in every record. If a required field is not present in a record, it will be rejected by EDMF storage routines. The next two bits inform the EDMF how the value is to be padded in the field if the field is fixed length, to the left with zeros in case of numbers, or to the right with blanks in case of alphabetic information.
The Level Number entry can be used to establish a hierarchical relationship between fields of a record. By use of this entry, attributes and values can be arranged in a tree structure in which a group of fields may be defined as a subtree of a record. This entry is similar to the COBOL level number. Figure 2.10 shows a record defined with level numbers for the fields, and the corresponding tree structure. (See [1], [10], and [19].) It should be noted that, when fields of different levels are used, the meanings of the required-or-optional-occurrence entry in the control information changes. If all fields in a record are defined as having the same level, then a field marked 'required' must appear in every record. On the other hand, if a field at level 1 is marked 'optional' and a field defined as being at level 2, a subfield of the optional level 1 field, is marked 'required', then the level 2 field must appear only if the level 1 field is present, and may not appear if the level 1 field is not present, i.e., fields are required relative to fields of which they are subfields.

9. Record-Template Connections

As an example of how the Record Format Block and Record Control Block connect together and are utilized in processing, consider the hierarchical structure shown in Figure 2.11. The Record Format Block corresponding to this structure is shown in the same figure. Notice that the attribute TITLE is repeated in the record, while it appears only once in the RFB. The format entry for the field will indicate that the attribute repeats. If attributes PUBLISHER and DATE are indicated as 'required', then they will repeat if TITLE does. Also shown in the figure is an EDMF record with the hierarchical structure shown. Note
Record

1 NAME
   2 FIRST NAME
   2 LAST NAME
   2 MIDDLE NAME

1 LOCATION
   2 HOME
      3 STREET ADDRESS
      3 CITY
      3 STATE
   2 BUSINESS
      3 STREET ADDRESS
      3 CITY
      3 STATE
      3 FIRM NAME

1 SOCIAL SECURITY NUMBER

Figure 2.10 A Tree-Structured Record
(AUTHOR, JOHN JONES)
(TITLE, COMPUTER PROGRAMMING)
(PUBLISHER, MCGRAW HILL)
(DATE, 1959)
(TITLE, ADVANCED TOPOLOGY)
(PUBLISHER, PRENTICE-HALL)
(DATE, 1962)
(ADDRESS, BOSTON, MASS.)

A user or logical record.

<table>
<thead>
<tr>
<th>Record Format Block</th>
<th>An EDMF Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>format number</td>
<td>format number</td>
</tr>
<tr>
<td>level number</td>
<td>relative address</td>
</tr>
<tr>
<td>relative address</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>(0)</td>
</tr>
<tr>
<td>1</td>
<td>(1)</td>
</tr>
<tr>
<td>2</td>
<td>(2)</td>
</tr>
<tr>
<td>3</td>
<td>(3)</td>
</tr>
<tr>
<td>4</td>
<td>(4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table of Contents</th>
<th></th>
<th>Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>AUTHOR</td>
<td>JOHN JONES</td>
</tr>
<tr>
<td>51</td>
<td>TITLE</td>
<td>COMPUTER PROGRAMMING</td>
</tr>
<tr>
<td>67</td>
<td>PUBLISHER</td>
<td>MCGRAW-HILL</td>
</tr>
<tr>
<td>78</td>
<td>DATE</td>
<td>1959</td>
</tr>
<tr>
<td>99</td>
<td>ADDRESS</td>
<td>1962</td>
</tr>
</tbody>
</table>

Values (text)

Figure 2.11 A Template and EDMF Record Before Updating
that the Record Control Block has as many entries as there are fields in the records. When the EDMF record is read into storage, the processing routines go down the entries in the Record Control Block, picking off from the text section the number of bytes specified in the RCB entry, and applying the specified format to the value.

Now, to show how the record and template organization facilitate updating, suppose that two new attributes LOCATION and AGE are to be added to the records of the file, so that the records of the file now have the structure shown in Figure 2.12. The procedure to be followed to add these new fields to the records of the file is as follows:

1. Add the new formats to the end of the Record Format Block, and assign them new format numbers.

2. Add entries for the new formats in the table of contents of the Record Format Block in the proper order to reflect the new record structure.

3. For each record which is to have the new fields added, add the new values to the end of the record.

4. Using the control information in the Record Format Block, generate a new Record Control Block for the record.

A new record and the new Record Format Block are also shown in Figure 2.12. Notice that the relative address technique used enables the new fields to be added with very little computation. This was illustrated in Section 2.7.2. The new values and formats are simply added at the end of the existing information, and the new control information is inserted in place without the necessity of recomputing the relative locations of the previous information. It is also important to note
Figure 2.12
A Template and EDMF Record After Updating
that old records, such as the one in Figure 2.11, may be processed without change by the new Record Format Block, in spite of the fact that the old record does not contain the new fields. Since the Record Control Block of the old record does not refer to either of the new format entries, their existence in the Record Format Block is ignored by the internal processing routines during retrieval of old records (of course, records with the old structure could not be stored in the system unless the new fields added have been specified as optional). This capability allows new information to be added to selected records in a file without requiring the reprocessing of every record in the file.
CHAPTER 3
FILES AND FILE STRUCTURES

1. Overview

In the previous section, the word "record" was used to describe a set of fields collected together for storage and retrieval purpose. In like manner, the word "file" is used to describe a set of records collected together for the same use in some applications. While they are not theoretically required, the records of a file are required in practice to be similar in the sense that they contain mostly the same attributes. In the EDMF, the similarity among records of a file is evident by the fact that all records in the file are specified by the same record template (RFB). The template determines the "organization" of each record, i.e. how the individual fields are related to one another in the record. Just as the record may have an organization, the file may have a structure which describes how the individual records are related to one another in the file. Since the previous section showed how it was possible to specify the organization of a record in terms of general characteristics such as field lengths, repetition numbers, etc., it is reasonable to suppose that the structure of a file may be similarly specified in terms of general characteristics of files. This, in fact, is the case. It is possible to describe files in terms of a generalized file structure, much as records are defined in terms of a generalized record organization by the template.
2. **Factors Determining File Structure**

Different ways of organizing records in files (file structures) have been developed for two basic reasons: 1) to meet the diverse processing demands of users for records in the files, and 2) to interface between the physical storage devices on which the records are stored and the logical relationships among the records as expected by the application programs. Some applications, such as information retrieval systems, require extremely fast retrieval, but permit slower storage and update. Other applications, such as real-time recording of test data, require rapid storage of large volumes of information, but permit slower retrieval and minimal update capabilities. Attempts to meet various critical requirements for these diverse applications have resulted in the development of file structures. These requirements may include the minimization of overhead in file processing operations, optimal use of storage space on high-speed storage devices, and reduction of access movement of disk storage devices. James Martin [17] lists such considerations as:

1. Whether the application demands random or sequential processing.
2. Percentage of the records in a file used during a typical run.
3. File and Record sizes.
5. Number of additions and deletions in a typical period.
6. Whether the file size remains static or is dynamic.
7. How easily the file can be protected from damage.
Despite all these factors that play a part in determining the proper file structure for a given application, all file structures can be described essentially in terms of sequential and associative types. Each of these types will be discussed in sequel. It will be shown that almost all the file structures can be described in terms of a generalized file structure as implemented in the EDMF which allows considerable flexibility in the choice of file structures.

3. **Keywords**

In a large data base file processing operations in which identical processing is performed on every record in the file are extreme cases. In normal file processing, specified operations are performed on various records of the file - retrieving some, updating others, storing still others. It is therefore important to be able to describe in some way which records are to be operated on for a specified operation, no matter what the file structure of the records is involved. There are two ways of identifying records for this purpose: identification by position, and identification by content.

When a record is identified by position, the location, address, or some similar piece of information must be provided. It will enable some system software to locate the record position and read the record there. This same method is used for locating data words in machine language computer instructions, as it is used for locating records on direct access storage devices. Unfortunately, it is not a natural one for people. A person would tend to refer to a record by its content or at least part of its content, say, "Bill Smith's personnel
record" in a personnel file instead of "The record on track 2, cylinder 3 of storage device 240". It would be useful if the data management system would permit the user or user's application program to refer to records by content.

From Chapter 2 we recall that a record may be considered as a set of attribute-value pairs, that is, a set of fields together with the attributes which describe the content (values) of the fields. Thus, the content of a record may be shown by displaying the attribute-value pairs which make up the record. This being true, it should be possible, and in fact is possible, to refer to records by using the attribute-value pairs of which they are formed. For example, the attribute-value pair LAST NAME=SMITH could certainly be said to refer to all those records in a file which contain the field named LAST NAME with the value SMITH, i.e., the attribute-value pair LAST NAME=SMITH. Thus any record may be referenced by means of all of the attribute-value pairs which make up the record. Furthermore, a group of records may be referenced by the attribute-value pairs which are common to the records.

In EDMF, any set of attribute-value pairs making up a record may be used to reference the record. In practice, however, only attribute-value pairs which are of reasonably short length are recommended to be used in this way. For example, the books in a library can be described in terms of many attributes such as the author, subject, date of publication, length, content (text), etc. However, the entire text of a book is not likely to be used to reference the book, even though one book may be distinguished from another by an examination of the text. Such short attribute-value pairs used to reference records will be called
keywords. Thus, keywords are used to reference records in the EDMF.

4. Sequential and Index Sequential File Structures

In a sequential file structure, records appear one after another both physically and logically. In order to retrieve a particular record in the file, the file must be searched from front to back until the desired record is found, since the records can only be accessed in the order in which they appear on the physical storage medium. Such a file structure is required for sequential access devices such as magnetic tape. Since, as we have seen, records are most often referred to by keywords (i.e., by their content), the sequential file structure can be extended to include keywords. The index sequential file is constructed with each record indexed by one or more keywords and the records in order by a particular keyword. For example, an inventory file would be maintained in order by part number, so that, assuming ascending order, part number 00000 would precede part number 99999. Since it is only possible to order records in this way by one keyword at a time, the file would have to be reordered by a special program if an order on some other keyword, such as supplier, were desired. This characteristic makes reference to records by more than one keyword, such as part number and supplier, very costly in terms of processing time.

The sequential file structure allows fast access to the next record in the file with very little overhead and is suitable in tape-oriented systems. This characteristic is ideal in applications where all or almost all records are processed each time the file is used,
but leads to very inefficient processing in applications where only a few records need to be processed, especially when the processing must be accomplished in real-time. When reliable random access devices, such as disks and drums, begin to be used as secondary storage devices, the sequential file structure was carried over to these devices from tape. The improved sequential file structure, known as the indexed sequential file structure, does not take full advantage of the capabilities of the direct access devices. File structures which are exceptionally well-suited for direct access devices begin to appear. These will be discussed in the following sections.

5. Associative File Structures

Associative file structures take advantages of the capabilities of direct access storage devices by allowing storage and retrieval of records on the basis of keywords. The correspondence between the records referenced by the keywords and the locations of the records on the storage device are maintained by the data management system. Once the locations on the storage device are known, they can be used to retrieve the records directly without processing other records. The above mentioned correspondence may be established in two ways: by hash coding or pointers.

5.1 Hash Coding

When hashing techniques are used, the referencing keyword is transformed into a location on the storage device. That is, the bits of the record keyword are manipulated to form a location on the storage device by means of a hashing algorithm. Often, during such a transformation, two or more keywords may be transformed into the same storage
location. In this case, their records must be linked if they are to be retrievable by either one of the keywords. For example, if a transformation which makes use of the first five letters of the attribute and the first five letters of the value is used, the two keywords AUTHOR=JOHNSON and AUTHOR=JOHNS will transform to the same location. Thus a link between the two records must be established, such as the one in Figure 3.1 so that either record may be successfully retrieved. Since links must be used, the chains of links may result in high access time. Another difficulty of employing hashing techniques is that they are difficult to implement when multiple keyword retrievals are required.

5.2 Pointers, Indexes, Lists and Directories

A pointer is defined as anything which will allow the system software to locate a record, and is the basis for two important elements in a file structure, indexes and lists.

In terms of files, an index is a set of keywords and one or more pointers for each keyword, which locates records with that keyword on some physical storage device. The index of a book is similar to this description, being a set of keywords in alphabetical order, and a set of pointers for each keyword which point to locations (pages) on a physical storage device (a book). An example of an index with one pointer per keyword is shown in Figure 3.2.

A list of records is a set of records linked by the pointers. The presence of pointers in the records themselves can preserve the logical order of the records even though their physical order may be different. For example, in Figure 3.3 four records of a list are
but leads to very inefficient processing in applications where only a few records need to be processed, especially when the processing must be accomplished in real-time. When reliable random access devices, such as disks and drums, begin to be used as secondary storage devices, the sequential file structure was carried over to these devices from tape. The improved sequential file structure, known as the indexed sequential file structure, does not take full advantage of the capabilities of the direct access devices. File structures which are exceptionally well-suited for direct access devices begin to appear. These will be discussed in the following sections.

5. Associative File Structures

Associative file structures take advantages of the capabilities of direct access storage devices by allowing storage and retrieval of records on the basis of keywords. The correspondence between the records referenced by the keywords and the locations of the records on the storage device are maintained by the data management system. Once the locations on the storage device are known, they can be used to retrieve the records directly without processing other records. The above mentioned correspondence may be established in two ways: by hash coding or pointers.

5.1 Hash Coding

When hashing techniques are used, the referencing keyword is transformed into a location on the storage device. That is, the bits of the record keyword are manipulated to form a location on the storage device by means of a hashing algorithm. Often, during such a transformation, two or more keywords may be transformed into the same storage
location. In this case, their records must be linked if they are to be retrievable by either one of the keywords. For example, if a transformation which makes use of the first five letters of the attribute and the first five letters of the value is used, the two keywords AUTHOR=JOHNSON and AUTHOR=JOHNS will transform to the same location. Thus a link between the two records must be established, such as the one in Figure 3.1 so that either record may be successfully retrieved. Since links must be used, the chains of links may result in high access time. Another difficulty of employing hashing techniques is that they are difficult to implement when multiple keyword retrievals are required.

5.2 Pointers, Indexes, Lists and Directories

A pointer is defined as anything which will allow the system software to locate a record, and is the basis for two important elements in a file structure, indexes and lists.

In terms of files, an index is a set of keywords and one or more pointers for each keyword, which locates records with that keyword on some physical storage device. The index of a book is similar to this description, being a set of keywords in alphabetical order, and a set of pointers for each keyword which point to locations (pages) on a physical storage device (a book). An example of an index with one pointer per keyword is shown in Figure 3.2.

A list of records is a set of records linked by the pointers. The presence of pointers in the records themselves can preserve the logical order of the records even though their physical order may be different. For example, in Figure 3.3 four records of a list are
Figure 3.1 Record Referencing by Hashing
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Pointer (storage location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHOR = SMITH</td>
<td>12459</td>
</tr>
<tr>
<td>AUTHOR = TAYLOR</td>
<td>00513</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBJECT = ANIMALS</td>
<td>57911</td>
</tr>
<tr>
<td>SUBJECT = BAKING</td>
<td>30487</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>YEAR = 1908</td>
<td>01972</td>
</tr>
<tr>
<td>YEAR = 1967</td>
<td>10047</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.2 An Index
logically in sequential order and physically in a different order. The logically sequential structure is obtained by using pointers. Physically, the first record is at location 129, the second at 34, the third at location 83, and the fourth at 67. Initially, there is a pointer to the first record. The first record contains a pointer, 34, to the second record, which contains a pointer, 83, to the third record, and so on. The last record in the list is designated by a special symbol, in this case ***.

Figure 3.3 simply shows how records may be connected in a list by means of pointers, without any special meaning being attached to the list. A better way of using a list is to connect all records with a common keyword onto one list. Thus in a personnel file there might be a list for OCCUPATION=PROGRAMMER, another list for BIRTH YEAR=1944, still another for LAST NAME=SMITH, and so on. This is illustrated in Figure 3.4. (In order to conserve space, we note that the pointers are not repeated in the Figures as in 3.3. However, arrows are retained.) When records are connected in this way in a file, it is possible to find all records in the file with OCCUPATION=PROGRAMMER by knowing the location of the first record in the list and tracing through the list using the pointers. Since any appropriate number of attribute-value pairs in a record may be treated as keywords, many lists can pass through the record. For example, in Figure 3.5, which is a more accurate version of Figure 3.4, the first record on the OCCUPATION=PROGRAMMER list has LAST NAME=SMITH as an attribute-value pair, and thus is on the LAST NAME=SMITH list as well. Allowing each record to be on as many lists as
Figure 3.3 A List of Records

Figure 3.4 Keyword Lists
Figure 3.5 Interconnected Keyword Lists
necessary avoids the duplication of records which would otherwise be necessary if separate lists were required, as in Figure 3.4. The location of the first record on any list for a keyword and the keyword are kept in a directory. A possible directory for the file of Figure 3.5 is shown in Figure 3.6. Many file structures are built around the concept of lists of records. The following sections discuss some of the more important file structures based on lists of records.

5.3 The Multilist File Structure

Figure 3.5 shows a portion of a file with a Multilist file structure. In such a file, each keyword is associated with one and only one list. The list threads its way through each and every record in the file which contains the associated keyword. Notice that the directory of such a file, as shown in Figure 3.6, has an entry for each keyword, followed by the location of the first record in the associated list. This location is known as the Head-Of-List-Address, abbreviated HOIA. Since there is only one HOIA per keyword, the directory, although it may be long, is relatively uncomplicated consisting of fixed length entries.

Retrievals of records containing any one keyword can be accomplished simply by looking up the keyword in the directory, finding the appropriate HOIA, and tracing down the keyword list. The fact that many lists may pass through a record means that the Multilist file structure allows for multiple keyword retrievals as well. To retrieve records containing two different keywords, for example, AUTHOR=SMITH OR SUBJECT=CALCULUS, each keyword is looked up in the
directory and the appropriate HOLAs are found. Both lists are then followed to the end, and the records appearing on either list are retained. A description of the contents of desired records, such as \texttt{AUTHOR=SMITH OR SUBJECT=CALCULUS}, is referred to as a \textit{logical and arithmetic expression of keywords}, since it utilizes the logical and arithmetic operators AND, OR, >, <, etc.

The number of file accesses can be reduced in many cases if the length of each list is kept in the directory along with the keyword and HOLA. This is shown in Figure 3.7. With this additional information, any retrieval based on the logical expression of the form, say, \texttt{A AND B AND C} can be satisfied by selecting the shortest list among the keyword list, \texttt{A}, \texttt{B} and \texttt{C} for the retrieval. This list, i.e., \texttt{LAST NAME=SMITH} list, is then searched to the end and all records containing all the specified keywords retained. This procedure is only possible if the keywords in question are connected by the \texttt{AND} operator, since only in this case can we be sure that all the required records are contained on the one selected list.

5.4 The Inverted File Structure

Figure 3.8 shows a portion of a file with an Inverted File structure. In such a file, all the pointers are removed from the records, and placed in the directory. Thus, along with the keyword appears the location of every record in the file containing that keyword. In terms of list structure, we can thus say that the directory of an Inverted File contains, for each keyword, a HOLA for every record in the file with that keyword, and in the file, each list is one record
BIRTH YEAR=1944 18

LAST NAME=SMITH 109

OCCUPATION=PROGRAMMER 23

Figure 3.6 Directory for File of Figure 3.5

<table>
<thead>
<tr>
<th>Keyword</th>
<th>List Length</th>
<th>HOLA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIRTH YEAR=1944</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST NAME=SMITH</td>
<td>3</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCCUPATION=PROGRAMMER</td>
<td>4</td>
<td>23</td>
</tr>
</tbody>
</table>

Figure 3.7 Directory of Figure 3.6 With List Lengths
Figure 3.8 Inverted File Structure
long. Retrievals are accomplished in this type of file by looking up the keywords in the directory as before. With an inverted file, however, there is sufficient information in the directory alone to determine exactly which records satisfy a given logical expression. Thus, for a logical expression A AND B, both keywords are found in the directory and the locations appearing with both words if different are used for retrieval. This means that there need be only as many file accesses as there are records satisfying the request. This characteristic is important if devices are used which have high access times. Another advantage to the Inverted File structure is that it can determine the number of records satisfying a logical expression before any file accesses are made. This permits the user to tally records in the file and to abort any retrieval which would return extremely large numbers of records. The disadvantage of this type of structure is that the directory can become extremely large.

5.5 The Generalized File Structure

Figure 3.9 shows a portion of the directory of a file using the Generalized File Structure. Besides the keyword and the Head-Of-List-Addresses, this directory contains two other pieces of information, the number of records in the file containing that keyword, and the number of Head-Of-List-Addresses for that keyword. This information allows the Generalized File Structure to support files with any number of lists per keyword, and the number of lists can vary from keyword to keyword. This means that a file could be constructed which is completely inverted with respect to one keyword, i.e., each record
location is in the directory, and pure Multilist with respect to
another keyword, i.e., there is only one HOLA for that keyword and
records containing that keyword are connected by pointers. Thus the
pure Inverted file and the pure Multilist file may be described as
special cases of the Generalized File Structure. Indeed, any file
structure describable in terms of list structures can be considered
as a special case of the Generalized File Structure. A complete
description of the Generalized File Structure is contained in reference
[15]. The EDMF, by utilizing the Generalized File Structure as its
basic file structure, can process any of the files describable by the
Generalized File Structure with the same set of processing programs.

5.6 The EDMF Implementation of the Generalized File Structure

A close examination of the entries contained in both the record
and Record Format Block (Chapter 2) will show that none of the entries
depend in any way on the file structure. That is, an EDMF record used
in the Generalized File Structure contains the same information
whether it is used in a Multilist file, Inverted file, or any of the
other files describable in the Generalized File Structure. The central
repository for information about the precise structure of each EDMF
file which can be defined by the Generalized File Structure is the
directory, as in section 4.5. The precise description of the EDMF
directory and its processing programs may be found in reference [4].
The contents of such a directory entry are shown in Figure 3.9. Since
the directory entry consists basically of a keyword — an attribute-
value pair, it would be possible to organize the entries in several ways.
(K, h, n; a_1, a_2, ..., a_h)

K -- the keyword for this directory entry

h -- the number of lists for this keyword in the file

n -- the number of records in the file containing the keyword K

a_1-a_h -- the Head-Of-List-Addresses of the h lists for the keyword K

Figure 3.9 Format of a Directory Entry

AUTHOR CARR (h_1, n_1, a_{1l}, a_{12}, ..., a_{1h_1})

HSIAO (h_2, n_2, a_{2l}, a_{22}, ..., a_{2h_2}) ...

YEAR 1964 (h_m, n_m, a_{ml}, ..., a_{mh_m})

1965 (h_{ml}, n_{ml}, ..., a_{mlh_{ml}})

1966 (h_{m2}, n_{m2}, ..., a_{m2h_{m2}})

Figure 3.10 Values Grouped Within Attributes
In the EDMF, an attribute or value can have up to 10 characters. A keyword has thus a maximal length of 20 characters. Directory entries are first sorted on attributes. Following each attribute, values that the attribute takes in the file and the appropriate Head-Of-List-Addresses are sorted and placed. This is shown in Figure 3.10. To cut down access times in the directory the whole directory is arranged in a three-level tree structure. Internal EDMF programs responsible for processing the directory return two types of information to calling programs: the number of records containing a given keyword, and the set of HOLAs for a given keyword. Other programs are responsible for updating the directory when new records are added to the file, or when records are deleted. By allowing variable numbers of HOLAs for each keyword, the EDMF directory completely describes the file structure to the EDMF processing programs. Other EDMF information about files is contained in the File Information Block, discussed in Chapter 6.

6. **Parallel Record Retrieval**

The EDMF makes use of an efficient record retrieval algorithm which takes full advantage of the type of storage devices (disks with movable heads) and the inherent capabilities of the Generalized File Structure. This algorithm attacks three outstanding problems associated with retrieval algorithms in the past: the retrieval of more records than necessary to satisfy a given logical expression of keywords, the duplicate retrieval of records on more than one list, and the random movement of the head mechanism on direct access storage devices with movable heads. The theory behind this algorithm is presented in [15],
and the details of its implementation are presented in [6]. A discussion of each of the above problems is presented below.

6.1 Selection of Prime Keywords

For a logical expression of the form, 'A AND B AND C ...', i.e., a conjunct, the EDMF retrieval algorithm attempts to minimize unnecessary record retrievals by selecting the keyword in the expression that is associated with the fewest records. This keyword is called the prime keyword for the conjunct. We note that no record which is not associated with the prime keyword can possibly satisfy the conjunct. The records which satisfy the conjunct must then be records associated with the prime keyword. Since there are fewer records associated with the prime keyword than with any other keyword in the conjunct, the retrieval algorithm should and does restrict its search to the list associated with the prime keyword. This tends to minimize the number of retrieved records. In an Inverted file, a special section of the algorithm comes into play which merges the locations (HOLAs) of all records for all keywords in the conjunct and uses only the unique locations for retrieval. In this case, the records retrieved are exactly the ones which satisfy the conjunct and, therefore, minimum.

6.2 Elimination of Duplicate Record Retrievals

Given a logical expression of keywords, it is possible to represent the expression in disjunctive normal form in which several conjuncts are concatenated by the operator 'OR'. From the previous section, we learned that each conjunct would result in a prime keyword list for retrieval. The EDMF retrieval algorithm eliminates duplicate record retrievals by searching all prime keyword lists for a logical
expression in parallel, rather than searching one list at a time (serial list search). The parallel list search means that a record appearing on more than one prime keyword list will be retrieved only once for the entire search, rather than once for every prime keyword list to which the record belongs. This further reduces the number of records retrieved for a logical expression.

6.3 Reduction of Movement of the Head-Mechanism

In carrying out the parallel list search, the EDMF retrieval algorithm reduces the movement of the head mechanism on direct access storage devices with movable heads by retrieving the records in order of their locations on the storage devices. This means that the head mechanism can make a "sweep" across the device without any back-and-forth movement. This eliminates completely the latency and repositioning delay which are characteristic of devices with movable heads.
CHAPTER 4

PRIVACY PROTECTION AND ACCESS CONTROL

1. Introduction

Early batching operating systems where magnetic tapes were used as the secondary storage medium did not require elaborate protection and access control procedures to be built into the operating systems. Since reels of tape did not need to be constantly "on-line" and were rather easy to be taken care of when dismounted, measures against theft and copying of the reels were sufficient. Access control could be easily provided by a tape librarian who would release reels for use to authorized persons.

The introduction and use of time-sharing systems altered this situation. The nature of time-sharing required that most files be on direct access devices and that these devices be constantly on-line. Even in the case of direct access devices with removable packs such as the IBM 2311 and 2314, the packs themselves are rather large and bulky, and for safety, are usually kept in the machine room even when dismounted. In addition, the storage capacity of a single pack can accommodate more than one file (and in most cases, files belonging to more than one user). This means that even though a pack may be mounted for the purpose of access to only one file, all the other files residing on the pack may be subject to unauthorized use. Thus means must be present to prevent any possible access to files by unauthorized users. Since the operating system is the only set of programs which
is present at all times, it is logical to incorporate the means in the operating system.

In the next section we examine the privacy protection and access control capabilities of the RCA Time Sharing Operating System (TSOS), the system on which the EDMF was implemented, as an example of the procedures to be found on a typical general purpose time sharing system.

2. Typical File Protection and Its Access Control

The RCA TSOS [23] stores the access control information associated with each file with the catalog entry for that file. The catalog entry allows the file to be found by the I/O routines. There are three types of information stored in this way: whether the file is sharable, whether the file is read-only or read-and-write, and whether any kind of password is required to access the file. This information is supplied initially when the file is created, and may be updated under conditions discussed below.

Under TSOS, no facilities are available to a potential user until he satisfies the log-on procedure. This consists of entering the LOGON command, followed by a valid user identification code. This user identification code is looked up in a table and verified by the system, which may also require a password to be provided before the log-on procedure is considered complete. Once the log-on procedure has been completed, the system considers the person logged on as a valid user, with access to all files which he "owns". When a user creates a file, he is recognized as its "owner". No other user can have any access to a file, unless the owner catalogs the file with the SHARE=YES
parameter specification in the CATALOG command. For a shared file, the manner in which the file may be used is controlled by two options. The ACCESS parameter may be used to limit the file to read-only use, or to specify both read and write options. In addition, two passwords can be associated with the file, a read-password and a write-password. If a read-password has been specified for a file, users desiring to read the file (provided they are allowed to do so by the ACCESS parameter) must give the read-password. To write and read the file they must provide the write-password. This provision for two passwords facilitates two levels of file sharing. Thus, a file marked as sharable can be accessed by any user who can provide either the identification code of the owner (at log-on), or the filename and the appropriate password if required. Figure 4.1 shows some examples.

The TSOS System Administrator can access any file in the system; that is, he can employ any Data Management System function consistent with the properties of the file. He has the same accessibility that the owner of a file has, but no more. For example, the administrator must supply the appropriate password to access the file. He cannot write to a file which is defined as read-only (nor can the owner). In other words, the administrator is viewed as a co-owner of every file in the system.

This discussion of TSOS file protection and access control capabilities serves as an introduction to the discussion of more advanced file protection and access control in the following sections.
User SMITH logs onto the system, and catalogs the following files:

/LOGON SMITH

Example 1:  /CATALOG F1, SHARE=NO, ACCESS=READ

Example 2:  /CATALOG F2, SHARE=YES, ACCESS=READ

Example 3:  /CATALOG F3, SHARE=YES, WRPASS=C'GOGO', ACCESS=WRITE

Example 1: User JONES cannot access F1 since SMITH specified that the file cannot be shared. Note that even SMITH, the owner, cannot write to the file.

Example 2: JONES, and in fact all other users, can read (but not write) F2. No password is required.

Example 3: JONES, and any other user, may read or write F3 by providing the password GOGO.

Figure 4.1 TSOS Access Control
3. **Desirable Privacy Protection and Access Control Capabilities**

To have "complete" privacy protection and access control is almost certainly impossible (see Reference [9] for some of the important considerations). Nevertheless, there are broad capabilities which can greatly extend the protection and access control on such systems as TSOS. In general, a system with privacy protection and access control should be based on the principle that there is a central "authority" having the power to grant or rescind access to data, and to make and change the classification of data as being more or less restricted in access. As a corollary, the system must enforce this authority through its system functions. This central "authority" should be the file owner, the person who created the file (as far as TSOS is concerned, there is a one-to-one correspondence between people and user identification codes, thus a whole group of people using the same user identification code is treated as one "owner"). To enforce the owner's authority, the operating system must provide the means with the following considerations:

a) different users can have widely varying access rights with respect to the same file,

b) data within a file may have widely varying protection classifications, and

c) procedures required for data access are not always static or predictable.

In other words, a file owner must be able to describe any piece of data in the system, to specify any appropriate access rights to that
data, and to assign any authorized user of the system with the specified access rights to the data described. In addition, the system must allow for the changing of access rights and data items described, and provide for future expansions of its access control capabilities by means of additional system programming.

4. **Deficiencies in Commercial Systems**

With respect to the desirable capabilities discussed above, the file protection and access control facilities of TSOS, and indeed of most commercial systems, are rather primitive. In Section 4.2 we have shown that the TSOS security arrangements are tied to the files as a whole. This means that there are no provisions for partitioning a file into sections with varying degrees of protection measures. Once access has been gained to a file, all data in the file are accessible. Furthermore, the file owner cannot vary the degrees of access which he gives to various users except by choosing the password he gives them. There are only two passwords available designating those authorized users with read or write option, and those authorized users with read-only option. It is also not possible to expand the protection features within the system. These are important limitations in dealing with data generated in the real world. For example, often it is not economically feasible to create multiple files for the same data just to enforce different protection of and access to the data. Furthermore, there are many more than two different access rights which users can have, and these access rights are not simply confined to read only and read and write.
5. **EDMF Privacy Protection and Access Control Facilities**

The privacy protection and access control facilities of the EDMF have been designed to overcome the above deficiencies.

EDMF privacy protection and access control facilities are user-oriented instead of file-oriented. This means that each user has his designated access right to any particular data item or group of data items, and this access right may vary from user to user. To enable the access right of each user to be individually specified, the system associates with each user a control block, called the Authority Item, which contains control information necessary for regulating access to data for that user. The Authority Item for a user has an entry for every file to which the user has any access rights, and is consulted whenever an access is made to the file. If the user does not have any access rights to a file, then there is no entry in the Authority Item for that file, so that files which are totally inaccessible to a user are also "invisible" to that user. The specification of the Authority Item and a description of the access control functions may be found in Chapter 6.

In addition to providing user-oriented access control, the EDMF allows the user to define access control at three different levels of data organization: at the file level, at the record level, and at the field level. Protection at the file level has to do with various rights to use the file as a whole, for example, whether the user shares, owns, may only read, or may both read and write a file. This is the level at which TSOS file protection security works as illustrated in section
4.2. Protection at the record level has to do with what records the user is allowed to use, and what records the user is not allowed to use. For this purpose records are specified by means of logical expressions of keywords which describe the characteristics of the records to be protected or not protected. These logical expressions are contained in the Authority Item entry for the file to which they refer, along with a code indicating how the descriptions are to be used. For example, an "allow" description indicates those records which are available to the user, while a "deny" description indicates those records which are not available to the user. Either type of description may be specified by means of a logical expression of keywords and applied to reading or writing of those records in the file. Protection at the field or data item level has to do with individual fields within records which the user may be restricted from using. For this purpose, fields are referred by the user to the system by their attribute names. Internally, protected fields are identified and recorded by their format numbers as used in the Record Format Block in the Authority Item.

Since access control can be defined at file, record, and field levels, the EDMF system routines must check user access at various levels in each storage or retrieval process. More specifically, the file level access is checked at the time the file is opened, whereas record level checking is done at the time each record is retrieved and verified, and before the record is presented to the user. In addition, file owners are allowed complete access rights to their own files by the EDMF. Thus, once a user is recognized at the file level as an owner, further
access control at record and field levels is inhibited.

Since keywords are fields, they are also subject to field protection. If a keyword is protected from use, then it will be removed from any logical expression containing the keyword. Only logical expressions of unrestricted keywords can be used for retrieval purposes. This is designed to prevent the user from obtaining information by default concerning fields which he is not allowed to access. For an example, consider a file of records in which NAME is the attribute of a field, and suppose that a user is not allowed to read the field whose attribute is NAME. If the user uses a logical expression, say, NAME=SMITH, and this keyword is not removed from the expression, he will receive the record, even though the system deletes the restricted NAME field from the retrieved record. In this case, the user is able to get the restricted information, namely NAME=SMITH by the fact that the retrieval took place. The deletion of the restricted fields, i.e., the keywords from the logical expression prior to retrieval and the deletion of the restricted fields from the record after retrieval can prevent such mishaps from occurring.

The file owner may specify any number of "allow" or "deny" logical expressions for record level protection. However, since "allow" and "deny" expressions must be separately checked, logical expressions of the same type (i.e., "allow" or "deny") should be combined for greater checking speed. For example, the two expressions "allow A" and "allow B" take longer to check than the equivalent single expression "allow A or B".
It is possible for the file owner to supply contradictory descriptions at the record level. For example, the addition of "deny A OR B" to a file which already has the entry "allow B OR C" results in contradictory instructions to the system for B. The system will resolve this contradiction by giving priority to the "deny" expression. This is based on the belief that privacy protection is better maintained if fewer records are made available. In EDMF, the following algorithm is used to resolve the contradiction:

1. If the record satisfies a "deny" description, suppress the record; otherwise, go to 2.
2. If there are no "allow" descriptions, pass the record. If there are "allow" descriptions, go to 3.
3. If the record satisfies an "allow" description, pass the record; otherwise, suppress the record.

Note that in the algorithm above, the particular descriptions must apply to the function being performed by the user in order to be checked by the algorithm. For example, if the file is being written into, those descriptions not applicable to writing in the file are ignored by the algorithm.

The provision in the EDMF of access control at several levels allows data with different protection requirements to be stored in the same file on the one hand and provides the user the capability to completely specify the exact access control over other users of this file on the other hand. Provision has also been made in the Authority Item for a future extension of the access control mechanism, allowing
users to provide access control programs, at any level, to be automatically executed by the system. Such user programs could be used to carry out enciphering and deciphering operations on records, and extensive user identification checks prior to access to data.
CHAPTER 5

COMMAND LANGUAGE PROCEDURES

1. Introduction

Individual machine language instructions or FORTRAN statements are useful by themselves, but it is only when they are combined into the lists of instructions or statements called programs that they become most effective. A somewhat similar situation holds true for EDMF or TSOS system commands. While the individual commands are very useful, it is the sequence of commands in which the power of the command language lies. In using the command language, there are often times when the same sequence of commands must be used repetitively. For this reason, TSOS provides the capability for the user to create and store files containing lists of commands and data, and for the user to invoke the file of commands by directing TSOS to read its system commands from this file. Such a file of commands is called a procedure or PROC file. Through the use of procedures, the user may combine existing programs, system commands, procedures, and data with new programs to form sequences of operations which will handle problems that could not be handled by any existing program, command, or procedure. The following sections deal with various phases of the procedure capability. The first section deals with the existing facilities of TSOS for creating and using procedures. The sections which follow deal with the enhancements provided by the EDMF for the creation and use of procedures.
2. Standard Procedure Creation and Use

TSOS provides no special facilities for creating procedures. Instead, procedure files must be created by using the standard TSOS file creation facilities, including the File Editor, the DATA command, and the various TSOS access methods. The procedure must begin with a PROC command, and with an ENDP command, and satisfy certain minor restrictions on the form of the file. The user may place anything he wishes into the procedure, as long as the file may be processed as a standard TSOS command stream.

A procedure is invoked when the user issues a DO command with the name of the procedure file to be invoked as a parameter. TSOS then reads its commands from the procedure file and performs the indicated operations until it reads another DO command or an ENDP command. The ENDP command causes TSOS to return to its original command source (user terminal or cards) for subsequent commands. The DO command with another procedure name causes another procedure to be invoked. Thus DO commands may be imbedded in procedures which would result in several levels of procedure executions. The first ENDP command processed does not return control to the invoking procedure, however, but to the original command source as previously stated. Thus procedures may not be 'nested' (see Figure 5.1).

Unacceptable commands or data appearing in the procedure, or the abnormal termination of a program executed within the procedure, will cause subsequent commands and data to be skipped until an ENDP or STEP command is found. The STEP command directs TSOS to process the remaining
Figure 5.1 Procedures Invoking Other Procedures
commands of a procedure due to an abnormal condition in a previous command. In addition, the STEP command resets certain task switches associated with the processing of the user's command sequence. These task switches may be tested by user programs, enabling conditions within the procedure to control user program execution. In addition, the user program may also set task switches, to indicate various conditions to the procedure. The task switches may be tested in the procedure by the SKIP command. If the actual task switch settings agree with the settings indicated in the SKIP command, then a forward skip is made to the command whose label is indicated in the SKIP command. Thus the SKIP command acts within a procedure like a conditional branch acts within a program (but only in a forward direction), controlling the sequence of commands processed by the system in the same way that a user might issue commands interactively based on the results of previous command executions.

The user may designate certain command parameters (not to be confused with input parameters to a user program) as formal parameters at the time of procedure creation. Such formal parameters are indicated by the presence of an ampersand (&) as the first character in the parameter's symbolic name. When the procedure is invoked, the user may substitute actual values for the formal parameters in the procedure. This process is shown in Figure 5.2. The user may write generalized procedures through the use of formal parameters and tailor them to a specific purpose at the time they are used, thus adding great flexibility to the procedures. The use of procedures in TSOS and the various commands discussed above are presented in [25].
/PROC C
/FILE INPUT.&NUMBER.,OPEN=INPUT
/EXEC PROGRAM1
/ENDP

... 

/DO PROCL,(2204) 
... 

user invokes PROCL with actual parameter 2204.

/PROC C
/FILE INPUT.2204,OPEN=INPUT
/EXEC PROGRAM1
/ENDP

commands executed as the result of user invoking PROCL

Figure 5.2 Formal Parameters in Procedures
3. **Extended Procedure Creation and Use**

The EDMF has enhanced the procedure capability by providing a facility for creating procedures and commands for incorporating user programs into procedures. These new facilities are discussed in the following sections.

3.1 Passing Parameters to Programs in a Procedure

The MEXEC command performs the same functions as the TSOS EXEC command. In addition, it allows the user to pass a parameter string to the program being executed. This command expands the utility of programs within a procedure because the user not only can pass parameters to the procedure as mentioned earlier but also pass parameters to the individual programs within the procedure. This means that the functions performed by the individual programs as well as the ones performed by the procedure may now be controlled by the user at the time he invokes the procedure.

3.2 Defining and Substituting Formal Command Names and Their Parameters

The FPARAM command is an extension of the formal parameter capability of the TSOS procedure. It allows the formal parameter to include the command name in addition to the parameters of the command. Figure 5.3 shows how FPARAM allows the user to substitute the command name and a part of a parameter for dummy ones - a capability which does not exist in the TSOS procedure. A more detailed description of both the FPARAM and MEXEC commands may be found in [18].
PROC C
/NAM1 FILEA
/EXEC PROCl,PARAM=C'&NAME2'
/ENDP

procedure NEWPROC with formal parameters

invoked by:

/FPARAM NEWPROC,TEMP1,NAM1,FILES,NAME2,TYPE=ADD
/DO TEMP1

results in execution of commands:

PROC C
/FILE FILEA
/EXEC PROCl,PARAM=C'TYPE=ADD'
/ENDP

invoked by:

/FPARAM NEWPROC,TEMP2,NAM1,ERASE,NAME2,TYPE=DELETE
/DO TEMP2

results in execution of commands:

PROC C
/ERASE FILEA
/EXEC PROCl,PARAM=C'TYPE=DELETE'
/ENDP

Figure 5.3 Use of FPARAM Command
3.3 Local Abbreviations and Command-Program Mix in Procedures

The EDMF procedure creation module, called PROC, allows the user to create the procedures he desires. In addition to cataloging the procedure file, allocating space for it, and placing the delimiting commands PROC and ENDP in the file for the user, the procedure creation module provides two services which simplify the creation process.

First, the PROC module enables the user to define and use abbreviations, called 'local names', for long expressions which are used repeatedly during procedure creation. This facility is especially important in using the EDMF functions, since their logical expressions or file names may be long and may repeat several times in a procedure. An example which demonstrates the use of a local name to abbreviate a long file name is shown in Figure 5.4. Local names are used for the ease of creating the procedure; unlike formal parameters they may not be referred to when the procedure is invoked.

Second, the PROC module makes transparent to the user the difference between a function implemented as a command, and a function implemented as a user program when the function is used from within a procedure, as shown in Figure 5.5. This facility is valuable in particular when interim changes are being made to the command language but the command handling routines are not ready to be added to the system. The procedure creation process remains the same, whether the routine is a user program called by the MEXEC command or a system program with its own command. A detailed discussion of the PROC module may be found in [7].
This terminal session:

/EXEC PROC
   ENTER PROCEDURE NAME
*PROCESSI
   BEGIN
*LOCNAME #AA := PERSONNEL.MASTER.22 ###
*OPEN  #AA,TYPE=READ
*RETRIEVE  #AA,C'SECURITY=SECRET',3
*CLOSE  #AA
*FSTATUS #AA
*END*

   END OF PROCEDURE

: :

produces the procedure named PROCESSI as shown:

/PROC C
/OPEN PERSONNEL.MASTER.22,TYPE=READ
/RETRIEVE PERSONNEL.MASTER.22,C'SECURITY=SECRET',3
/CLOSE PERSONNEL.MASTER.22
/FSTATUS PERSONNEL.MASTER.22
/ENDP

Figure 5.4 Use of Local Abbreviation During Procedure Generation
Procedure definition:

/EXEC PROC
.
.
*OPEN FILEA, TYPE=READ
*RETRIEVE FILEA, C'NAME=JOHN SMITH', 0
*COMPUTE 36.7 + 16**3
.
.
Commands generated by EDMF:
.
.
/OPEN FILEA, TYPE=READ
/RETRIEVE FILEA, C'NAME JOHN SMITH', 0
/MEEXEC COMPUTE, PARAM=C'36.7 + 16**3'
.
.
System recognizes OPEN and RETRIEVE as valid commands and treats them accordingly.

User issues COMPUTE within procedure (definition) as if it were a command. EDMF recognizes this as an invalid command, and generates an MEEXEC command for the program COMPUTE with the proper parameter.

Figure 5.5 Program Simulating a System Command Within a Procedure
CHAPTER 6
EDMF FUNCTIONS AND PROGRAMS

In this chapter, we will provide an overview of the major EDMF functions provided to the user. For this purpose, a narrative description of each function is provided. It also includes the names of the programs which make up the function and some detailed information about specific programs which cannot be found elsewhere. Other information about the mentioned programs may be found in cited references. A flow-chart showing the interconnections between programs in the EDMF is included in the Appendix.

1. The OPEN Function

The EDMF OPEN function serves two purposes: to check a user's authority to use a file in the system and to prepare the file being opened for further processing. Further processing may include the use of the STORE, RETRIEVE, UPDATE, and DELETE functions of the EDMF. Thus, all these functions require that the file be opened before any processing takes place. The EDMF user has two methods of calling on the OPEN function: one is through the OPEN command, issued conversationally at the terminal or non-conversationally through the card reader while the system is in command mode. The other is from a user program through the OPN macro-instruction. The OPEN function takes three parameters: the name of the file to be opened, a specification of whether the file is to be opened for read-only or for update (i.e. reading and writing),
the OPEN request. If an SSB is present, it will contain the address of the user's authority item in the storage, even though it may not apply to the file being opened. Furthermore, the FIB for the file will be in the storage, and its address in the SSB. After OPNPROC completes these tasks, control is returned to the user.

We note that the OPEN function involves a set of interrelated control blocks and processes designed to overcome some of the restrictions imposed by TSOS, and to afford protection for files being shared by more than one user. These control blocks are the authority item, Service Status Block (SSB), File Status Block (FSB), and User Description Block (UDB). They will be described in detail in subsequent sections.

The restrictions imposed by TSOS may be summarized as follows. The TSOS restricts file opening (using the TSOS OPEN function, not the EDMF OPEN function) in an attempt to keep two users who may be using the same file from conflicting with each other. Generally, two rules are enforced: 1) if one or more users have opened a file for reading, no user may then open the file for writing, 2) if a user has opened a file for writing, no user may then open the file for any purpose. The objective is to prevent one user from reading information which will shortly be changed by another user. This poses a problem for the EDMF OPEN function because, under the EDMF, a portion of a file may be opened, whereas under the TSOS this may not be done. For example, a user of a file of technical data could inform the EDMF that he is interested in only those records containing the keyword SUBJECT= AIRCRAFT by opening only that portion of the file. If another user opens a portion of the same file
and a logical expression of keywords specifying the portion of the file to be opened. If such an expression is present, records retrieved by the user will be limited to the ones that satisfy the expression. A complete discussion of the OPEN process, its control blocks, and the specification for the OPN macro-instruction may be found in [8]. However, a brief description, which shows the overall flow of the process, is presented below.

The main program in the OPEN function is OPNPROC, which is in overall control of the entire process. OPNPROC is accessed directly via the OPN macro and its associated SVC call. The OPEN command is directed by the Terminal Command Processor (TCP) to the pre-processing routine MULTOPEN, which decodes the command parameters and places them in the proper input format to OPNPROC. OPNPROC's first job is to call SSBOPTR in order to acquire the address of the user's chain of Service Status Blocks (SSB). It then searches to see whether there is already an SSB entry for the file being opened. The absence of any SSB in the chain will prompt OPNPROC to set up a SSB for the file being opened. To this end, OPNPROC must first call AIRETR to retrieve the user's authority item into main storage, and then call FIFDIRSL to retrieve the File Information Block (FIB) for the file being opened from the File Information File (FIF). The FIB contains the record template for the file and information required by the ISOS I/O routines. Its format is shown in Figure 6.1. Subsequently, a SSB for the particular request is created, the file is marked open, a File Status Block (FSB) is created by calling FSBOPTR, and a User Description Block (UDB) is created if an expression was specified in
<table>
<thead>
<tr>
<th>3 bytes</th>
<th>Size of FIB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>Count</td>
</tr>
<tr>
<td>5 bytes</td>
<td>Disk Address (ISAM Key)</td>
</tr>
<tr>
<td>54 bytes</td>
<td>Filename</td>
</tr>
<tr>
<td>2 bytes</td>
<td>File size (half pages)</td>
</tr>
<tr>
<td>6 bytes</td>
<td>Volume Serial Number (VSN)</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Type of file</td>
</tr>
<tr>
<td>2 bytes</td>
<td>Share Status</td>
</tr>
<tr>
<td>4 bytes</td>
<td>RDPASS if any</td>
</tr>
<tr>
<td>4 bytes</td>
<td>WRPASS if any</td>
</tr>
<tr>
<td>5 bytes</td>
<td>Last ISAM key assigned in file</td>
</tr>
<tr>
<td>4 bytes</td>
<td>Flag and pointer for User Processing Program</td>
</tr>
<tr>
<td>160 bytes</td>
<td>FCB for file (standard TSO6 format, minus the logical areas)</td>
</tr>
<tr>
<td>variable</td>
<td>Record Format Block (RFB) for file</td>
</tr>
</tbody>
</table>

Notes:

1. Type of file may be (1) facility standard ISAM (2) ISAM (3) PAM (4) SAM (5) BTAM (6) other.
2. Share status is either shareable or non-shareable.
3. Flag and pointer for user processing program contains at least the program name and the name of the Object Module Library (OML) on which the program resides.

Figure 6.1 The File Information Block (FIB)
having the keyword SUBJECT=CAMELLS, and no records have both SUBJECT=
AIRCRAFT and SUBJECT=CAMELLS as keywords, then file processing for these
two users will not likely result in any conflict. Hence, they should
be permitted to process the file in any way they want, subject to their
access rights to the file. Because the EDMF system routines make use
of the TSOS file opening facilities, the TSOS considers partial opening
for the same file as discussed above as two openings for the whole file.
If the second opening attempt violates one of the two rules mentioned
above, TSOS will not allow access to it. Thus it is necessary for EDMF
routines to overcome this "file lock" of the TSOS.

The EDMF OPEN minimizes the conflicts in file opening by not issuing
a TSOS OPEN for the file until a storage or retrieval process has begun.
Instead, the TSOS OPEN is issued by the STORE, RETRIEVE, UPDATE, or
DELETE function just before an actual storage or retrieval process begins.
Furthermore, these functions issue a TSOS CLOSE right after the process
is finished. This procedure reduces the amount of time that a file is
open under TSOS, and thus minimizes the chances for conflicts. At the
same time, however, the EDMF must provide for protection of shared files,
ensuring that a user who opens a portion of a file receives records
only from that portion of the open file. Furthermore, it must ensure
that any update would not conflict with any read of the same record.
In conclusion, we note that the EDMF does not restrict different users
to open the same files.
Once a file is opened by several users, the storage and retrieval of records for the users are subject to the following checking. If a user opened a file first, i.e., his FSB is at the head of the list for the file, then he is confined to the open portion of the file and his access rights to the file. Assuming the user opened the file for read-only, then a newly-arrived user (whose FSB must follow the previous FSB) opening the same file for record update will not be able to enter updated records into the portion of the file which was opened by the previous user. Once a user opened the file for update, no subsequent users may retrieve records from or store records into the portion of the file which was opened by the user. We note that the FSB and its position in the FSB list for a given file play a critical role in this procedure. As an example, consider the FSB lists shown in Figure 6.2. The file ALPHA has been partially opened for record update by Smith, for read-only by Jones, and for record update by Thomas, in that order. Each of the three opened portions is indicated by a logical expression of keywords. Similarly, the file BETA has been opened for read-only by Jones and Thomas, and for record update by Smith, in that order. Thus, in file ALPHA, Smith can store or retrieve any record satisfying the expression YEAR=1966. Jones can retrieve any record satisfying the expression SUBJECT=AIRPLANES, provided it does not satisfy the expression YEAR=1966. Thomas can store or retrieve any record satisfying the expression YEAR=1967, provided it does not satisfy either of the expressions provided by Smith or Jones. In file BETA, Jones can retrieve any record satisfying the expression NAME=YOOSER. Thomas can retrieve
Figure 6.2 File Status Blocks and User Description Blocks for Two Files
any record satisfying the expression LOCATION=NEWARK, whether or not it also satisfies Jones' expression. However, Smith may store or retrieve any record satisfying the expression PERCENTAGE=10, provided it does not satisfy either of the expressions provided by Jones or Thomas. Each time a user opens a file, a new FSB is created and attached at the end of the FSB list for the file. Thus, if user Smith issues another EDMF OPEN for the file ALPHA, there will be a FSB placed at the end of the list for file ALPHA, in spite of the fact that he already has an FSB on the list. This enables the user to open different portions of the same file for different processing. We note that the STORE, UPDATE, and DELETE functions are all considered by the OPEN function to be updates for the purpose of the preceding discussion.

2. **Control Blocks and Access Management**

The control of user access to files once they are opened is supported by two sets of control blocks, the user-oriented Service Status Block (SSB) and the file-oriented File Status Block (FSB). Whenever a user opens a file using the EDMF OPEN function, both SSB and FSB are created; the SSB is linked to a list of SSB's for that user and the FSB is linked to a list of FSB's for that file.

The SSB contains information concerning the mode of file use such as read-only or read/write, and the whereabouts of control information for the file such as the TSO8 File Control Block, the FSB, and the User Description Block (UDB). Thus, any time the EDMF needs information about the opened files of a particular user, it will go to the SSB list for that user. On the other hand, the FSB contains the user's identification, control information for the file, and also a pointer to the UDB.
Thus, any time the EDMF needs information about users of a particular opened file, it will go to the FSB list for that file. In addition, the FSB list for a file is in order by time of file opening, so that the EDMF can determine which user opened a given file first (and thus has priority in use of the file) by examining the first FSB on the list for that file.

2.1 Service Status Block (SSB)

The Service Status Block (SSB) contains status information about every file processed by a user during an interactive session. It is created when a user logs onto the system, associated with the user's task throughout its existence, and destroyed when the user logs off. The purpose of the SSB is to eliminate duplicate retrievals of control information about user files. This information is kept in the SSB in privileged system memory. The first file to be opened by a user results in its control information being stored in the SSB. All subsequent file openings cause additional SSB sections to be chained to the initial section in a linked list. Thus the SSB can grow as the number of files referenced grows during an interactive session. Note that a file need not be open to be in the SSB. If the file had been previously opened and closed, then the control information is already in storage in one of the SSB sections. There is no need to retrieve this control information from the FIF. The Service Status Block is shown in Figure 6.3. Similar control information is stored in the File Control Block by the TSO for handling TSO files. However, the File Control Block is placed in the user's program and is not protected from misuse.
### Service Status Block (SSB)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes</td>
<td>Control--Address of User's Authority Item</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>Control--Address of FCB for FIF</td>
<td></td>
</tr>
<tr>
<td>2 bytes</td>
<td>Length of Filename</td>
<td></td>
</tr>
<tr>
<td>54 bytes</td>
<td>Filename</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>Control Information</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>Control--Address of FIB for Filename</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>Control--Address of FCB for Filename</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>Control--Address of User Description Block</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>Control--Address of Core Format Record</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>Control--Address of corresponding FSB</td>
<td></td>
</tr>
<tr>
<td>4 bytes</td>
<td>Control--pointer to next SSB entry</td>
<td></td>
</tr>
</tbody>
</table>

### Notes on the SSB:

1. Unless stated explicitly, all control information is 1 byte, all addresses are 3 bytes.

2. The header appears on the first SSB block for a given task only - all subsequent SSB entries contain only the text.

   1st SSB block = 8 + 84 bytes = 92 bytes

   all subsequent SSB blocks = 84 bytes

---

Figure 6.3 The Service Status Block (SSB)
The SSB processing routine SSBOPTR keeps track of the list of SSB's for each user by means of a table, SSBTAB, which contains a pointer to the list of SSB's for each user of the system. This table contains 128 entries, corresponding to the 128 possible tasks operating under the EDMF at one time. Each user task has a task number from 0 to 127 associated with it, which is used as an index into SSBTAB to find the appropriate pointer to the SSB list for the task. This table is illustrated in Figure 6.4.

2.2 Interrogation of the List of Service Status Blocks

The SSBOPTR routine provides services to programs needing to use or interrogate the list of Service Status Blocks for a particular user's task. SSBOPTR has four entry points as follows, each of which provides a different function.

SSBLOGON: establish the initial SSB for the task number specified in register 1.

SSBAOQR: give the user in register 1 the address of the initial SSB for the task number specified in register 1.

SSBGTNU: establish a new SSB at the end of the current SSB list for the task number specified in register 1. The address of the new SSB is returned in register 1.

SSBLOGGF: release memory for all SSB's on list for task number specified in register 1, i.e., delete all SSB's for task.

External Subroutine Calls - none
User task 15 entry in SSBTAB is 16th entry in table.

Figure 6.4 Organization of SSBTAB and SSB Lists
DSECTS used:

<table>
<thead>
<tr>
<th>Name</th>
<th>Bytes</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSB</td>
<td>DSECT</td>
<td>SSB Header</td>
</tr>
<tr>
<td>SSBHDR</td>
<td>0-7</td>
<td>Address of User's Authority Item</td>
</tr>
<tr>
<td>SSBUAI</td>
<td>0-3</td>
<td>Address of FCB for File Information File</td>
</tr>
<tr>
<td>SSBFIF</td>
<td>4-7</td>
<td>SSB Text</td>
</tr>
<tr>
<td>SSBTXT</td>
<td>8-91</td>
<td>2 bytes - length of filename</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5½ bytes - filename</td>
</tr>
<tr>
<td>SSBFNAM</td>
<td>8-63</td>
<td>Control Information</td>
</tr>
<tr>
<td>SSBCL</td>
<td>64-67</td>
<td>Type of request</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>Indicator - EDMF open</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>Unused</td>
</tr>
<tr>
<td></td>
<td>66-67</td>
<td>Address of File Information Block (FIB)</td>
</tr>
<tr>
<td>SSBFIB</td>
<td>68-71</td>
<td>Address of File Control Block (FCB)</td>
</tr>
<tr>
<td>SSBFCB</td>
<td>72-75</td>
<td>Open description indicator</td>
</tr>
<tr>
<td>SSBDTBIN</td>
<td>76</td>
<td>Address of User Description Block</td>
</tr>
<tr>
<td>SSBDTAB</td>
<td>77-79</td>
<td>Address of core format of record</td>
</tr>
<tr>
<td>SSBCREC</td>
<td>80-83</td>
<td>Address of File Status Block</td>
</tr>
<tr>
<td>SSBFSB</td>
<td>84-87</td>
<td>Control Information for Pointer</td>
</tr>
<tr>
<td>SSBCTL6</td>
<td>88</td>
<td>Pointer to next SSB Block</td>
</tr>
<tr>
<td>SSBPTR</td>
<td>89-91</td>
<td></td>
</tr>
</tbody>
</table>
Note: PSTAT is register 2.
Figure 6.5b Flowchart of SSBOPTR (continued)
Figure 6.5c Flowchart of SSBQPTR (continued)
Figure 6.5d Flowchart of SSBOPTR (continued)
Figure 6.5e Flowchart of SSBOPTR (continued)
Figure 6.5f Flowchart of SSBOPTR (continued)
Input parameters: user task number in right-most byte of register 1, rest of register cleared to zero.

Register Conventions:

<table>
<thead>
<tr>
<th>Register</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Not used</td>
</tr>
<tr>
<td>1</td>
<td>REQMed memory address, I/O parameter</td>
</tr>
<tr>
<td>2</td>
<td>work register for return code</td>
</tr>
<tr>
<td>3</td>
<td>task number index for SSBTAB</td>
</tr>
<tr>
<td>4</td>
<td>base register for program</td>
</tr>
<tr>
<td>5</td>
<td>SSB block address</td>
</tr>
<tr>
<td>6</td>
<td>base register for SSB DSECT</td>
</tr>
<tr>
<td>7</td>
<td>address of memory to be released</td>
</tr>
<tr>
<td>8</td>
<td>next SSB block address</td>
</tr>
<tr>
<td>9</td>
<td>work</td>
</tr>
<tr>
<td>10,11,12</td>
<td>not used</td>
</tr>
<tr>
<td>13</td>
<td>save area address</td>
</tr>
<tr>
<td>14</td>
<td>return address</td>
</tr>
<tr>
<td>15</td>
<td>return code</td>
</tr>
</tbody>
</table>

Internal Work Area: SSBTAB

**Typical Entry:**

<table>
<thead>
<tr>
<th>entry code</th>
<th>pointer to first SSB in chain</th>
</tr>
</thead>
</table>

**entry code:** a000 bbbb

**where:**

- \( a = 0 \)  no user file open for update
- \( a = 1 \)  some user file is open for update
bbbb = 0000 no SSB for this task number
0011 system task entry
1100 existing SSB not yet referenced by EDMF
1111 existing SSB referenced by EDMF

SSBTAB has 128 entries, one for every possible task number, the
first 7 of which are reserved for permanent system tasks and have entry
codes of X'03'. The remaining 121 entries are for user tasks, and
point to the first SSB in the list of SSB's for that user task.

Return Codes: Return codes depend on entry used

SSBACQR: same as PSTAT
SSBLOGON: X'00' normal return
X'04' SSB already exists
X'08' SSBOPTR could not get memory to set up SSB
X'0C' task number given by user is that of a system
task

SSBCTNU: X'00' same as PSTAT
X'04' "
X'08' "
X'0C' "
X'10' SSBOPTR could not get memory to set up SSB

SSBLOGOF: X'00' normal return
X'04' SSBOPTR could not release memory
Internal Codes:

**PSTAT** - present status of SSB list for user task number given

- \( X'\varnothing' \) an SSB exists and has not been referenced yet by EDMF
- \( X'\varnothing^4' \) an SSB exists and has been referenced by EDMF
- \( X'\varnothing^8' \) no SSB exists for this task
- \( X'\varnothing^C' \) task number given by user is that of a system task - error

2.3 The File Status Block (FSB)

**FILE STATUS BLOCK (FSB)**

- 8 bytes - userid
- 4 - header (control information)
- 4 - description address
- 11 - control
- 3 - back link; points to previous FSB in list
- 1 - \( X'00' = \) null pointer, \( X'FF' = \) good pointer
- 3 - forward link; points to next FSB in list

**FSB TABLE (FSBTAB) ENTRY**

- 2 bytes - length of filename
- 54 - filename
- 1 - \( X'00' = \) null, \( X'FF' = \) good
- 3 - pointer to first FSB with filename
- 1 - \( X'00' = \) null, \( X'FF' = \) good
- 3 - forward link; points to next entry
Entry FSBADBLK

Set MODE=1

Get address of first FSB

Does FSB exist?

Yes

MODE=2?

Yes

Return to Caller

No

No

MODE=2?

Yes

Get last FSB

Set pointers

No

Set pointers

Get new FSB

Return to Caller

Return with error X'04'
Figure 6.6b Flowchart of FSBOPTR
(continued)
2.4 Creating and Finding a File Status Block

**FSBAD** - get address of file status block, and create a new block
if one does not already exist for the filename.

Input: reg. 1 - 2 bytes - length of filename variable up to 54
       bytes - filename
       reg. 13 - register save area

Output: reg. 1 - address of block with filename (may be newly created
         block)
       reg. 15 - X'00' = OK
       X'08' = REQM error from attempt to create new block

**FSBNU** - get new file status block (regardless of whether a block
already exists for the filename)

Input, output: same as FSBAD

**FSBREM** - remove file status block

Input: reg. 1 - list of 4-byte addresses of blocks to be removed;
       X'FF' at end indicates end of list
       reg. 13 - register save area

Output: reg. 15 - X'00' = OK

Otherwise, address of block during which RELM error
occurred. Blocks pointed to earlier in parameter
list have been removed.
Register Conventions:

<table>
<thead>
<tr>
<th>Register</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Not used</td>
</tr>
<tr>
<td>1</td>
<td>Input parameter</td>
</tr>
<tr>
<td>2</td>
<td>Miscellaneous use</td>
</tr>
<tr>
<td>3</td>
<td>Base register</td>
</tr>
<tr>
<td>4</td>
<td>Address and base of FSBTAB entry</td>
</tr>
<tr>
<td>5</td>
<td>Length of input filename</td>
</tr>
<tr>
<td>6</td>
<td>Mode switch</td>
</tr>
<tr>
<td>7</td>
<td>Address and base of FSB</td>
</tr>
<tr>
<td>8</td>
<td>Hold Control Byte</td>
</tr>
<tr>
<td>9</td>
<td>Backward Link</td>
</tr>
<tr>
<td>10</td>
<td>Forward Link</td>
</tr>
<tr>
<td>11</td>
<td>Address and base for input parameter</td>
</tr>
<tr>
<td>12</td>
<td>Not used</td>
</tr>
<tr>
<td>13</td>
<td>Save Area Address</td>
</tr>
<tr>
<td>14</td>
<td>Return Address</td>
</tr>
<tr>
<td>15</td>
<td>Return Codes</td>
</tr>
</tbody>
</table>

3. The CLOSE Function

The CLOSE function serves to deactivate control blocks of a file which were set up during the OPEN process for the file, and to indicate that the file (or specified portion of the file) may no longer be accessed by EDMF routines. The EDMF user may call on the CLOSE function by means of the CLOSE command or the CLSE macro-instruction. The CLOSE function takes 2 parameters: the filename of the file to be closed,
and a logical expression of keywords indicating a portion of the file to be closed. A complete discussion of the CLOSE process and its control blocks as well as the specification for the CLSE macro-instruction may be found in [8]. A brief description of the process is included as follows.

The main program in the CLOSE function is CLSEPROC, which is in overall control of the entire process. CLSEPROC is accessed directly via the CLSE macro and its associated SVC call. The CLOSE command is routed by the TCP to the pre-processing routine MULTCLSE, which decodes the command parameters and places them in the proper input format for CLSEPROC. CLSEPROC first calls SSBOPTR, to get the address of the user's SSB chain, which in turn checks to see whether there is an SSB entry for the file being closed. The absence of any SSB entry means that this file was never opened, so an error message is output and the control returned to the user. If the user has omitted a logical expression in his CLOSE request, then it means that he wishes the whole file closed. In this case, all SSB's for this file will be found and marked closed, and all FSB's pointed to by the SSB's will also be eliminated using FSBOPTR. Control will then be returned to the user. If, however, the user provided a logical expression in his CLOSE request, this means that only the portion of the file described by the logical expression is to be closed. In this case, CLSEPROC checks all SSB's with the given filename for a logical expression which matches the logical expression given in the CLOSE request. If none is found, an error message is
output. Otherwise, CLSEPROC marks the file closed in the SSB, calls
FSBOPTR to eliminate the FSB for that portion of the file, and returns
control to the user.

4. The RETRIEVE Function

The RETRIEVE function allows the user to receive from a file
records which satisfy a given expression of keywords. These records
may be directed by the user or the user's program to a specific output
device or buffer storage. Along with each output record, the user
also receives a dynamically-generated record number which uniquely
identifies the record. This record number may be used in conjunction
with the UPDATE function (see section 6.7) to identify the record for
modification. The RETRIEVE function is available through the RETRIEVE
command, or the RETR macro. The RETRIEVE function takes 5 parameters:
the file name, the logical expression of keywords describing the records
to be retrieved, the maximum number of records to be retrieved, the
output specification, and a label to be used in connection with the
CONTINUE function to be described later. The output specification
includes the current user's SYSCUT file (i.e., terminal or printer),
the printer, the main storage, or a count of the number of records
satisfying the expression. A complete discussion of the RETRIEVE func-
tion and its control blocks may be found in [6] and [8]. The
specification for the RETR macro may be found in [8]. The overall
flow is presented below.
The first major program in the RETRIEVE process is MACPROC [8]. MACPROC is accessed directly via the RETR macro and its associated SVC call. For handling macros, MACPROC calls the logical expression translator LOGTRAN [7] to decode the user's request. If the function is requested via the RETRIEVE command, then the TCP routes the request to MULTIRETR, which sets up the proper parameter list for MACPROC and calls LOGTRAN, so that for commands, MACPROC does not call LOGTRAN. Once MACPROC has called LOGTRAN, it calls SSBOPTR to find the user's SSB chain, and makes sure the file was opened. MACPROC then issues the TSOS OPEN macro for the file, readying it for processing by TSOS ISAM. MACPROC then sets up parameter lists, and passes control to the program RETRIEVE.

RETRIEVE first retrieves from disk the highest level of the EDMF directory for the file in question, using RETRDIR. Then RETRIEVE calls the directory search routine DECODE [4] to return the number of records associated with each keyword in the user's request expression. RETRIEVE then determines the prime keywords of the request. As discussed in Chapter 3, the use of the prime keywords minimizes the number of records retrieved for a given request. Once the prime keywords are known, RETRIEVE calls DECODE again to obtain the Head-Of-List-Addresses (HOLAs) for the prime keywords. The prime keywords and their HOLAs are set up in parameter lists for use in later stages of retrieval. Then RETRIEVE, having finished its processing, passes control to FORPROG.

FORPROG takes the user's request expression in internal form which was set up by LOGTRAN and, uses the information in the Record Format Block for the file (which was part of the FIB retrieved during
file opening), and associates the appropriate format number with the attribute in each keyword of the request expression. This enables the record checking routine RCDCHK to determine more easily whether the record satisfies the user's request. When proper format numbers are obtained, FORPROG is finished, and passes control to RETALG for record retrieval, checking, and output.

RETALG first sorts the MOLAs obtained by RETRIEVE (which are actually ISAM keys) into a retrieval address list. By calling on READWRIT [10] to get and assemble the ISAM records from disk, RETALG obtains an EDMF record. With the EDMF record, RETALG then calls RCDCHK to determine whether the retrieved record contains any of the prime keywords in the request expression. If so, the corresponding pointers (i.e., ISAM keys) in the record are obtained and added to the retrieval address list, making sure that no duplicate pointers are entered. The ISAM key used to retrieve the EDMF record is marked used, so that duplicate retrievals of the same record will not take place. After the pointers are obtained, RCDCHK then determines whether the record satisfies the user's request, and then returns to RETALG. If the record was satisfactory, RETALG then calls RCDLVNL for checking on privacy protection, and either RCDFRMT or COREFMT for outputting the record depending on the output specification. Once a given record is outputted, RETALG returns to the retrieval address list, selects the next unused address (i.e., an ISAM key), and proceeds to retrieve, process, and output the next EDMF record. This process continues until the number of records specified by the user have been retrieved, or all lists being searched
are exhausted (i.e. all records satisfying the expression have been found). If all records satisfying the expression have been found, RETALG returns control to MACPROC which issues a TSOS CLOSE for the file and returns to the user.

If only the number of records specified by the user have been retrieved, it is possible that the user might later wish to continue this retrieval from where it left off by utilizing the CONTINUE function. In this case, RETALG calls CONRET to save sufficient information to restart the retrieval from where it left off. This information is collected in a control block associated with the user's task. The control block consists of general register contents and a user-specified label as it appeared in his retrieval request. When CONRET finishes, control is returned to RETALG, which returns control to MACPROC to issue the TSOS CLOSE as described above.

5. The CONTINUE Function

The CONTINUE function allows the user to continue a retrieval request which was initiated earlier by a RETRIEVE function. For example, if a user wishes to receive one record at a time from a set of records satisfying a given logical expression of keywords, he may initiate the retrieval by issuing the RETRIEVE function with an indication that only one record is to be output. Subsequent records may then be received by repetitively issuing the CONTINUE function. The CONTINUE function is available through either the CONTINUE command or the CONT macro, and takes two parameters: the label of a retrieval request previously
executed that is to be continued, and the maximum number of records to be retrieved for this request (if this number is not specified, the number specified in the original retrieval request will be used). The overall flow of the process is presented below.

CONTINUE commands are routed by the TCP to the MULTCONT routine, which puts the parameters of the command into proper form for passing to the main processing routine, CONRET. CONT macros go directly to CONRET via their SVC call. CONRET searches its list of control blocks for the same user task to find one whose label matches the label in the CONTINUE request. If one such control block is found, CONRET then reloads the proper registers from the control block, restores various types of information, and passes control to RETALG. RETALG then continues the retrieval processing as if it had never been interrupted, and completes its processing as described in the previous section.

CONRET is composed of two separate sections, CONTLIST and CONTCAL. The CONTLIST is called by the retrieval program RETALG to save all necessary information to restart the retrieval. This information consists of all of RETALG's general register contents, the label supplied by the user with the RETRIEVE request, and a code used internally by RETALG. The restart information is stored in a control block whose format is shown in Figure 6.7. When CONTLIST is called by RETALG, it creates the control block and links it onto a list of similar control blocks for the user's task, each control block representing a retrieval which may be restarted by the user through the CONTINUE function.
<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXT</td>
<td>DSECT</td>
<td></td>
</tr>
<tr>
<td>LABEL</td>
<td>5C</td>
<td>Label</td>
</tr>
<tr>
<td>OPNCDE</td>
<td>C</td>
<td>Opcodes supplied by RETALG</td>
</tr>
<tr>
<td>USED</td>
<td>H</td>
<td>Indication if label and registers have already been used to continue X'FF' if so.</td>
</tr>
<tr>
<td>LINKAD</td>
<td>F</td>
<td>Link to next list for task no.</td>
</tr>
<tr>
<td>RREGS</td>
<td>16F</td>
<td>Registers of RETALG 14 - 13 when list set up</td>
</tr>
</tbody>
</table>

**Figure 6.7** Control Block for the CONTINUE Function

<table>
<thead>
<tr>
<th>Name</th>
<th>Length</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPARAM</td>
<td>DSECT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4F</td>
<td></td>
</tr>
<tr>
<td>RCONNUM</td>
<td>H</td>
<td>Number of records</td>
</tr>
<tr>
<td></td>
<td>C1 5</td>
<td></td>
</tr>
<tr>
<td>RLABEL</td>
<td>C1 5</td>
<td>Label</td>
</tr>
<tr>
<td></td>
<td>5C</td>
<td></td>
</tr>
<tr>
<td>RCODE</td>
<td>C1 1</td>
<td>Open code</td>
</tr>
</tbody>
</table>

**Figure 6.8** Input Parameter List for CONTCAL
<table>
<thead>
<tr>
<th>Register</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>1</td>
<td>Parameter list address</td>
</tr>
<tr>
<td>2</td>
<td>Task number and table address for Contlist</td>
</tr>
<tr>
<td>3</td>
<td>Beginning of table</td>
</tr>
<tr>
<td>4</td>
<td>Task number and table address for Contcal</td>
</tr>
<tr>
<td>5</td>
<td>Misc.</td>
</tr>
<tr>
<td>6</td>
<td>Misc.</td>
</tr>
<tr>
<td>7,8</td>
<td>Unused</td>
</tr>
<tr>
<td>9</td>
<td>Parameter list address (RETALG's)</td>
</tr>
<tr>
<td>10</td>
<td>Program base for work area</td>
</tr>
<tr>
<td>11</td>
<td>Parameter base - input</td>
</tr>
<tr>
<td>12</td>
<td>Program base register</td>
</tr>
<tr>
<td>13</td>
<td>Register save area address</td>
</tr>
<tr>
<td>14</td>
<td>Return address</td>
</tr>
<tr>
<td>15</td>
<td>Codes - error and return</td>
</tr>
</tbody>
</table>

Figure 6.9 CONRET Register Conventions
The CONTCAL is called by the command processor or interrupt analyzer when the user requests the CONTINUE function. This section searches the linked list of control blocks associated with the user's task to find the control block with a label which matches the one supplied by the user in his CONTINUE request. If the proper control block is found, CONTCAL restores the proper information and passes control to RETALG, which continues the retrieval process. If CONTCAL cannot find the proper control block due to,(e.g., an improper label supplied by the user), an error will be indicated. It is important to note that the command pre-processor MULTCONT will reject any CONTINUE command which does not have a label. The input parameter list for the CONTCAL section and the register conventions for the entire CONRET routine are shown in Figures 6.8 and 6.9 respectively.

6. The STORE Function

The STORE function allows the user to store new records to a specified EDMF file from terminals, main storage, card reader, or catalogued files. The STORE function is available through the STORE command or the ESTOR macro, and takes one parameter - the name of the file into which records are to be stored. A complete discussion of the STORE function and its control blocks may be found in [10]. The overall flow of the process is presented below.

The STORE routine is the main program in overall control of the STORE process. It can be used directly via the ESTOR macro and its associated SVC call. The STORE command is routed by the TCP to the
pre-processing routine MULTSTOR, which prepares the proper input parameter list for STORE. Before the STORE function is used, it is important to note that the file in which records are to be stored must be opened using the OPEN function. At the beginning of the record storing process, STORE calls the directory processing routine GENDIR for updating or creating the directory of the file. If the STORE function is used to put additional records into the file which already contains ENDM records, then GENDIR will retrieve the highest level of the file's directory into storage for directory update. If the STORE function is used to put new records into an empty file (i.e., no directory exists), then GENDIR will create a skeleton high-level directory for the file. In either case, GENDIR returns to STORE upon completion of the above work.

What STORE does next depends on whether the function is being accessed via a command or an SVC call. If STORE is accessed via an SVC call (i.e., through the ESTOR macro), then the record to be stored must be in main storage and is ready for immediate processing. In this case, STORE calls the routine SVCRCCHK which, along with the subroutines ATTRCHK and CHKATTR, checks to see whether the record organization is consistent with the record organization as specified in the record template. When STORE is accessed via the STORE command, STORE enters an interactive mode of processing by asking the user to enter the record, line-by-line, at the terminal. As each line comes in, the subroutine GETREC checks the line in relation to the other lines. When all lines of a record have been input, the record is created in main storage.
This record in main storage is then validated against the record template by ATTCHK and CHKATTR as before. The STORE command may also be used in batch processing mode where each field (i.e., a line) is punched on a card. Records on punched cards may be entered through the card reader in batches.

The record template (or Record Format Block) for the file is contained in the file's FIB, retrieved at the beginning of the OPEN process. If the input record through the STORE function does not satisfy the template (i.e., the record does not conform to the general record organization of the file) the record is rejected. If the record is acceptable, STORE calls on the subroutine GETNEWAD to prepare for storing the record on secondary storage, then calls the directory processing routine ADDIR [4] to add and merge the keywords and their associated information from the record to the directory of the file. ADDIR returns to STORE upon completion. STORE then converts the record in main storage to standard EDMF internal form, breaks up the record into fixed-size ISAM records, and calls the I/O routine READWRIT to store the ISAM records on disk. The reasons why the EDMF record must be divided into ISAM records will be discussed below. Once the record has been stored on disk, the STORE routine continues in either one of the two processing modes. If it was originally accessed via a command, STORE goes back to read another user record line-by-line, convert it to internal form, and so on, until the user indicates that he is finished with inputting records. At this time, STORE returns control to the user. If it was
accessed via an SVC all, STORE immediately returns control to the user. We note that each time the SVC is used the STORE processes one and only one record.

The process of breaking up an EDMF record into a number of ISAM records is as follows:

As was discussed in Chapter 2, the header of the EDMF internal record begins as shown in Figure 6.10 below. The third field in this header is a 5 byte packed decimal value which contains an ISAM key. This key is the ISAM of the first of a series of ISAM records which contain the EDMF record. The EDMF record must be broken up into several ISAM records because ISAM will not process records as long as the lengths that EDMF records are allowed to attain. The original version of ISAM under which the EDMF was implemented only allowed records of a maximum of 255 bytes, of which 12 were required for length and key information as shown in Figure 6.11. Thus, only records containing a maximum of 243 bytes of text could be accepted by the original

```
| 3 byte length | 2 byte count | 5 byte packed decimal key | ... |
```

Figure 6.10

ISAM. Thus EDMF records had to be broken up into blocks to fit into many ISAM records. The process by which this is done is as follows. The

```
| 4 byte length of record | 8 byte key-unpacked numeric | ... |
```

Figure 6.11
ISAM key on which the first block of the EDMF record is to reside is determined, using standard ISAM information and information stored in the FIB. For reasons which will be discussed later, this key must be a multiple of 1000. The first 242 bytes of the EDMF record are then stored in the ISAM record, along with a one byte marker which indicates whether or not the EDMF record is continued on the "next" ISAM record (the ISAM record with a key one greater than the current record). The next 242 bytes are then stored in this "next" ISAM record, along with another marker, and so on until the entire EDMF record is stored in ISAM records. We note that the EDMF does not care about the relative locations of the ISAM records. ISAM record 100 may be 500 records removed from ISAM record 101, but they are logically "next to" each other by virtue of their ISAM keys. This procedure means that, for example, an EDMF record which takes 1000 bytes will be broken up into 5 ISAM records. If the first ISAM key which is a multiple of 1000 is key 1000, then the EDMF record will be stored in ISAM records 1000, 1001, 1002, 1003, and 1004. Since there are 1000 possible ISAM keys between 1000 and 2000, this means that it is only possible to store EDMF records which require less than 1001 ISAM records, or up to 242,000 bytes using the current implementation. In the future, ISAM will support a greater record size (65,536 bytes) which, assuming 12 bytes for key and length as before, allows EDMF records of 65,524,000 bytes in length with the technique described above. Notice that the key field in the EDMF record is packed decimal, while that in the ISAM record is unpacked. The packing technique allows a saving of storage in the EDMF record, while the unpacked
key is required in the current ISAM (future ISAM will not require un-
packed keys).

7. The UPDATE Function

The Update function allows the user to modify existing records in
his EDMF files, and is available in either command or macro form. The
UPDATE function takes one parameter, the record number of the record
to be modified. We note from section 6.4 that the unique record number
of the record is dynamically generated by the RETRIEVE function. This
number is either displayed at the terminal when the record is printed
at the terminal or embedded in the record when the record is needed in
main storage by the user's program.

The main program in the UPDATE process is the command servicing
routine RESTREC. Both the macro and command enter RESTREC via different
entry points. All programs called by RESTREC return to RESTREC upon
completion. The first program called by RESTREC is SSBCHECK which
checks whether the file has been opened for update by the user and
locates the Service Status Block associated with the open file.

Subsequent processing depends on whether the command or the macro
is being used. If the macro is being used, the updated record must be
in main memory and is ready for immediate processing. In this case,
RESTREC calls DELREC to delete the existing record from the file, and
then calls UPDATCOM (part of the program STORE) to add the updated record
to the file. The STORE routine processes the updated record as if it
were a new record, as described in section 6.6. After STORE returns
control to RESTREC, RESTREC returns control to the user. If the command
is being used, several intermediate steps are required before DELREC and STORE are called. Since the user cannot physically update the record in command mode, the UPDATE function must perform this service for the user guided by user's specifications. In order to do this, RESTREC first retrieves the record to be updated into main storage by calling READWRIT and COREPMT, then calls RESTORE. RESTORE requests the user to input his update specifications to add, delete, or modify information in the record [7]. When modifications to the record are complete, RESTORE places the modified record in the file by calling DELREC and STORE as described above. Record and field level access control checks are performed as always in the STORE routine.

8. The DELETE Function

The DELETE function allows the user to delete existing records from his EDMF files, and is available in either command or macro form. The DELETE function, like the UPDATE function, requires the record number of the record to be deleted.

The main program in the DELETE process is the routine DELETE. DELETE is entered directly by the macro. The DELETE command is routed first through the command servicing routine MULTDELT, which sets up the proper parameter list for the DELETE routine. DELETE first calls SSBCHECK to ensure that the file was opened for update as it is required for record deletion. Next, DELETE calls READWRIT to retrieve the record to be deleted, and then passes the address of the retrieved record to DELREC which performs the actual deletion by setting a deletion bit in the record. Furthermore, the record is condensed in size by erasing values which are not on keyword lists. Once the deletion
process is completed, control is returned to the user.

9. Access Control Functions and Routines

The EDMF access control functions allow a file owner to assign or revoke access rights to any user of his files, and to display the access rights he has assigned to any other user. Other functions enable the EDMF to assign the file owner access rights to any new files which he creates, and to remove information concerning access rights to any files which have been deleted from the system.

9.1 The Authority Item

The authority item contains the access control information for a single user and is in standard EDMF record format. The authority items for all users of the system are contained in a single EDMF file, called the Authority Item File (AIF), which can be used only by the system access control programs.

The logical organization of the authority item is shown in Figure 6.12. It consists of the user identification, and a series of sections each of which contains the access control information for one file. The access control information for a file consists of the file name, file access control data, and a set of "protection entries" concerning restricted keywords for field-level protection and logical expressions of keywords for record-level protection. Since the authority item is stored as an EDMF record, each type of access control information is treated as a value and is therefore associated with an attribute. The attributes for the various types of access control information in the authority item are shown in Figure 6.13.
Figure 6.12 Logical Structure of Authority Item
<table>
<thead>
<tr>
<th>ATTRIBUTE NAME</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>USRID</td>
<td>user identification</td>
</tr>
<tr>
<td>GCINF</td>
<td>general control information</td>
</tr>
<tr>
<td>FLNAM</td>
<td>file name</td>
</tr>
<tr>
<td>FLACD</td>
<td>file access control data</td>
</tr>
<tr>
<td>FLAPN</td>
<td>file access program entry*</td>
</tr>
<tr>
<td>RCPDE</td>
<td>record protection description entry</td>
</tr>
<tr>
<td>RCPPE</td>
<td>record protection program entry*</td>
</tr>
<tr>
<td>FDPDE</td>
<td>field protection description entry</td>
</tr>
<tr>
<td>FDPPE</td>
<td>field protection program entry*</td>
</tr>
</tbody>
</table>

* entry provided for future expansion of access control capabilities

Figure 6.13 Attributes Associated With Fields in EDMF Authority Item
Access control at the file level is centered around the file access control data with attribute FIACD as shown in Figure 6.14. In addition to the file-level access control information, this field also indicates whether record- or field-level control information is present in the authority item.

Access control at the record and field levels are based on the protection entries shown in Figure 6.15. Each entry contains a code indicating whether it is an 'allow' or 'deny' entry. The entry also contains the internal form of a logical expression of an attribute name to be used for access control.

9.2 Storage, Retrieval, and Update of Authority Items

All of the access control functions to be discussed in the following sections involve the storage, retrieval, or update of users' authority items. Since the authority items were implemented as standard EDMF records, the access control functions may utilize the standard EDMF STORE, RETRIEVE, and UPDATE functions for processing the authority items. Each of these functions contains a special entry point for user by system routines. By entering through these entry points, system routines can bypass the privacy measures and access controls. In the case of the STORE and UPDATE functions, the calling system routine uses the system entry point and passes a parameter list containing the address of the authority item to be stored or restored in the file, the address of the record template for the authority item file, and other information. In the case of the RETRIEVE function, the system program calls the
4 bytes of control information where:

\[\begin{array}{c|c|c|c|c}
\text{aabb} & \text{codd} & \text{ef00} & \text{gh10} & 0000 \\
\hline
00 & 00 & 00 & 00 & 00 \\
01 & 01 & 01 & 01 & 01 \\
11 & 11 & 11 & 11 & 11 \\
\end{array}\]

\text{aa:} \\
00 user unauthorized \\
01 user shares file \\
11 user owns file

\text{bb:} \\
00 no access to file \\
01 standard access control via EDMF functions only \\
10 access control via user-supplied programs only \\
11 both types of access control are applied (EDMF first)

\text{c:} \\
0 file protection program present \\
1 file protection program absent

\text{dd:} \\
00 no read or write to file \\
01 read only file \\
10 write only file \\
11 read and write to file

\text{e:} \\
0 record protection description present \\
1 record protection description absent

\text{f:} \\
0 record protection program present \\
1 record protection program absent

\text{g:} \\
0 standard field protection active \\
1 standard field protection inactive

\text{h:} \\
0 field protection description present \\
1 field protection description absent

\text{i:} \\
0 field protection program present \\
1 field protection program absent

\text{Figure 6.14 File Access Control Data}
for logical expression

| aabb xxxx | DCB | KIB | Format Number Stack for logical expression |

RCPDE entry (record-level checking)

| aabb xxxx | 2 byte attribute length | n byte attribute name |
| format number stack for attribute |

FDPEE entry (field-level checking)

aabb xxxx is one byte entry type code where:

aa: 00 not applicable to read
10 deny read according to entry (i.e. a 'deny' description)
11 allow read according to entry (i.e. an 'allow' description)

bb: 00 not applicable to write
10 deny write according to entry (i.e. a 'deny' description)
11 allow write according to entry (i.e. an 'allow' description)

xxxx: 0000 entry deleted or no longer active
otherwise xxxx is ignored

Figure 6.15 Authority Item Protection Entries
authority item retrieval routine AIRETR with the logical expression
for the retrieval of the required authority item or items. This logical
expression is based on the keywords whose attributes are USRID (user
identification) or FLNAME (file name). We note that the authority items
are characterized by user identifications and file names. AIRETR then
calls LOGTRAN to translate the logical expression, and enters the RETRIEVE
function via the program RETRIEVE, supplying a code indicating the
retrieval of an authority item. The code informs the RETALG program
that the retrieved records are to be placed in privileged memory, and
that control should be returned to AIRETR when retrieval is completed.

In addition to serving as the main routine for authority item
retrieval, AIRETR also contains the record template for the authority
item file as a part of the routine. This means that the template is
always in memory and available to any other system routine, and that the
template for the authority item file does not appear in the File Informa-
tion File (see section 6.1) along with the user templates. This provides
additional privacy protection to the system file since it is impossible
for a user to reference the authority item without the template.

9.3 Access Control Considerations for File Creation and Deletion

File creation and deletion in the EDMF are carried out by the
functions ADDFILE and DELFILE, described in [10]. Since the creation
of a new file involves the assignment of access rights to the file
owner, and the deletion of a file involves the revocation of all access
rights, these two functions have access control responsibilities.
The **ADDFILE** function must create a new entry for the file owner's authority item, granting him ownership to the file. The file owner's authority item is retrieved using **AIRETR**, and updated using the **UPDATE** function. If the owner does not yet have an authority item, one will be created for him containing an entry for the file he just created.

The **DELFILE** function must delete all references to the deleted file in all user authority items, since the file is no longer accessible. **DELFILE** first checks to see whether the user issuing the request is the file owner. If he is an owner, **DELFILE** calls **AIRETR** to retrieve all authority items containing references to the deleted file (this can be easily done since the file name results in a keyword list in the authority item file; and the **EDMF RETRIEVE** function may be used). **AIRETR** returns all these authority items to **DELFILE**. By calling the **UPDATE** function, **DELFILE** can then remove the entries associated with the deleted file from these authority items.

9.4 Assignment of Access Rights

For each user of his files the file owner may assign or update access rights to his files by using the **ACCESS** function. The owner gives the function the user identification of the user to be given access, the name of the file to which he is giving access, and the specification of the access rights being granted.

The **ACCESS** function first verifies that the user issuing the request is actually the file owner. This is done by calling **AIRETR** to retrieve the authority item and checking the appropriate entry. Then
the authority item of the user to receive the access rights must be retrieved again using AIRETR (if the user has no authority item, the ACCESS function will create one for him). The ACCESS function then creates a new authority item entry using the specification provided by the owner, unless an update has been indicated, in which case the existing entry for the given file is appropriately modified. During the creations of the authority item entry, the routines LOGTRAN and FORPROG are called to translate logical expressions into internal form. Once the updated authority item has been created, the ACCESS function calls on the UPDATE function to restore the updated authority item to the file (or the STORE function if a new authority item has been created).

9.5 Revocation of Access Rights

The file owner may deny any user access rights to his files by using the DENY function. The owner must provide the name of the file to which access is being denied and the name of the user whose access rights are being revoked. The DENY function verifies that the user issuing the request is actually the file owner, as described in the previous section. Then, the authority item of the user whose access rights are being revoked is retrieved using AIRETR, the entry for the specified file name is deleted, and the updated authority item is restored using the UPDATE function. An additional function, DELUSER, may be used by the system administrator to delete a user's entire authority item from the system, thus preventing him from using any EDMP file. In this function, the system administrator provides the identification of the user whose authority item is to be deleted. The authority item is retrieved using
AIRETR and deleted using the DELETE function.

9.6 Display of Access Rights

The file owner may display the access rights of users of his files by using the RIGHTS function. He may specify that only rights granted to a particular user be displayed, or that rights granted to all users be displayed. The function checks to see whether the person issuing the request is the file owner. If so, it then retrieves all authority items containing entries for the file, transforms the internal codes to a readable form, and displays the information.

9.7 Access Verification and Control of User Requests

Access verification and control of user requests involves the file, record, and field-level checking described in Chapter 4, and is accomplished by several EDMF system modules, two of which are expressly for the purpose of access verification. The first of these two modules is AUTHCHK which performs the file-level check. AUTHCHK is called by the module OPNPROC during the EDMF OPEN function. It first performs the file-level checking for read-only or update privileges, then sets up field-level control information in the template for subsequent access control. For each restricted field as indicated in the authority item, there is an entry of control information for that field in the template.

The second of the access verification modules is RCDLVL, which performs the record-level checking. RCDLVL is called when a record has satisfied a retrieval request and is about to be given to the user, or when a record is about to be stored into a file. It calls the record checking program RCDCCHK to see if the record should not be given
to the user or stored into the file. This amounts to checking the record against the logical expressions ('deny' expression or 'allow' expression) and control information in the user's authority item.

Other EDMF system modules involved in access verification and control are the directory decoding routine DECODE, and the record output formatting routines RCDFRMT and COREFMT, all of which handle field-level checking. DECODE ensures that restricted keywords in logical expressions are not used for directory decoding. This is done by checking each attribute name of the keywords against its template entry. If the control information for the attribute has been set by AUTHCHK, as explained above, the keyword of the attribute will not be used for directory decoding. RCDFRMT and COREFMT ensure that no restricted field will appear in any output to the user, whether the output is a record in main storage, a display on a terminal, or a hard copy on a printer. By assembling the output record from the EDMF record and field control information in the template, these routines can avoid placing the restricted fields in the output records.
CHAPTER 7
SYSTEMS PROGRAMMING OF THE EDMF

In the previous chapters, we have presented the major concepts and facilities of the EDMF. These concepts include the generalized file structure and record organization, the authority item and its use for privacy protection and access control, and the problem-solving procedure. They are the core around which the design and programming of the system revolve. A large portion of the work is involved in the overall system design to integrate these concepts and facilities. In this chapter, we will present how the system is implemented. The implementation procedures and an overall picture of the modules of the EDMF will be discussed. Some of the lessons we have learned in the implementation process will also be enumerated.

1. Interface Considerations

As considered in Chapter 1, the EDMF is intended to be a general purpose time-sharing system. To implement it on the RCA Spectra 70/46G, we must write the EDMF as a time-sharing operating system of the Spectra 70. Since the existing RCA TSOS is also a general purpose system, we can incorporate the TSOS as part of the EDMF. The combined EDMF and TSOS can then be used by people who desire the facilities of the EDMF, or the TSOS, or both. Furthermore, the EDMF is designed so that people who do not wish to use EDMF facilities may continue to use the standard TSOS facilities without the need of reprogramming. These considerations
are of major importance in determining the EDMF-TSOS interfacing techniques and implementation decisions as discussed in the following sections.

The basic components of the RCA Time-Sharing Operating System (TSOS) concerning users' service requests (requests for memory, for I/O, etc.) and commands, with which the EDMF interfaces, are shown in Figure 7.1. The two at the top, the Interrupt Analyzer and the Terminal Command Processor, handle user service requests from user programs (via SVC instructions) and terminal commands (interactively or from cards), respectively. Once the service being requested is determined, they give control to an appropriate system program or sequence of programs to handle the request. Most of these servicing routines in turn call on common system routines, such as the memory management routines, for service of various needs. In an attempt to completely integrate the TSOS system routines into the EDMF design, the EDMF makes use of the same system routines to handle EDMF commands and service requests.

Thus, as far as TSOS is concerned, each EDMF service request (or command) is just another service request (command) to be switched by the Interrupt Analyzer (or Terminal Command Processor) to the appropriate EDMF routine or sequence of routines for processing. This is shown in Figure 7.2. This approach has the advantage of requiring little modification of TSOS. It means that the implementation can be concentrated on the EDMF, rather than on the redesign of the TSOS. Furthermore, services such as memory management are also available to the EDMF system routines. Since the TSOS was fairly new and untested at the time of the beginning
user requests via TSOS system macro call

SVC instruction

Interrupt Analyzer

user requests via TSOS system commands

Command Processor

TSOS higher level executive routines

A program or sequence of programs for handling each system macro or command.

Calls from executive routines requiring service

Lower-level executive routines

Device control, memory management

Figure 7.1 TSOS Control Program Flow
user requests via TSOS or EDMF system macro call

SVC instruction

Interrupt Analyzer

Command Processor

EDMF-TSOS higher level executive routines

A program or sequence of programs for handling each SVC call or command.

calls from executive routines requiring service

lower-level executive routines
device control, memory management

Figure 7.2 EDMF Control Program Flow
of the EDMF implementation, this approach is by far the best one to separate the EDMF bugs from the TSOS bugs. Now that implementation is nearly complete, this approach also enables the TSOS users who do not wish to utilize the EDMF facilities to continue on the new system without having to learn new conventions and vocabularies.

The utilization of the concept of design parallelism is prevalent in the EDMF. That is, the EDMF provides the same services to terminal users as it does to user programs. For example, the EDMF has a RETRIEVE command as well as a RETRIEVE macro, a STORE command and STORE macro, and so on. This parallelism enables the EDMF to serve not only non-conversational users (user programs run in batch or "background" mode) but also conversational users (interactive or terminal users). For example, a natural way of using a retrieval request (e.g. macro) in a user program to get multiple records satisfying a logical expression is to put the request in a loop, getting one record at a time through the loop. This is not, however, the natural way of using a retrieval request from a terminal, since loops are seldom used with commands. For this reason, the EDMF retrieval function is so designed that it can retrieve a specified number of records, or all the records satisfying a given logical expression. If it is desired to get a specified number of records, say just one at a time, the CONTINUE function may be used to continue the process of getting the next record after the RETRIEVE function is used once. Subsequent records can then be retrieved by repeated use of the CONTINUE function.
Finally, in considering the overall design of the EDMF, a point raised earlier in this chapter should be emphasized. The EDMF is designed to include the TSOS without major problems. However, minor problems occasionally do occur. For an example of difficulties with the TSOS, see the discussion of the EDMF OPEN function in section 1 of Chapter 6.

1.1 The Basic EDMF Input/Output Operations

Like the consideration of the overall EDMF design discussed above, the determination of the input and output methods to be used by the EDMF is influenced by the TSOS. This factor is especially important in the case of I/O operations because the TSOS, like many other time-sharing systems, prohibits the use of what is commonly called "physical I/O" by programs. This means that even system programs cannot set up their own chains of channel commands and perform I/O directly. Because the data channels are designed to place data in physical addresses, while the programs make references only to virtual addresses, virtual pages involved in I/O operations must be "frozen" in physical memory (i.e., not swapped out) until the data transfer to or from the channel is completed in order to insure that data is transferred to the proper place for the program. For this reason, I/O operations involve the coordination of both device control and memory management routines, as well as the task scheduler. Any attempt by programs, whether user or system, to perform their own I/O directly would result in chaos. Thus, the EDMF performs its I/O operations through the existing TSOS. The EDMF's choice of the higher level I/O - the Indexed-Sequential Access Method (ISAM) of the TSOS - as the basic I/O is influenced by a major change
in the lowest level TSOS logical I/O during the implementation of EDMF. By making such a choice of higher level I/O, any change of lowest level logical I/O or physical I/O will not affect the EDMF programs.

1.2 Interfacing the TSOS File Protection Facilities with the EDMF

Since the programs of the EDMF are designed to use I/O provided by the TSOS, it follows that all files created by the EDMF will be handled internally as standard TSOS files and will therefore be subject to the standard TSOS file protection measures. To enforce EDMF privacy protection and access control, the EDMF access control facilities are placed "on top of" those of TSOS. This arrangement works as follows: If the user performs all operations on EDMF files (i.e., files in the EDMF standard format) using the EDMF, he is required to issue the EDMF OPEN request, rather than the TSOS OPEN. The EDMF OPEN function routes the user's request through the EDMF access control mechanism, as well as setting up for a later TSOS OPEN for the file during storage or retrieval (see Chapter 6). However, just because the user satisfies the EDMF access control mechanism does not guarantee that the TSOS OPEN will be successfully carried out, because the EDMF access control mechanism is "on top of" that of TSOS. For example, a file owner may have granted a user access to a file, and indicated this in the user's EDMF authority item, but the file must also be catalogued with SHARE=YES under TSOS. Otherwise, the EDMF OPEN function will not be carried through. If the file catalogued under TSOS also requires TSOS passwords, then those passwords must be supplied. Otherwise, the same OPEN procedure will not take place.
Obviously, the EDMF privacy protection can only be applied if access to files is through the EDMF. Since all files created by the EDMF are TSO/S files, they might conceivably be accessed through TSO/S directly, thus bypassing the EDMF access control routines. This possibility is made unlikely by the following considerations. First, the EDMF records are split up internally, part being on the file, part (e.g. the template) being in a system control block which is stored in privileged memory and is inaccessible to users. In order to reconstruct the record in user-readable form, the data from the system control block and the data in the EDMF file must be combined. Without authorized access the unavailability of the system control blocks would render such records illegible. Secondly, user file protection programs, which can be introduced for their EDMF files, could be used to defeat unauthorized access to illegible EDMF records by enciphering the values in the EDMF record. Thirdly, the user must still satisfy the TSO/S file protection measures to do even this.

2. Systems Programming Techniques

In the preceding section we have discussed how the TSO/S interfaces with the EDMF. In this section we will discuss the systems programming techniques utilized in implementing and testing the programs of the EDMF. It also includes a discussion of the system backup and problems encountered during the implementation.

2.1 Program Coding

2.1.1 Choice of Systems Programming Language

The entire EDMF is written in assembly language, the traditional language of systems programming. At a time when higher-level languages
such as FORTRAN and COBOL are being used for almost every application, it may seem strange that the EDMF was not implemented in one of the higher languages. As with many other techniques used in the implementation, the choice of the assembly language was determined by the design requirements of the EDMF instead of the traditional considerations. The EDMF programs are designed to be time-sharing control programs, which should be reentrant. Special systems programming techniques must therefore be used to produce reentrant code. It was found that none of the compilers available on the RCA Spectra 70 produced reentrant code. The possibility of converting from another computer a high-level language compiler (e.g., the IBM 360 PL/1) which could produce reentrant code was investigated, but it was learned that the conversion would be too extensive and time-consuming to become immediately useful. For this reason, the EDMF was written in assembly language using reentrant coding techniques.

2.1.2 Use of Interactive Program Creation Techniques

One of the major benefits in using a time-sharing system is the ability of the programmer to create and assemble his program on-line, correct program errors dynamically, and re-assemble the program for subsequent execution. Thus all the programming work can be done at terminals. Unfortunately, during the period when much of the implementation work for the EDMF was taking place, these on-line facilities were either unavailable, or unreliable. During a major portion of the EDMF implementation, for instance, the only assembler available under TSOS was a tape-oriented assembler (ASMDOS) which required punched-card
input. Since the user was required to punch a program deck for input to the assembler, dynamic program creation by the File Editor at the terminal was of little use to the system programmer. Another factor which did not encourage the use of the File Editor was its unreliability. Frequently, source program files would be destroyed when the File Editor was used. The use of either card backup or duplicate source file was necessary. The maintenance of duplicate source files strained the capacity of the system disk storage on the one hand, and the use of card backup had rendered the interactive facility worthless on the other hand since it is just as easy to make corrections at a keypunch as it is at a terminal. Later, however, an improved version of the File Editor, and a disk-oriented (but not yet interactive) assembler became available. Most recent work on the EDMF has made use of one or both of these facilities. In particular, the File Editor has proved useful in the preparation of input to new EDMF programs.

2.2 Program Testing

2.2.1 System Generation and TESTSYS

One of the most important developments was a technique for combining new EDMF programs with the existing EDMF and T60S programs in such a way that they could be easily tested as parts of the total operating system.

To put this development in perspective, we note that most third-generation computers have what is known as a "privileged mode" of operation to protect their operating systems from user programs. Certain machine instructions and system macro-instructions are classed
as "privileged" and may only be used by components of the operating system which reside in protected memory. In order to be able to use privileged instructions, a program must be linked together with all the other routines of the operating system in an extended process known as "system generation". The EDMF system generation is a very time-consuming process which takes 3 hours.

A method to minimize the need of system generation was developed [1] as follows: At system initialization the file containing the existing system programs is read from disk into virtual memory. A dynamic linkage editor, EIDYLL, is then used to read from tape and cards what is known as a "loader". A loader is a file consisting of patches to system routines and new versions of system programs to overlay the old ones. EIDYLL reads the patch of complete program, locates the program to be modified or overlayed in virtual memory, and then performs the modification or overlay in virtual memory (the disk version of the system program is unchanged so that the same loaders must be read for each run). The method is used to modify existing system programs without the necessity of a system generation. Furthermore, this technique may be used for testing new system programs as part of the operating system, provided there is room in the operating system for them to overlay. The room was made available in a system generation by including in the system some dummy programs whose only purpose was to reserve space for EDMF programs to be tested as part of the operating system. The newly generated operating system was called TESTSYS [1]. Utilizing this method, the EDMF programs were tested as overlays onto dummy programs.
until they were checked out. When several checked-out programs were accumulated, a system generation was then performed to include the EDMF programs as permanent part of the operating system and to reserve the dummy programs for future use. The above process was used repeatedly until the completion of the EDMF. In addition, of course, bugs found in EDMF programs after a system generation were corrected temporarily either by patches or complete decks through the EIDYLL process and eliminated permanently at the next system generation.

2.2.2 Interactive Debugging

Another important advantage in time-sharing programming is the availability of interactive program testing facilities which allow programs to be loaded, executed, and stopped at specific points, displayed and modified dynamically by the programmer. The Interactive Debugging Aid (IDA) of TSOS is such a facility. Using IDA, stand-alone EDMF programs could be tested interactively in unprivileged mode until they were considered ready to be checked out as part of the operating system. They were then tested again using the technique described in the previous section. In the latter case, the privileged version of the program containing the checked-out logic would be used. Even though the EDMF program might be part of the operating system, the IDA could still be used for checkout. Using the system administrator's password IDA is allowed to be used for displaying and modifying parts of the operating system. Since the EDMF programs are clearly delineated from TSOS programs, it is possible to modify the EDMF programs without impairing the TSOS programs. Through the use of IDA, bugs can be found more rapidly than
using conventional testing procedures, and corrections can be made and tested immediately. Once corrections are made and verified via IDA, they can then be made more permanent through the EIDYLL process.

2.3 System Backup

Early in the development of the EDMF, the only convenient way to back up the system was via card decks, because disk and tape utility programs were unavailable. Thus copies of each source program deck and each object program deck were kept. In addition, the Object Module Library (OML) tape used in the system generation process was saved. This OML tape contained all the EDMF and TSOS object modules. The need to resort to bulky card decks was clumsy, but necessary, in view of the problems mentioned in section 2.1.2. Also, of course, source listings were kept in a master binder. Prior to a system generation, both source and object modules, and source listings, were required to be submitted and dated with the date of the system generation. In this way, the decks and listings for one system generation could be kept together. When a new version of EDMF was generated, the old version would still be retained while the new version was undergoing tests. With several versions of EDMF active at the same time, confusion was created because the same deck was often used in different versions of the system. The use of the same deck in different versions required either the punching of duplicate decks, or some indication that the same deck (and listing) applied to more than one system. This requires a unique identification procedure for different versions of the same program and listings keyed to the decks.
Later, with the development of disk to tape and tape to disk utility programs, and an improved File Editor, the use of source modules on disk as the prime backup became feasible. Secondary backup is provided by tape copies of the disk files and punched decks. The unique keys for program versions can be included in the filename, for an example, "SOURCE.EDMF3.RETALG". However, the problem of keying the source listing to the proper program version in the correct system remains.
CHAPTER 8

FUTURE WORK AND EXPERIMENTATION

It is anticipated that use of the EDMF will expand, involving increasing numbers of user programs which can make good use of the EDMF data management functions as well as additional systems programs which can extend the capabilities of the EDMF itself. This is based on the fact that the EDMF is useful not only as an operating system to support advanced applications, which have diverse data management requirements, but also as an experimental system to investigate new data management ideas.

1. Use of the EDMF to Support Advanced Applications

The generalized file structure and record organization provided by the EDMF, as well as its storage and retrieval capability in batch or interactive modes, make the EDMF an ideal system for use in advanced applications which require extensive data handling. The ability to call on EDMF functions through systems macros from user programs is especially valuable in this regard. In addition, a set of "front-end" subroutines have been written by Mr. D. Kountanis of the Moore School, which allow EDMF functions to be used from FORTRAN or COBOL programs. Thus, users of higher-level languages may also utilize the advanced capabilities of the EDMF, which further encourages the development of applications programs based on the EDMF.

Many new computer applications require the type of facilities provided by the EDMF. For example, hospitals are becoming increasingly
interested in medical data base systems to provide a wide range of services, such as providing administrative information for the hospital staff, storage and retrieval or pathology data, on-line diagnostic aids, etc. Such services require the ability to access system functions from user-written programs which produce specialized reports or displays, the ability to store, retrieve, and update data in an interactive mode, and the existence of elaborate access control capabilities, since the protection of confidential information concerning patients is vitally important when patients' records in the data base are used for medical inquiry and research. In addition, it is often important that all hospitals in a given area be able to use the same data base, since, for example, victims of accidents are usually taken to the closest hospital, rather than to the hospital which last treated the patient. The EDMF provides the user-program interface, on-line processing, and access control capabilities required by this application. Work is currently in progress by Mr. K. Nakanishi of the Moore School to use the EDMF to process patients' records generated in the Cardiac Catheterization Laboratory of the Hospital of the University of Pennsylvania. It is anticipated that a set of FORTRAN programs will be written to accept patient data in a convenient format and to produce several kinds of on-line displays and batch reports.

Another type of application is the development of data bases containing various types of urban data for use by city departments. This application requires the use of elaborate record organizations and file
structures, the ability to provide answers to complex queries, and the ability to produce many different reports with the information required by different city departments. The EDMF provides the record organizations, file structures, and efficient retrieval capability required by this application. Mr. J. Morrissen of the Moore School is currently creating files containing such complex urban data, and is developing an advanced report generator and query facility using the FORTRAN EDMF functions [20].

Still another application is the management of the components of a product in a research, development, and manufacturing environment. This requires the ability to define complicated file structures which define individual parts of the product, how they are used to form assemblies and subsystems, and how parts and assemblies fit together to form various configurations of the same product. In addition, facilities for on-line update and retrieval must be available. These are all basic capabilities of the EDMF. Currently, Mr. T. W. McIntyre is applying the EDMF to such a configuration problem in the development and manufacturing of experimental re-entry vehicles.

2. Use of the EDMF as an Experimental Tool

The flexibility of the EDMF makes it a natural experimental tool for investigating many areas in the field of data management. For instance, Lance J. Hoffman [9] noted: "The implementation of a (real or simulated) system which uses many counter-measure techniques would be a very desirable undertaking. It would enable us to evaluate the
effectiveness and the costs of each technique. A suitably designed system would at the same time allow us to vary the structure of a file. Since the structure of a file may affect quite strongly the access control method used, a number of interesting experiments could be performed. ... [For] example, the existence of indexes into a tree-structured file (i.e. the use of an inverted file) might strongly alter the operating characteristics of the access control mechanism by allowing control information to reside in the indexes rather than (say) with the data itself. Further investigation of this relationship is also warranted." The EDMF has both the capabilities mentioned above. In the following sections, we suggest some areas for experimenting with the EDMF.

2.1 Multiple Record Format Types

It was noted in Chapter 2 that the "type of format" entry in the template is currently unused. The introduction of this entry is the result of theoretical study of format types as presented in [12,13]. The current EDMF format represents the most general type of those discussed in [12,13], however, if a way was found to implement these other types of formats, savings in storage requirements for the Record Control Block of each record could result. As an example of the (secondary) storage requirement for pointers and control information required by the EDMF to provide generality in record organization, a graph in Figure 8.1 shows the percentage of the storage used for useful text in a record versus the number of fields in a record, with different curves for different average field sizes. This graph shows, as might be expected, that longer fields require relatively less storage for pointers and
Figure 8.1 Percentage of EDMF Record Containing Non-Control Information as Field Size Increases
control information than shorter fields. If every field in the template were of fixed length and had a fixed number of repetitions, for example, then there would be no need to append a Record Control Block in the record at all, since record organization would be uniform throughout the file. By knowing the format type, it is possible to conserve storage by reducing or eliminating the Record Control Blocks.

2.2 Memory Acquisition and Release

Presently, most routines in the EDMF acquire their own memory for parameter lists and work areas, using the TSOS dynamic memory management macro-instructions. Because the memory is acquired separately by many routines for different purposes, it is difficult to determine when and how much of this memory can be safely released to the system. At present, this is not a problem because use of system memory is not heavy and use of the EDMF routines is not extensive. Furthermore, the acquired memory is automatically released when the user logs off the system. However, with extensive use of TSOS and the EDMF, increasing use of memory will obviously degrade the performance of the system. The development of a central memory acquisition and releasing algorithm for the EDMF would improve overall system performance.

2.3 Program Substitution Facility

A facility allows the user to define his own directory decoding and file search programs if he chooses not to use the ones provided by the EDMF.
2.4 List Processing Capability

Currently, the EDMF allows file structures to be defined with any number of lists per keyword (as explained in Chapter 3), and utilizes an algorithm which attempts to keep each list the same length for a given keyword. However, the system does not permit the user to specify on which list an input record should be placed during storage, or from which list records are to be retrieved. The addition of the capability for specifying lists for the storage and retrieval of records via commands and macros would enable the user to designate lists for special processing purposes, to create still more new file structures, and to optimize (say, disk) access time by placing all records in one list in the same storage unit (say, a disk cylinder).

2.5 Lower-Level I/O

For reasons discussed in Chapter 7, the EDMF utilizes the Indexed Sequential Access Method (ISAM) of OS as its basic I/O method. However, it might be useful to investigate possible savings in I/O time by utilizing the Primitive Access Method (PAM), which was not available at the beginning of the EDMF implementation, as the basic I/O method. Although it may turn out that this investigation would involve just as much overhead as using ISAM, it is possible that the use of PAM would enhance directory set-up and processing, and control over space used on direct-access devices.
REFERENCES


REFERENCES (continued)


REFERENCES (continued)


REFERENCES (continued)

26. R. L. Wexelbian and H. A. Freedman, "The MULTILANG On-Line Pro-
    gramming System," Proc. AFIPS 1967 Spring Joint Computer Con-
APPENDIX

SYSTEM FLOWCHART OF EDMF ROUTINES

Note: indicates a connection to a program which appears on another page.
The RETRIEVE and CONTINUE Functions
The STORE Function
The UPDATE and DELETE Functions
The OPEN and CLOSE Functions
The ADDFILE, DELFILE, MEXEC, and FPARAM Functions
The DENY, ACCESS, RIGHTS, and DELUSER Functions